CSE 3139

DATABASE MANAGEMENT SYSTEMS

FALL 2024

COURSE PROJECT

E-COMMERCE DATABASE DESIGN WITH API CODE IMPLEMENTATION

Student Id	Name Surname	1st /2nd Ed.
210316016	BERKE ALPASLAN	2nd Education
210316076	MERT DOĞAN AYGÜN	2nd Education
220316055	GÜL YEŞİLGİL	2nd Education

TABLE OF CONTENTS

INTRODUCTION 1

1. Project Overview 1
2. Database Design Approach 1
3. Data Population 1
4. Query Implementation 1
5. API Integration 1
6. Purpose and Objectives 1

APPLICATION SCENARIO 2

1. Scenario Description 2
2. Key Entities and Their Roles 2
3. Relationship Between Entities 3
4. Key Functionalities Enabled by the Database 3

DATABASE DESIGN 5

1. E-R Diagram 5
2. Table and Relations 6
   - Customers 6
   - Sellers 7
   - Products 8
   - Categories 8
   - Advertises 8
   - Orders 9
   - ProductOrder 9
   - Baskets 9
   - BasketProduct 10
   - Reviews 10

Data Population 11

- Advertises Table 11
- BasketProduct CrossTable 12
- Baskets Table 12
- Categories Table 13

- Customers Table 13
- Orders Table 14
- ProductOrder CrossTable 14
- Products Table 15
- Reviews Table 15
- Sellers Table 16

Queries and Results 17

1. Number of Products and Average Price in Each Category 17
2. Products in Categories Without Subcategories 18
3. Customers with Both Baskets and Purchases 19
4. Average Basket Value per Customer 20
5. Price Comparison of Products within the Same Category 21
6. Total Advertisements per Category 22
7. Top 5 Most Expensive Products 23
8. Products with Low Stock Levels 24
9. Customer Segmentation by Spending 25
10. Category-based Price Ranges and Product Distribution 25

API DEVELOPMENT 27

- Purpose and Features of the API 27
- Technologies and Frameworks Used 27
- Onion Architecture Layers 28

API TESTING RESULTS WITH SWAGGER 52

- **Add Customer Operation** 54

  Implementation with Mediator Pattern of Add Customer Operation 54

  Swagger Test of Add Customer Operation 56

  Database Confirmation of Add Customer Operation 57
- **Get All Customers** 58

  Implementation with Mediator Pattern of Get All Customers Operation 58

  Swagger Test of Get All Customers Operation 59

  Database Confirmation of Get All Customers Operation 60
- **Remove Customer** 61

Implementation with Mediator Pattern of Remove Customer Operation 61
Swagger Test of Remove Customer Operation 62
Database Confirmation of Delete Customer Operation 63
Summary 63
CONCLUSION 64

TABLE OF FIGURES

Figure 1 E-Commerce Database E-R Diagram 5
Figure 2 Advertises Table 11
Figure 3 Continuation of Advertises Table 11
Figure 4 BasketProduct CrossTable 12
Figure 5 Basket Table 12
Figure 6 Categories Table 13
Figure 7 Customers Table 13
Figure 8 Continuation of Customers Table 13
Figure 9 Orders Table 14
Figure 10 ProductOrder CrossTable 14
Figure 11 Products Table 15
Figure 12 Reviews Table 15
Figure 13 Continuation of Reviews Table 15
Figure 14 Sellers Table 16
Figure 15 Continuation of Sellers Table 16
Figure 16 Query_1 - Number of Products and Average Price in Each Category 17
Figure 17 Result_1 - Number of Products and Average Price in Each Category 17
Figure 18 Query_2 - Products in Categories Without Subcategories 18
Figure 19 Result_2 - Products in Categories Without Subcategories 18
Figure 20 Query_3 - Customers with Both Baskets and Purchases 19
Figure 21 Result_3 - Customers with Both Baskets and Purchases 19
Figure 22 Query_4 - Average Basket Value per Customer 20
Figure 23 Result_4 - Average Basket Value per Customer 20
Figure 24 Query_5 - Price Comparison of Products within the Same Category 21
Figure 25 Result_5 - Price Comparison of Products within the Same Category 21
Figure 26 Query_6 - Total Advertisements per Category 22
Figure 27 Result_6 - Total Advertisements per Category 22
Figure 28 Query_7 - Top 5 Most Expensive Products 23
Figure 29 Result_7 - Top 5 Most Expensive Products 23
Figure 30 Query_8 - Products with Low Stock Levels 24
Figure 31 Result_8 - Products with Low Stock Levels 24
Figure 32 Query_9 - Customer Segmentation by Spending 25
Figure 33 Result_9 - Customer Segmentation by Spending 25
Figure 34 Query_10 - Category-based Price Ranges and Product Distribution 26
Figure 35 Result_10 - Category-based Price Ranges and Product Distribution 26
Figure 36 BaseEntity 29
Figure 37 User Abstract Class Entity 30
Figure 38 Advertise Entity Class 31

Figure 39 Basket Entity Class 31
Figure 40 Category Entity Class 32
Figure 41 Customer Entity Class 32
Figure 42 Order Entity Class 33
Figure 43 Product Entity Class 33
Figure 44 Review Entity Class 34
Figure 45 Seller Entity Class 34
Figure 46 BasketProduct CrossTable Entity Class 36
Figure 47 ProductOrder CrossTable Entity Class 37
Figure 48 Rating Enum Class 38
Figure 49 IRepository Interface 39
Figure 50 IReadRepository Interface 40
Figure 51IWriteRepository Interface 40
Figure 52 ICategoryReadRepository Interface 41
Figure 53 ICategoryReadRepository Interface 41
Figure 54 IUnitOfWork Interface 42
Figure 55 AddCustomerCommandRequest Class 43
Figure 56 AddCustomerCommandHandler Class 44
Figure 57 AddCustomerCommandResponse Class 44
Figure 58 ReadRepository Class 45
Figure 59 WriteRepository Class 46
Figure 60 BasketWriteRepository Class 47
Figure 61 BasketReadRepositoryClass 47
Figure 62 UnitOfWork Class 48
Figure 63 DbmsAPIDbContext Class 49
Figure 64 DbmsAPI Class OnModelCreating Method 50
Figure 65 DbmsAPIDbContext Class SaveChangesAsync Method 50
Figure 66 CustomerController Class 51
Figure 67 appsetting.json PostgreSQL Connection String 52
Figure 68 Swagger UI Part_1 52
Figure 69 Swagger UI Part_2 53
Figure 70 Swagger UI Part_3 53
Figure 71 Customers Controller 54
Figure 72 AddCustomerCommandRequest Mediator Pattern Class 55
Figure 73 AddCustomerCommandHandler Mediator Pattern Class 55
Figure 74 AddCustomerCommandResponse Mediator Pattern Class 56
Figure 75 Swagger AddCustomer Operation Request Page 56
Figure 76 Swagger AddCustomer Operation Response Page 57
Figure 77 PostgreSQL Customers Table After AddCustomer with Swagger 57
Figure 78 GetAllCustomersQueryRequest Mediator Pattern Class 58

Figure 79 GetAllCustomersQueryHandler Mediator Pattern Class..... 58
Figure 80 GetAllCustomersQueryResponse Mediator Pattern Class 59
Figure 81 Swagger GetAllCustomers Operation Request Page 59
Figure 82 Swagger GetAllCustomers Operation Response Page 60
Figure 83 PostgreSQL Customers Table to Confirmation GetAllCustomers Operation from Swagger..... 60
Figure 84 DeleteCustomerCommandRequest Mediator Pattern Class 61
Figure 85 DeleteCustomerCommandHandler Mediator Pattern Class 61
Figure 86 DeleteCustomerCommandResponse Mediator Pattern Class 62
Figure 87 Swagger DeleteCustomer Operation Request Page 62
Figure 88 Swagger DeleteCustomer Operation Response Page 63
Figure 89 PostgreSQL Customers Table After DeleteCustomer Operation From Swagger 63

INTRODUCTION

1. Project Overview

In this project, the **E-Commerce** scenario was selected to design and implement a database system. The project focuses on creating a robust and well-structured database to manage key entities such as customers, products, orders, and order details. These entities and their relationships are modeled to ensure efficient data storage, retrieval, and manipulation in alignment with the requirements of an e-commerce platform.

2. Database Design Approach

The database was designed using **PostgreSQL**, following the principles of database normalization to minimize redundancy and ensure data integrity. An **Entity-Relationship (E-R) Diagram** was created to visualize the entities and relationships before implementation. Each table was carefully structured with appropriate primary keys, foreign keys, and constraints to enforce referential integrity and support complex queries.

3. Data Population

After defining the schema, sample data was populated into the tables to represent realistic scenarios in an e-commerce environment. Each table contains 10 sample records, demonstrating various interactions such as customer purchases, product availability, and order details. These records are used to validate the functionality of the database and test different query scenarios.

4. Query Implementation

Ten critical queries were written to address common business needs in the e-commerce scenario. These queries provide insights such as customer order history, popular products, and revenue tracking. Screenshots of query results are included in the report to illustrate their outputs and relevance to the application.

5. API Integration

To enhance the usability of the database, a web API was developed using **ASP.NET Core**. While the primary focus remains on database design and functionality, the API demonstrates how the database can be effectively utilized for real-world applications by enabling seamless data interaction for external clients.

6. Purpose and Objectives

The primary goal of this project is to showcase proficiency in database design and implementation, ensuring the system aligns with real-world application scenarios. The database has been structured to support scalability, maintainability, and efficient data processing, reflecting best practices in modern database management.

APPLICATION SCENARIO

1. Scenario Description

The project is designed to model a database system for an **E-Commerce platform**, where sellers can list products for sale, and customers can browse, review, and purchase these products. The database supports key features such as product categorization, order management, basket functionality, and customer reviews. The primary objective is to provide a robust backend for an e-commerce website where users can interact seamlessly with various entities like products, orders, and advertisements.

In this system:

- Sellers can list products by creating advertisements, specifying the price, and providing descriptions for each product.
- Customers can add products to their baskets, place orders, and leave reviews for purchased products.
- Products are organized into categories, with the ability to create hierarchical category structures using parent-child relationships.
- Orders can include multiple products, and an order's details are managed via a many-to-many relationship between the **Orders** and **Products** tables.

2. Key Entities and Their Roles

- **Customers:** Represent individuals purchasing products on the platform. They can create baskets, place orders, and leave reviews for products they have interacted with.
- **Sellers:** Represent individuals or organizations listing products for sale. Sellers can manage their advertisements and view order details for their products.
- **Products:** Core entity representing items available for sale. Each product belongs to a category and can have multiple advertisements associated with it.
- **Categories:** Hierarchical structure to organize products. Each category can have subcategories, creating a parent-child relationship.
- **Advertisements:** A table to link products with sellers, allowing sellers to define pricing and additional details for specific products they wish to sell.
- **Orders:** Represents purchases made by customers. Each order is linked to a customer and can include multiple products.
- **Baskets:** Allows customers to group products they intend to purchase before placing an order.

- **Reviews:** Enables customers to provide feedback and ratings for products they have purchased.

3. Relationship Between Entities

- A **Customer** can have one or more **Baskets**, but a basket belongs to only one customer.
- A **Basket** can contain multiple **Products**, creating a many-to-many relationship modeled via the **BasketProduct** table.
- An **Order** is associated with one **Customer** and can include multiple **Products**, represented via the **ProductOrder** table.
- Each **Product** is assigned to one **Category**, and a category can have subcategories, forming a self-referential relationship.
- **Sellers** create **Advertisements** to sell specific **Products**, with pricing and stock defined at the advertisement level.
- **Reviews** are linked to both **Customers** and **Products**, ensuring that only verified customers can leave feedback for products they purchased.

4. Key Functionalities Enabled by the Database

- **Product Listing and Categorization:**
    - Products are listed by sellers through advertisements, and these products are organized into categories for easier navigation.
- **Order Management:**
    - Customers can place orders containing multiple products, and sellers can track orders linked to their advertisements.
- **Basket Management:**
    - Customers can add products to baskets before placing orders, with the flexibility to modify basket contents as needed.
- **Customer Reviews:**
    - Verified customers can leave feedback and ratings for products, enhancing trust and transparency on the platform.

- **Dynamic Pricing via Advertisements:**
    - Sellers can define different prices and stock amounts for the same product, enabling competitive pricing strategies.

- **Design Decision Highlights**

- Separate tables for **Customers** and **Sellers** were created instead of using a single **User** table with a discriminator column. This decision was made to keep the data structures simpler and avoid overloading a single table with distinct roles and attributes.
- The **Advertisements** table was introduced to allow sellers to manage product pricing and stock independently, providing flexibility in the e-commerce model.
- A hierarchical category system was implemented to allow products to be organized effectively, supporting both general and specific classifications.

This design ensures scalability, maintainability, and the ability to handle complex queries, meeting the needs of a modern e-commerce platform.

DATABASE DESIGN

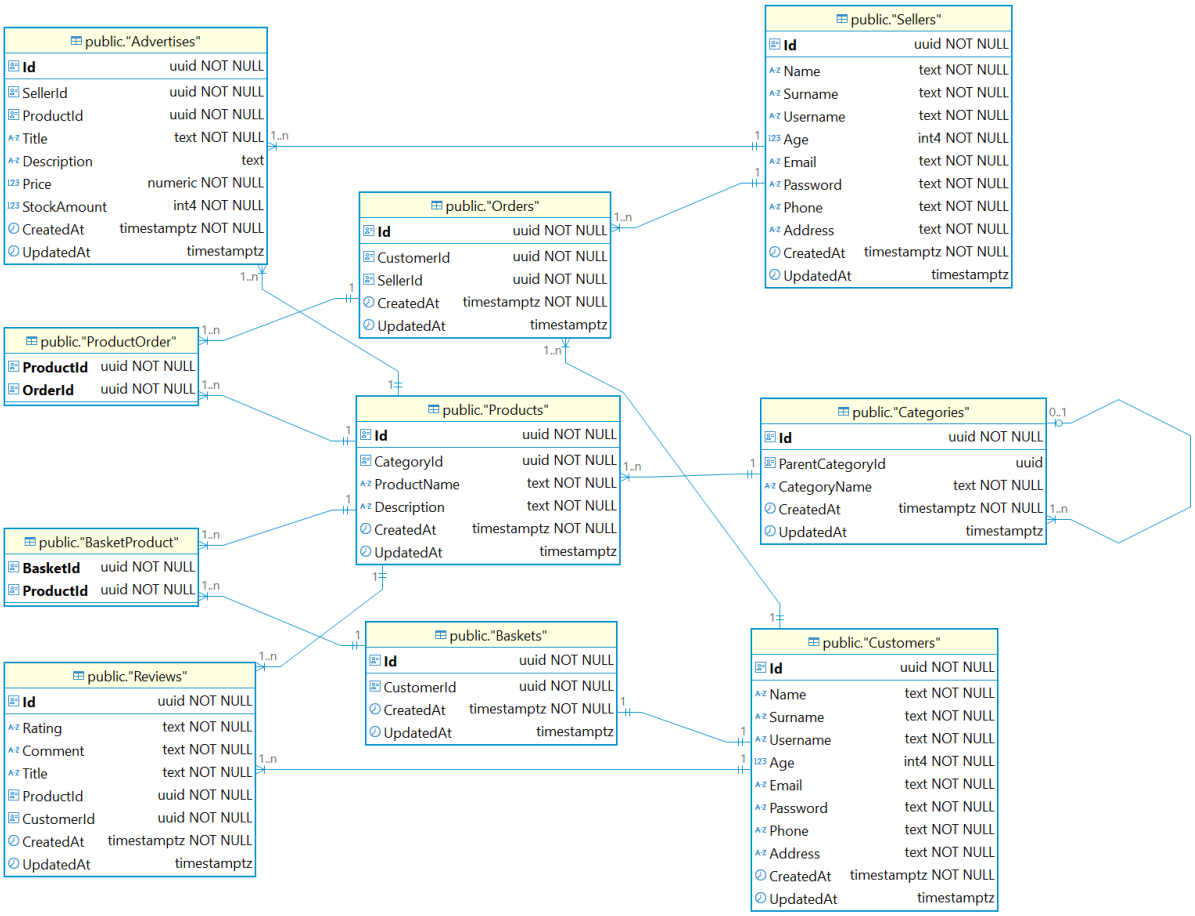


Figure 1 E-Commerce Database E-R Diagram

1. E-R Diagram

The Entity-Relationship (E-R) diagram illustrates the database structure for our e-commerce platform. It consists of several entities (tables) and their relationships, designed to meet the system's functional requirements. Below is a detailed explanation of the entities and their relationships:

- **Customers**: Represents users of the platform who can purchase products. Each customer can create multiple orders and maintain a shopping basket.
- **Sellers**: Represents users who sell products on the platform. Each seller can create multiple advertisements to list their products for sale.
- **Products**: Contains details about available products, such as their name, description, and category. Each product belongs to one category.
- **Categories**: Organizes products into a hierarchical structure, allowing for parent-child category relationships.

- **Advertises**: Represents product listings created by sellers, including pricing, stock amount, and descriptive information.
- **Orders**: Tracks customer purchases, including the seller and the ordered products.
- **ProductOrder**: Acts as a join table between Orders and Products, capturing the details of products in each order.
- **Baskets**: Represents customers' shopping baskets, which can hold multiple products.
- **BasketProduct**: Acts as a join table between Baskets and Products, capturing the products added to a customer's basket.
- **Reviews**: Captures customer feedback for products, including ratings and comments.

Key relationships:

- A customer can create multiple orders (one-to-many).
- A seller can create multiple advertisements (one-to-many).
- A product belongs to a single category, but categories can have a hierarchical structure (one-to-many).
- An order can include multiple products, and a product can be part of multiple orders (many-to-many via ProductOrder).
- A basket can include multiple products, and a product can be part of multiple baskets (many-to-many via BasketProduct).

2. Table and Relations

The tables are implemented in PostgreSQL with the following structure:

- Customers
    - **Columns**:
        - Id (uuid, Primary Key)
        - Name (text)

        - Surname (text)
        - Username (text)
        - Age (int4)
        - Email (text)
        - Password (text)
        - Phone (text)
        - Address (text)
        - CreatedAt (timestamptz)
        - UpdatedAt (timestamptz)
    - **Relations**: One-to-many with Orders and Baskets.

- Sellers
    - **Columns**:
        - Id (uuid, Primary Key)
        - Name (text)
        - Surname (text)
        - Username (text)
        - Email (text)
        - Password (text)
        - Phone (text)
        - Address (text)
        - Age (int4)
        - CreatedAt (timestamptz)
        - UpdatedAt (timestamptz)
    - **Relations**: One-to-many with Advertises and Orders.

- Products
    - **Columns**:
        - Id (uuid, Primary Key)
        - CategoryId (uuid, Foreign Key -> Categories(Id))
        - ProductName (text)
        - Description (text)
        - CreatedAt (timestamptz)
        - UpdatedAt (timestamptz)
    - **Relations**: One-to-many with Advertises, many-to-many with Orders via ProductOrder, and many-to-many with Baskets via BasketProduct.

- Categories
    - **Columns**:
        - Id (uuid, Primary Key)
        - ParentCategoryId (uuid, Nullable, Foreign Key -> Categories(Id))
        - CategoryName (text)
        - CreatedAt (timestamptz)
        - UpdatedAt (timestamptz)
    - **Relations**: Self-referential one-to-many (parent-child relationship with categories).

- Advertises
    - **Columns**:
        - Id (uuid, Primary Key)
        - SellerId (uuid, Foreign Key -> Sellers(Id))
        - ProductId (uuid, Foreign Key -> Products(Id))
        - Title (text)
        - Description (text)

        - Price (numeric)
        - StockAmount (int4)
        - CreatedAt (timestamptz)
        - UpdatedAt (timestamptz)
    - **Relations**: One-to-many with Sellers.

- Orders
    - **Columns**:
        - Id (uuid, Primary Key)
        - CustomerId (uuid, Foreign Key -> Customers(Id))
        - SellerId (uuid, Foreign Key -> Sellers(Id))
        - CreatedAt (timestamptz)
        - UpdatedAt (timestamptz)
    - **Relations**: Many-to-many with Products via ProductOrder.

- ProductOrder
    - **Columns**:
        - ProductId (uuid, Foreign Key -> Products(Id))
        - OrderId (uuid, Foreign Key -> Orders(Id))
    - **Relations**: Join table for Orders and Products (many-to-many) (CrossTable).
- Baskets
    - **Columns**:
        - Id (uuid, Primary Key)
        - CustomerId (uuid, Foreign Key -> Customers(Id))
        - CreatedAt (timestamptz)
        - UpdatedAt (timestamptz)
    - **Relations**: Many-to-many with Products via BasketProduct.

- BasketProduct
    - **Columns**:
        - BasketId (uuid, Foreign Key -> Baskets(Id))
        - ProductId (uuid, Foreign Key -> Products(Id))
    - **Relations**: Join table for Baskets and Products (many-to-many) (CrossTable).

- Reviews
    - **Columns**:
        - Id (uuid, Primary Key)
        - Rating (int4)
        - Comment (text)
        - Title (text)
        - ProductId (uuid, Foreign Key -> Products(Id))
        - CustomerId (uuid, Foreign Key -> Customers(Id))
        - CreatedAt (timestamptz)
        - UpdatedAt (timestamptz)
    - **Relations**: One-to-many with Products and Customers.

This structure ensures a robust, scalable, and normalized design that meets the requirements of the e-commerce platform. Each table has clearly defined relationships, keys, and constraints to maintain data integrity.

Data Population

- Advertises Table

Advertises | Enter a SQL expression to filter results (use Ctrl+Space)

	Id	SellerId	ProductId	Title	Description	Price
1	73d9e3a4-8615-4d1a-aae5-13fba501020d	a14b1ebf-bad4-4c5e-b60b-f9538e3f14f2	78cffb14-c078-46d6-9285-678b14204957	Factory New	Best Price	750
2	1de62716-1b71-48cb-9a99-b3c83b4de907	ee035cf8-e44b-4a3f-adc2-6e24e83a938d	0ba945c5-409f-4724-8767-72ce3e1ae686	Well Worn	Old Product but Good Price	20
3	884fcb59-1c2c-4a18-a476-3738b508fe65	95de2d74-3085-406c-9376-c493f18167b4	d652cf51-642c-4fdd-8fdd-e3ba8211e125	Factory New	It makes you icon	125
4	8efd9fc3-7d5a-4e74-9dd1-80efd9c70dc8	6f543965-88b3-4bd0-aeb1-643c8739a10c	67a2ecf4-bcf7-4088-bf65-0e53e720a3a1	Special Design	Limited Edition	25
5	890bea2b-a9b0-41e4-bc02-2062735f9272	77fcb774-8900-4776-ada4-2e7fa73bd6db	01ecae5f-7229-4187-b9ee-0ad68bba1072	Fake	Fake Product but Cheap	12
6	83f05ab6-792b-459b-af1b-dd25da1045ad	0067aa0a-6300-4c43-b53e-6d9e5c417f81	009064e8-f1ef-45d0-9800-91d5a48196e3	Best Android Phone	Good Camera	1,000
7	3f2d5a06-6f47-4276-b559-9dad0594e040	9b280d6b-2b81-41eb-8f8f-4754919a99de	7744b33a-6444-4620-a6a1-4f49117086c9	Good Quality	Good but not as good as Levi's	80
8	99b367a8-8b81-4437-8e5c-1789567cbd10	e70b43a5-2b24-44f4-819f-2b9916d86828	60402ffa-acd3-4334-bc16-997fe0de9f45	Windows Laptop	Powerful Laptop	300
9	4f534156-ef10-4bc1-b853-7d09091e0945	c7ebb014-f2de-4596-889a-80e1673db9c8	a42cf113-5d57-4a86-810b-6c3c4f950180	Snapdragon Version	I'm Apple CEO but I need money	250
10	5cff5cc2-7714-4889-a702-6ecd87a61db3	4aec5682-7da1-4917-940d-1bec8cd30594	97227474-175b-4aa8-8203-96f35b1cd9b3	Best on the Market	I'm Microsoft CEO but power is power	750

Figure 2 Advertises Table

StockAmount	CreatedAt	UpdatedAt
20	2025-01-01 16:15:54.582 +0300	[NULL]
10	2025-01-01 16:17:06.847 +0300	[NULL]
100	2025-01-01 16:17:48.049 +0300	[NULL]
100	2025-01-01 16:18:29.679 +0300	[NULL]
50	2025-01-01 16:19:04.653 +0300	[NULL]
10	2025-01-01 16:19:38.744 +0300	[NULL]
40	2025-01-01 16:20:19.410 +0300	[NULL]
5	2025-01-01 16:21:25.739 +0300	[NULL]
6	2025-01-01 16:22:22.119 +0300	[NULL]
3	2025-01-01 16:22:58.285 +0300	[NULL]

Figure 3 Continuation of Advertises Table

- BasketProduct CrossTable

BasketProduct | Enter a SQL expression to filter results (use Ctrl+Space)

Grid | Text

	BasketId	ProductId
1	8dd505b6-5b05-46be-b939-8f204c1fe6(	78cffb14-c078-46d6-9285-678b14204957
2	d7bb9c77-a9c1-4a20-8f3c-c6fc82e6938f	0ba945c5-409f-4724-8767-72ce3e1ae686
3	3fb9fde0-7370-40db-bfb3-d0787a0a3127	d652cf51-642c-4fdd-8fdd-e3ba8211e125
4	89191532-5bc6-421f-8a59-d18983b348c4	67a2ecf4-bcf7-4088-bf65-0e53e720a3a1
5	b9e323fa-811d-4653-8759-3f248a0182fa	01ecae5f-7229-4187-b9ee-0ad68bba1072
6	d96cf601-9219-4154-adfb-8ac44cca67da	009064e8-f1ef-45d0-9800-91d5a48196e3
7	f46271d9-d333-488d-8c0e-5e864e9cb535	7744b33a-6444-4620-a6a1-4f49117086c9
8	adeac5a3-4e30-48ef-9cd7-28bdf41c4598	60402ffa-acd3-4334-bc16-997fe0de9f45
9	c9292e4b-c493-401b-b7e1-930476f96fd6	a42cf113-5d57-4a86-810b-6c3c4f950180
10	9136f878-1d5a-4772-a439-a8e2dd2ccb1d	97227474-175b-4aa8-8203-96f35b1cd9b3

Figure 4 BasketProduct CrossTable

- Baskets Table

Baskets | Enter a SQL expression to filter results (use Ctrl+Space)

Grid | Text

	Id	CustomerId	CreatedAt	UpdatedAt
1	8dd505b6-5b05-46be-b939-8f204c1fe660	049a69bb-38d6-4212-b8e2-65574344671a	2024-12-31 23:04:06.093 +0300	[NULL]
2	d7bb9c77-a9c1-4a20-8f3c-c6fc82e6938f	d9aaa2ba-305e-414b-8d59-77365634c890	2024-12-31 23:12:00.761 +0300	[NULL]
3	3fb9fde0-7370-40db-bfb3-d0787a0a3127	2577a62e-b88b-483e-93ed-d89e74c0c210	2024-12-31 23:12:16.734 +0300	[NULL]
4	89191532-5bc6-421f-8a59-d18983b348c4	05a09696-47cf-4d7a-ac02-b364ab82b17d	2024-12-31 23:12:37.646 +0300	[NULL]
5	d96cf601-9219-4154-adfb-8ac44cca67da	b69b3f25-5107-4133-b777-1a23caa4de0d	2025-01-01 09:34:28.543 +0300	[NULL]
6	f46271d9-d333-488d-8c0e-5e864e9cb535	08307427-a94c-43f8-b812-2b69e76012e6	2025-01-01 09:34:42.748 +0300	[NULL]
7	adeac5a3-4e30-48ef-9cd7-28bdf41c4598	f502fc49-b7dc-42ae-9024-789035c13ef0	2025-01-01 09:34:57.919 +0300	[NULL]
8	c9292e4b-c493-401b-b7e1-930476f96fd6	5999c6ce-33d5-4318-b06b-45777cd5967e	2025-01-01 15:10:25.830 +0300	[NULL]
9	9136f878-1d5a-4772-a439-a8e2dd2ccb1d	b91c1202-a50d-4c97-a8a4-ac2c0388d2fa	2025-01-01 15:10:44.130 +0300	[NULL]
10	b9e323fa-811d-4653-8759-3f248a0182fa	66727aee-d51d-4828-8405-b27ebd3d7d5a	2024-12-31 23:12:55.272 +0300	[NULL]

Figure 5 Basket Table

- Categories Table

Categories | Enter a SQL expression to filter results (use Ctrl+Space)

	Id	ParentCategoryId	CategoryName	CreatedAt	UpdatedAt
1	ef93d8d3-bebc-434d-9195-6ec635941e76	[NULL]	Technology	2024-12-31 21:59:07.978 +0300	[NULL]
2	b93a7302-1798-432e-877a-3a978430ed45	ef93d8d3-bebc-434d-9195-6ec635941e76	Phone	2024-12-31 21:59:40.563 +0300	[NULL]
3	4ec46d4a-51a7-460e-8408-38baf9e29766	[NULL]	Home	2024-12-31 22:30:13.581 +0300	[NULL]
4	79bbb4b8-9ad5-4eb2-bff2-e3294ebaf2ad	4ec46d4a-51a7-460e-8408-38baf9e29766	Kitchen	2024-12-31 22:30:42.561 +0300	[NULL]
5	a61bf461-adc8-46b1-bac0-6cce377ad959	4ec46d4a-51a7-460e-8408-38baf9e29766	Decor	2024-12-31 22:36:24.932 +0300	[NULL]
6	2925849d-de0b-4cdc-90b4-707d250a511b	[NULL]	Fashion	2024-12-31 22:36:47.599 +0300	[NULL]
7	0309b623-bd0a-40b4-86dd-ddec37524a03	2925849d-de0b-4cdc-90b4-707d250a511b	Jeans	2024-12-31 22:37:20.042 +0300	[NULL]
8	4dd58b7e-73d5-46b8-b53e-05610d44b28d	2925849d-de0b-4cdc-90b4-707d250a511b	T-Shirt	2024-12-31 22:37:24.420 +0300	[NULL]
9	abd1c73a-a694-4dd0-8fb5-6857ed8d49e2	ef93d8d3-bebc-434d-9195-6ec635941e76	Laptop	2025-01-01 15:11:34.431 +0300	[NULL]
10	9dd63c9d-9782-4c82-8549-e6e00e72db59	ef93d8d3-bebc-434d-9195-6ec635941e76	Tablet	2025-01-01 15:11:37.335 +0300	[NULL]

Figure 6 Categories Table

- Customers Table

Customers | Enter a SQL expression to filter results (use Ctrl+Space)

	Id	Name	Surname	Username	Age	Email	Password	Phone	Address
1	049a69bb-38d6-4212-b8e2-65574344671a	Berke	Alpaslan	BaraCuda	22	210316016@ogr.cbu.edu.tr	210316016	+905555555555	Manisa Celal Bayar University
2	d9aaa2ba-305e-414b-8d59-77365634c890	Mert Doğan	Aygün	Merdo	22	210316076@ogr.cbu.edu.tr	210316076	+905555555555	Manisa Celal Bayar University
3	2577a62e-b88b-483e-93ed-d89e74c0c210	Gül	Yeşilgil	Rose	21	220316055@ogr.cbu.edu.tr	220316055	+905555555555	Manisa Celal Bayar University
4	05a09696-47cf-4d7a-ac02-b364ab82b17d	Joe	Biden	joe_biden	82	joe_biden@gmail.com	usa	+905555555555	USA
5	66727aee-d51d-4828-8405-b27ebd3d7d5a	Donald	Trump	donald_trump	78	donald_trump@gmail.com	usa	+905555555555	USA
6	b69b3f25-5107-4133-b777-1a23caa4de0d	Hayko	Cepkin	hayfan	46	hayko_cepkin@gmail.com	hayfan	+905555555555	Turkey
7	08307427-a94c-43f8-b812-2b69e76012e6	Sezen	Aksu	Serçe	70	sezen_aksu@gmail.com	minikserce	+905555555555	İstanbul
8	f502fc49-b7dc-42ae-9024-789035c13ef0	Nazan	Öncel	Naz	68	nazan_oncel@gmail.com	naz	+905555555555	İstanbul
9	5999c6ce-33d5-4318-b06b-45777cd5967e	Oğuz	Çağlar	Kısa Dönem	24	kisa_dönem@dag.com	bekir	+905555555555	İstanbul
10	b91c1202-a50d-4c97-a8a4-ac2c0388d2fa	Bekir	Özbey	M1_Abrams	26	zaten_bekir@dag.com	oguz	+905555555555	Ankara

Figure 7 Customers Table

CreatedAt	UpdatedAt
2024-12-30 00:23:39.391 +0300	[NULL]
2024-12-30 00:24:04.550 +0300	[NULL]
2024-12-30 00:24:30.268 +0300	[NULL]
2024-12-30 00:26:20.379 +0300	[NULL]
2024-12-30 00:26:52.229 +0300	[NULL]
2024-12-30 00:28:13.696 +0300	[NULL]
2024-12-31 21:52:29.669 +0300	[NULL]
2024-12-31 21:53:08.253 +0300	[NULL]
2025-01-01 15:08:56.023 +0300	[NULL]
2025-01-01 15:09:30.148 +0300	[NULL]

Figure 8 Continuation of Customers Table

- Orders Table

Orders | Enter a SQL expression to filter results (use Ctrl+Space)

	Id	CustomerId	SellerId	CreatedAt	UpdatedAt
1	15b83c4c-f991-4c73-9a12-6d2231846a8d	049a69bb-38d6-4212-b8e2-65574344671a	a14b1ebf-bad4-4c5e-b60b-f9538e3f14f2	2025-01-01 16:51:52.806 +0300	[NULL]
2	8dafa52b-56fe-4c9f-86a9-f6bcb2990367	d9aaa2ba-305e-414b-8d59-77365634c890	ee035cf8-e44b-4a3f-adc2-6e24e83a938d	2025-01-01 16:52:05.818 +0300	[NULL]
3	fe640120-e160-4dfb-9d85-e3e293b25574	05a09696-47cf-4d7a-ac02-b364ab82b17d	6f543965-88b3-4bd0-aeb1-643c8739a10c	2025-01-01 16:52:28.803 +0300	[NULL]
4	b0967157-f1f1-4e24-b550-49dbad25e5fa	66727aee-d51d-4828-8405-b27ebd3d7d5a	77fcb774-8900-4776-ada4-2e7fa73bd6db	2025-01-01 16:52:50.272 +0300	[NULL]
5	40fec431-18d1-4e69-be71-b434436f9d98	b69b3f25-5107-4133-b777-1a23caa4de0d	0067aa0a-6300-4c43-b53e-6d9e5c417f81	2025-01-01 16:53:17.571 +0300	[NULL]
6	599d77e8-6fb5-4888-bab3-2944a6a213fc	08307427-a94c-43f8-b812-2b69e76012e6	9b280d6b-2b81-41eb-8f8f-4754919a99de	2025-01-01 16:53:32.447 +0300	[NULL]
7	81863d0a-9ffb-4edb-96fd-da45e795d548	f502fc49-b7dc-42ae-9024-789035c13ef0	e70b43a5-2b24-44f4-819f-2b9916d86828	2025-01-01 16:53:40.930 +0300	[NULL]
8	71748c2c-6b54-4598-8eb2-b1101ecca236	5999c6ce-33d5-4318-b06b-45777cd5967e	c7ebb014-f2de-4596-889a-80e1673db9c8	2025-01-01 16:53:52.644 +0300	[NULL]
9	7e01e702-d840-42f1-b69e-bee695e572ca	b91c1202-a50d-4c97-a8a4-ac2c0388d2fa	4aec5682-7da1-4917-940d-1bec8cd30594	2025-01-01 16:54:03.093 +0300	[NULL]
10	1e29c3ec-58ee-4391-881b-6b045f730de8	2577a62e-b88b-483e-93ed-d89e74c0c210	95de2d74-3085-406c-9376-c493f18167b4	2025-01-01 16:52:14.709 +0300	[NULL]

Figure 9 Orders Table

- ProductOrder CrossTable

ProductOrder | Enter a SQL expression to filter results (use Ctrl+Space)

	ProductId	OrderId
1	78cffb14-c078-46d6-9285-678b1420495	15b83c4c-f991-4c73-9a12-6d2231846a8d
2	0ba945c5-409f-4724-8767-72ce3e1ae686	8dafa52b-56fe-4c9f-86a9-f6bcb2990367
3	d652cf51-642c-4fdd-8fdd-e3ba8211e125	1e29c3ec-58ee-4391-881b-6b045f730de8
4	67a2ecf4-bcf7-4088-bf65-0e53e720a3a1	fe640120-e160-4dfb-9d85-e3e293b25574
5	01ecae5f-7229-4187-b9ee-0ad68bba1072	b0967157-f1f1-4e24-b550-49dbad25e5fa
6	009064e8-f1ef-45d0-9800-91d5a48196e3	40fec431-18d1-4e69-be71-b434436f9d98
7	7744b33a-6444-4620-a6a1-4f49117086c9	599d77e8-6fb5-4888-bab3-2944a6a213fc
8	60402ffa-acd3-4334-bc16-997fe0de9f45	81863d0a-9ffb-4edb-96fd-da45e795d548
9	a42cf113-5d57-4a86-810b-6c3c4f950180	71748c2c-6b54-4598-8eb2-b1101ecca236
10	97227474-175b-4aa8-8203-96f35b1cd9b3	7e01e702-d840-42f1-b69e-bee695e572ca

Figure 10 ProductOrder CrossTable

- Products Table

Products | Enter a SQL expression to filter results (use Ctrl+Space)

Ctrl+click to open SQL console

	Id	CategoryId	ProductName	Description	CreatedAt	UpdatedAt
1	78cffb14-c078-46d6-9285-678b14204957	b93a7302-1798-432e-877a-3a978430ed45	iPhone 13	Apple A15 Bionic	2024-12-31 22:40:53.082 +0300	[NULL]
2	0ba945c5-409f-4724-8767-72ce3e1ae686	79bbb4b8-9ad5-4eb2-bff2-e3294ebaf2ad	Tefal Pan	Teflon	2024-12-31 22:42:18.477 +0300	[NULL]
3	d652cf51-642c-4fdd-8fdd-e3ba8211e125	0309b623-bd0a-40b4-86dd-ddec37524a03	Levi's 501	Timeless Design	2024-12-31 22:45:35.095 +0300	[NULL]
4	67a2ecf4-bcf7-4088-bf65-0e53e720a3a1	4dd58b7e-73d5-46b8-b53e-05610d44b28d	Mecanik Casual Neo Organic T-Shirt	Ertan Balaban Design	2024-12-31 22:47:15.638 +0300	[NULL]
5	01ecae5f-7229-4187-b9ee-0ad68bba1072	a61bf461-adc8-46b1-bac0-6cce377ad959	Picture Table	Vincent Van Gogh	2024-12-31 22:56:47.290 +0300	[NULL]
6	009064e8-f1ef-45d0-9800-91d5a48196e3	b93a7302-1798-432e-877a-3a978430ed45	Galaxy S24 Ultra	Samsung	2024-12-31 23:02:13.237 +0300	[NULL]
7	7744b33a-6444-4620-a6a1-4f49117086c9	0309b623-bd0a-40b4-86dd-ddec37524a03	Mavi Black Pro	Slim Straight	2024-12-31 23:02:52.362 +0300	[NULL]
8	60402ffa-acd3-4334-bc16-997fe0de9f45	abd1c73a-a694-4dd0-8fb5-6857ed8d49e2	Lenovo IdeaPad 5 Pro 16ACH6	AMD Ryzen 7 5800H	2025-01-01 15:13:04.720 +0300	[NULL]
9	a42cf113-5d57-4a86-810b-6c3c4f950180	9dd63c9d-9782-4c82-8549-e6e00e72db59	Galaxy Tab S10	Samsung	2025-01-01 15:14:03.798 +0300	[NULL]
10	97227474-175b-4aa8-8203-96f35b1cd9b3	abd1c73a-a694-4dd0-8fb5-6857ed8d49e2	Macbook Pro	Apple M4 Chip	2025-01-01 15:22:24.993 +0300	[NULL]

Figure 11 Products Table

- Reviews Table

Reviews | Enter a SQL expression to filter results (use Ctrl+Space)

	Id	Rating	Comment	Title	ProductId
1	59620688-458a-4dfe-a47b-751912255cce	VeryGood	Good Camera	Best Phone Ever	78cffb14-c078-46d6-9285-678b14204957
2	a4d69760-8296-48ba-b86e-1ece873933de	Normal	Late Shipping	Cargo Issue	7744b33a-6444-4620-a6a1-4f49117086c9
3	fcce7b38-a71d-445e-b9b5-3b4e66068b06	VeryBad	Snapdragon version is better than Exynos	Bad Chip	009064e8-f1ef-45d0-9800-91d5a48196e3
4	15282355-6ed0-4b58-9935-3e88c08fcaab	VeryGood	Non-Sticks	Best Quality	0ba945c5-409f-4724-8767-72ce3e1ae686
5	9f7ca535-94fc-448b-84e5-a7d652dd49bd	Good	Organic Cotton	Soft T-Shirt	67a2ecf4-bcf7-4088-bf65-0e53e720a3a1
6	f5c65343-9e62-4967-a75e-c8c0434727f3	VeryGood	Iconic and Timeless Model of Levi's	Timeless Cut	d652cf51-642c-4fdd-8fdd-e3ba8211e125
7	82b5f714-c94b-4cb1-a1fe-fd5773c03d09	Bad	It isn't Original	Fake Product	01ecae5f-7229-4187-b9ee-0ad68bba1072
8	d0632521-dbbd-4d8f-b7d6-f51091e9c7b1	Normal	Not as Fast as Apple Chips	Slow	a42cf113-5d57-4a86-810b-6c3c4f950180
9	986025a1-7748-468f-aab9-c46d9838d90b	VeryGood	AMD chips are better than Intel	Good Performance	60402ffa-acd3-4334-bc16-997fe0de9f45
10	1217ebba-62ce-4f99-9a18-f4e0d2414b69	VeryGood	Fastest laptop on the market	Incredible	97227474-175b-4aa8-8203-96f35b1cd9b3

Figure 12 Reviews Table

CustomerId	CreatedAt	UpdatedAt
049a69bb-38d6-4212-b8e2-65574344671a	2025-01-01 09:40:36.229 +0300	[NULL]
d9aaa2ba-305e-414b-8d59-77365634c890	2025-01-01 15:37:51.498 +0300	[NULL]
2577a62e-b88b-483e-93ed-d89e74c0c210	2025-01-01 15:46:19.391 +0300	[NULL]
05a09696-47cf-4d7a-ac02-b364ab82b17d	2025-01-01 15:48:07.205 +0300	[NULL]
66727aee-d51d-4828-8405-b27ebd3d7d5a	2025-01-01 15:49:29.262 +0300	[NULL]
b69b3f25-5107-4133-b777-1a23caa4de0d	2025-01-01 15:50:22.518 +0300	[NULL]
08307427-a94c-43f8-b812-2b69e76012e6	2025-01-01 15:51:12.051 +0300	[NULL]
f502fc49-b7dc-42ae-9024-789035c13ef0	2025-01-01 16:09:42.757 +0300	[NULL]
5999c6ce-33d5-4318-b06b-45777cd5967e	2025-01-01 16:10:52.871 +0300	[NULL]
b91c1202-a50d-4c97-a8a4-ac2c0388d2fa	2025-01-01 16:11:41.673 +0300	[NULL]

Figure 13 Continuation of Reviews Table

- Sellers Table

Sellers | Enter a SQL expression to filter results (use Ctrl+Space)

	Id	Name	Surname	Username	Age	Email	Password	Phone	Address
1	a14b1ebf-bad4-4c5e-b60b-f9538e3f14f2	Ali	Koç	Fenerbahçe	57	ali_koç@gmail.com	fenerbahce	+905555555555	İstanbul
2	ee035cf8-e44b-4a3f-adc2-6e24e83a938d	Sakıp	Sabancı	SabancıHolding	71	sakıp_sabanci@gmail.com	sabanci	+905555555555	İstanbul
3	95de2d74-3085-406c-9376-c493f18167b4	Ahmet	Arslan	philosopher	81	ahmet_arslan@gmail.com	philosopher	+905555555555	İstanbul
4	6f543965-88b3-4bd0-aeb1-643c8739a10c	Celal	Şengör	geologist	69	celal_sengor@gmail.com	ilber	+905555555555	İstanbul
5	77fcb774-8900-4776-ada4-2e7fa73bd6db	İlber	Ortaylı	CahilSavarHistorian	77	hepiniz_cahilsiniz@gmail.com	celal	+905555555555	İstanbul
6	0067aa0a-6300-4c43-b53e-6d9e5c417f81	Selim	Erdoğan	War_Geography_Specialist	52	selim_erdogan@gmail.com	war	+905555555555	İstanbul
7	9b280d6b-2b81-41eb-8f8f-4754919a99de	Linus	Torvalds	linux	55	linux_penguin@gmail.com	kernel	+905555555555	Finland
8	e70b43a5-2b24-44f4-819f-2b9916d86828	Lee	Byung-Chul	Samsung	77	samsung_galaxy@gmail.com	korean	+905555555555	South Korea
9	c7ebb014-f2de-4596-889a-80e1673db9c8	Steve	Jobs	Apple	56	apple_iphone@gmail.com	ios	+905555555555	USA
10	4aec5682-7da1-4917-940d-1bec8cd30594	Bill	Gates	Microsoft	69	microsoft_windows@outlook.com	windows	+905555555555	USA

Figure 14 Sellers Table

CreatedAt	UpdatedAt
2025-01-01 13:59:00.267 +0300	[NULL]
2025-01-01 14:00:07.839 +0300	[NULL]
2025-01-01 14:01:21.085 +0300	[NULL]
2025-01-01 14:02:51.146 +0300	[NULL]
2025-01-01 14:03:37.144 +0300	[NULL]
2025-01-01 14:04:54.444 +0300	[NULL]
2025-01-01 14:06:34.806 +0300	[NULL]
2025-01-01 14:10:42.233 +0300	[NULL]
2025-01-01 15:06:09.321 +0300	[NULL]
2025-01-01 15:07:27.020 +0300	[NULL]

Figure 15 Continuation of Sellers Table

Queries and Results

1. Number of Products and Average Price in Each Category

This query provides insights into the distribution of products across categories and their pricing trends. It is vital for analyzing which categories are performing well in terms of product variety and pricing.

```
select c."CategoryName", count(p."Id") as product_count, avg(a."Price") as avg_price from "Categories" c
left join "Products" p on c."Id"  = p."CategoryId"
left join "Advertises" a on p."Id" = a."ProductId"
group by c."Id", c."CategoryName";
```

Figure 16 Query_1 - Number of Products and Average Price in Each Category

select c."CategoryName", count(p."Id") as product_c Enter a SQL e

Grid | Text

	CategoryName	product_count	avg_price
1	Home	0	[NULL]
2	Laptop	2	525
3	Kitchen	1	20
4	Jeans	2	102.5
5	T-Shirt	1	25
6	Tablet	1	250
7	Technology	0	[NULL]
8	Decor	1	12
9	Fashion	0	[NULL]
10	Phone	2	875

Figure 17 Result_1 - Number of Products and Average Price in Each Category

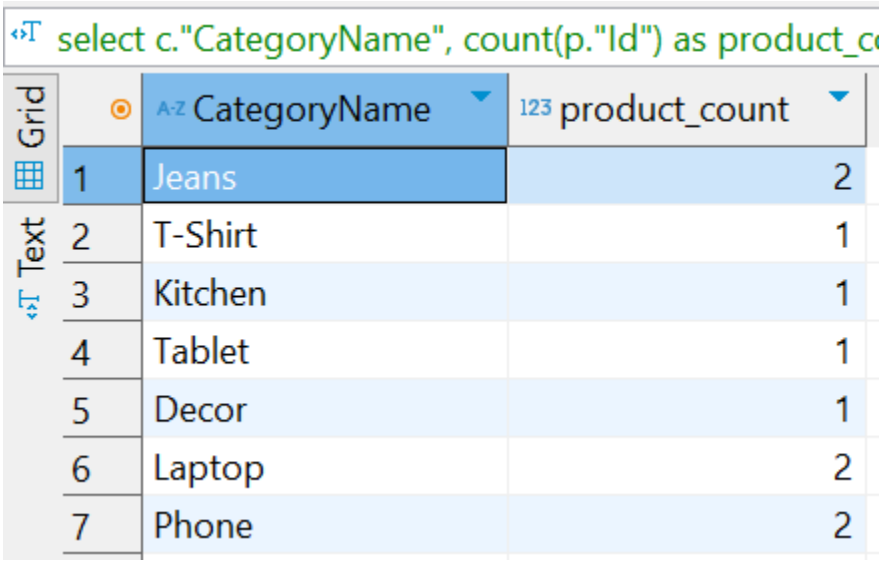
Result: Displays each category's name, the count of products, and their average price. This helps identify high-demand or underrepresented categories.

2. Products in Categories Without Subcategories

This query helps identify standalone categories and their product counts.

```
select c."CategoryName", count(p."Id") as product_count from "Categories" c
left join "Categories" c2 on c."Id" = c2."ParentCategoryId"
left join "Products" p on c."Id" = p."CategoryId"
where c2."Id" is null
group by c."Id", c."CategoryName";
```

Figure 18 Query_2 - Products in Categories Without Subcategories



	CategoryName	product_count
1	Jeans	2
2	T-Shirt	1
3	Kitchen	1
4	Tablet	1
5	Decor	1
6	Laptop	2
7	Phone	2

Figure 19 Result_2 - Products in Categories Without Subcategories

Result: Highlights the categories with no subcategories and the number of products in them. This aids in assessing organizational structure.

3. Customers with Both Baskets and Purchases

This query identifies customers who have added products to their basket and completed purchases, indicating active engagement.

```
select distinct c."Name", c."Surname", c."Email" from "Customers" c
join "Baskets" b on c."Id" = b."CustomerId"
join "BasketProduct" bp on b."Id" = bp."BasketId"
where c."Id" in (select o."CustomerId" from "Orders" o);
```

Figure 20 Query_3 - Customers with Both Baskets and Purchases

select distinct c."Name", c."Surname", c."Email" from Enter a SQ

Grid | Text

	Name	Surname	Email
1	Hayko	Cepkin	hayko_cepkin@gmail.com
2	Bekir	Özbey	zaten_bekir@dag.com
3	Nazan	Öncel	nazan_oncel@gmail.com
4	Sezen	Aksu	sezen_aksu@gmail.com
5	Mert Doğan	Aygün	210316076@ogr.cbu.edu.tr
6	Berke	Alpaslan	210316016@ogr.cbu.edu.tr
7	Gül	Yeşilgil	220316055@ogr.cbu.edu.tr
8	Oğuz	Çağlar	kisa_dönem@dag.com
9	Donald	Trump	donald_trump@gmail.com
10	Joe	Biden	joe_biden@gmail.com

Figure 21 Result_3 - Customers with Both Baskets and Purchases

Result: Lists engaged customers, helping to focus on retention strategies.

4. Average Basket Value per Customer

This query calculates the average monetary value of each customer's basket.

```
select c."Name", c."Surname", avg(a."Price") as avg_basket_price from "Customers" c
join "Baskets" b on c."Id" = b."CustomerId"
join "BasketProduct" bp on b."Id" = bp."BasketId"
join "Advertises" a on bp."ProductId" = a."ProductId"
group by c."Id", c."Name", c."Surname"
order by avg_basket_price desc;
```

Figure 22 Query_4 - Average Basket Value per Customer

select c."Name", c."Surname", avg(a."Price") as avg_b

Grid | Text

	Name	Surname	avg_basket_price
1	Hayko	Cepkin	1,000
2	Bekir	Özbey	750
3	Berke	Alpaslan	750
4	Nazan	Öncel	300
5	Oğuz	Çağlar	250
6	Gül	Yeşilgil	125
7	Sezen	Aksu	80
8	Joe	Biden	25
9	Mert Doğan	Aygün	20
10	Donald	Trump	12

Figure 23 Result_4 - Average Basket Value per Customer

Result: Shows high-value customers based on their basket value, aiding in premium service offerings.

5. Price Comparison of Products within the Same Category

This query highlights price variations of products within the same category.

```
SELECT
    p1."ProductName",
    cat."CategoryName",
    a1."Price",
    AVG(a2."Price") as avg_category_price,
    a1."Price" - AVG(a2."Price") as price_difference
FROM "Products" p1
JOIN "Categories" cat ON p1."CategoryId" = cat."Id"
JOIN "Advertises" a1 ON p1."Id" = a1."ProductId"
join "Products" p2 ON p1."CategoryId" = p2."CategoryId"
JOIN "Advertises" a2 ON p2."Id" = a2."ProductId"
GROUP BY p1."ProductName", cat."CategoryName", a1."Price"
ORDER BY price_difference DESC;
```

Figure 24 Query_5 - Price Comparison of Products within the Same Category

SELECT p1."ProductName", cat."CategoryName", a1. Enter a SQL expression to filter results (use Ctrl+Space)

Grid | Text

	ProductName	CategoryName	Price	avg_category_price	price_difference
1	Macbook Pro	Laptop	750	525	225
2	Galaxy S24 Ultra	Phone	1,000	875	125
3	Levi's 501	Jeans	125	102.5	22.5
4	Mecanik Casual Neo Organic T-Shirt	T-Shirt	25	25	0
5	Galaxy Tab S10	Tablet	250	250	0
6	Tefal Pan	Kitchen	20	20	0
7	Picture Table	Decor	12	12	0
8	Mavi Black Pro	Jeans	80	102.5	-22.5
9	iPhone 13	Phone	750	875	-125
10	Lenovo IdeaPad 5 Pro 16ACH6	Laptop	300	525	-225

Figure 25 Result_5 - Price Comparison of Products within the Same Category

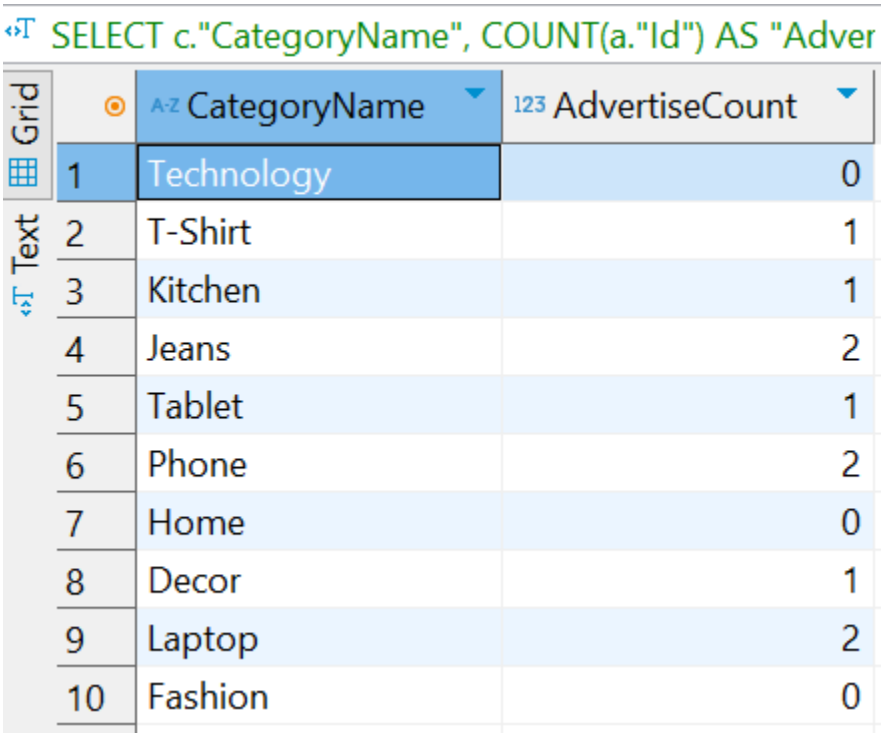
Result: Identifies overpriced or underpriced products, enabling pricing optimization.

6. Total Advertisements per Category

This query summarizes the advertising efforts in each category.

```
SELECT c."CategoryName", COUNT(a."Id") AS "AdvertiseCount"
FROM "Categories" c
LEFT JOIN "Products" p ON c."Id" = p."CategoryId"
LEFT JOIN "Advertises" a ON p."Id" = a."ProductId"
GROUP BY c."CategoryName";
```

Figure 26 Query_6 - Total Advertisements per Category



	CategoryName	AdvertiseCount
1	Technology	0
2	T-Shirt	1
3	Kitchen	1
4	Jeans	2
5	Tablet	1
6	Phone	2
7	Home	0
8	Decor	1
9	Laptop	2
10	Fashion	0

Figure 27 Result_6 - Total Advertisements per Category

Result: Evaluates advertising reach and effectiveness by category.

7. Top 5 Most Expensive Products

This query lists the most expensive products and their categories.

```
SELECT p."ProductName", c."CategoryName", a."Price"
FROM "Advertises" a
JOIN "Products" p ON a."ProductId" = p."Id"
JOIN "Categories" c ON p."CategoryId" = c."Id"
ORDER BY a."Price" DESC
LIMIT 5;
```

Figure 28 Query_7 - Top 5 Most Expensive Products

SELECT p."ProductName", c."CategoryName", a."Pric Enter a SQL expres

Grid Text

	ProductName	CategoryName	Price
1	Galaxy S24 Ultra	Phone	1,000
2	Macbook Pro	Laptop	750
3	iPhone 13	Phone	750
4	Lenovo IdeaPad 5 Pro 16ACH6	Laptop	300
5	Galaxy Tab S10	Tablet	250

Figure 29 Result_7 - Top 5 Most Expensive Products

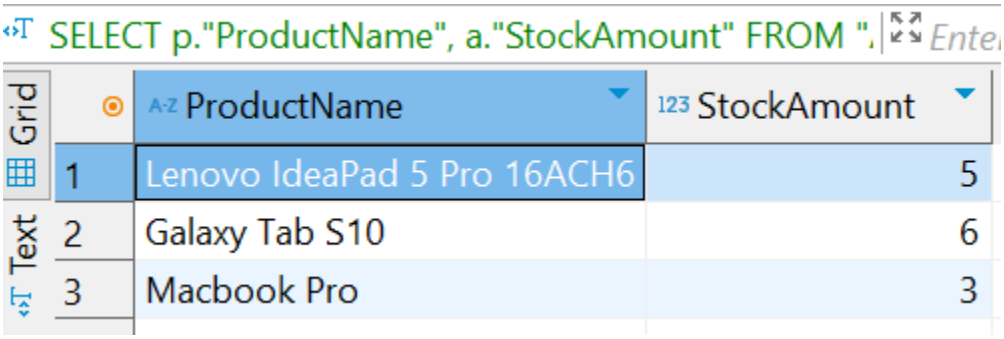
Result: Highlights luxury items for potential premium marketing strategies.

8. Products with Low Stock Levels

This query identifies products with stock amounts below a critical threshold.

```
SELECT p."ProductName", a."StockAmount"
FROM "Advertises" a
JOIN "Products" p ON a."ProductId" = p."Id"
WHERE a."StockAmount" < 10
ORDER BY a."UpdatedAt" DESC;
```

Figure 30 Query_8 - Products with Low Stock Levels



	ProductName	StockAmount
1	Lenovo IdeaPad 5 Pro 16ACH6	5
2	Galaxy Tab S10	6
3	Macbook Pro	3

Figure 31 Result_8 - Products with Low Stock Levels

Result: Facilitates inventory management by focusing on restocking low-stock items.

9. Customer Segmentation by Spending

This query categorizes customers based on their order frequency and total spending.

```
SELECT
    c."Name",
    c."Surname",
    COUNT(DISTINCT o."Id") as order_count,
    SUM(a."Price") as total_spent,
    CASE
        WHEN COUNT(DISTINCT o."Id") >= 5 AND SUM(a."Price") >= 1000 THEN 'VIP'
        WHEN COUNT(DISTINCT o."Id") >= 3 OR SUM(a."Price") >= 500 THEN 'Regular'
        ELSE 'New'
    END as customer_segment
FROM public."Customers" c
LEFT JOIN public."Orders" o ON c."Id" = o."CustomerId"
LEFT JOIN public."ProductOrder" po ON o."Id" = po."OrderId"
LEFT JOIN public."Advertises" a ON po."ProductId" = a."ProductId"
GROUP BY c."Id", c."Name", c."Surname"
ORDER BY total_spent DESC;
```

Figure 32 Query_9 - Customer Segmentation by Spending

SELECT c."Name", c."Surname", COUNT(DISTINCT o." Enter a SQL expression to filter results (use Ctrl+

Grid Text

	Name	Surname	order_count	total_spent	customer_segment
1	Hayko	Cepkin	1	1,000	Regular
2	Berke	Alpaslan	1	750	Regular
3	Bekir	Özbey	1	750	Regular
4	Nazan	Öncel	1	300	New
5	Oğuz	Çağlar	1	250	New
6	Gül	Yeşilgil	1	125	New
7	Sezen	Aksu	1	80	New
8	Joe	Biden	1	25	New
9	Mert Doğan	Aygün	1	20	New
10	Donald	Trump	1	12	New

Figure 33 Result_9 - Customer Segmentation by Spending

Result: Enables tailored communication and offers based on customer segmentation.

10. Category-based Price Ranges and Product Distribution

This query breaks down products into price ranges (Low, Mid, High) within each category, providing a detailed view of product positioning.

```sql
SELECT
    cat."CategoryName",
    CASE
        WHEN a."Price" < 100 THEN 'Low Price'
        WHEN a."Price" < 500 THEN 'Mid Price'
        ELSE 'High Price'
    END as price_range,
    COUNT(*) as product_count
FROM public."Categories" cat
JOIN public."Products" p ON cat."Id" = p."CategoryId"
JOIN public."Advertises" a ON p."Id" = a."ProductId"
GROUP BY cat."CategoryName", price_range
ORDER BY cat."CategoryName", price_range;
```

Figure 34 Query_10 - Category-based Price Ranges and Product Distribution

SELECT cat."CategoryNa | Enter a SQL expression to filter results (use C

Grid | Text

	CategoryName	price_range	product_count
1	Decor	Low Price	1
2	Jeans	Low Price	1
3	Jeans	Mid Price	1
4	Kitchen	Low Price	1
5	Laptop	High Price	1
6	Laptop	Mid Price	1
7	Phone	High Price	2
8	T-Shirt	Low Price	1
9	Tablet	Mid Price	1

Figure 35 Result_10 - Category-based Price Ranges and Product Distribution

Result: Provides insights into the distribution of products by price range within each category, assisting in inventory and pricing strategies.

API DEVELOPMENT

- Purpose and Features of the API

The API was developed to enable seamless interaction between the application and the database, ensuring efficient data management for the e-commerce platform. Key features of the API include:

- User authentication and authorization.
- CRUD operations for managing customers, sellers, products, and orders.
- Complex queries for advanced filtering and reporting.
- Performance optimization using Entity Framework Core.

The API also follows a scalable, modular structure to support future enhancements.

- Technologies and Frameworks Used
- **Framework**: ASP.NET Core was chosen for its cross-platform capabilities and robust tooling support.

- **Architecture**: The Onion Architecture was implemented to achieve a clear separation of concerns and maintainability.

- **Design Patterns**:
    - **Generic Repository Pattern**: To abstract database operations and ensure reusability.
    - **Marker Pattern**: To differentiate object behavior at runtime.
    - **UnitOfWork Pattern**: To handle transactions efficiently.
    - **Mediator Pattern**: To decouple requests and handlers, improving readability and testability.

- **ORM and Query Tools**:
    - **Entity Framework Core**: Used for general CRUD operations, leveraging its ease of use for table creation and relationships.
    - The database was created using **the Code-First** approach in Entity Framework Core, enabling seamless mapping between the object-oriented domain models and the relational database structure.

- Onion Architecture Layers

- **Domain Layer**
  Contains “Entity” classes for create tables on database.

  Entity Design Approach
  To avoid repetitive code and enhance maintainability, an **abstract class-based inheritance structure** was implemented for the entity design. This approach ensures that common properties are shared across multiple entities while enabling flexibility for specific use cases.

  Benefits of This Approach
- **Code Reusability**: Common attributes are written once and reused across all entities.
- **Scalability**: Adding new shared attributes or entity types is straightforward.
- **Consistency**: Ensures uniformity in naming and behavior of shared attributes.
- **Separation of Concerns**: Each class focuses only on its specific responsibilities while inheriting the relevant attributes from the parent classes.
  This structure reflects object-oriented programming principles and is aligned with best practices for database modeling and API development.

    BaseEntity Abstract Class
    The BaseEntity abstract class defines attributes that are common across all entities in the system. These include:

    - **Id:** A unique identifier for each entity.
    - **CreatedAt:** The timestamp indicating when the entity was created.
    - **UpdatedAt:** The timestamp indicating when the entity was last updated. It’s nullable.

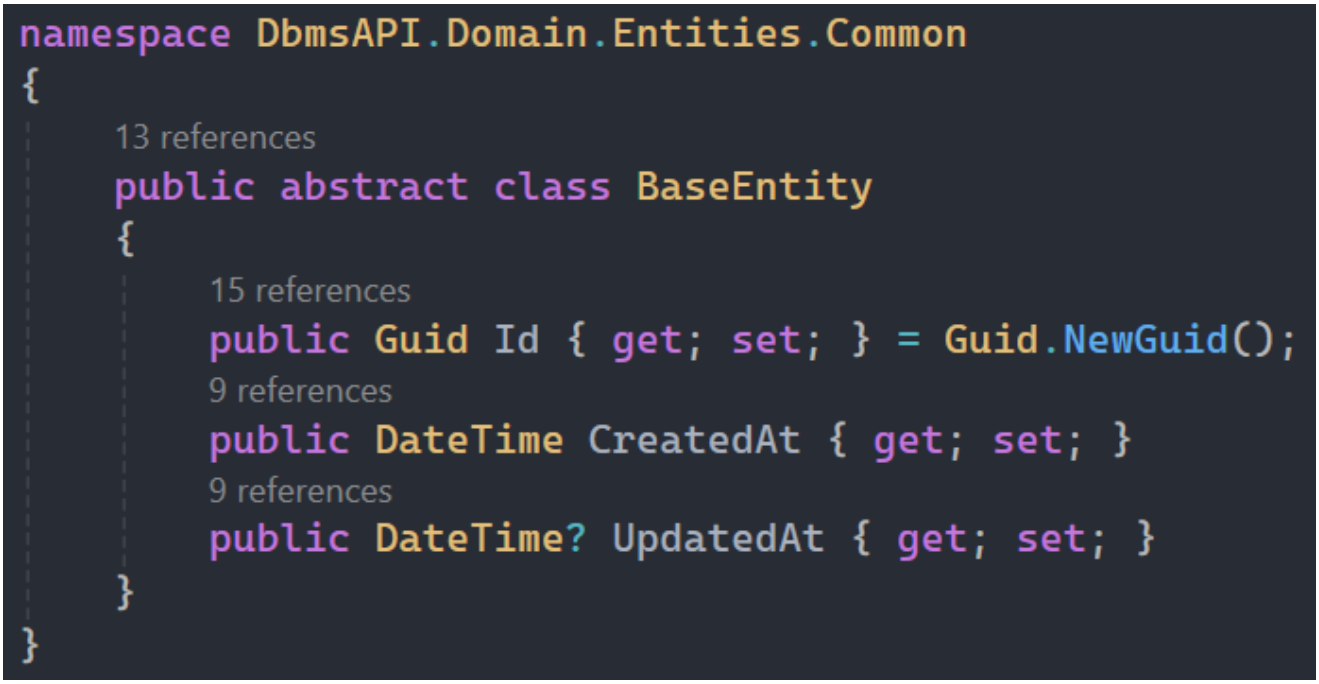


Figure 36 BaseEntity

User Abstract Class

To handle the shared properties between **Customer** and **Seller** entities, a **User** abstract class was created, inheriting from **BaseEntity**. This class includes attributes such as:

- **Name:** The first name of the user.
- **Surname:** The last name of the user.
- **Username, Age, Email, Password, Phone, Address:** Other shared user-specific properties.

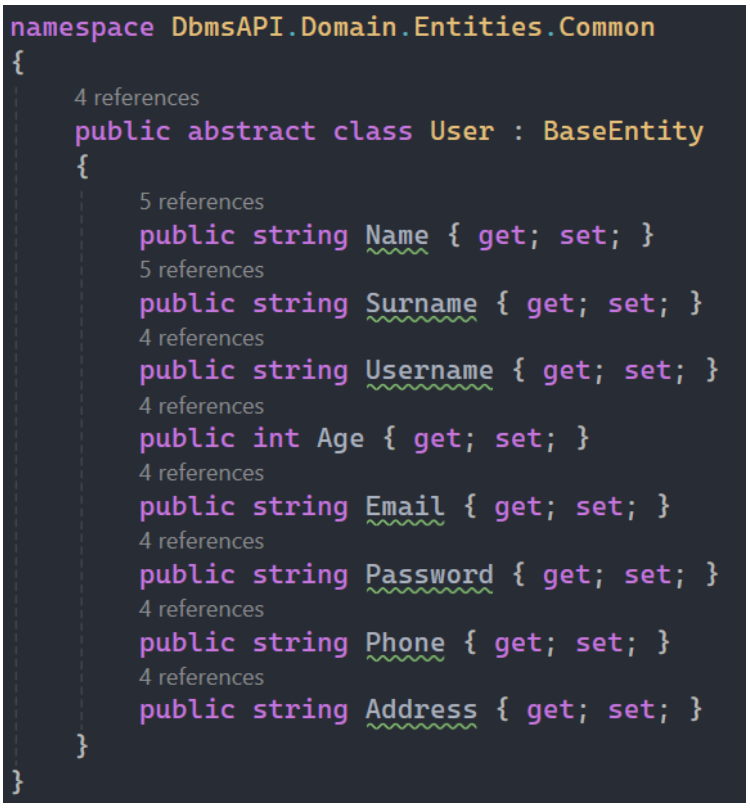


Figure 37 User Abstract Class Entity

Other Entities:

```
namespace DbmsAPI.Domain.Entities
{
    10 references
    public class Advertise : BaseEntity
    {
        3 references
        public Guid SellerId { get; set; }
        1 reference
        public Seller Seller { get; set; }
        3 references
        public Guid ProductId { get; set; }
        1 reference
        public Product Product { get; set; }
        2 references
        public string Title { get; set; }
        2 references
        public string? Description { get; set; }
        2 references
        public decimal Price { get; set; }
        2 references
        public int StockAmount { get; set; }
    }
}
```

Figure 38 Advertise Entity Class

```
namespace DbmsAPI.Domain.Entities
{
    11 references
    public class Basket : BaseEntity
    {
        1 reference
        public ICollection<BasketProduct> Products { get; set; }
        3 references
        public Guid CustomerId { get; set; }
        1 reference
        public Customer Customer { get; set; }
    }
}
```

Figure 39 Basket Entity Class

```
namespace DbmsAPI.Domain.Entities
{
    9 references
    public class Category : BaseEntity
    {
        2 references
        public Guid? ParentCategoryId { get; set; }
        0 references
        public Category ParentCategory { get; set; }
        1 reference
        public ICollection<Product> Products { get; set; }
        3 references
        public string CategoryName { get; set; }
    }
}
```

Figure 40 Category Entity Class

```
namespace DbmsAPI.Domain.Entities
{
    12 references
    public class Customer : User
    {
        1 reference
        public Basket? Basket { get; set; }
        1 reference
        public ICollection<Review> Reviews { get; set; }
        1 reference
        public ICollection<Order> Orders { get; set; }
    }
}
```

Figure 41 Customer Entity Class

```
namespace DbmsAPI.Domain.Entities
{
    16 references
    public class Order : BaseEntity
    {
        1 reference
        public ICollection<ProductOrder> Products { get; set; }
        3 references
        public Guid CustomerId { get; set; }
        1 reference
        public Customer Customer { get; set; }
        3 references
        public Guid SellerId { get; set; }
        1 reference
        public Seller Seller { get; set; }
    }
}
```

Figure 42 Order Entity Class

```
namespace DbmsAPI.Domain.Entities
{
    14 references
    public class Product : BaseEntity
    {
        1 reference
        public ICollection<Advertise>? Advertises{ get; set; }
        1 reference
        public ICollection<Review>? Reviews { get; set; }
        1 reference
        public ICollection<BasketProduct>? Baskets { get; set; }
        1 reference
        public ICollection<ProductOrder>? Orders { get; set; }
        1 reference
        public Category Category { get; set; }
        3 references
        public Guid CategoryId { get; set; }
        2 references
        public string ProductName { get; set; }
        2 references
        public string Description { get; set; }
    }
}
```

Figure 43 Product Entity Class

```
namespace DbmsAPI.Domain.Entities
{

    public class Review : BaseEntity
    {
        3 references
        public Rating Rating { get; set; }
        2 references
        public string Comment { get; set; }
        2 references
        public string Title { get; set; }
        3 references
        public Guid ProductId { get; set; }
        1 reference
        public Product Product { get; set; }
        3 references
        public Guid CustomerId { get; set; }
        1 reference
        public Customer Customer { get; set; }
    }
}
```

Figure 44 Review Entity Class

```
namespace DbmsAPI.Domain.Entities
{
    8 references
    public class Seller : User
    {
        1 reference
        public ICollection<Advertise>? Advertises { get; set; }
        1 reference
        public ICollection<Order>? Orders { get; set; }
    }
}
```

Figure 45 Seller Entity Class

Cross-Table Entities

In the system, two cross-table entities, **BasketProduct** and **ProductOrder**, are designed to manage many-to-many relationships between other entities. These classes do not inherit from the **BaseEntity** class, as they serve solely as junction tables without requiring attributes like Id, **CreatedAt**, or **UpdatedAt.**

Why Cross-Table Entities Do Not Inherit from BaseEntity

Unlike other entities in the system, BasketProduct and ProductOrder are purely functional, existing only to facilitate many-to-many relationships. They do not need:

- **Unique Identifiers (Id)**: Their uniqueness is defined by the composite keys (BasketId + ProductId or ProductId + OrderId).
- **Timestamps (CreatedAt and UpdatedAt)**: These attributes are not relevant to junction tables, as they do not have standalone behaviors or states to track.

Benefits of This Design

- **Simplification**: By excluding unnecessary properties, these entities remain lightweight and focused on their purpose.
- **Integrity**: The use of composite keys ensures the integrity of many-to-many relationships without the need for additional identifiers.
- **Alignment with Database Schema**: This approach mirrors the relational database design, where junction tables typically include only foreign keys.

By following this design, the system effectively manages complex relationships while maintaining a clear and concise entity structure.

BasketProduct Entity

The **BasketProduct** entity represents the many-to-many relationship between **Baskets** and **Products**. Each record in this table connects a specific Basket to a Product included in that basket.

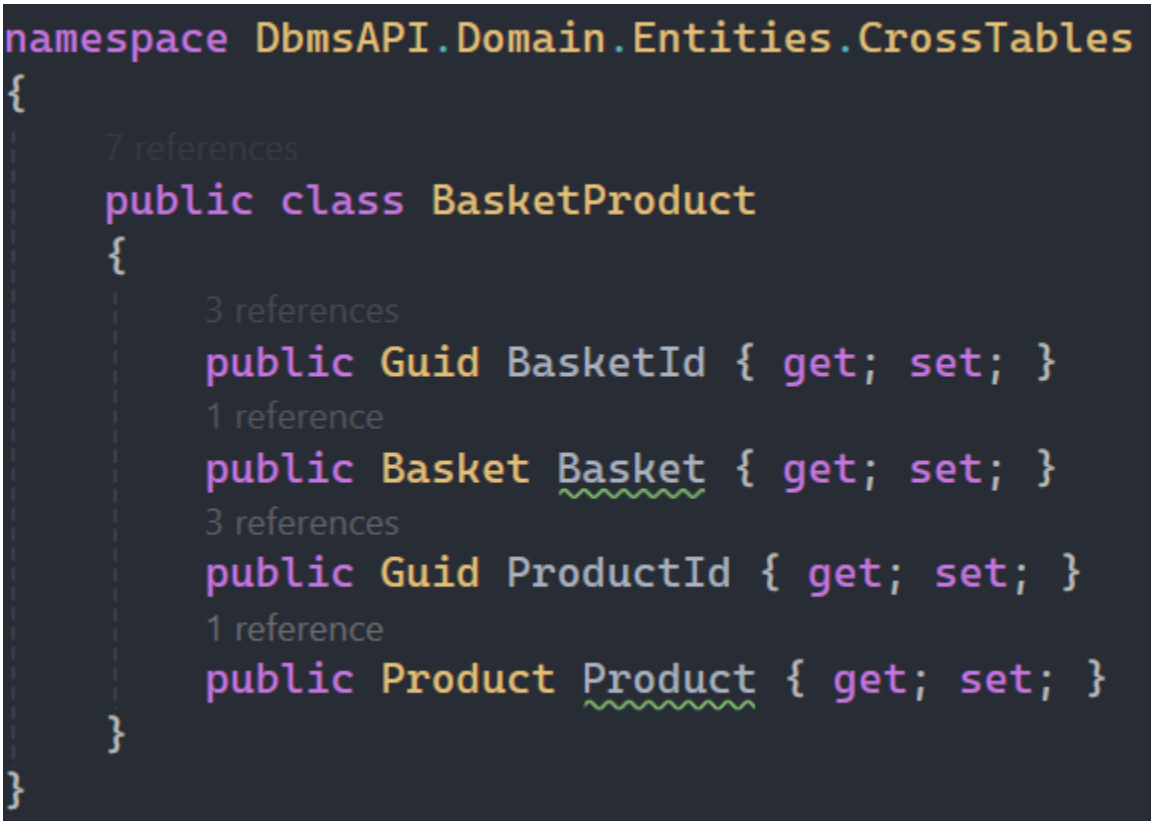


Figure 46 BasketProduct CrossTable Entity Class

ProductOrder Entity

The **ProductOrder** entity represents the many-to-many relationship between **Orders** and **Products**. Each record in this table connects a specific Order to the Products it contains.

```
namespace DbmsAPI.Domain.Entities.CrossTables
{
    7 references
    public class ProductOrder
    {
        3 references
        public Guid ProductId { get; set; }
        1 reference
        public Product Product { get; set; }
        3 references
        public Guid OrderId { get; set; }
        1 reference
        public Order Order { get; set; }
    }
}
```

Figure 47 ProductOrder CrossTable Entity Class

Rating Enum

The Rating enum defines a range of possible values for product ratings. These values represent customer satisfaction levels, typically on a 1–5 scale.

Usage in the Database

In the database, the Rating enum is stored as an integer (int). This mapping ensures compatibility with PostgreSQL while maintaining the semantic meaning of the values through the enum in the application.

```
namespace DbmsAPI.Domain.Enums
{
    2 references
    public enum Rating
    {
        VeryBad = 1,
        Bad = 2,
        Normal = 3,
        Good = 4,
        VeryGood = 5
    }
}
```

Figure 48 Rating Enum Class

- **Application Layer**
  The Application Layer is responsible for implementing business logic, ensuring a clear separation of concerns while adhering to the Onion Architecture. This layer includes interfaces and patterns that promote code reusability, maintainability, and scalability. Key design patterns employed in this layer are **Generic Repository Pattern**, **Marker Pattern**, and **Unit of Work Pattern**.

  1. Repository Interfaces
  To implement the **Generic Repository Pattern**, three primary interfaces were created: IRepository, IReadRepository, and IWriteRepository. These interfaces ensure a clean and reusable structure for interacting with the database.

  IRepository Interface
  The IRepository interface employs the **Marker Pattern** and restricts its generic type to BaseEntity. This ensures that all repositories work exclusively with entities that inherit from BaseEntity. It includes shared functionality between IReadRepository and IWriteRepository.

```
namespace DbmsAPI.Application.Repositories
{
    2 references
    public interface IRepository<T> where T : BaseEntity
    {
        14 references
        DbSet<T> Table { get; }
    }
}
```

Figure 49 IRepository Interface

IReadRepository and IWriteRepository Interfaces

These interfaces inherit from IRepository and define methods specific to reading and writing operations.

- **IReadRepository** includes methods for querying the database

```
namespace DbmsAPI.Application.Repositories
{
    9 references
    public interface IReadRepository<T> : IRepository<T> where T : BaseEntity
    {
        9 references
        IQueryable<T> GetAll(bool tracking = true);
        1 reference
        IQueryable<T> GetWhere(Expression<Func<T, bool>> method, bool tracking = true);
        1 reference
        Task<T> GetSingleAsync(Expression<Func<T, bool>> method, bool tracking = true);
        1 reference
        Task<T> GetByIdAsync(string id, bool tracking = true);
    }
}
```

Figure 50 IReadRepository Interface

- **IWriteRepository** includes methods for modifying the database

```
namespace DbmsAPI.Application.Repositories
{
    9 references
    public interface IWriteRepository<T> : IRepository<T> where T : BaseEntity
    {
        10 references
        Task<bool> AddAsync(T entity);
        1 reference
        Task<bool> AddRangeAsync(List<T> datas);
        1 reference
        Task<bool> RemoveAsync(string id);
        1 reference
        bool RemoveRange(List<T> datas);
        1 reference
        Task<bool> UpdateAsync(string id);
        1 reference
        bool UpdateRange(List<T> datas);
        11 references
        Task<int> SaveAsync();
    }
}
```

Figure 51IWriteRepository Interface

Entity-Specific Repository Interfaces

For each entity (excluding cross-tables), separate interfaces such as **ICategoryReadRepository** and **ICategoryWriteRepository** are created by inheriting from **IReadRepository** and **IWriteRepository.**

Example for the **Category** entity:

```
namespace DbmsAPI.Application.Repositories.Category
{
    4 references
    public interface ICategoryReadRepository : IReadRepository<Domain.Entities.Category>
    {
    }
}
```

Figure 52 ICategoryReadRepository Interface

```
namespace DbmsAPI.Application.Repositories.Category
{
    4 references
    public interface ICategoryWriteRepository : IWriteRepository<Domain.Entities.Category>
    {
    }
}
```

Figure 53 ICategoryReadRepository Interface

2. Unit of Work Interface

The **Unit of Work Pattern** is implemented through the **IUnitOfWork** interface, which ensures atomicity of operations by managing transactions.

```
namespace DbmsAPI.Application.Repositories
{
    22 references
    public interface IUnitOfWork : IAsyncDisposable
    {
        11 references
        Task BeginTransactionAsync();
        11 references
        Task CommitAsync();
        11 references
        Task RollbackAsync();
    }
}
```

Figure 54 IUnitOfWork Interface

Benefits of the Repository and Unit of Work Pattern

1. **Code Reusability**: Shared logic is centralized in base interfaces, reducing redundancy across repositories.
2. **Type Safety**: Marker Pattern ensures repositories operate only on valid entities.
3. **Transaction Management**: Unit of Work ensures that multiple database operations are performed atomically.
4. **Separation of Concerns**: Repository pattern separates data access logic from business logic, making the application easier to maintain and test.

By structuring the Application Layer with these patterns, the system achieves a scalable and robust architecture that adheres to software design best practices.

Mediator Pattern in the Application Layer

To implement the **Mediator Pattern**, the Application Layer includes a **Features** folder that organizes business logic into two main categories:

- **Queries**: Handles read operations (e.g., fetching data).
- **Commands**: Handles write operations (e.g., creating, updating, or deleting data).

This approach ensures a clean separation between **commands** and **queries**, adhering to the **CQRS (Command Query Responsibility Segregation)** principle. Each operation is represented by:

1. A **Request** class that defines the input parameters.
2. A **Handler** class that processes the request and returns a response.
3. An optional **Response** object to encapsulate the output.

AddCustomer Command Example

```
namespace DbmsAPI.Application.Features.Commands.Customer.AddCustomer
{
    3 references
    public class AddCustomerCommandRequest : IRequest<AddCustomerCommandResponse>
    {
        1 reference
        public string Name { get; set; }
        1 reference
        public string Surname { get; set; }
        1 reference
        public string Username { get; set; }
        1 reference
        public int Age { get; set; }
        1 reference
        public string Email { get; set; }
        1 reference
        public string Password { get; set; }
        1 reference
        public string Phone { get; set; }
        1 reference
        public string Address { get; set; }
    }
}
```

Figure 55 AddCustomerCommandRequest Class

```
namespace DbmsAPI.Application.Features.Commands.Customer.AddCustomer
{
    1 reference
    public class AddCustomerCommandHandler : IRequestHandler<AddCustomerCommandRequest, AddCustomerCommandResponse>
    {
        private readonly ICustomerWriteRepository _customerWriteRepository;
        private readonly IUnitOfWork _unitOfWork;

        0 references
        public AddCustomerCommandHandler(ICustomerWriteRepository customerWriteRepository, IUnitOfWork unitOfWork)
        {
            _customerWriteRepository = customerWriteRepository;
            _unitOfWork = unitOfWork;
        }

        0 references
        public async Task<AddCustomerCommandResponse> Handle(AddCustomerCommandRequest request, CancellationToken cancellationToken)
        {
            await _unitOfWork.BeginTransactionAsync();

            try
            {
                await _customerWriteRepository.AddAsync(new()
                {
                    Name = request.Name,
                    Surname = request.Surname,
                    Username = request.Username,
                    Age = request.Age,
                    Email = request.Email,
                    Password = request.Password,
                    Phone = request.Phone,
                    Address = request.Address
                });
                await _customerWriteRepository.SaveAsync();

                await _unitOfWork.CommitAsync();
                return new() { Success = true };
            }
            catch (Exception)
            {
                await _unitOfWork.RollbackAsync();
                return new() { Success = false };
            }
        }
    }
```

Figure 56 AddCustomerCommandHandler Class

```
namespace DbmsAPI.Application.Features.Commands.Customer.AddCustomer
{
    6 references
    public class AddCustomerCommandResponse
    {
        3 references
        public bool Success { get; set; }
    }
}
```

Figure 57 AddCustomerCommandResponse Class

- **Persistence Layer of Onion Architecture**

  The Persistence Layer is responsible for implementing the repository interfaces defined in the Application Layer. This layer acts as the data access layer, bridging the application logic and the database.

  Repository Implementations

  To maintain a clean and reusable structure, the following repository classes were implemented:

  Base Repositories

  - **ReadRepository:** Implements **IReadRepository<T>,** providing methods for retrieving data.

```
namespace DbmsAPI.Persistence.Repositories
{
    17 references
    public class ReadRepository<T> : IReadRepository<T> where T : BaseEntity
    {
        private readonly Contexts.DbmsAPIDbContext _context;

        8 references
        public ReadRepository(Contexts.DbmsAPIDbContext context)
        {
            _context = context;
        }

        5 references
        public DbSet<T> Table => _context.Set<T>();

        9 references
        public IQueryable<T> GetAll(bool tracking = true)
        {
            var query = Table.AsQueryable();

            if (!tracking)
                query = query.AsNoTracking();

            return query;
        }

        1 reference
        public async Task<T> GetByIdAsync(string id, bool tracking = true)
        {
            var query = Table.AsQueryable();

            if (!tracking)
                query = query.AsNoTracking();

            T? entity = await query.FirstOrDefaultAsync(x => x.Id == Guid.Parse(id));

            if (entity == null)
                throw new Exception("Entity not found");

            return entity;
        }
```

Figure 58 ReadRepository Class

- **WriteRepository:** Implements **IWriteRepository<T>,** handling data modification operations (Create, Update, Delete).

```
namespace DbmsAPI.Persistence.Repositories
{
    17 references
    public class WriteRepository<T> : IWriteRepository<T> where T : BaseEntity
    {
        private readonly Contexts.DbmsAPIDbContext _context;

        8 references
        public WriteRepository(Contexts.DbmsAPIDbContext context)
        {
            _context = context;
        }

        9 references
        public DbSet<T> Table => _context.Set<T>();

        10 references
        public async Task<bool> AddAsync(T entity)
        {
            EntityEntry<T> entityEntry = await Table.AddAsync(entity);
            return entityEntry.State == EntityState.Added;
        }

        1 reference
        public async Task<bool> AddRangeAsync(List<T> datas)
        {
            await Table.AddRangeAsync(datas);
            return true;
        }
```

Figure 59 WriteRepository Class

- **Both classes utilize a Marker Pattern to restrict the generic type T to BaseEntity, ensuring that only entities inheriting from BaseEntity can be used.**

Entity-Specific Repositories For each entity (except cross-table entities), entity-specific repository classes were created by inheriting from **ReadRepository<T>** and **WriteRepository<T>.** The **Marker Pattern** ensures type safety and enforces the correct entity type.

Example for the **Basket** entity:

```
namespace DbmsAPI.Persistence.Repositories.Basket
{
    2 references
    public class BasketWriteRepository : WriteRepository<Domain.Entities.Basket>, IBasketWriteRepository
    {
        0 references
        public BasketWriteRepository(DbmsAPIDbContext context) : base(context)
        {
        }
    }
}
```

Figure 60 BasketWriteRepository Class

```
namespace DbmsAPI.Persistence.Repositories.Basket
{
    2 references
    public class BasketReadRepository : ReadRepository<Domain.Entities.Basket>, IBasketReadRepository
    {
        0 references
        public BasketReadRepository(DbmsAPIDbContext context) : base(context)
        {
        }
    }
}
```

Figure 61 BasketReadRepositoryClass

Unit of Work Implementation

To coordinate transactional operations across multiple repositories, the **Unit of Work Pattern** was implemented:

```
namespace DbmsAPI.Persistence.Repositories
{

    public class UnitOfWork : IUnitOfWork
    {
        private readonly DbmsAPIDbContext _context;
        private IDbContextTransaction _transaction;


        public UnitOfWork(DbmsAPIDbContext context)
        {
            _context = context;
        }

        11 references
        public async Task BeginTransactionAsync()
        {
            _transaction = await _context.Database.BeginTransactionAsync();
        }
```

Figure 62 UnitOfWork Class

DbContext Configuration

The Persistence Layer includes a **DbmsAPIDbContext** class, inheriting from **DbContext.** This class acts as the main configuration for the database connection and entity mappings.

Entity Configuration

Each entity (except cross-table entities) is mapped to the database using **DbSet** properties.

```
namespace DbmsAPI.Persistence.Contexts
{
    29 references
    public class DbmsAPIDbContext : DbContext
    {
        0 references
        public DbmsAPIDbContext(DbContextOptions options) : base(options)
        {
        }

        0 references
        public DbSet<Advertise> Advertises { get; set; }
        0 references
        public DbSet<Basket> Baskets { get; set; }
        0 references
        public DbSet<Category> Categories { get; set; }
        0 references
        public DbSet<Customer> Customers { get; set; }
        0 references
        public DbSet<Order> Orders { get; set; }
        0 references
        public DbSet<Product> Products { get; set; }
        0 references
        public DbSet<Review> Reviews { get; set; }
        0 references
        public DbSet<Seller> Sellers { get; set; }
        0 references
        public DbSet<User> Users { get; set; }
```

Figure 63 DbmsAPIDbContext Class

Fluent API Configuration

Using the **OnModelCreating** method, entity relationships are configured **with Fluent API**. This ensures proper setup of **primary keys, foreign keys, and table relationships.**

```
protected override void OnModelCreating(ModelBuilder builder)
{
    builder.Entity<Advertise>()
        .HasOne(a => a.Product)
        .WithMany(p => p.Advertises)
        .HasForeignKey(a => a.ProductId)
        .OnDelete(DeleteBehavior.Cascade);

    builder.Entity<Advertise>()
        .HasOne(a => a.Seller)
        .WithMany(s => s.Advertises)
        .HasForeignKey(a => a.SellerId)
        .OnDelete(DeleteBehavior.Cascade);

    builder.Entity<Product>()
        .HasOne(p => p.Category)
        .WithMany(c => c.Products)
        .HasForeignKey(p => p.CategoryId)
        .OnDelete(DeleteBehavior.Cascade);
```

Figure 64 DbmsAPI Class OnModelCreating Method

Overriding SaveChangesAsync

The **SaveChangesAsync** method was overridden to automatically populate the **CreatedAt** property when new records are added:

```
public override async Task<int> SaveChangesAsync(CancellationToken cancellationToken = default)
{
    var datas = ChangeTracker.Entries().Where(e => e.State == EntityState.Added || e.State == EntityState.Modified);

    foreach (var data in datas)
    {
        if (data.Entity is BaseEntity entity)
        {
            entity.Id = entity.Id == Guid.Empty ? Guid.NewGuid() : entity.Id;

            if (data.State == EntityState.Added)
                entity.CreatedAt = DateTime.UtcNow;
            else
                entity.UpdatedAt = DateTime.UtcNow;
        }

    }

    return await base.SaveChangesAsync(cancellationToken);
}
```

Figure 65 DbmsAPIDbContext Class SaveChangesAsync Method

- Presentation Layer

The **Presentation Layer** serves as the entry point of the application, handling HTTP requests and returning appropriate responses to the client. This layer is implemented using ASP.NET Core **Controllers**.

Controllers

Controllers in this layer process requests and interact with the Application and Persistence layers to execute business logic and database operations. Each controller corresponds to a specific entity or feature, providing methods for CRUD operations and other functionalities.

For example, the **CustomerController** includes endpoints for managing customer data:

```
namespace DbmsAPI.API.Controllers
{
    [Route("api/[controller]/[action]")]
    [ApiController]
    1 reference
    public class CustomersController : ControllerBase
    {
        private readonly IMediator _mediator;

        0 references
        public CustomersController(IMediator mediator)
        {
            _mediator = mediator;
        }

        [HttpPost]
        0 references
        public async Task<IActionResult> AddCustomer([FromForm] AddCustomerCommandRequest request)
        {
            AddCustomerCommandResponse response = await _mediator.Send(request);
            return response.Success ? Ok(response) : BadRequest();
        }

        [HttpGet]
        0 references
        public async Task<IActionResult> GetAllCustomers([FromQuery] GetAllCustomersQueryRequest request)
        {
            List<GetAllCustomersQueryResponse> response = await _mediator.Send(request);
            return Ok(response);
        }
    }
}
```

Figure 66 CustomerController Class

Only one controller is provided here as an example to avoid making the report excessively lengthy.

Database Configuration

The connection string for the PostgreSQL database is stored in the **appsettings.json** file to enable seamless integration. This configuration ensures secure and flexible management of database connections.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "ConnectionStrings": {
    "PostgreSQL": "Server=localhost;Port=5432;Database=DbmsAPIDb;User Id=postgres;Password=admin;"
  },
  "AllowedHosts": "*"
}
```

Figure 67 appsetting.json PostgreSQL Connection String

API TESTING RESULTS WITH SWAGGER

To demonstrate that my API works correctly, I tested CRUD (**Create, Read, Delete**) operations for **Customer** using **Swagger**. Below are the screenshots and explanations of the test process.

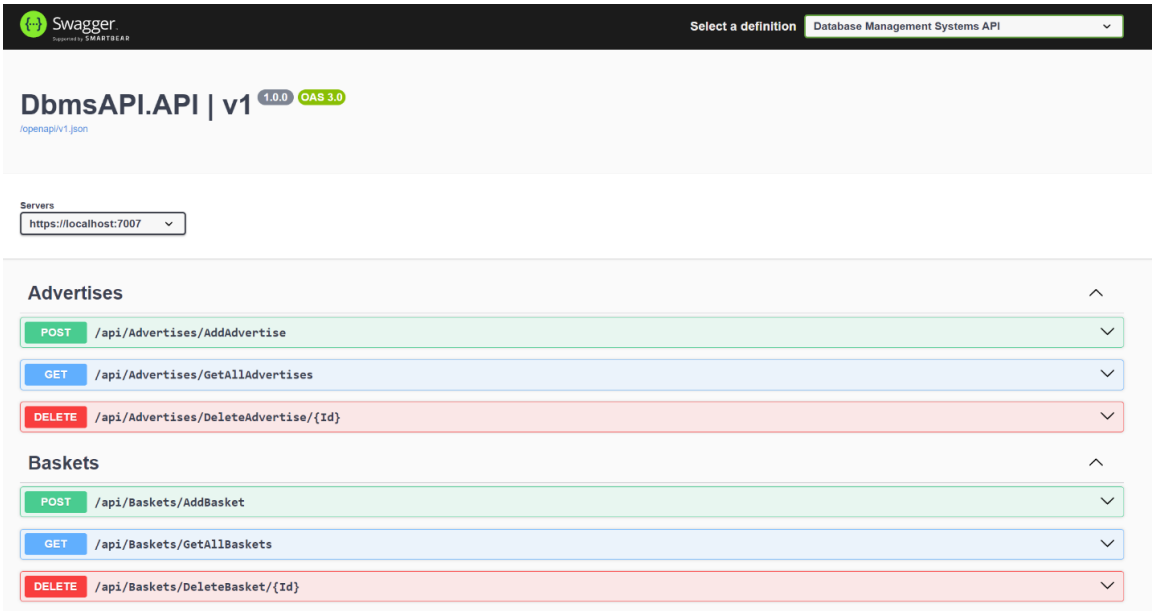


Figure 68 Swagger UI Part_1

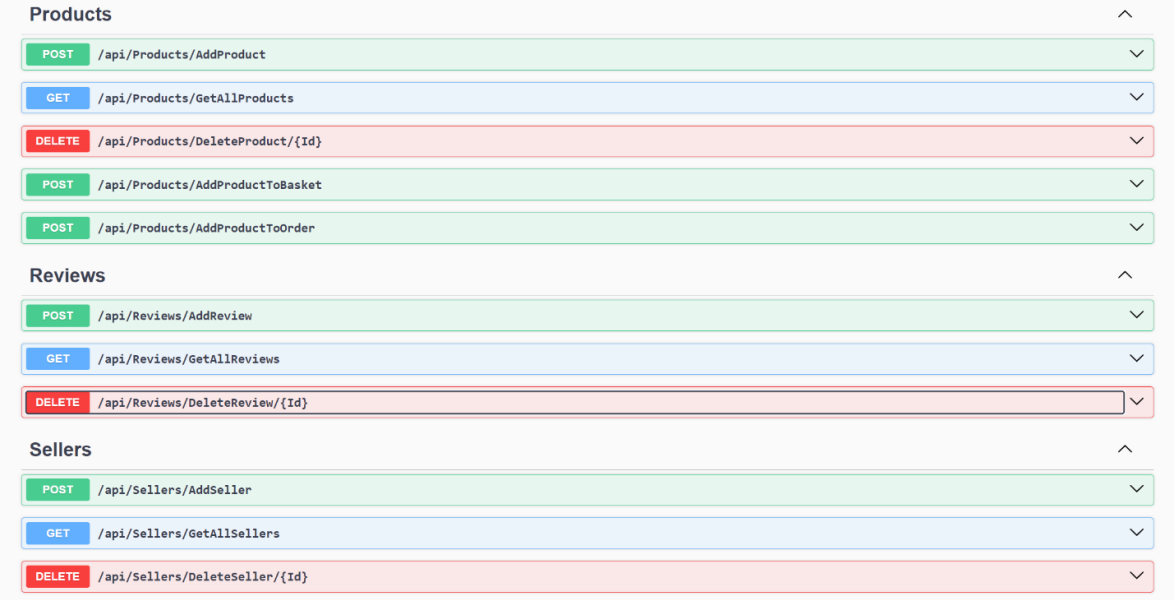


Figure 69 Swagger UI Part_2

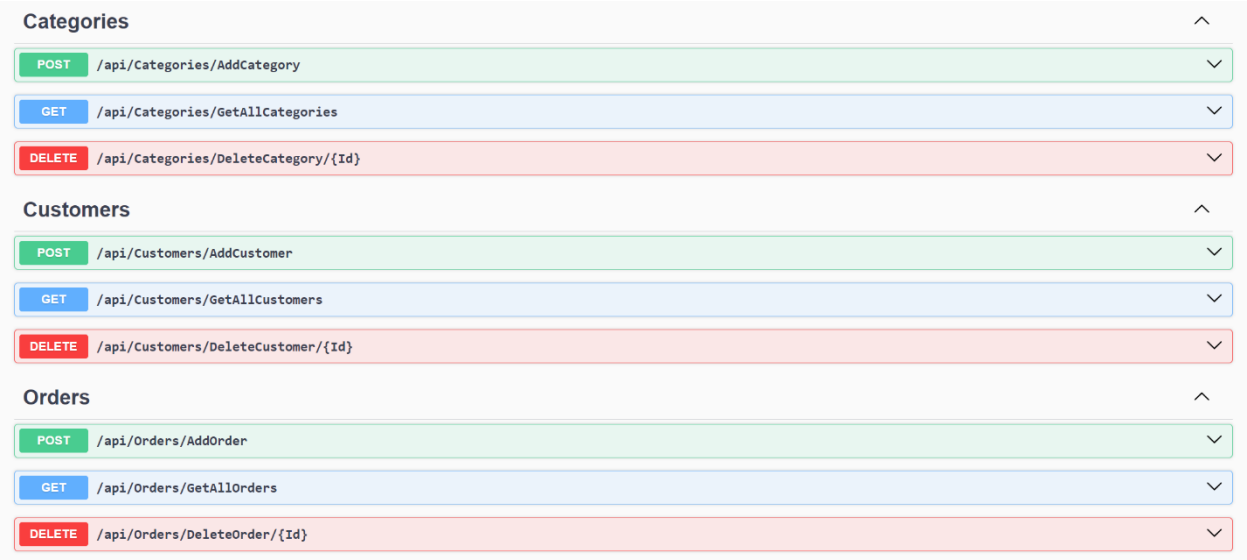


Figure 70 Swagger UI Part_3

```
namespace DbmsAPI.API.Controllers
{
    [Route("api/[controller]/[action]")]
    [ApiController]
    1 reference
    public class CustomersController : ControllerBase
    {
        private readonly IMediator _mediator;

        0 references
        public CustomersController(IMediator mediator)
        {
            _mediator = mediator;
        }

        [HttpPost]
        0 references
        public async Task<IActionResult> AddCustomer([FromForm] AddCustomerCommandRequest request)
        {
            AddCustomerCommandResponse response = await _mediator.Send(request);
            return response.Success ? Ok(response) : BadRequest();
        }

        [HttpGet]
        0 references
        public async Task<IActionResult> GetAllCustomers([FromQuery] GetAllCustomersQueryRequest request)
        {
            List<GetAllCustomersQueryResponse> response = await _mediator.Send(request);
            return Ok(response);
        }

        [HttpDelete("{Id}")]
        0 references
        public async Task<IActionResult> DeleteCustomer([FromRoute] DeleteCustomerCommandRequest request)
        {
            DeleteCustomerCommandResponse response = await _mediator.Send(request);
            return response.Success ? Ok(response) : BadRequest();
        }
    }
}
```

Figure 71 Customers Controller

- **Add Customer Operation**

Implementation with Mediator Pattern of Add Customer Operation

The following classes implement the Add Customer operation using the Mediator Pattern:

- **Request**: Contains the required fields to add a customer.
- **Handler**: Processes the request and adds the customer to the database.
- **Response**: Returns the result of the operation.

```
namespace DbmsAPI.Application.Features.Commands.Customer.AddCustomer
{
    3 references
    public class AddCustomerCommandRequest : IRequest<AddCustomerCommandResponse>
    {
        1 reference
        public string Name { get; set; }
        1 reference
        public string Surname { get; set; }
        1 reference
        public string Username { get; set; }
        1 reference
        public int Age { get; set; }
        1 reference
        public string Email { get; set; }
        1 reference
        public string Password { get; set; }
        1 reference
        public string Phone { get; set; }
        1 reference
        public string Address { get; set; }
    }
}
```

Figure 72 AddCustomerCommandRequest Mediator Pattern Class

```
public class AddCustomerCommandHandler : IRequestHandler<AddCustomerCommandRequest, AddCustomerCommandResponse>
{
    private readonly ICustomerWriteRepository _customerWriteRepository;
    private readonly IUnitOfWork _unitOfWork;

    0 references
    public AddCustomerCommandHandler(ICustomerWriteRepository customerWriteRepository, IUnitOfWork unitOfWork)
    {
        _customerWriteRepository = customerWriteRepository;
        _unitOfWork = unitOfWork;
    }

    0 references
    public async Task<AddCustomerCommandResponse> Handle(AddCustomerCommandRequest request, CancellationToken cancellationToken)
    {
        await _unitOfWork.BeginTransactionAsync();

        try
        {
            await _customerWriteRepository.AddAsync(new()
            {
                Name = request.Name,
                Surname = request.Surname,
                Username = request.Username,
                Age = request.Age,
                Email = request.Email,
                Password = request.Password,
                Phone = request.Phone,
                Address = request.Address
            });
            await _customerWriteRepository.SaveAsync();

            await _unitOfWork.CommitAsync();
            return new() { Success = true };
        }
        catch (Exception)
        {
            await _unitOfWork.RollbackAsync();
            return new() { Success = false };
        }
    }
}
```

Figure 73 AddCustomerCommandHandler Mediator Pattern Class

```
namespace DbmsAPI.Application.Features.Commands.Customer.AddCustomer
{
    6 references
    public class AddCustomerCommandResponse
    {
        3 references
        public bool Success { get; set; }
    }
}
```

Figure 74 AddCustomerCommandResponse Mediator Pattern Class

Swagger Test of Add Customer Operation

The following request was sent to the POST /api/customers endpoint via Swagger to add a customer:

- **Endpoint**: /api/customers/AddCustomer

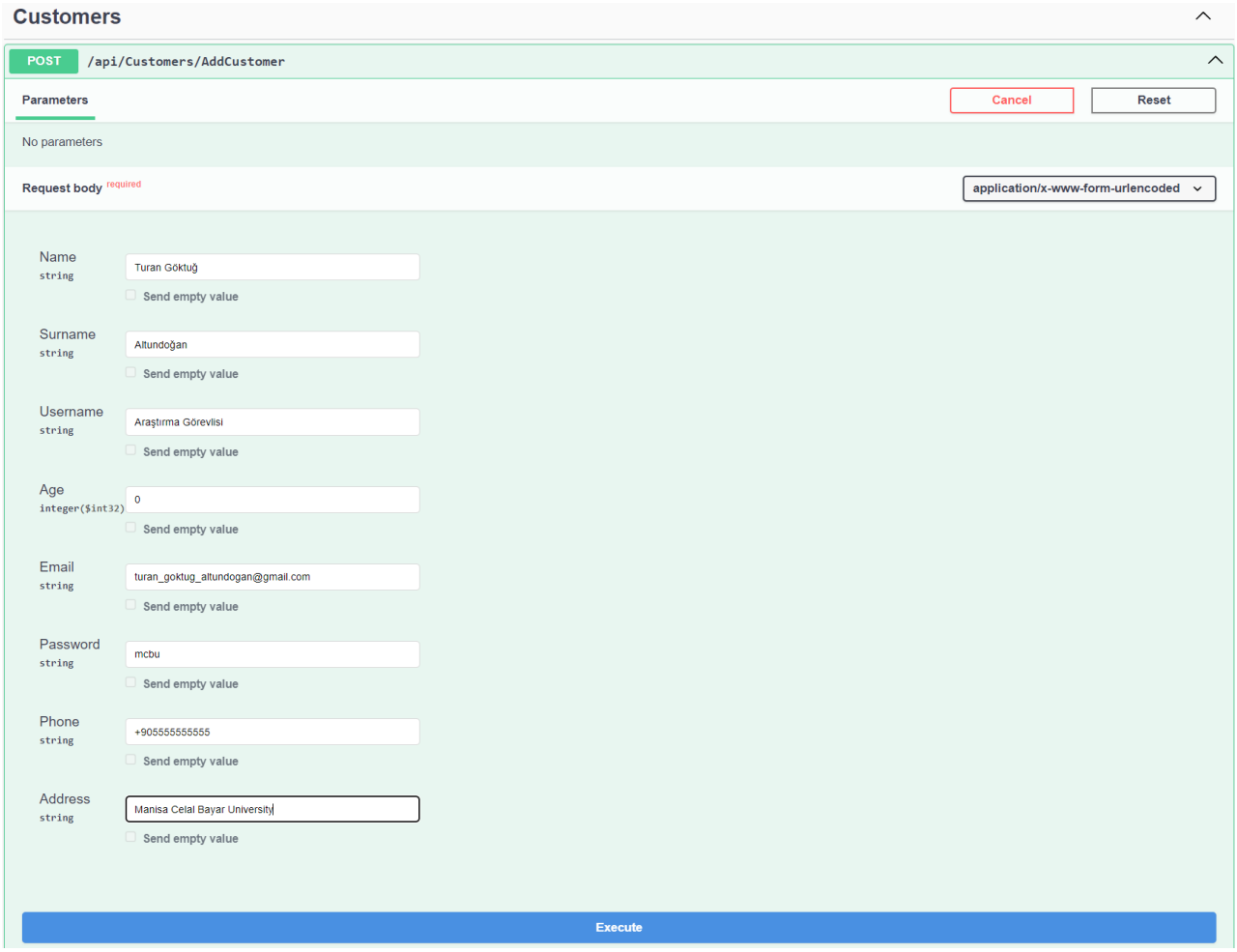


Figure 75 Swagger AddCustomer Operation Request Page

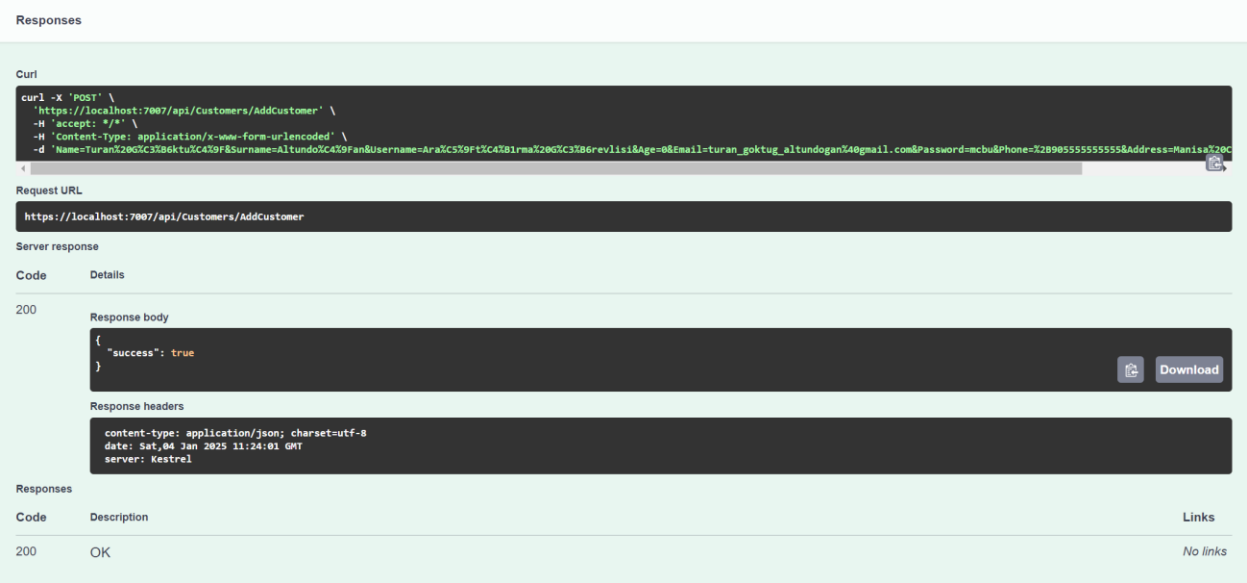


Figure 76 Swagger AddCustomer Operation Response Page

Database Confirmation of Add Customer Operation

The Customers table in the database was queried to confirm that the new customer was successfully added:

	Id	Name	Surname	Username	Age	Email	Password	Phone	Address
1	049a69bb-38d6-4212-b8e2-65574344671a	Berke	Alpaslan	BaraCuda	22	210316016@ogr.cbu.edu.tr	210316016	+905555555555	Manisa Celal Bayar Univer
2	d9aaa2ba-305e-414b-8d59-77365634c890	Mert Doğan	Aygün	Merdo	22	210316076@ogr.cbu.edu.tr	210316076	+905555555555	Manisa Celal Bayar Univer
3	2577a62e-b88b-483e-93ed-d89e74c0c210	Gül	Yeşilgil	Rose	21	220316055@ogr.cbu.edu.tr	220316055	+905555555555	Manisa Celal Bayar Univer
4	05a09696-47cf-4d7a-ac02-b364ab82b17d	Joe	Biden	joe_biden	82	joe_biden@gmail.com	usa	+905555555555	USA
5	66727aee-d51d-4828-8405-b27ebd3d7d5a	Donald	Trump	donald_trump	78	donald_trump@gmail.com	usa	+905555555555	USA
6	b69b3f25-5107-4133-b777-1a23caa4de0d	Hayko	Cepkin	hayfan	46	hayko_cepkin@gmail.com	hayfan	+905555555555	Turkey
7	08307427-a94c-43f8-b812-2b69e76012e6	Sezen	Aksu	Serçe	70	sezen_aksu@gmail.com	miniksерce	+905555555555	İstanbul
8	f502fc49-b7dc-42ae-9024-789035c13ef0	Nazan	Öncel	Naz	68	nazan_oncel@gmail.com	naz	+905555555555	İstanbul
9	5999c6ce-33d5-4318-b06b-45777cd5967e	Oğuz	Çağlar	Kısa Dönem	24	kisa_dönem@dag.com	bekir	+905555555555	İstanbul
10	b91c1202-a50d-4c97-a8a4-ac2c0388d2fa	Bekir	Özbey	M1_Abrams	26	zaten_bekir@dag.com	oguz	+905555555555	Ankara
11	48bb9f43-daa2-4be9-ad2f-cb8b59a73766	Turan Göktuğ	Altundoğan	Araştırma Görevlisi	0	turan_goktug_altundogan@gmail.com	mcbu	+905555555555	Manisa Celal Bayar Univer

Figure 77 PostgreSQL Customers Table After AddCustomer with Swagger

- **Get All Customers**

Implementation with Mediator Pattern of Get All Customers Operation

The following classes implement the Get All Customers operation using the Mediator Pattern:

- **Request**: Does not require any specific parameters for fetching all customers.
- **Handler**: Fetches all customer records from the database.
- **Response**: Returns the list of customers.

```
namespace DbmsAPI.Application.Features.Queries.Customer.GetAllCustomers
{
    3 references
    public class GetAllCustomersQueryRequest : IRequest<List<GetAllCustomersQueryResponse>>
    {
    }
}
```

Figure 78 GetAllCustomersQueryRequest Mediator Pattern Class

```
public class GetAllCustomersQueryHandler : IRequestHandler<GetAllCustomersQueryRequest, List<GetAllCustomersQueryResponse>>
{
    private readonly ICustomerReadRepository _customerReadRepository;
    0 references
    public GetAllCustomersQueryHandler(ICustomerReadRepository customerReadRepository)
    {
        _customerReadRepository = customerReadRepository;
    }

    0 references
    public async Task<List<GetAllCustomersQueryResponse>> Handle(GetAllCustomersQueryRequest request, CancellationToken
      cancellationToken)
    {
        var customers = await _customerReadRepository.GetAll(false)
            .Select(customer => new GetAllCustomersQueryResponse
            {
                Id = customer.Id.ToString(),
                Name = customer.Name,
                Surname = customer.Surname,
                Username = customer.Username,
                Age = customer.Age,
                Email = customer.Email,
                Password = customer.Password,
                Phone = customer.Phone,
                Address = customer.Address,
                CreatedAt = customer.CreatedAt,
                UpdatedAt = customer.UpdatedAt
            }).ToListAsync();

        return customers;
    }
}
```

Figure 79 GetAllCustomersQueryHandler Mediator Pattern Class

```
namespace DbmsAPI.Application.Features.Queries.Customer.GetAllCustomers
{

    public class GetAllCustomersQueryResponse
    {

        public string Id { get; set; }
        1 reference
        public string Name { get; set; }
        1 reference
        public string Surname { get; set; }
        1 reference
        public string Username { get; set; }
        1 reference
        public int Age { get; set; }
        1 reference
        public string Email { get; set; }
        1 reference
        public string Password { get; set; }
        1 reference
        public string Phone { get; set; }
        1 reference
        public string Address { get; set; }
        1 reference
        public DateTime CreatedAt { get; set; }
        1 reference
        public DateTime? UpdatedAt { get; set; }
    }
}
```

Figure 80 GetAllCustomersQueryResponse Mediator Pattern Class

Swagger Test of Get All Customers Operation

The GET /api/customers endpoint was tested via Swagger to retrieve all customers:

- **Endpoint**: /api/customers/GetAllCustomers

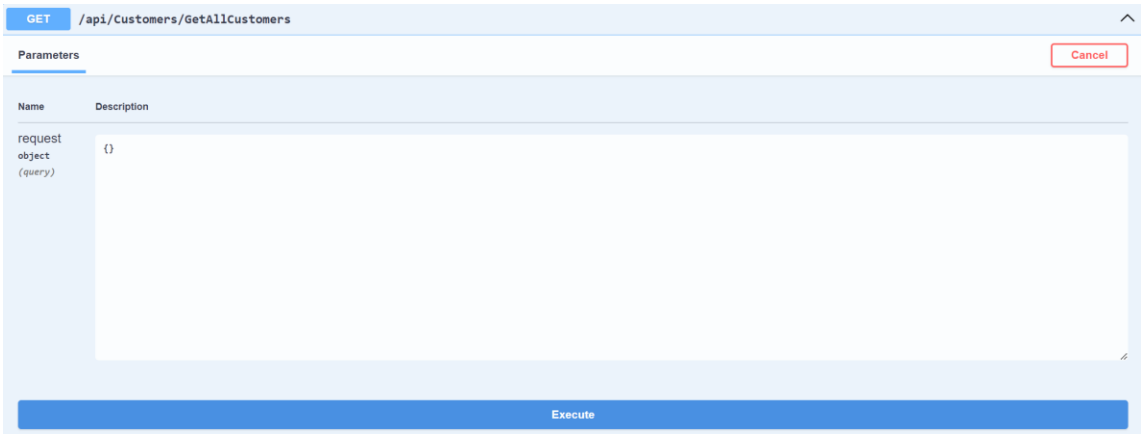


Figure 81 Swagger GetAllCustomers Operation Request Page

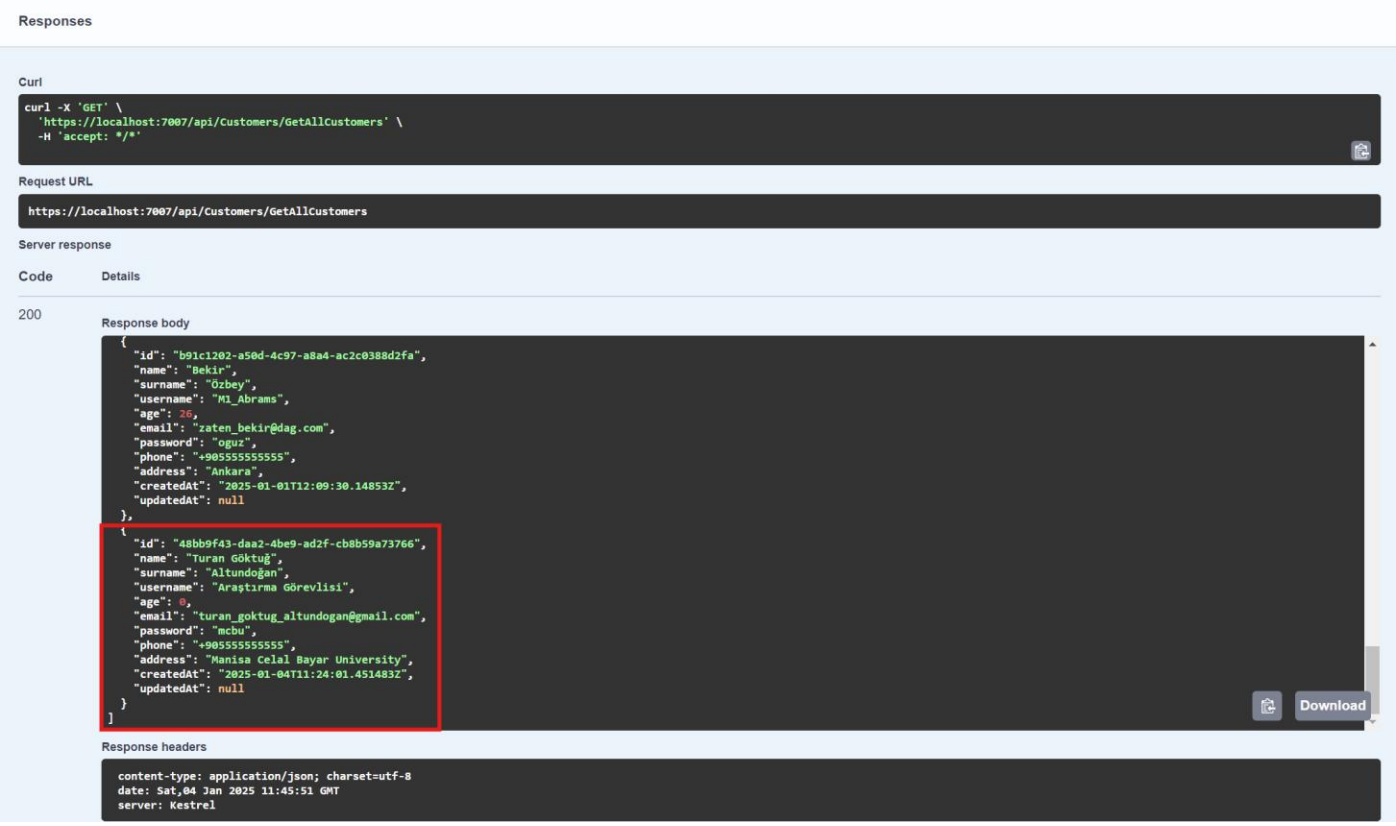


Figure 82 Swagger GetAllCustomers Operation Response Page

Database Confirmation of Get All Customers Operation

The Customers table in the database was queried to confirm that.

	Id	Name	Surname	Username	Age	Email	Password	Phone	Address
1	049a69bb-38d6-4212-b8e2-65574344671a	Berke	Alpaslan	BaraCuda	22	210316016@ogr.cbu.edu.tr	210316016	+905555555555	Manisa Celal Bayar Univer
2	d9aaa2ba-305e-414b-8d59-77365634c890	Mert Doğan	Aygün	Merdo	22	210316076@ogr.cbu.edu.tr	210316076	+905555555555	Manisa Celal Bayar Univer
3	2577a62e-b88b-483e-93ed-d89e74c0c210	Gül	Yeşilgil	Rose	21	220316055@ogr.cbu.edu.tr	220316055	+905555555555	Manisa Celal Bayar Univer
4	05a09696-47cf-4d7a-ac02-b364ab82b17d	Joe	Biden	joe_biden	82	joe_biden@gmail.com	usa	+905555555555	USA
5	66727aee-d51d-4828-8405-b27ebd3d7d5a	Donald	Trump	donald_trump	78	donald_trump@gmail.com	usa	+905555555555	USA
6	b69b3f25-5107-4133-b777-1a23caa4de0d	Hayko	Cepkin	hayfan	46	hayko_cepkin@gmail.com	hayfan	+905555555555	Turkey
7	08307427-a94c-43f8-b812-2b69e76012e6	Sezen	Aksu	Serçe	70	sezen_aksu@gmail.com	miniksерce	+905555555555	İstanbul
8	f502fc49-b7dc-42ae-9024-789035c13ef0	Nazan	Öncel	Naz	68	nazan_oncel@gmail.com	naz	+905555555555	İstanbul
9	5999c6ce-33d5-4318-b06b-45777cd5967e	Oğuz	Çağlar	Kısa Dönem	24	kisa_dönem@dag.com	bekir	+905555555555	İstanbul
10	b91c1202-a50d-4c97-a8a4-ac2c0388d2fa	Bekir	Özbey	M1_Abrams	26	zaten_bekir@dag.com	oguz	+905555555555	Ankara
11	48bb9f43-daa2-4be9-ad2f-cb8b59a73766	Turan Göktuğ	Altundoğan	Araştırma Görevlisi	0	turan_goktug_altundogan@gmail.com	mcbu	+905555555555	Manisa Celal Bayar Univer

Figure 83 PostgreSQL Customers Table to Confirmation GetAllCustomers Operation from Swagger

- **Remove Customer**

Implementation with Mediator Pattern of Remove Customer Operation

The following classes implement the Remove Customer operation using the Mediator Pattern:

- **Request**: Contains the ID of the customer to be removed.
- **Handler**: Processes the request and removes the customer from the database.
- **Response**: Confirms whether the removal was successful.

```
namespace DbmsAPI.Application.Features.Commands.Customer.DeleteCustomer
{
    3 references
    public class DeleteCustomerCommandRequest : IRequest<DeleteCustomerCommandResponse>
    {
        1 reference
        public string Id { get; set; }
    }
}
```

Figure 84 DeleteCustomerCommandRequest Mediator Pattern Class

```
public class DeleteCustomerCommandHandler : IRequestHandler<DeleteCustomerCommandRequest, DeleteCustomerCommandResponse>
{
    private readonly ICustomerWriteRepository _customerWriteRepository;
    private readonly IUnitOfWork _unitOfWork;

    0 references
    public DeleteCustomerCommandHandler(ICustomerWriteRepository customerWriteRepository, IUnitOfWork unitOfWork)
    {
        _customerWriteRepository = customerWriteRepository;
        _unitOfWork = unitOfWork;
    }

    0 references
    public async Task<DeleteCustomerCommandResponse> Handle(DeleteCustomerCommandRequest request, CancellationToken cancellationToken)
    {
        await _unitOfWork.BeginTransactionAsync();

        try
        {
            await _customerWriteRepository.RemoveAsync(request.Id);
            await _customerWriteRepository.SaveAsync();
            await _unitOfWork.CommitAsync();
            return new() { Success = true };
        }
        catch (Exception)
        {
            await _unitOfWork.RollbackAsync();
            return new() { Success = false };
        }
    }
}
```

Figure 85 DeleteCustomerCommandHandler Mediator Pattern Class

```
namespace DbmsAPI.Application.Features.Commands.Customer.DeleteCustomer
{
    6 references
    public class DeleteCustomerCommandResponse
    {
        3 references
        public bool Success { get; set; }
    }
}
```

Figure 86 DeleteCustomerCommandResponse Mediator Pattern Class

Swagger Test of Remove Customer Operation

The DELETE /api/customers/{id} endpoint was tested via Swagger to delete a customer:

- **Endpoint**: /api/customers/{id}
- **Request**: Replace {id} with the ID of the **Turan Göktuğ Altundoğan** to delete (**/api/customers/ 48bb9f43-daa2-4be9-ad2f-cb8b59a73766**).

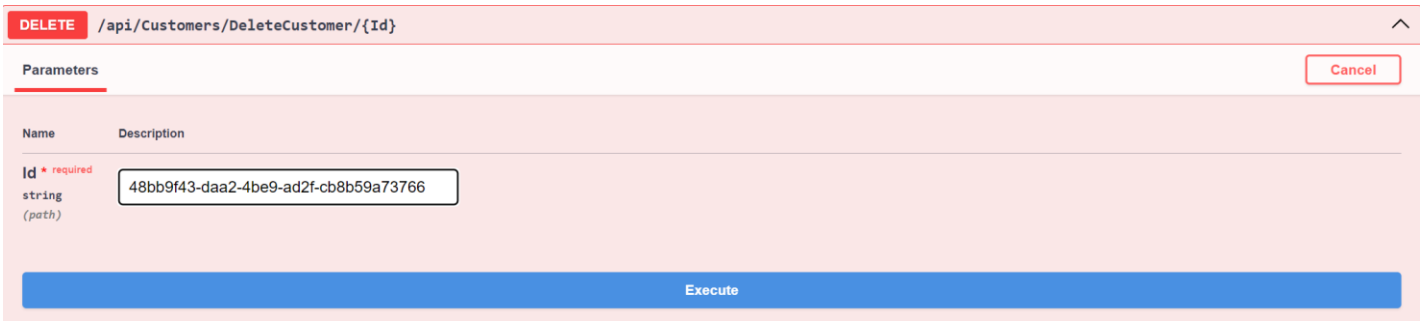


Figure 87 Swagger DeleteCustomer Operation Request Page

Figure 88 Swagger DeleteCustomer Operation Response Page

Database Confirmation of Delete Customer Operation

The Customers table in the database was queried after the deletion to confirm the **Turan Göktuğ Altundoğan (Id = 48bb9f43-daa2-4be9-ad2f-cb8b59a73766)**record was removed:

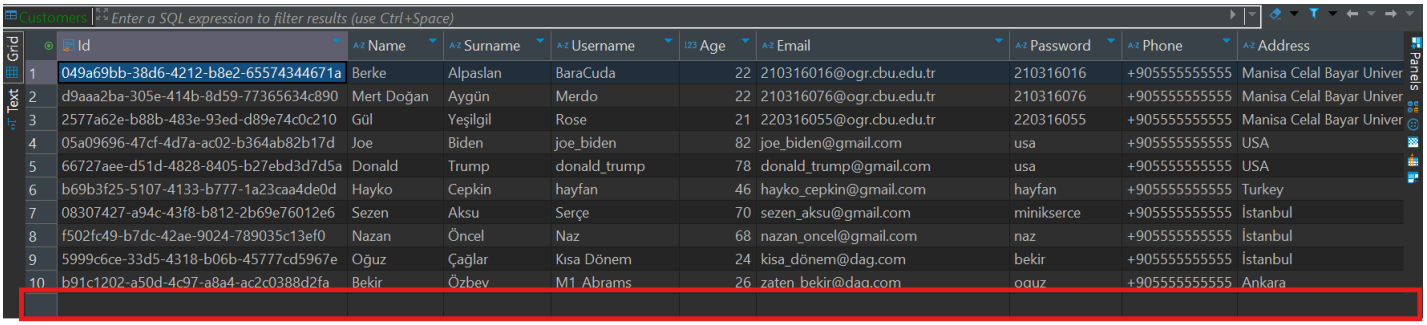


Figure 89 PostgreSQL Customers Table After DeleteCustomer Operation From Swagger

Summary

The above steps validate the functionality of the API using the Mediator Pattern and Swagger testing. Screenshots of the implementations, Swagger tests, and database states clearly demonstrate that the Add, Get All, and Remove operations work as expected.

CONCLUSION

This project successfully demonstrates the design and development of an e-commerce platform API using modern software engineering principles, with a strong emphasis on both database design and clean code implementation. By integrating **Onion Architecture** with PostgreSQL as the database backend, the project achieved a balance between a scalable architecture and an optimized data structure.

Key Accomplishments:

1. **Database Design and Implementation**:
    - An **ER Diagram** was created to model the relationships between entities such as Customers, Orders, Products, and Categories. Special attention was given to representing real-world relationships like One-to-Many (e.g., Orders to OrderDetails) and Many-to-Many (e.g., Products and Baskets via cross-table entities like BasketProduct).
    - **Cross-table entities** such as BasketProduct and ProductOrder were introduced to handle many-to-many relationships effectively. These entities were optimized to focus solely on mapping relationships without redundant attributes.
    - **Fluent API** configurations ensured the correct enforcement of primary keys, foreign keys, and constraints within the database, aligning the schema with the project's business requirements.
2. **SQL Queries and Reporting**:
    - Complex SQL queries were written and tested to extract valuable insights, such as:
        - The count of products in categories without subcategories.
        - Customers who both added products to their basket and completed purchases.
        - The average basket price for each customer.
    - The use of Dapper allowed the seamless execution of these queries within the API, ensuring both performance and flexibility in handling advanced reporting requirements.
3. **API Development**:
    - A multi-layered architecture was implemented to separate concerns effectively:
        - **Domain Layer** encapsulated the core business logic, with reusable base classes like BaseEntity and User to minimize redundancy.

        - **Application Layer** utilized **Mediator Pattern** for modular and reusable CRUD operations, with design patterns like **Generic Repository** and **Unit of Work** enhancing code efficiency and maintainability.
        - **Persistence Layer** handled database interactions, leveraging repositories and Fluent API for efficient data management.
        - **Presentation Layer** served as the entry point, exposing API endpoints and enabling interactive testing with Swagger.
    - Key API operations such as Add, Get All, and Remove were implemented and tested, showcasing the seamless integration of database queries, business logic, and presentation.

4. **Database-Driven Approach**:
    - The entire implementation was rooted in a well-structured database design. Entity relationships and table constraints were carefully modeled to ensure data consistency.
    - The choice of **PostgreSQL** as the database ensured robust handling of complex queries while maintaining high performance.
    - SQL queries were validated against the database using real-world test scenarios, and results were visualized through Swagger tests and database table snapshots.

Challenges Addressed:

- Managing complex relationships between entities such as Products, Orders, and Categories required careful design and testing.
- The integration of cross-table entities (e.g., BasketProduct) streamlined many-to-many relationships while avoiding redundancy.
- Use EF Core for standard CRUD operations provided the flexibility needed for diverse use cases.

Outcomes:

This project successfully delivered a functional, scalable, and maintainable e-commerce platform API. Key outcomes include:

- **A robust database design** that reflects real-world relationships and supports complex queries for reporting and analytics.
- **A clean and modular architecture** that ensures maintainability and scalability for future enhancements.

- **Seamless integration of database operations and business logic**, demonstrated through end-to-end testing and validation.

In conclusion, the project not only fulfilled its objectives but also provided valuable insights into the integration of database design and modern coding practices. The methodologies, tools, and patterns used here serve as a strong foundation for developing scalable and efficient software systems in the future.