Case study

Ecommerce Application

By

JASWANTH KUMAR S

BATCH 4

Table of content

1. Introduction

The rapid growth of online shopping has increased the demand for robust, scalable, and user-friendly ecommerce platforms. This project aims to simulate a simplified backend for an ecommerce system, focusing on core concepts of software development such as object-oriented programming, SQL database interaction, exception handling, and unit testing. By modeling real-world entities like customers, products, carts, and orders, this system demonstrates how data flows through different components in a layered architecture. The application is built using modular principles, making it easier to test, maintain, and extend. The project serves as a hands-on case study to bridge theoretical knowledge with practical implementation.

2. Purpose of the Project

The purpose of this E-commerce Application project is to simulate a real-world online shopping backend system using core concepts of object-oriented programming, SQL database integration, control structures, exception handling, and unit testing. This project aims to develop a menu-driven, modular application that manages customers, products, carts, and orders through a structured and layered architecture. By incorporating entity modeling, a DAO layer for database operations, custom exception handling, and utility-based DB connectivity, the project provides a hands-on understanding of full-stack backend development principles. It is intended to strengthen problem-solving and software engineering skills in a scalable, testable, and maintainable manner.

3. Scope of the Project

This project covers the complete development of an Ecommerce backend system with the following scope:

Entity Modeling: Implement real-world entities like Customer, Product, Cart, Order, and OrderItem with private attributes, constructors, and getter/setter methods.

Data Access Layer (DAO): Create interfaces and an implementation class (OrderProcessorRepositoryImpl) to handle all database operations using JDBC.

Custom Exceptions: Define and handle user-defined exceptions such as CustomerNotFoundException, ProductNotFoundException, and OrderNotFoundException.

Database Integration: Design and implement relational tables in MySQL with proper foreign key relationships. Use PropertyUtil and DBConnection utilities to manage secure and dynamic DB connections.

Functionalities: Provide customer registration, product creation/deletion, cart operations, and order processing with support for listing customer orders.

Menu-Driven Application: Develop a EcomApp class in the main package to drive all operations through user interaction.

Unit Testing: Ensure system reliability by writing test cases for product creation, cart addition, order placement, and exception handling using the unittest module.

4. Structure of the project

4.1 SQL Structure (Database Schema)

The Ecommerce database schema is designed to efficiently store and manage customer information, product details, cart operations, and order processing. It follows a relational structure using MySQL and ensures data integrity using primary and foreign key constraints.

Database creation:

```
create database ecomm_db;

use ecomm_db;
```

Table creation:

1. customers table:

• customer_id (Primary Key)

• name

• email

• password

Description:

This table stores details of all users who register on the platform.

- Primary Key: customer_id uniquely identifies each customer.
- Fields: name, email, and password.
- The email is unique to prevent duplicate accounts.

SQL Query:

```
create table customers (
    customer_id int primary key auto_increment,
    name varchar(100) not null,
```

email varchar(100) unique not null,

password varchar(255) not null

);

2. products table

• product_id (Primary Key)

• name

• price

• description

• stockQuantity

Description:

This table holds information about products available for purchase.

- Primary Key: product_id.
- Fields: name, price, description, and stockQuantity.
- price is stored as a decimal to handle currency format.
- stockQuantity tracks the available units in inventory.

SQL Query:

create table products (

product_id int primary key auto_increment,

name varchar(100) not null,

price decimal(10,2) not null,

description text,

stockquantity int not null

);

3. cart table:

• cart_id (Primary Key)

• customer_id (Foreign Key)

• product_id (Foreign Key)

• quantity

Description:

This table maintains temporary product selections by customers.

- Primary Key: cart_id.
- Foreign Keys:
  - customer_id references customers
  - product_id references products
- Field: quantity indicates how many units the customer wants.
- ON DELETE CASCADE ensures cart items are removed if related customer/product is deleted.

SQL Query:

```
create table cart (
    cart_id int primary key auto_increment,
    customer_id int,
    product_id int,
    quantity int not null,
    foreign key (customer_id) references customers(customer_id) on delete cascade,
    foreign key (product_id) references products(product_id) on delete cascade
);
```

4. orders table:

• order_id (Primary Key)

• customer_id (Foreign Key)

• order_date

• total_price

• shipping_address

Description:

Stores the finalized purchases made by customers.

- Primary Key: order_id.
- Foreign Key: customer_id references customers.
- Fields:
    - order_date automatically captures the timestamp of the order.
    - total_price stores the total amount.
    - shipping_address contains delivery details.

SQL Query:

```
create table orders (
    order_id int primary key auto_increment,
    customer_id int,
    order_date timestamp default current_timestamp,
    total_price decimal(10,2) not null,
    shipping_address text not null,
    foreign key (customer_id) references customers(customer_id) on delete cascade
);
```

5. order_items table (to store order details):

• order_item_id (Primary Key)

• order_id (Foreign Key)

• product_id (Foreign Key)

• quantity

Description:

This is a junction table to handle many-to-many relationships between orders and products.

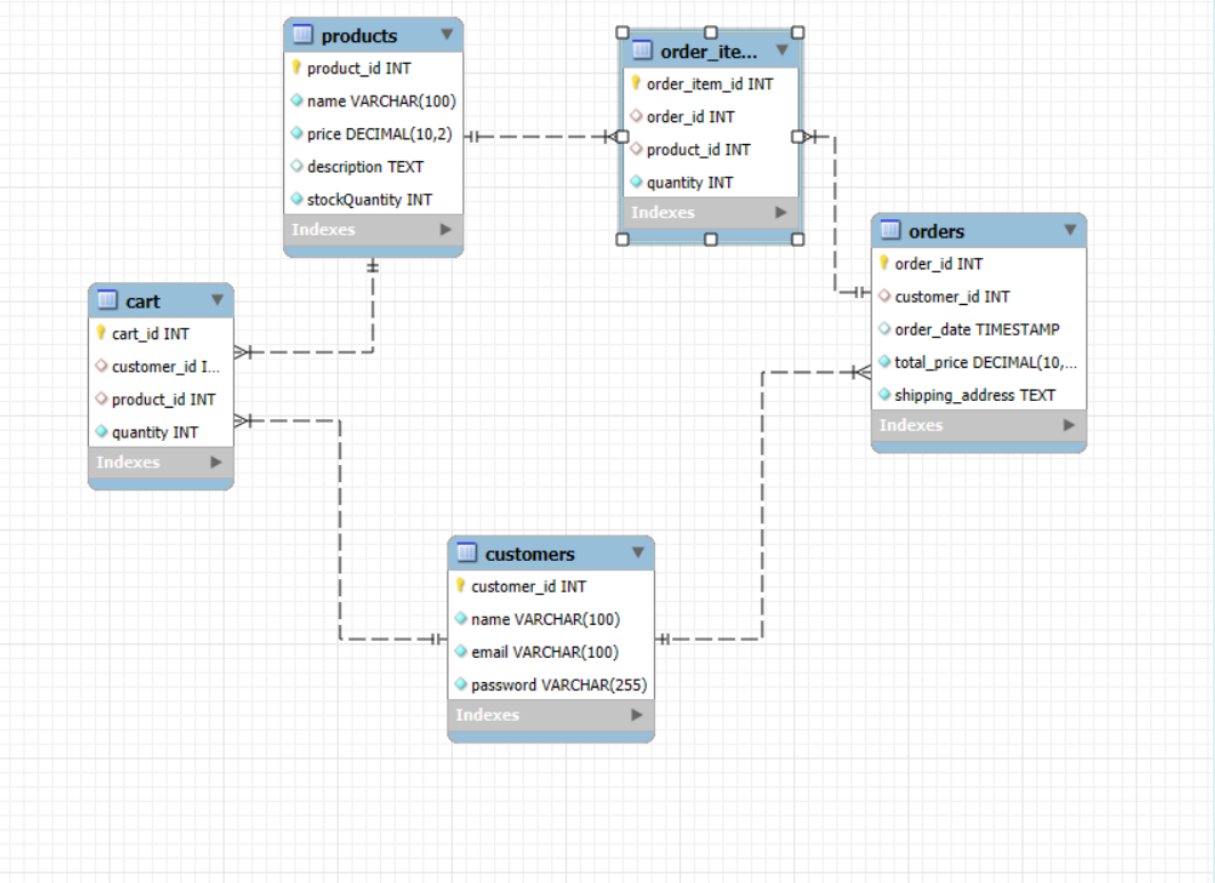
- Primary Key: order_item_id.
- Foreign Keys:
  - order_id references orders
  - product_id references products
- Field: quantity denotes how many units of the product are in that order.

SQL Query:

```
create table order_items (
    order_item_id int primary key auto_increment,
    order_id int,
    product_id int,
    quantity int not null,
    foreign key (order_id) references orders(order_id) on delete cascade,
    foreign key (product_id) references products(product_id) on delete cascade
);
```

ER Diagram

An ER (Entity-Relationship) Diagram is a visual representation of the entities (tables) in a database and the relationships between them. It helps in designing and understanding the logical structure of the database. Entities represent real-world objects like Customer, Product, Order, and the relationships represent how these entities are related



4.2 OOP Structure (Object-Oriented Programming)

1.Entity Module (entity/):

Defines model classes representing database tables. These are pure data holders with attributes but no business logic.

Structure: Each class includes:

- Private variables
- Default and parameterized constructors
- Getters and setters

No business logic here, only data representation

Customer.py

- Attributes: customer_id, name, email, phone, address.
- Represents a customer who places orders.
- Central to linking users to transactions.

CODE:

```
class Customer:
    def __init__(self, customer_id=None, name=None, email=None, password=None):
        self.__customer_id = customer_id
        self.__name = name
        self.__email = email
        self.__password = password

    # Getters
    def get_customer_id(self):
```

```
        return self.__customer_id

    def get_name(self):
        return self.__name

    def get_email(self):
        return self.__email

    def get_password(self):
        return self.__password

# Setters
    def set_customer_id(self, customer_id):
        self.__customer_id = customer_id

    def set_name(self, name):
        self.__name = name

    def set_email(self, email):
        self.__email = email

    def set_password(self, password):
        self.__password = password
```

Product.py

- Attributes: product_id, name, price, category.
- Serves as a superclass for Clothing and Electronics.
- Avoids duplication of base product fields.

CODE:

```
class Product:
    def __init__(self, product_id=None, name=None, price=None, description=None, stock_quantity=None):
        self.__product_id = product_id
        self.__name = name
        self.__price = price
        self.__description = description
        self.__stock_quantity = stock_quantity

    # Getters
    def get_product_id(self):
        return self.__product_id

    def get_name(self):
        return self.__name

    def get_price(self):
        return self.__price

    def get_description(self):
```

```
    return self.__description

def get_stock_quantity(self):
    return self.__stock_quantity

# Setters
def set_product_id(self, product_id):
    self.__product_id = product_id

def set_name(self, name):
    self.__name = name

def set_price(self, price):
    self.__price = price

def set_description(self, description):
    self.__description = description

def set_stock_quantity(self, stock_quantity):
    self.__stock_quantity = stock_quantity
```

Cart.py

- Defines a class that represents the **shopping cart** functionality in an e-commerce system. Each instance of Cart stores information about a customer's selected product and the quantity they intend to purchase.
- Represents an item in a user's cart. Helps the system temporarily track products before placing an order.
- This class can be used by the DAO layer to map a cart database table.
- Useful for features like **"Add to Cart"**, **"View Cart"**, or **"Update Cart"**.
- It provides a structured object to transfer cart data between the database and application logic.

CODE:

```
class Cart:
    def __init__(self, cart_id=None, customer_id=None, product_id=None, quantity=None):
        self.__cart_id = cart_id
        self.__customer_id = customer_id
        self.__product_id = product_id
        self.__quantity = quantity

    # Getters
    def get_cart_id(self):
        return self.__cart_id

    def get_customer_id(self):
        return self.__customer_id
```

```
def get_product_id(self):
    return self.__product_id

def get_quantity(self):
    return self.__quantity

# Setters
def set_cart_id(self, cart_id):
    self.__cart_id = cart_id

def set_customer_id(self, customer_id):
    self.__customer_id = customer_id

def set_product_id(self, product_id):
    self.__product_id = product_id

def set_quantity(self, quantity):
    self.__quantity = quantity
```

Order.py
Tracks information related to a **placed order**, such as which customer placed it, when, the cost, and delivery address.

- Maps directly to the **orders table** in the database.
- DAO methods use this class to:
- Insert new orders
- Retrieve order history
- Track individual customer orders
- Helps decouple business logic from database logic.

CODE:

```
class Order:

  def __init__(self, order_id=None, customer_id=None, order_date=None, total_price=None, shipping_address=None):

      self.__order_id = order_id

      self.__customer_id = customer_id


      self.__order_date = order_date

      self.__total_price = total_price

      self.__shipping_address = shipping_address


  # Getters

  def get_order_id(self):

      return self.__order_id

  def get_customer_id(self):

      return self.__customer_id
```

```
    def get_order_date(self):
        return self.__order_date

    def get_total_price(self):
        return self.__total_price

    def get_shipping_address(self):
        return self.__shipping_address
```

```
# Setters
def set_order_id(self, order_id):
    self.__order_id = order_id

def set_customer_id(self, customer_id):
    self.__customer_id = customer_id


def set_order_date(self, order_date):
    self.__order_date = order_date


def set_total_price(self, total_price):
    self.__total_price = total_price
```

```
def set_shipping_address(self, shipping_address):
    self.__shipping_address = shipping_address
```

OrderItem.py

Represents an individual product and its quantity as part of a larger order. Multiple OrderItem instances collectively make up an Order.

- Allows the system to support **orders with multiple products**.
- Used by DAO methods to:
- Insert each item when placing an order.
- Retrieve itemized breakdown of orders.
- Supports calculating total prices and tracking inventory per product.

```
class OrderItem:

  def __init__(self, order_item_id=None, order_id=None, product_id=None, quantity=None):

      self.__order_item_id = order_item_id

      self.__order_id = order_id

      self.__product_id = product_id

      self.__quantity = quantity


  # Getters

  def get_order_item_id(self):

      return self.__order_item_id


  def get_order_id(self):

      return self.__order_id


  def get_product_id(self):

      return self.__product_id
```

```
def get_quantity(self):
    return self.__quantity

# Setters
def set_order_item_id(self, order_item_id):
    self.__order_item_id = order_item_id

def set_order_id(self, order_id):
    self.__order_id = order_id

def set_product_id(self, product_id):
    self.__product_id = product_id

def set_quantity(self, quantity):
    self.__quantity = quantity
```

2.DAO Module (dao/):

Handles business logic and database interaction.

Files:

- OrderProcessorRepository.py
    - Abstract interface with method signatures like createCustomer(), placeOrder(), etc.
- OrderProcessorRepositoryImpl.py
    - Implements all the methods defined in the interface
    - Uses SQL queries to interact with MySQL DB

Responsibilities:

- Add/delete customer/product
- Cart operations (add/remove/view)
- Place order and retrieve orders
- Handle data persistence

Files:

OrderProcessorRepository.py

- Declares abstract methods for all operations (add user, view products, create order, etc.).
- Ensures a consistent contract for implementation.

CODE:

```
from abc import ABC, abstractmethod
from typing import List, Dict
import sys
import os


sys.path.append(os.path.abspath(os.path.dirname(__file__)))
sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), '..')))

from entity.Customer import Customer
from entity.Product import Product

class OrderProcessorRepository(ABC):
    @abstractmethod
    def create_product(self, product: Product) -> bool:
        pass
    @abstractmethod
    def create_customer(self, customer: Customer) -> bool:
        pass
```

```
    @abstractmethod
    def delete_product(self, product_id: int) -> bool:
        pass

    @abstractmethod
    def delete_customer(self, customer_id: int) -> bool:
        pass
    @abstractmethod
    def add_to_cart(self, customer: Customer, product: Product, quantity: int) ->
bool:
        pass

    @abstractmethod
    def remove_from_cart(self, customer: Customer, product: Product) -> bool:
        pass
    @abstractmethod
    def get_all_from_cart(self, customer: Customer) -> List[Product]:
        pass
    @abstractmethod
    def place_order(self, customer: Customer, products: List[Dict[Product, int]],
shipping_address: str) -> bool:
        pass

    @abstractmethod
    def get_orders_by_customer(self, customer_id: int) -> List[Dict[Product, int]]:
        Pass
```

OrderProcessorRepositoryImpl.py

- Implements all abstract methods declared in the interface.
- Uses SQL queries to interact with the MySQL database.
- Performs all CRUD operations on users, products, orders, and order_items.

CODE:

```
From dao.OrderProcessorRepository import OrderProcessorRepository

from entity.Customer import Customer

from entity.Product import Product

from util.db_connection import DBConnection

from exception.customer_not_found_exception import CustomerNotFoundException

from exception.product_not_found_exception import ProductNotFoundException

from exception.order_not_found_exception import OrderNotFoundException

class OrderProcessorRepositoryImpl(OrderProcessorRepository):

 def create_product(self, product: Product) -> bool:

      conn = None

      cursor = None

      success = False

      try:

         conn = DBConnection.get_connection()

         cursor = conn.cursor()

         query = "INSERT INTO products (name, price, description, stockQuantity) VALUES (%s, %s, %s, %s)"

         cursor.execute(query, (

            product.get_name(),

            product.get_price(),
```

```
            product.get_description(),
            product.get_stock_quantity()
        ))
        product.set_product_id(cursor.lastrowid)

        conn.commit()
        success = True
    except Exception as e:
        print(f"Error creating product: {e}")
    finally:
        if cursor:
            cursor.close()
        if conn:
            conn.close()
    return success
def create_customer(self, customer: Customer) -> bool:
    conn = None
    cursor = None
    success = False
    try:
        conn = DBConnection.get_connection()
        cursor = conn.cursor()
        query = "INSERT INTO customers (name, email, password) VALUES
(%s, %s, %s)"
        cursor.execute(query, (
            customer.get_name(),
            customer.get_email(),
            customer.get_password()
```

))

customer.set_customer_id(cursor.lastrowid)

```
            conn.commit()
            success = True
        except Exception as e:
            print(f"Error creating customer: {e}")
        finally:
            if cursor:
                cursor.close()
            if conn:
                conn.close()
        return success


    def delete_product(self, product_id: int) -> bool:
        conn = None
        cursor = None
        success = False
        try:
            conn = DBConnection.get_connection()
            cursor = conn.cursor()
            cursor.execute("SELECT * FROM products WHERE product_id = %s",
(product_id,))
            if cursor.fetchone() is None:
                raise ProductNotFoundException(f"Product with ID {product_id} not
found.")


            query = "DELETE FROM products WHERE product_id = %s"
            cursor.execute(query, (product_id,))
            conn.commit()
```

```
            success = cursor.rowcount > 0
        except ProductNotFoundException as e:
            print(e)
            raise
        except Exception as e:
            print(f"Error deleting product: {e}")
        finally:
            if cursor:
                cursor.close()
            if conn:
                conn.close()
        return success


    def delete_customer(self, customer_id: int) -> bool:
        conn = None
        cursor = None
        success = False
        try:
            conn = DBConnection.get_connection()
            cursor = conn.cursor()
            cursor.execute("SELECT * FROM customers WHERE customer_id = %s", (customer_id,))
            if cursor.fetchone() is None:
                raise CustomerNotFoundException(f"Customer with ID {customer_id} not found.")


            query = "DELETE FROM customers WHERE customer_id = %s"
```

```
            cursor.execute(query, (customer_id,))
            conn.commit()
            success = cursor.rowcount > 0
        except CustomerNotFoundException as e:
            print(e)
            raise
        except Exception as e:
            print(f"Error deleting customer: {e}")
        finally:
            if cursor:
                cursor.close()
            if conn:
                conn.close()
        return success


    def add_to_cart(self, customer: Customer, product: Product, quantity: int) ->
bool:
        conn = None
        cursor = None
        success = False
        try:
            conn = DBConnection.get_connection()
            cursor = conn.cursor()


            cursor.execute("SELECT * FROM customers WHERE customer_id =
%s", (customer.get_customer_id(),))
            if cursor.fetchone() is None:
```

```
            raise CustomerNotFoundException(f"Customer with ID {customer.get_customer_id()} not found.")


        cursor.execute("SELECT * FROM products WHERE product_id = %s", (product.get_product_id(),))
        if cursor.fetchone() is None:
            raise ProductNotFoundException(f"Product with ID {product.get_product_id()} not found.")


        query = "INSERT INTO cart (customer_id, product_id, quantity) VALUES (%s, %s, %s)"
        cursor.execute(query, (
            customer.get_customer_id(),
            product.get_product_id(),
            quantity
        ))
        conn.commit()
        success = True
    except (CustomerNotFoundException, ProductNotFoundException) as e:
        print(e)
        raise
    except Exception as e:
        print(f"Error adding to cart: {e}")
    finally:
        if cursor:
            cursor.close()
        if conn:
```

```
            conn.close()
        return success

    def remove_from_cart(self, customer: Customer, product: Product) -> bool:
        conn = None
        cursor = None
        success = False
        try:
            conn = DBConnection.get_connection()
            cursor = conn.cursor()

            cursor.execute("SELECT * FROM customers WHERE customer_id = %s", (customer.get_customer_id(),))
            if cursor.fetchone() is None:
                raise CustomerNotFoundException(f"Customer with ID {customer.get_customer_id()} not found.")

            cursor.execute("SELECT * FROM products WHERE product_id = %s", (product.get_product_id(),))
            if cursor.fetchone() is None:
                raise ProductNotFoundException(f"Product with ID {product.get_product_id()} not found.")

            query = "DELETE FROM cart WHERE customer_id = %s AND product_id = %s"
            cursor.execute(query, (
                customer.get_customer_id(),
                product.get_product_id()
```

```
            ))
            conn.commit()
            success = cursor.rowcount > 0
        except (CustomerNotFoundException, ProductNotFoundException) as e:
            print(e)
            raise
        except Exception as e:
            print(f"Error removing from cart: {e}")
        finally:
            if cursor:
                cursor.close()
            if conn:
                conn.close()
        return success


    def place_order(self, customer, cart_items: list, shipping_address: str) -> bool:
        conn = None
        cursor = None
        success = False
        try:
            customer_id = customer.get_customer_id() if isinstance(customer, Customer) else customer

            conn = DBConnection.get_connection()
            cursor = conn.cursor()
```

```
            total_price = sum(item["product"].get_price() * item["quantity"] for item
in cart_items)

            order_query = "INSERT INTO orders (customer_id, order_date,
total_price, shipping_address) VALUES (%s, NOW(), %s, %s)"
            cursor.execute(order_query, (customer_id, total_price, shipping_address))
            order_id = cursor.lastrowid

            for item in cart_items:
                product_id = item["product"].get_product_id()
                quantity = item["quantity"]
                order_item_query = "INSERT INTO order_items (order_id, product_id,
quantity) VALUES (%s, %s, %s)"
                cursor.execute(order_item_query, (order_id, product_id, quantity))

            conn.commit()
            success = True
        except Exception as e:
            print(f"Error placing order: {e}")
        finally:
            if cursor:
                cursor.close()
            if conn:
                conn.close()
        return success

    def get_orders_by_customer(self, customer_id: int):
```

```
    conn = None
    cursor = None
    orders = []
    try:
        conn = DBConnection.get_connection()
        cursor = conn.cursor()
        cursor.execute("SELECT * FROM customers WHERE customer_id = %s", (customer_id,))
        if cursor.fetchone() is None:
            raise CustomerNotFoundException(f"Customer with ID {customer_id} not found.")

        query = "SELECT order_id, order_date, total_price, shipping_address FROM orders WHERE customer_id = %s"
        cursor.execute(query, (customer_id,))
        orders = cursor.fetchall()

        if not orders:
            raise OrderNotFoundException(f"No orders found for Customer ID {customer_id}.")

    except (CustomerNotFoundException, OrderNotFoundException) as e:
        print(e)
        raise
    except Exception as e:
        print(f"Error retrieving orders: {e}")
    finally:
```

```
        if cursor:
            cursor.close()
        if conn:
            conn.close()
    return orders

def get_all_from_cart(self, customer_id: int) -> list:
    conn = None
    cursor = None
    cart_items = []
    try:
        if isinstance(customer_id, Customer):
            customer_id = customer_id.get_customer_id()

        conn = DBConnection.get_connection()
        cursor = conn.cursor(dictionary=True)
        query = """
            SELECT p.product_id, p.name, p.price, c.quantity
            FROM cart c
            JOIN products p ON c.product_id = p.product_id
            WHERE c.customer_id = %s
        """
        cursor.execute(query, (customer_id,))
        cart_items = cursor.fetchall()
    except Exception as e:
        print(f"Error retrieving cart items: {e}")
```

```
finally:
    if cursor:
        cursor.close()
    if conn:
        conn.close()
return cart_items
```

3.Exception Module (exception/):

Defines custom exceptions to handle specific error scenarios.

Custom Exceptions files:

- CustomerNotFoundException
- ProductNotFoundException
- OrderNotFoundException

Purpose: To manage errors gracefully and avoid system crashes with meaningful messages.

Files:

Customer_Not_Found_Exception.py:

```
class CustomerNotFoundException(Exception):
    def __init__(self, message="Customer not found in the database"):
        self.message = message
        super().__init__(self.message)
```

Order_Not_Found_Exception.py:

```
class OrderNotFoundException(Exception):
    def __init__(self, message="Order not found in the database"):
        self.message = message
        super().__init__(self.message)
```

Product_Not_Found_Exception.py:

```
class ProductNotFoundException(Exception):
    def __init__(self, message="Product not found in the database"):
```

```
        self.message = message
        super().__init__(self.message)
```

4.Utility Module (util/)

Provides reusable utility functions, especially for DB connection.

Files:

- PropertyUtil.py
  - Reads database connection details from db.properties
- DBConnection.py
  - Establishes DB connection using property values

Files:

DB_connection.py:

```
import mysql.connector
from util.property_util import PropertyUtil

class DBConnection:
  _connection = None

  @staticmethod
  def get_connection():
        try:
    if DBConnection._connection is None or not DBConnection._connection.is_connected():
            db_config = PropertyUtil.get_database_config()
            DBConnection._connection = mysql.connector.connect(
              host=db_config["host"],
              user=db_config["user"],
              password=db_config["password"],
```

```
                database=db_config["database"]
            )
            return DBConnection._connection
        except mysql.connector.Error as e:
            print(f"Error connecting to MySQL: {e}")
            return None

    @staticmethod
    def close_connection():
        if DBConnection._connection and DBConnection._connection.is_connected():
            DBConnection._connection.close()
            DBConnection._connection = None
```

Property_Util.py:

```
import configparser
import os
class PropertyUtil:
    @staticmethod
    def get_database_config():
        config = configparser.ConfigParser()

        # Absolute Path
        config_path = os.path.join(os.path.dirname(__file__), "../config.ini")

        if not os.path.exists(config_path):
            raise FileNotFoundError(f"Config file not found: {config_path}")

        config.read(config_path)

        return {
            "host": config.get("database", "host"),
            "user": config.get("database", "user"),
            "password": config.get("database", "password"),
            "database": config.get("database", "database")
        }
```

5.Main Application (app/):

Menu-driven interface to interact with the system.

File:

- EcomApp.py

Responsibilities:

- Prompt user with options like:
  - Register customer
  - Add/delete product
  - Add/remove/view cart
  - Place order
  - View orders
- Call corresponding methods from the DAO layer
- Handle exceptions

Files:

Main.py:

```
import sys
from dao.OrderProcessorRepositoryImpl import OrderProcessorRepositoryImpl
from entity.Customer import Customer
from entity.Product import Product
from exception.customer_not_found_exception import CustomerNotFoundException
```

```
from exception.product_not_found_exception import ProductNotFoundException
from exception.order_not_found_exception import OrderNotFoundException



class EcomApp:
    def __init__(self):
        self.order_repo = OrderProcessorRepositoryImpl()

    def menu(self):
        while True:
            print("\n===== E-Commerce System =====")
            print("1. Register Customer")
            print("2. Create Product")
            print("3. Delete Product")
            print("4. Delete Customer")
            print("5. Add to Cart")
            print("6. Remove from Cart")
            print("7. View Cart")
            print("8. Place Order")
            print("9. View Customer Order")
            print("10. Exit")

            choice = input("Enter your choice: ")

            if choice == "1":
```

```
            self.create_customer()
        elif choice == "2":
            self.create_product()
        elif choice == "3":
            self.delete_product()
        elif choice == "4":
            self.delete_customer()
        elif choice == "5":
            self.add_to_cart()
        elif choice == "6":
            self.remove_from_cart()
        elif choice == "7":
            self.view_cart()
        elif choice == "8":
            self.place_order()
        elif choice == "9":
            self.view_orders()
        elif choice == "10":
            print("Exiting... Goodbye!")
            sys.exit()
        else:
            print("Invalid choice! Please enter a number from 1 to 10.")

def create_product(self):
    name = input("Enter product name: ")
    price = float(input("Enter product price: "))
```

```
        description = input("Enter product description: ")
        stock_quantity = int(input("Enter stock quantity: "))
        product = Product(name=name, price=price, description=description, stock_quantity=stock_quantity)
        if self.order_repo.create_product(product):
            print("Product created successfully!")
        else:
            print("Failed to create product.")

    def create_customer(self):
        name = input("Enter customer name: ")
        email = input("Enter customer email: ")
        password = input("Enter password: ")
        customer = Customer(name=name, email=email, password=password)
        if self.order_repo.create_customer(customer):
            print("Customer created successfully!")
        else:
            print("Failed to create customer.")

    def delete_product(self):
        try:
            product_id = int(input("Enter product ID to delete: "))
            if self.order_repo.delete_product(product_id):
                print("Product deleted successfully!")
            else:
                print("Product not found.")
```

```
        except ProductNotFoundException:
            print("Error: Product not found.")

    def delete_customer(self):
        try:
            customer_id = int(input("Enter customer ID to delete: "))
            if self.order_repo.delete_customer(customer_id):
                print("Customer deleted successfully!")
            else:
                print("Customer not found.")
        except CustomerNotFoundException:
            print("Error: Customer not found.")

    def add_to_cart(self):
        try:
            customer_id = int(input("Enter customer ID: "))
            product_id = int(input("Enter product ID: "))
            quantity = int(input("Enter quantity: "))
            customer = Customer(customer_id=customer_id, name="", email="", password="")
            product = Product(product_id=product_id, name="", price=0, description="", stock_quantity=0)
            if self.order_repo.add_to_cart(customer, product, quantity):
                print("Product added to cart successfully!")
            else:
                print("Failed to add product to cart.")
        except ProductNotFoundException:
```

```
            print("Error: Product not found.")
        except CustomerNotFoundException:
            print("Error: Customer not found.")


    def remove_from_cart(self):
        try:
            customer_id = int(input("Enter customer ID: "))
            product_id = int(input("Enter product ID: "))
            customer = Customer(customer_id=customer_id, name="", email="", password="")
            product = Product(product_id=product_id, name="", price=0, description="", stock_quantity=0)
            if self.order_repo.remove_from_cart(customer, product):
                print("Product removed from cart successfully!")
            else:
                print("Product not found in cart.")
        except ProductNotFoundException:
            print("Error: Product not found.")
        except CustomerNotFoundException:
            print("Error: Customer not found.")


    def view_cart(self):
        try:
            customer_id = int(input("Enter customer ID: "))



            cart_items = self.order_repo.get_all_from_cart(customer_id)
```

```
            if cart_items:
                print("Cart Items:")
                for item in cart_items:
                    print(
                        f"Product ID: {item['product_id']}, Name: {item['name']}, Price: {item['price']}, Quantity: {item['quantity']}")
            else:
                print("Cart is empty.")
        except CustomerNotFoundException:
            print("Error: Customer not found.")


    def place_order(self):
        try:
            customer_id = int(input("Enter customer ID: "))
            shipping_address = input("Enter shipping address: ")



            cart_items = self.order_repo.get_all_from_cart(customer_id)


            if cart_items:
                order_items = [
                    {"product": Product(
                        product_id=item["product_id"],
                        name=item["name"],
                        price=item["price"],
```

```
                description="",
                stock_quantity=0
            ),
                "quantity": item["quantity"]}
            for item in cart_items
        ]

        if self.order_repo.place_order(customer_id, order_items,
                                       shipping_address):
            print("Order placed successfully!")
        else:
            print("Failed to place order.")
    else:
        print("Cart is empty.")
except OrderNotFoundException:
    print("Error: Unable to place order.")

def view_orders(self):
    try:
        customer_id = int(input("Enter customer ID: "))
        orders = self.order_repo.get_orders_by_customer(customer_id)
        if orders:
            print("Orders:")
            for order in orders:
                print(
```

```
                f"Order ID: {order[0]}, Date: {order[1]}, Total Price: {order[2]},
Shipping Address: {order[3]}")
        else:
            print("No orders found.")
    except CustomerNotFoundException:
        print("Error: Customer not found.")



if __name__== "__main__":
  app = EcomApp()
  app.menu()
```

2. Testing Module (tests/)

Unit tests to validate correctness and reliability.

File:

- test_order_processor.py

Test Cases Cover:

- Product creation
- Cart addition
- Order placement
- Exception handling when customer/product not found

Test_order_processor.py:

```
import unittest
from dao.OrderProcessorRepositoryImpl import OrderProcessorRepositoryImpl
from entity.Product import Product
```

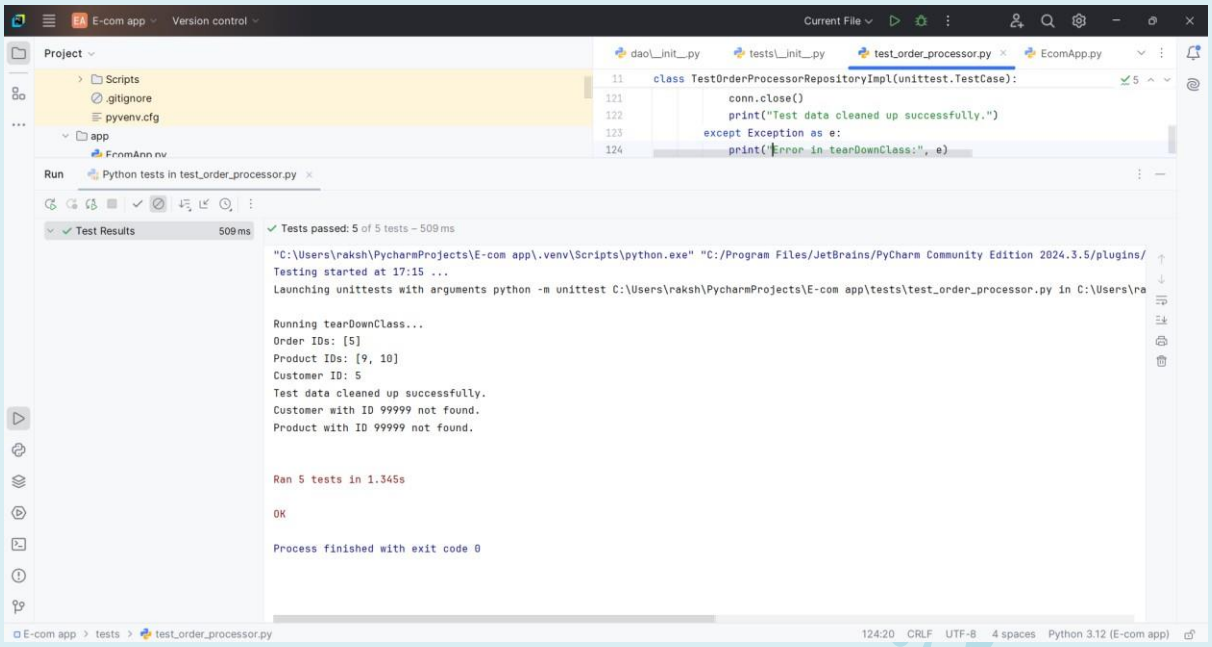
```
from entity.Customer import Customer
from exception.customer_not_found_exception import CustomerNotFoundException
from exception.product_not_found_exception import ProductNotFoundException
import random
import mysql.connector
class TestOrderProcessorRepositoryImpl(unittest.TestCase):
  @classmethod
  def setUpClass(cls):
    cls.repo = OrderProcessorRepositoryImpl()
    cls.customer = Customer(
      name="UnitTest User",
      email=f"testuser_{random.randint(1000, 9999)}@test.com",
      password="test123"
    )
    cls.repo.create_customer(cls.customer)
    cls.test_customer_id = cls.customer.get_customer_id()


    cls.product = Product(
      name="Test Product",
      price=99.99,
      description="Unit test product",
      stock_quantity=50
    )
    cls.repo.create_product(cls.product)
```

```
    cls.test_product_ids = [cls.product.get_product_id()]
    cls.test_order_ids = []

def test_1_create_product_success(self):
    new_product = Product(
        name="New Test Product",
        price=49.99,
        description="Another test product",
        stock_quantity=30
    )
    result = self.repo.create_product(new_product)
    self.assertTrue(result)
    self.assertIsNotNone(new_product.get_product_id())



    type(self).test_product_ids.append(new_product.get_product_id())

def test_2_add_to_cart_success(self):
    result = self.repo.add_to_cart(self.customer, self.product, quantity=2)
    self.assertTrue(result)

def test_3_place_order_success(self):
    cart_items = [{"product": self.product, "quantity": 2}]
    result = self.repo.place_order(self.customer, cart_items, "123 Test Lane")
    self.assertTrue(result)
```

```
        conn = mysql.connector.connect(
            host='localhost',
            user='root',
            password='Jash@2512',
            database='ecomm_db'
        )
        cursor = conn.cursor()
        cursor.execute("SELECT MAX(order_id) FROM orders WHERE customer_id = %s", (self.test_customer_id,))
        order_id = cursor.fetchone()[0]
        if order_id:
            type(self).test_order_ids.append(order_id)
        cursor.close()
        conn.close()


    def test_4_customer_not_found_exception(self):
        fake_customer = Customer(name="Fake", email="fake@test.com", password="fake123")
        fake_customer.set_customer_id(99999)
        with self.assertRaises(CustomerNotFoundException):
            self.repo.add_to_cart(fake_customer, self.product, quantity=1)


    def test_5_product_not_found_exception(self):
        fake_product = Product(name="Ghost", price=0.0, description="Ghost", stock_quantity=0)
        fake_product.set_product_id(99999)
```

```
        with self.assertRaises(ProductNotFoundException):
            self.repo.add_to_cart(self.customer, fake_product, quantity=1)


    @classmethod
    def tearDownClass(cls):
        print("Running tearDownClass...")
        print("Order IDs:", cls.test_order_ids)
        print("Product IDs:", cls.test_product_ids)
        print("Customer ID:", cls.test_customer_id)


        try:
            conn = mysql.connector.connect(
                host='localhost',
                user='root',
                password='Jash@2512',
                database='ecomm_db'
            )
            cursor = conn.cursor()



            for order_id in cls.test_order_ids:
                cursor.execute("DELETE FROM order_items WHERE order_id = %s",
(order_id,))
                cursor.execute("DELETE FROM orders WHERE order_id = %s",
(order_id,))
```

```
            for product_id in cls.test_product_ids:
                cursor.execute("DELETE FROM cart WHERE customer_id = %s AND
product_id = %s",
                               (cls.test_customer_id, product_id))



            for product_id in cls.test_product_ids:
                cursor.execute("DELETE FROM products WHERE product_id = %s",
(product_id,))



            cursor.execute("DELETE FROM customers WHERE customer_id = %s",
(cls.test_customer_id,))


            conn.commit()
            cursor.close()
            conn.close()
            print("Test data cleaned up successfully.")
        except Exception as e:
            print(“ Error in tearDownClass:", e)


if __name__== "__main__":
    unittest.main()
```

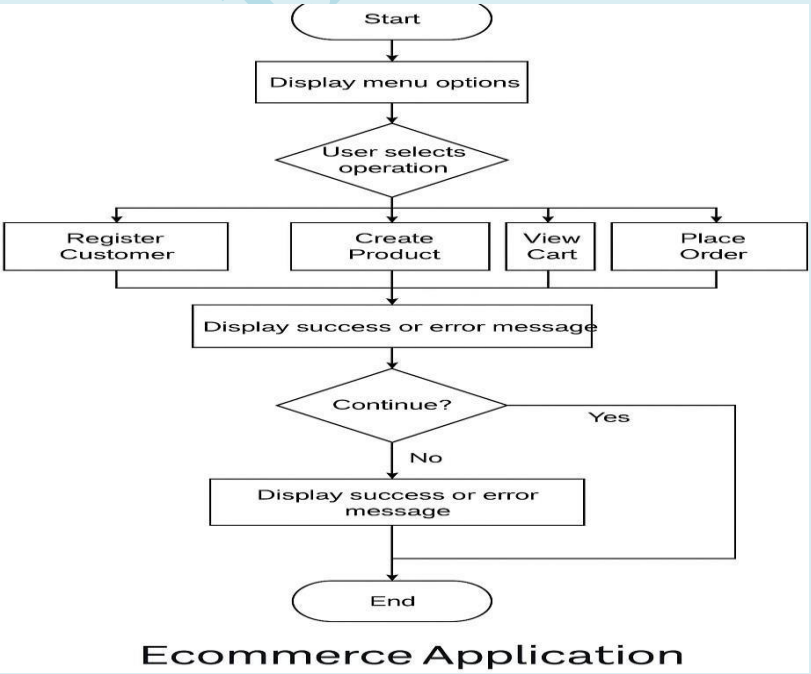


5. Technologies Used

- Programming Language- Python
- Database- **MySQL Workbench**.
- Database Connectivity- MySQL Connector/Python
- Software Development Concepts- Object-Oriented Programming (OOP) and Exception Handling
- Testing Framework- unittest (Python Standard Library)
- IDE / Tools- PyCharm

6. Flow Chart:

A flowchart is a graphical representation of the logical flow of processes or steps in a system. It uses symbols like ovals (start/end), rectangles (processes), diamonds (decisions), and arrows (flow direction) to show how data or control moves through the system.



Ecommerce Application

7.Output:

Create class named EcomApp with main method in app Trigger all the methods in service implementation class by user choose operation from the following menu.

1. Register Customer.

2. Create Product.

3. Delete Product.

4. Delete customer

5. Add to cart.

6. Delete cart

7. View cart.

8. Place order.

9. View Customer Order

10. Exit

1. Register Customer.

```
Enter your choice: 1
Enter customer name: jash
Enter customer email: jash@gmail.com
Enter password: jash
Customer created successfully!
```

	customer_id	name	email	password
▶	3	jash	jash@gmail.com	jash
*	NULL	NULL	NULL	NULL

2. Create Product.

```
Enter your choice: 2
Enter product name: choco bites
Enter product price: 10
Enter product description: tasty
Enter stock quantity: 100
Product created successfully!
```

	product_id	name	price	description	stockQuantity
▶	5	choco bites	10.00	tasty	100
*	NULL	NULL	NULL	NULL	NULL

3.delete product

product_id	name	price	description	stockQuantity
5	choco bites	10.00	tasty	100
6	milky bar	10.00	divine	100
7	jimjam	5.00	biscuit	50
NULL	NULL	NULL	NULL	NULL

```
Enter your choice: 3
Enter product ID to delete: 7
Product deleted successfully!
```

product_id	name	price	description	stockQuantity
5	choco bites	10.00	tasty	100
6	milky bar	10.00	divine	100
NULL	NULL	NULL	NULL	NULL

4. delete customer

customer_id	name	email	password
3	jash	jash@gmail.com	jash
4	vara	vara@gmail.com	vara
5	abc	abc@gamil.com	abc
NULL	NULL	NULL	NULL

```
Enter your choice: 4
Enter customer ID to delete: 5
Customer deleted successfully!
```

customer_id	name	email	password
3	jash	jash@gmail.com	jash
4	vara	vara@gmail.com	vara
NULL	NULL	NULL	NULL

5. Add to cart

```
Enter your choice: 5
Enter customer ID: 3
Enter product ID: 6
Enter quantity: 100
Product added to cart successfully!
```

	cart_id	customer_id	product_id	quantity
▶	3	3	6	100
*	NULL	NULL	NULL	NULL

6. Delete from cart

```
Enter your choice: 6
Enter customer ID: 3
Enter product ID: 5
Product removed from cart successfully!
```

7. View cart

```
Enter your choice: 7
Enter customer ID: 3
Cart Items:
Product ID: 6, Name: milky bar, Price: 10.00, Quantity: 100
```

8. Place order

```
Enter your choice: 8
Enter customer ID: 3
Enter shipping address: jash heart
Order placed successfully!
```

order_id	customer_id	order_date	total_price	shipping_address
3	3	2025-04-11 01:35:53	1000.00	jash heart
NULL	NULL	NULL	NULL	NULL

9. View cart

```
Enter your choice: 9
Enter customer ID: 3
Orders:
Order ID: 3, Date: 2025-04-11 01:35:53, Total Price: 1000.00, Shipping Address: jash heart
```

10. EXIT:

```
===== E-Commerce System =====
1. Register Customer
2. Create Product
3. Delete Product
4. Delete Customer
5. Add to Cart
6. Remove from Cart
7. View Cart
8. Place Order
9. View Customer Order
10. Exit
Enter your choice: 10
Exiting... Goodbye!
```

11. Future Enhancements

The Ecommerce Application has vast potential for future enhancements that can significantly improve user experience, business efficiency, and scalability. One of the key areas for improvement is the shopping cart system, which can be enhanced to support session persistence, discount coupons, and cart expiration mechanisms. In terms of security, implementing advanced user authentication with encrypted passwords, OTP verification, and role-based access control would add a strong layer of protection. Integrating a payment gateway such as Razorpay or Stripe would allow secure and seamless online transactions. Additionally, a shipping and logistics module could be introduced to track delivery status and sync with third-party couriers.

Another enhancement would be developing a product recommendation engine using machine learning to offer personalized suggestions to users based on their browsing and purchase history. For admins, an interactive dashboard could be created to provide insights into sales, revenue, inventory, and customer behavior through visual reports. Customers could also be given the ability to rate and review products, enhancing transparency and trust. Supporting multiple languages and currencies would allow the application to serve a global user base, and creating a mobile app would provide more accessibility to customers on smartphones.

12. Conclusion

The Ecommerce Application Project successfully demonstrates the development of a fully functional backend system using core principles of object-oriented programming, MySQL database integration, exception handling, and unit testing. Through the implementation of modules such as customer management, product management, cart operations, and order processing, this project replicates the foundational operations of a real-world ecommerce platform. It effectively showcases the interaction between frontend input, backend logic, and persistent data storage.By following a modular and layered architecture — including the use of entity classes, DAO interfaces, utility handlers, and custom exceptions — the project ensures scalability, maintainability, and code reusability. The use of SQL for structured data management, along with robust error handling and unit testing, enhances the reliability and stability of the application.In conclusion, this project not only meets academic and practical requirements but also lays a strong foundation for real-time ecommerce solutions. It can be further extended with advanced features like payment integration, analytics, and enhanced security to make it production-ready.

End of the document