### **Coding Skills Assessment**

#### **Instructions:**

* You are required to complete three parts of this assignment: **System Design**, **Business Logic Implementation**, and **Database Query Writing**.
* You may use any programming language and database management system you are comfortable with.
* Ensure that your code is modular, scalable, and follows clean coding practices.

### **Part 1: System Design**

**Problem Statement:**Design a simplified **e-commerce system** that handles users, products, orders, and payments.

#### **Requirements:**

* The system should support multiple users with the ability to create, view, and manage orders.
* Each order can contain multiple products.
* A payment can be made for each order, and an order can have different statuses (e.g., pending, completed, shipped).

#### **Deliverables:**

1. **Class Diagram** that outlines the relationships between User, Product, Order, and Payment.
2. Write code stubs for each of the main components, ensuring that relationships (e.g., Order contains multiple Products) are appropriately handled.

**Solution:** **Class Diagram**

+--------------------+ +--------------------+ +--------------------+ +--------------------+

| User | | Product | | Order | | Payment |

+--------------------+ +--------------------+ +--------------------+ +--------------------+

| - user\_id | | - product\_id | | - order\_id | | - payment\_id |

| - name | | - name | | - user\_id | | - order\_id |

| - email | | - price | | - products[] | | - amount |

| | | | | - status | | - payment\_date |

| + placeOrder() | | + viewProduct() | | + addProduct() | | + processPayment() |

+--------------------+ +--------------------+ +--------------------+ +--------------------+

# Code in python

class User:

def \_\_init\_\_(self, user\_id, name, email):

self.user\_id = user\_id

self.name = name

self.email = email

self.orders = []

def place\_order(self, order):

"""Method for placing an order"""

self.orders.append(order)

print(f"Order {order.order\_id} placed by {self.name}")

def view\_orders(self):

"""Method for viewing orders placed by the user"""

for order in self.orders:

print(f"Order ID: {order.order\_id}, Status: {order.status}")

class Product:

def \_\_init\_\_(self, product\_id, name, price):

self.product\_id = product\_id

self.name = name

self.price = price

def view\_product(self):

"""Display product details"""

print(f"Product ID: {self.product\_id}, Name: {self.name}, Price: ${self.price}")

class Order:

def \_\_init\_\_(self, order\_id, user\_id):

self.order\_id = order\_id

self.user\_id = user\_id

self.products = []

self.status = "Pending"

def add\_product(self, product):

"""Add a product to the order"""

self.products.append(product)

print(f"Product {product.name} added to Order {self.order\_id}")

def view\_order(self):

"""View the order details and its products"""

print(f"Order ID: {self.order\_id}, Status: {self.status}")

for product in self.products:

product.view\_product()

class Payment:

def \_\_init\_\_(self, payment\_id, order\_id, amount):

self.payment\_id = payment\_id

self.order\_id = order\_id

self.amount = amount

self.payment\_date = None

def process\_payment(self):

"""Process the payment for the given order"""

print(f"Payment of ${self.amount} processed for Order {self.order\_id}")

# Example

# Create users

user1 = User(user\_id=1, name="Alice", email="alice@example.com")

user2 = User(user\_id=2, name="Bob", email="bob@example.com")

# Create products

product1 = Product(product\_id=101, name="Laptop", price=1000)

product2 = Product(product\_id=102, name="Phone", price=500)

# User places an order

order1 = Order(order\_id=201, user\_id=user1.user\_id)

order1.add\_product(product1)

order1.add\_product(product2)

# User places another order

order2 = Order(order\_id=202, user\_id=user2.user\_id)

order2.add\_product(product1)

# View orders

user1.place\_order(order1)

user1.view\_orders()

# Process payment for order

payment1 = Payment(payment\_id=301, order\_id=order1.order\_id, amount=1500)

payment1.process\_payment()

# Order Status Update

order1.status = "Shipped"

user1.view\_orders()

### Explanation:

* **User** class: Contains methods to place and view orders.
* **Product** class: Defines products and allows viewing their details.
* **Order** class: Handles adding products to an order and viewing order details.
* **Payment** class: Processes payments related to an order.

### **Part 2: Business Logic Implementation**

**Problem Statement:**You are tasked with implementing an **inventory management system** for a warehouse. The system should be able to track stock levels and manage restocking.

#### **Requirements:**

1. Implement a function that:
   * Takes a list of products with their current stock levels and a list of incoming sales orders.
   * Reduces the stock levels based on the orders.
   * If the stock level of any product drops below a certain threshold (e.g., 10 units), an alert should be triggered to restock the item.
2. Implement a function to **restock** items. The function should:
   * Take a list of products that need restocking and their required quantities.
   * Update the stock levels accordingly.

#### **Deliverables:**

* Provide the code implementation for the two functions: process\_orders() and restock\_items().
* Ensure error handling is in place for invalid input (e.g., trying to process an order when the product is out of stock).

**Solution:**

**Code:**

class Product:

def \_\_init\_\_(self, product\_id, name, stock\_level, threshold=10):

self.product\_id = product\_id

self.name = name

self.stock\_level = stock\_level

self.threshold = threshold

def \_\_str\_\_(self):

return f"Product ID: {self.product\_id}, Name: {self.name}, Stock: {self.stock\_level}"

def process\_orders(products, orders):

"""

Takes a list of products and a list of orders, reduces stock levels based on the orders,

and triggers a restocking alert if stock level drops below the threshold.

:param products: List of Product objects representing the current stock

:param orders: Dictionary where keys are product IDs and values are the quantity to be ordered

"""

for product\_id, order\_quantity in orders.items():

product = next((p for p in products if p.product\_id == product\_id), None)

if product is None:

print(f"Error: Product ID {product\_id} not found.")

continue

if product.stock\_level < order\_quantity:

print(f"Error: Insufficient stock for product {product.name}. Available: {product.stock\_level}, Requested: {order\_quantity}")

continue

product.stock\_level -= order\_quantity

print(f"Order processed for {product.name}. New stock level: {product.stock\_level}")

if product.stock\_level < product.threshold:

print(f"Alert: Stock for {product.name} has dropped below the threshold. Current stock: {product.stock\_level}. Restocking needed.")

def restock\_items(products, restock\_list):

"""

Takes a list of products that need restocking and their required quantities,

updates the stock levels accordingly.

:param products: List of Product objects representing the current stock

:param restock\_list: Dictionary where keys are product IDs and values are the quantity to be restocked

"""

for product\_id, restock\_quantity in restock\_list.items():

product = next((p for p in products if p.product\_id == product\_id), None)

if product is None:

print(f"Error: Product ID {product\_id} not found.")

continue

product.stock\_level += restock\_quantity

print(f"Product {product.name} restocked. New stock level: {product.stock\_level}")

# Example :

# Initialize products

products = [

Product(product\_id=1, name="Laptop", stock\_level=15),

Product(product\_id=2, name="Phone", stock\_level=8),

Product(product\_id=3, name="Tablet", stock\_level=20),

]

# Incoming sales orders (Product ID and Quantity Ordered)

orders = {

1: 5, # Order 5 Laptops

2: 3, # Order 3 Phones

3: 15, # Order 15 Tablets

4: 2 # Invalid Product ID

}

# Process the orders

process\_orders(products, orders)

# Restocking items (Product ID and Quantity to Restock)

restock\_list = {

1: 10, # Restock 10 Laptops

2: 15 # Restock 15 Phones

}

# Restock items

restock\_items(products, restock\_list)

### **Explanation:**

1. **Product Class**:
   * Each product has an ID, name, stock level, and a restock threshold.
   * The threshold is used to trigger restock alerts when stock drops below the specified level.
2. **process\_orders() Function**:
   * Takes a list of products and a dictionary of incoming sales orders.
   * Reduces stock based on the order quantities.
   * If stock drops below the threshold, an alert is printed to notify that restocking is required.
   * Handles errors such as:
     + Product ID not found.
     + Insufficient stock for the requested order.
3. **restock\_items() Function**:
   * Takes a list of products and a dictionary containing product IDs and quantities for restocking.
   * Updates the stock levels accordingly.
   * Handles errors such as:
     + Product ID not found.

### **Error Handling:**

* **Invalid Product ID**: If an order or restock involves an invalid product ID, an error is printed and processing continues.
* **Insufficient Stock**: If a product does not have enough stock to fulfill an order, an error is printed.

### **Part 3: Database Query Handling**

**Problem Statement:**You are given a relational database schema for an online bookstore with the following tables:

**Tables:**

Customers (customer\_id, name, email)  
Books (book\_id, title, author, price)  
Orders (order\_id, customer\_id, order\_date)  
OrderDetails (order\_id, book\_id, quantity)

#### **Requirements:**

1. Write a SQL query to retrieve the top 5 customers who have purchased the most books (by total quantity) over the last year.
2. Write a SQL query to calculate the total revenue generated from book sales by each author.
3. Write a SQL query to retrieve all books that have been ordered more than 10 times, along with the total quantity ordered for each book.

#### **Deliverables:**

* Provide the SQL queries for the three requirements.
* Ensure that the queries are optimized for performance, considering indexing where necessary.

**Solution:**

Code:

1.

SELECT c.customer\_id, c.name, SUM(od.quantity) AS total\_books\_purchased

FROM Customers c

JOIN Orders o ON c.customer\_id = o.customer\_id

JOIN OrderDetails od ON o.order\_id = od.order\_id

WHERE o.order\_date >= DATE\_SUB(CURDATE(), INTERVAL 1 YEAR)

GROUP BY c.customer\_id, c.name

ORDER BY total\_books\_purchased DESC

LIMIT 5;

#### Explanation:

* **Join**: We use JOIN to link the Customers, Orders, and OrderDetails tables.
* **Filter**: The WHERE clause ensures we only consider orders from the last year using DATE\_SUB(CURDATE(), INTERVAL 1 YEAR).
* **Aggregation**: The SUM(od.quantity) calculates the total number of books purchased by each customer.
* **Sorting and Limiting**: We sort the results in descending order by total\_books\_purchased and limit it to the top 5 customers.

#### Optimization Considerations:

* Ensure indexes on customer\_id in Customers, Orders, and OrderDetails for fast joins.
* Index on order\_date in Orders for faster filtering by date.

2.

SELECT b.author, SUM(od.quantity \* b.price) AS total\_revenue

FROM Books b

JOIN OrderDetails od ON b.book\_id = od.book\_id

GROUP BY b.author

ORDER BY total\_revenue DESC;

#### Explanation:

* **Join**: We join the Books table with OrderDetails to link book sales data.
* **Revenue Calculation**: The total revenue is calculated by multiplying the quantity of books sold by the price of each book (SUM(od.quantity \* b.price)).
* **Grouping**: We group by b.author to aggregate revenue per author.
* **Sorting**: The results are ordered by total revenue in descending order.

#### Optimization Considerations:

* Ensure indexes on book\_id in Books and OrderDetails to speed up the join.

3.

SELECT b.book\_id, b.title, SUM(od.quantity) AS total\_quantity\_ordered

FROM Books b

JOIN OrderDetails od ON b.book\_id = od.book\_id

GROUP BY b.book\_id, b.title

HAVING SUM(od.quantity) > 10

ORDER BY total\_quantity\_ordered DESC;

#### Explanation:

* **Join**: We join the Books and OrderDetails tables to link each book to its sales details.
* **Aggregation**: The SUM(od.quantity) calculates the total quantity of each book ordered.
* **Filter**: The HAVING clause ensures only books with a total quantity ordered greater than 10 are retrieved.
* **Sorting**: Results are ordered by total quantity ordered in descending order.

#### Optimization Considerations:

* Index on book\_id in both Books and OrderDetails tables to optimize the join and aggregation performance.

### Indexing Strategy:

To optimize these queries, it is essential to have the following indexes:

1. **Customers Table**: Index on customer\_id.
2. **Books Table**: Index on book\_id and author.
3. **Orders Table**: Index on customer\_id and order\_date.
4. **OrderDetails Table**: Index on order\_id, book\_id, and quantity.

### **Submission Guidelines:**

* Submit your code and any necessary files (diagrams, database scripts) in a single archive.
* Include a **README** explaining your solution, how to run it, and any assumptions you have made.

#### **Evaluation Criteria:**

* **System Design**: Clarity, scalability, and correctness of the design.
* **Business Logic**: Efficiency, correctness, and robustness of the logic.
* **Database Query**: Correctness, performance, and understanding of SQL.
* **Clean Code**: Adherence to coding best practices, readability, and maintainability.