Optimizing Inventory and Order Tracking: A Scalable eCommerce Solution

Introduction:

The **Online Shopping Platform – Customer Order and Inventory Management System** is designed to address critical inefficiencies in order tracking, inventory management, and reporting within eCommerce operations. In today's competitive retail environment, ensuring seamless inventory updates, accurate reporting, and real-time customer notifications is essential to maintain customer satisfaction and operational efficiency.

This project aims to replace the existing system with an enhanced, integrated solution that leverages automation and advanced reporting capabilities. The new system will streamline inventory management, eliminate manual processes, and provide real-time data insights. By integrating robust SQL-based reporting, automated notifications, and real-time inventory synchronization, this project seeks to reduce order cancellations, improve stock availability, and enhance overall customer experience.

With a focus on scalability, performance, and security, the proposed solution will support high transaction volumes and ensure data integrity. By addressing current gaps and enabling seamless integration with existing systems, this project will empower stakeholders with actionable insights and efficient workflows, fostering growth and customer loyalty.

.

BUSINESS REQUIREMENT DOCUMENT:

1.Project Overview

Objective : To develop an enhanced Customer Order and Inventory Management System that streamlines order tracking, automates inventory management, and provides real-time reporting to improve overall customer experience and operational efficiency.

Justification : The current system's inefficiencies, including delayed inventory updates and limited reporting, lead to customer dissatisfaction and revenue loss. A new system will address these issues by providing real-time data and automation.

2. Business Problem

- Customers can place orders for out-of-stock items, resulting in cancellations.
- Inventory mismatches and delayed updates lead to stockouts.
- Limited reporting capabilities hinder business insights.

3. AS-IS vs. TO-BE

Aspect	AS-IS (Current State)	TO-BE (Future State)
Inventory Management	Manual data entry for inventory updates.	Automated real-time inventory updates to ensure accurate stock levels and eliminate manual errors.
Order Processing	Delayed stock updates lead to order cancellations and customer dissatisfaction.	Real-time stock updates and validation prevent customers from ordering out-of-stock items.

Reporting	Limited SQL-based reporting capabilities.	Comprehensive SQL-based reporting for orders, returns, refunds, sales, and inventory in real time.
Notifications	No real-time notifications for stock replenishment or order status updates.	Proactive notifications for low stock levels and real-time updates on order status (Processing, Shipped).
Integration	Standalone systems for inventory and order management.	Seamless integration between IMS and OMS, enabling synchronized updates and notifications.
Customer Experience	Customers place orders for out-of-stock items, leading to cancellations and frustration.	Customers receive accurate product availability and real-time notifications, enhancing satisfaction.
System Scalability	Limited capability to handle high transaction volumes.	Scalable system supporting up to 500,000 concurrent users with robust performance.
Data Integrity	High risk of errors due to manual processes and lack of automation.	Improved data integrity with SQL triggers and real-time synchronization.
Performance	SQL queries are slow and inefficient for large datasets.	Optimized SQL queries return results in under 3 seconds for datasets up to 1 million rows.
Availability	System downtime impacts operations and customer service.	99.9% uptime ensures 24/7 availability and smooth operations.

4. Project Scope:

- Inventory automation in real-time.
- Real-time status updates for orders.
- SQL-based comprehensive reporting.
- Stock-level notifications.

Out of Scope

- Changes to payment gateway systems.
- Development of new features for the eCommerce storefront.
- Integration with third-party logistics systems.

5. Stakeholders

- Product Managers
- Inventory Managers
- Customer Support Team
- Developers
- Operations Team
- Customers

6. Business Requirements

- Automate real-time inventory updates.
- Provide real-time customer notifications for order status changes.
- Enable inventory managers to generate SQL-based reports for stock levels, sales, returns, and refunds.
- Notify product managers about low stock levels for timely reorders.

7. Functional Requirements

- Inventory Management: Automatically update stock levels when orders are placed or products are received.
- Order Tracking: Real-time status updates and customer notifications for each stage of the order.
- Reporting: Allow authorized users to generate SQL-based reports.
- Notifications: Trigger notifications for low stock and order updates.

8. Non-Functional Requirements

- Performance: SQL queries should return results in under 3 seconds for datasets up to 1 million rows.
- Scalability: Handle up to 500,000 concurrent users.
- Security: Encrypt all data in transit and at rest; restrict access based on user roles.
- Availability: Ensure 99.9% uptime and 24/7 system availability.

9**. API Requirements**

- Inventory Update API: Update stock levels in real-time.
- Order Status API: Fetch real-time order updates.
- Notification API: Trigger notifications for stock alerts and order updates.

10. Integration Requirements

- Integration with the existing Inventory Management System (IMS).
- Integration with the existing Order Management System (OMS).
- Integration with email/SMS services for customer notifications.

11. Database Requirements

- Orders Table: Tracks orders with details like OrderID, CustomerID, OrderDate, OrderStatus, and TotalAmount.
- Inventory Table: Maintains inventory details such as ProductID, ProductName, QuantityInStock, and ReorderLevel.
- Returns Table: Captures returns data, including ReturnID, OrderID, ReturnDate, and RefundAmount.

12. Transition Requirements

- Data Migration: Ensure all existing data is migrated to the new system accurately.
- Testing: Conduct thorough system testing (unit, integration, and UAT).
- Training: Train stakeholders on how to use the new system and generate reports.

13. Risks and Mitigation

- Data Integrity Risk: Validate migrated data.
- System Downtime Risk: Transition during non-peak hours with a rollback strategy.

14. Assumptions

- The existing inventory and order data is accurate.
- Stakeholders will be available for requirements gathering and testing

15. RACI Matrix:

Role	Responsible	Accountable	Consulted	Informed
Business Analyst	Requirements Gathering	Product Manager	Developers	Customer Support
Product Manager	Prioritize Features	CEO	Business Analyst	Customers
Developers	Implementation	Lead Developer	Business Analyst	QA Team
Customer Support	Support Integration	Support Manager	Business Analyst	Product Managers

Software Requirements Specification (SRS) Document

1. Introduction

1.1 Purpose

The purpose of this document is to detail the requirements for the development of the Customer Order and Inventory Management System for the online shopping platform. The system will streamline order processing, automate inventory management, and provide real-time reporting, improving customer satisfaction and operational efficiency.

1.2 Scope

This SRS outlines the functional and non-functional requirements, system architecture, and constraints necessary to develop and implement the new system. The system will:

- Integrate with existing OMS and IMS.
- Provide SQL-based reporting.
- Send real-time notifications for order status and low stock levels.

1.3 Definitions, Acronyms, and Abbreviations

- **SQL**: Structured Query Language
- **OMS**: Order Management System
- **IMS**: Inventory Management System
- **API**: Application Programming Interface
- **UAT**: User Acceptance Testing

1.4 References

- Business Requirement Document (BRD) for this project.
- Existing OMS and IMS documentation.

1.5 Overview

This document is organized to cover functional and non-functional requirements, system interfaces, architecture, database design, and transition requirements. It also addresses risks, mitigation, and assumptions.

2. System Overview

The Customer Order and Inventory Management System will:

1. Automate inventory updates for order placement and stock reception.
2. Provide real-time reporting using SQL queries.
3. Seamlessly integrate with existing OMS and IMS systems.
4. Trigger notifications for low stock and order status changes.

3. Functional Requirements

3.1 Inventory Management

Description:
The system must:

- Update inventory levels automatically when an order is placed or when stock is received.
- Trigger low stock notifications when the QuantityInStock falls below the ReorderLevel.

Use Case:

- **Actors**: Inventory Managers, System Admin
- **Precondition**: Stock levels exist in the database.
- **Postcondition**: Updated stock levels and low stock alerts.

SQL Example:

```
UPDATE Inventory
SET QuantityInStock = QuantityInStock - 1
WHERE ProductID = 101 AND QuantityInStock > 0;
```

3.2 Order Management

The system must:

- Update order statuses in real time (e.g., Processing, Shipped, Delivered).
- Prevent orders for out-of-stock products.
- Notify customers of status changes via email/SMS

Use Case:

- **Actors**: Customer, Order Manager, System
- **Precondition**: Product is in stock.
- **Postcondition**: Updated order status and notifications sent.

SQL Example:

```
UPDATE Orders
SET OrderStatus = 'Shipped'
WHERE OrderID = 202;
```

3.3 Reporting (SQL-Based Reports)
The system must allow users to generate reports on:

1. Orders and sales.
2. Returns and refunds.
3. Stock levels.

Use Case:

- **Actors**: Product Manager, Inventory Manager, Admin
- **Precondition**: Data is up to date in the system.
- **Postcondition**: SQL query returns requested data.

SQL Example:

```
SELECT MONTH(OrderDate) AS OrderMonth,
       SUM(TotalAmount) AS TotalSales
FROM Orders
WHERE OrderStatus = 'Delivered'
GROUP BY MONTH(OrderDate);
```

3.4 Notifications
The system must:

- Notify inventory managers when stock falls below the reorder threshold.
- Send customers order status updates via email/SMS.

Use Case:

- **Actors**: Inventory Managers, Customers
- **Precondition**: Stock or order data exists in the system.
- **Postcondition**: Notifications are sent automatically.

4. Non-Functional Requirements

1. **Performance**: SQL queries must execute within 3 seconds for up to 1 million rows.
2. **Scalability**: The system must support up to 500,000 concurrent users.
3. **Security**: All data must be encrypted at rest and in transit.
4. **Availability**: Ensure 99.9% uptime and 24/7 availability.

5. System Architecture

5.1 High-Level Architecture

- Relational database for inventory, orders, and reporting.
- SQL for stock updates, order tracking, and reporting.
- API integration for seamless communication with OMS and IMS.

5.2 Database Schema

1. **Orders Table**:
    - **Columns**: OrderID, CustomerID, OrderDate, OrderStatus, TotalAmount.
2. **Inventory Table**:
    - **Columns**: ProductID, ProductName, QuantityInStock, ReorderLevel.
3. **Returns Table**:
    - **Columns**: ReturnID, OrderID, ReturnDate, RefundAmount.

5.3 API Architecture

1. Inventory Update API: Updates inventory in real time.
2. Order Status API: Fetches real-time order status.
3. Notification API: Sends low stock and order status updates.

6. Database Requirements

1. **Design**:

    - Use SQL Server or MySQL.
    - Define primary and foreign keys for relationships.
2. **Data Integrity**:
    - Use SQL triggers for automatic stock updates.
3. **Indexing**:
    - Index columns like OrderDate and ProductID for query optimization.

SQL Example:

```
CREATE INDEX idx_orderdate ON Orders (OrderDate)
```

7. Transition Requirements

1. **Data Migration**: Migrate legacy system data accurately.
2. **Testing**: Perform unit testing, integration testing, and UAT.
3. **Training**: Train stakeholders on new system features and SQL reporting.

8. Risks and Mitigation

1. **Data Integrity Risk**: Validate all migrated data and run parallel operations.
2. **Performance Risk**: Optimize SQL queries and implement indexing.

9. Assumptions and Constraints

1. **Assumptions**:
    - Legacy data is accurate and clean.
    - Stakeholders will provide feedback during the project.
2. **Constraints**:
    - The project must operate within existing budget and infrastructure.
    - No changes to payment gateway integrations.

GAP Analysis:

Aspect	Current State (AS-IS)	Future State (TO-BE)	Gap Identified	Steps to Cover the Gap (using SQL)
Inventory Management	Inventory updates are manually performed and often delayed, leading to inaccurate stock levels and stockouts.	Real-time automated inventory updates ensure accurate stock levels are always visible to both customers and staff.	Manual processes create delays and inaccuracies, leading to stockouts and poor customer experience.	1. Implement real-time SQL queries to update stock levels. 2. Automate inventory updates using SQL UPDATE queries. 3. Use SQL TRIGGERS to ensure stock updates upon transactions.
SQL Example				UPDATE Inventory SET QuantityInStock = QuantityInStock - 1 WHERE ProductID = 1;
Order Status Tracking	Order statuses are inconsistent and delayed, leading to customer frustration and increased support workload.	Order statuses are updated in real time, and customers are promptly notified at each stage (e.g., processing, shipped).	Delayed status updates cause customer dissatisfaction and inefficiencies for the support team.	1. Use SQL queries to fetch real-time order status updates. 2. Automate SQL UPDATE queries for dynamic status updates. 3. Implement SQL-based notifications for each status change.

Aspect	Current State (AS-IS)	Future State (TO-BE)	Gap Identified	Steps to Cover the Gap (using SQL)
SQL Example				UPDATE Orders SET OrderStatus = 'Shipped' WHERE OrderID = 1;
Customer Order Reporting	Limited reporting capabilities on customer orders; no ability to generate customized SQL-based reports.	Comprehensive SQL-based reporting with real-time data for orders, sales, returns, and refunds.	Business lacks insights into sales, returns, and customer orders due to limited reporting capabilities.	1. Implement SQL SELECT queries to generate reports on orders, sales, and returns. 2. Use SQL JOINqueries for complex reporting. 3. Automate report generation using SQL.
SQL Example				SELECT MONTH(OrderDate) AS OrderMonth, SUM(TotalAmount) AS TotalSales FROM Orders WHERE OrderStatus = 'Delivered' GROUP BY MONTH(OrderDate);
Product Availability Notification	No automatic notifications when products go out of stock or reach the reorder threshold.	Automatic alerts when product stock falls below the reorder level, enabling proactive stock replenishment.	No proactive notifications for low stock levels lead to stockouts, causing a loss in sales.	1. Set up SQL TRIGGERS to monitor stock levels and alert managers when a product hits the reorder level. 2. Use SQL SELECT queries to fetch low stock products.
SQL Example				SELECT ProductName, QuantityInStock FROM Inventory WHERE QuantityInStock < ReorderLevel;

Aspect	Current State (AS-IS)	Future State (TO-BE)	Gap Identified	Steps to Cover the Gap (using SQL)
Return & Refund Management	Limited visibility into returns and refunds data, making it difficult to assess financial impact and improve processes.	SQL-based reporting on returns and refunds allows for monthly analysis of financial impact and process improvement.	Inability to generate detailed reports on returns and refunds results in poor financial tracking and inefficiencies.	1. Use SQL SELECTqueries to track and analyze returns. 2. Automate refund tracking using SQL JOIN queries between Orders and Returns tables. 3. Build automated SQL reports for refund analysis.
SQL Example				SELECT MONTH(ReturnDate) AS ReturnMonth, SUM(RefundAmount) AS TotalRefunds FROM Returns GROUP BY MONTH(ReturnDate);
System Performance	SQL queries on large datasets (e.g., orders and inventory) are slow, impacting user experience and operational efficiency.	Optimized SQL queries with faster response times, enabling smooth reporting and real-time system performance.	Poor query performance on large datasets slows down reporting and operations.	1. Optimize SQL queries with indexes. 2. Regularly monitor query performance with SQL EXPLAIN. 3. Implement caching for frequent SQL reports.
SQL Example				CREATE INDEX idx_orderdate ON Orders (OrderDate);
Customer Experience	Customers face delays in order tracking and product availability, leading to complaints and poor brand perception.	Customers receive real-time order updates, product availability alerts, and faster order fulfillment, improving satisfaction.	Delayed order updates and stockouts negatively affect customer experience and brand perception.	1. Use SQL TRIGGERS to notify customers of order updates. 2. Automate SQL queries for stock availability alerts. 3. Improve real-time customer experience with faster SQL queries for order tracking.

Aspect	Current State (AS-IS)	Future State (TO-BE)	Gap Identified	Steps to Cover the Gap (using SQL)
SQL Example				UPDATE Inventory SET QuantityInStock = QuantityInStock - 1 WHERE ProductID = 1;

ROOT CAUSE ANALYSIS:

Problem Statement

- Customers are reporting that the system is allowing them to place orders for products that are marked as "out of stock" in the inventory, resulting in order cancellations and customer dissatisfaction.

Root Cause Analysis

Step 1: Investigate Inventory Management

Action: Check if the inventory levels are updated correctly when orders are placed.
SQL Query: Check Stock Levels for Ordered Products

```
SELECT O.OrderID, O.CustomerID, O.OrderDate, O.OrderStatus, I.ProductName, I.QuantityInStoc

FROM Orders O

JOIN Inventory I ON O.ProductID = I.ProductID

WHERE I.QuantityInStock = 0 AND O.OrderStatus = 'Processing';
```

Expected Output:

OrderID	CustomerID	OrderDate	OrderStatus	ProductName	QuantityInStock

101	**2005**	**2024-01-01**	**Processing**	**Smartphone**	**0**
102	**2006**	**2024-01-02**	**Processing**	**Laptop**	**0**

Root Cause: Orders are being processed for products with zero stock, indicating a failure in stock validation.

Step 2: Investigate Order Placement Logic

Action: Check if the logic prevents orders from being placed when the inventory is out of stock.
SQL Query: Review Orders with Invalid Stock Levels

```
SELECT ProductID, COUNT(OrderID) AS OrderCount, MAX(OrderDate) AS
LastOrderDate

FROM Orders

GROUP BY ProductID

HAVING ProductID IN (SELECT ProductID FROM Inventory WHERE
QuantityInStock = 0);
```

Output:

ProductID	OrderCount	LastOrderDate
1	**5**	**2024-01-10**
2	**3**	**2024-01-09**

Root Cause: Multiple orders were placed even when inventory levels were zero, confirming a failure in the order placement logic.

Step 3: Investigate Inventory Updates After Order Placement

Action: Verify if the inventory is updated correctly after an order is placed.

SQL Query: Verify Inventory Update After Order

```
SELECT I.ProductID, I.ProductName, I.QuantityInStock,SUM(O.TotalAmount) AS TotalSales

FROM Orders O

JOIN Inventory I ON O.ProductID = I.ProductID

WHERE O.OrderStatus = 'Delivered'

GROUP BY I.ProductID, I.ProductName, I.QuantityInStock

HAVING I.QuantityInStock < 0;
```

Expected Output:

ProductID	ProductName	QuantityInStock	TotalSales
1	Smartphone	-5	500.00

Root Cause: The system allowed orders to be placed for products beyond the available stock, leading to negative stock levels.

Step 4: Check SQL Triggers for Stock Updates

Action: Check if SQL triggers are set up correctly to update inventory after each order.
SQL Query: Review SQL Triggers on Inventory Updates

```
SHOW TRIGGERS LIKE 'Inventory';
```

EXPECTED OUTPUT:

rigger	Event	Table	Statement	Timing

StockUpdate	UPDATE	Inventory	UPDATE Inventory SET QuantityInStock = QuantityInStock - X WHERE ProductID = Y;	AFTER

Root Cause: Missing or incorrectly configured SQL triggers caused inventory not to update after order placement.

Step 5: Investigate Order Validation Logic

Action: Verify if the system is preventing customers from placing orders when stock is unavailable.

SQL Query: Review Order Placement Logic

```
SELECT O.OrderID, O.CustomerID, I.ProductID, I.QuantityInStock, O.OrderStatus
FROM Orders O
JOIN Inventory I ON O.ProductID = I.ProductID
WHERE I.QuantityInStock = 0;
```

Expected Output:

OrderID	CustomerID	ProductID	QuantityInStock	OrderStatus
101	2005	1	0	Processing
102	2006	2	0	Processing

Root Cause: The system failed to validate stock levels before order placement.

Steps to Fix the Root Cause

1.Implement Stock Validation Logic : Add a validation step to the order placement process.

- **SQL Example:**

```
IF (SELECT QuantityInStock FROM Inventory WHERE ProductID = 1) > 0 THEN
   -- Proceed with order
ELSE
   -- Reject order
END IF;
```

2. Fix SQL Trigger Configuration: Ensure that the AFTER UPDATE trigger on the Inventory table updates stock levels correctly.

- SQL Example:

```
CREATE TRIGGER StockUpdate AFTER UPDATE ON Orders
FOR EACH ROW
BEGIN
UPDATE Inventory
   SET QuantityInStock = QuantityInStock - NEW.Quantity
   WHERE ProductID = NEW.ProductID;
END;
```

3. Real-Time Inventory Sync : Synchronize inventory levels with the order system in real-time.

SQL Example:

```
UPDATE Inventory
SET QuantityInStock = QuantityInStock - (
   SELECT SUM(Quantity)
   FROM Orders
   WHERE ProductID = Inventory.ProductID AND OrderStatus = 'Processing'
);
```

SQL Use Cases

SQL is used to analyze customer orders and generate reports for the business team.

- **SQL Query to Fetch Monthly Sales Report**:

```
SELECT MONTH(OrderDate) AS OrderMonth,
    SUM(TotalAmount) AS TotalSales
FROM Orders
WHERE OrderStatus = 'Delivered'
GROUP BY MONTH(OrderDate);
```

Output:

OrderMonth	TotalSales
1	250.00

This query provided monthly sales data based on delivered orders, helping the business team track performance.

2. Inventory Analysis for Stock Management

SQL Query to Identify Low Stock Products:

```
SELECT ProductName, QuantityInStock
FROM Inventory
WHERE QuantityInStock < ReorderLevel;
```

Output:

ProductName	QuantityInStock
Smartphone	5

This query was used to alert the inventory team when product stock was running low.

3. Customer Order Status Reporting

SQL Query to Fetch Pending Orders:

```
SELECT OrderID, CustomerID, OrderDate
FROM Orders
```

```
WHERE OrderStatus = 'Processing' OR OrderStatus = 'Pending';
```

Output:

OrderID	CustomerID	OrderDate
3	1003	2024-01-12

This helped the operations team identify pending or delayed orders for faster resolution

4. Returns and Refunds Reporting

SQL Query to Calculate Monthly Refunds:

```
SELECT MONTH(ReturnDate) AS ReturnMonth,
    SUM(RefundAmount) AS TotalRefunds
FROM Returns
GROUP BY MONTH(ReturnDate);
```

Output (assuming no returns data yet):

ReturnMonth	TotalRefunds
NULL	NULL

Sample Output (with returns data)

ReturnMonth	TotalRefunds
1	50.00

This report was used to monitor monthly refund trends and assess the financial impact of returns.

5. Product Sales Insights

SQL Query to Fetch Top-Selling Products:

```
SELECT ProductID, SUM(TotalAmount) AS ProductSales
FROM Orders
```

```
GROUP BY ProductID
ORDER BY ProductSales DESC
LIMIT 5;
```

Output (assuming ProductID refers to an actual product):

ProductID	ProductSales
NULL	500.00

This query helped the product managers identify the top-selling products, enabling better inventory planning

USER STORY AND ACCEPTANCE CRITERIA

User Story 1: Real-Time Inventory Updates

Story:
As an Inventory Manager, I want the system to automatically update inventory levels when an order is placed or products are received, so that I can always have accurate stock levels in real-time.

Acceptance Criteria:

1. When an order is placed and confirmed, the system reduces the corresponding product's stock by the order quantity.
2. When new products are received and inventory is updated manually, the system increases the stock for the corresponding product.
3. If a product is out of stock and a customer attempts to place an order, the system displays an "Out of Stock" message and prevents the order.
4. When a query is run on the inventory table, updated stock levels are reflected in real-time.

User Story 2: Customer Order Status Updates

Story:
As a Customer, I want to receive real-time updates on my order status (Processing, Shipped, Delivered), so that I can track my order.

Acceptance Criteria:

1. When the order status is updated (e.g., to Processing, Shipped, or Delivered), the customer receives an email and SMS notification.
2. When a customer logs into their account, the current order status is displayed in real-time.
3. When an order is shipped, the system records the shipping date and time.

User Story 3: Low Stock Notification

Story:
As an Inventory Manager, I want to receive a notification when the stock level of any product falls below the reorder threshold, so that I can reorder products in time.

Acceptance Criteria:

1. When a product's stock falls below the defined reorder level, the system automatically sends an email to the inventory manager.
2. When a product reaches the low stock threshold, the system triggers a notification.
3. When a product is restocked and inventory is updated, the notification for that product is no longer active.

User Story 4: SQL-Based Sales Report

Story:
As a Product Manager, I want to generate monthly sales reports using SQL queries, so that I can analyze product performance and make data-driven decisions.

Acceptance Criteria:

1. When a monthly report query is run, the total sales for each product are calculated for that month.
2. Only orders with status "Delivered" are included in the total sales amount when the SQL query is executed.
3. For datasets of up to 1 million rows, the query returns results in under 3 seconds.

User Story 5: Real-Time Order Blocking for Out-of-Stock Products

Story:
As a Customer, I want the system to block me from placing orders for out-of-stock products, so that I don't experience cancellations or delays.

Acceptance Criteria:

1. When a customer tries to add an out-of-stock product to their cart, the system prevents the product from being added and displays an "Out of Stock" message.
2. When the stock level of a product is zero, the system automatically flags the product as "Out of Stock."
3. When the product stock is replenished, the system makes the product available for ordering

User Story 6: Return and Refund Management

Story:
As a Customer Support Representative, I want to track customer returns and refunds, so that I can handle return requests efficiently and process refunds in a timely manner.

Acceptance Criteria:

1. When a return is processed, the return data (ReturnID, OrderID, ReturnDate, RefundAmount) is stored in the system.
2. When a refund is processed, the customer's refund status is updated, and a notification is sent to the customer.
3. When SQL queries are executed for monthly reports, the total refund amount for each month is calculated and made available for analysis.

User Story 7: SQL-Based Inventory Reporting

Story:
As an Inventory Manager, I want to generate real-time reports on stock levels using SQL queries, so that I can monitor inventory levels and identify low stock products.

Acceptance Criteria:

1. When a report query is executed, the current stock level of all products is displayed.
2. When the query is run, products with stock levels below the reorder threshold are displayed.
3. For datasets of up to 1 million rows, the query returns results in under 3 seconds.

User Story 8: Order Management API Integration

Story:
As a Developer, I want to integrate the Order Management System (OMS) with the Inventory Management System (IMS) via APIs, so that stock levels are updated in real time when an order is placed.

Acceptance Criteria:

1. When an order is placed, the OMS triggers an API call to the IMS to update the stock level.
2. When an order is canceled, the IMS receives an API call to update the stock by adding the returned quantity.
3. When orders are placed, stock levels are updated without delays across both systems.

User Story 9: Notification API Integration

Story:
As a Developer, I want to integrate the notification system with the inventory and order management systems, so that customers and managers receive real-time updates.

Acceptance Criteria:

1. When an order status changes, the notification API sends an email/SMS to the customer with the updated status.
2. When the stock level falls below the reorder level, the notification API sends a low stock alert to the inventory manager.
3. Notifications are sent without delay when a trigger event occurs (e.g., order status change, low stock).

User Story 10: Data Migration and Validation

Story:
As a Data Analyst, I want to ensure that all existing data from the legacy system is migrated and validated in the new system, so that no data is lost or corrupted during the transition.

Acceptance Criteria:

1. When migration is performed, all customer, order, and inventory data are accurately transferred to the new system.
2. After migration, all records match between the legacy and new systems during validation.
3. After the system goes live, migrated data is accessible without loss or discrepancies.

USE CASES:

Use Case 1: Real-Time Inventory Updates

Description:
This use case describes how the system automatically updates inventory levels when an order is placed or stock is received.

Primary Actor: Inventory Manager and **Supporting Actor**: System

Pre-Condition: Product data exists in the inventory table with valid stock levels.
Post-Condition: Inventory levels are updated in real time after an order is placed or stock is received.

Main Flow:

1. **Actor Step**: The Inventory Manager updates stock when new products are received.
   System Response: The system increases the `QuantityInStock` for the corresponding product in the database.

2. **Actor Step**: A customer places an order for a product.
   System Response: The system checks the current stock. If stock is available, it reduces the `QuantityInStock` by the ordered quantity.
3. **Actor Step**: The Inventory Manager queries the stock levels.
   System Response: The system displays current inventory levels, reflecting real-time updates.

Alternate Flow:

- **AF1**: If the stock is insufficient, the system prevents the order and notifies the customer that the product is "Out of Stock."

Exception Flow:

- **EF1**: If the system fails to update inventory due to a technical issue (e.g., database connection failure), an error message is displayed, and the order is flagged for manual review.

Additional Requirements:

- **Functional**: The system must track stock changes and generate inventory update reports.
- **Non-Functional**: Inventory updates should be processed within 2 seconds.
- **Database**: Stock levels are stored in the Inventory table.
- **Technical**: Use SQL triggers for automatic stock updates after every order transaction.

Use Case 2: Customer Order Status Updates

Description:
This use case describes how customers receive real-time updates on their order status.

Primary Actor: Customer
Supporting Actor: System, Customer Support

Pre-Condition: The customer has placed an order, and the order is stored in the system.
Post-Condition: The customer receives real-time notifications at each stage of the order's progress.

Main Flow:

1. **Actor Step**: The customer places an order.
   System Response: The system sets the order status to "Processing" and sends a confirmation email/SMS to the customer.

2. **Actor Step**: The order is shipped by the fulfillment team.
   System Response: The system updates the order status to "Shipped" and notifies the customer.
3. **Actor Step**: The order is delivered to the customer.
   System Response: The system updates the order status to "Delivered" and notifies the customer.

Alternate Flow:

- **AF1**: If the customer cancels the order before shipping, the system updates the status to "Cancelled" and notifies the customer.

Exception Flow:

- **EF1**: If the notification system fails (e.g., SMS gateway or email server is down), the system logs the error and retries.

Additional Requirements:

- **Functional**: Notifications for status changes ("Processing," "Shipped," "Delivered").
- **Non-Functional**: Notifications must be sent within 5 seconds.
- **Database**: Order statuses are stored in the Orders table.
- **Technical**: Integration with third-party SMS and email gateways.

Use Case 3: Low Stock Notification

Description:
This use case describes how the system notifies inventory managers when stock levels fall below a defined threshold.

Primary Actor: Inventory Manager

Supporting Actor: System

Pre-Condition: Product stock levels exist in the inventory system, and the reorder threshold is set.
Post-Condition: Inventory Manager is notified when stock levels fall below the reorder threshold.

Main Flow:

1. **Actor Step**: The system monitors the `QuantityInStock` for all products in real time.
   System Response: When stock falls below `ReorderLevel`, the system triggers a notification.
2. **Actor Step**: The system sends an email to the Inventory Manager.
   System Response: The system logs the notification and waits for an inventory update.
3. **Actor Step**: The Inventory Manager reorders stock.
   System Response: The system updates inventory when new stock is received.

Alternate Flow:

- **AF1**: If inventory is restocked before reaching the reorder threshold, the system does not trigger a notification.

Exception Flow:

- **EF1**: If the notification fails (e.g., email server is down), the system logs the failure and retries.

Additional Requirements:

- **Functional**: System must send notifications automatically when stock falls below `ReorderLevel`.
- **Non-Functional**: Notifications must be sent within 5 seconds.
- **Database**: `ReorderLevel` and stock levels are stored in the Inventory table.
- **Technical**: Integration with email servers and retry capability for failed notifications.

Use Case 4: SQL-Based Sales Report

Description:
This use case describes how product managers generate monthly sales reports using SQL queries.

Primary Actor: Product Manager and **Supporting Actor**: System

Pre-Condition: Orders have been placed and delivered, and all transactions are recorded in the system.
Post-Condition: A monthly sales report is generated, showing total sales for each product.

Main Flow:

1. **Actor Step**: The Product Manager requests a monthly sales report.
   System Response: Runs an SQL query to calculate total sales for all delivered orders in the given month.

2. **Actor Step**: The system displays the total sales for each product.
   System Response: The report shows the sum of `TotalAmount` grouped by month and product.

Alternate Flow:

- **AF1**: If a different time range (e.g., weekly report) is specified, the system adjusts the SQL query.

Exception Flow:

- **EF1**: If the SQL query takes too long due to large datasets, the system notifies the user and optimizes the query.

Additional Requirements:

- **Functional**: System must allow SQL queries for monthly and weekly reports.
- **Non-Functional**: Query response time must be under 3 seconds for up to 1 million rows.
- **Database**: Sales data is stored in the Orders table.
- **Technical**: Ensure optimized SQL queries with indexing.

Use Case 5: Return and Refund Management

Description:
This use case describes how the system handles customer returns and refunds.

Primary Actor: Customer Support Representative and **Supporting Actor**: System

Pre-Condition: An order has been placed and delivered. Customer initiates a return.
Post-Condition: The return is processed, and a refund is issued to the customer.

Main Flow:

1. **Actor Step**: The Customer Support Representative initiates the return process.
   System Response: Creates a record in the Returns table with `ReturnID`, `OrderID`, `ReturnDate`, and `RefundAmount`.

2. **Actor Step**: The return is approved.
   System Response: Updates the `RefundAmount` and changes order status to "Returned."
3. **Actor Step**: The customer is refunded.
   System Response: Sends a notification to the customer confirming the refund.

Alternate Flow:

- **AF1**: If the return is rejected, the system updates the status to "Return Rejected" and notifies the customer.

Exception Flow:

- **EF1**: If the refund fails (e.g., payment gateway issue), the system logs the error and retries.

Additional Requirements:

- **Functional**: Track return details and calculate refunds.
- **Non-Functional**: Refunds must be processed within 24 hours of approval.
- **Database**: Return details are stored in the Returns table.
- **Technical**: Integration with payment gateway for issuing refunds.

CONCLUSION:

The **Online Shopping Platform – Customer Order and Inventory Management System** is a transformative initiative aimed at resolving critical inefficiencies in inventory management, order tracking, and data reporting within eCommerce operations. By implementing real-time automation, advanced SQL-based reporting, and robust notification systems, this project addresses existing gaps that negatively impact customer satisfaction and operational performance.

With a focus on scalability, security, and seamless integration with existing systems, the proposed solution empowers stakeholders—including product managers, inventory managers, and customers—with accurate, actionable data and efficient workflows. It minimizes manual

processes, reduces order cancellations due to stockouts, and enhances decision-making through comprehensive reporting capabilities.

Upon successful implementation, this system will not only improve the customer experience through timely updates and accurate stock visibility but also position the organization as a leader in efficient and technology-driven eCommerce operations. The result will be increased customer loyalty, operational efficiency, and business growth.