EX.NO.:12

ENHANCING CUSTOMER EXPERIENCE USING RETAIL DOMAIN DEEP DIVE

Aim:

To implement Inventory Management Optimization and enhance customer experience for a fictional retail store using a retail domain deep dive.

Domain Deep Dive:

A Domain Deep Dive involves a comprehensive exploration and analysis of the retail domain to understand inventory challenges, opportunities, and workflows. This ensures solutions are tailored to optimize inventory management processes effectively.

Key Objectives:

- Ensure accurate stock tracking and real-time visibility.
- Minimize overstocking and stockouts.
- Improve demand forecasting and inventory turnover.

Benefits:

- Reduced operational costs through efficient stock management.
- Improved order fulfillment rates and customer satisfaction.
- Data-driven decision-making to handle seasonal demand and trends.

Installation Steps for Inventory Management Optimization :

To implement inventory management optimization, follow these steps using specific tools and frameworks:

Step 1: Install Inventory Tracking Tools

- Tools like Zoho Inventory, Fishbowl, or Odoo for centralized inventory management.
    - **Odoo Installation:**
        1. Install Odoo: sudo apt-get install odoo
        2. Configure product categories, warehouses, and suppliers.
        3. Enable barcode or RFID tracking.

Step 2: Setup Real-Time Monitoring

- Implement barcode or RFID systems for stock movement tracking.
    - Use scanners integrated with the inventory system.
    - Train warehouse staff to use scanning devices for stock-in and stock-out updates.

Step 3: Configure Demand Forecasting Models

- Use ML models or built-in forecasting tools.
    - Install TensorFlow for demand prediction: pip install tensorflow.
    - Train models with historical sales data to predict trends.

Step 4: Integrate With Existing Systems

- Connect the inventory system to POS, ERP, and e-commerce platforms for real-time updates.
    - Integration with Shopify:
        - Use Shopify API to synchronize inventory levels and sales.

Step 5: Automate Reordering

- Set reorder thresholds and automate purchase orders.
    - Configure safety stock levels in the inventory management system.

Module Taken for Implementation: Real-Time Inventory Monitoring :

Objective of the Module:

The Real-Time Inventory Monitoring Module ensures that inventory levels are accurately updated with every transaction, preventing discrepancies and enabling quick decision-making.

Features of the Real-Time Inventory Monitoring Module:

1. **Barcode Scanning:** Automatically update stock levels during sales and restocking.
2. **Real-Time Alerts:** Notifications for low stock or overstocked items.
3. **Demand-Based Reordering:** Automated reordering based on sales trends.
4. **Centralized Dashboard:** Provides visibility across multiple warehouses or stores.

Code Implementation:

1. Database Setup:

- Add fields for quantity and reorder_threshold in the products table.

```
rails generate migration AddInventoryFieldsToProducts quantity:integer
reorder_threshold:integer
```

```
rails db:migrate
```

2. Model Modifications:

- Update the Product model to manage inventory logic.

PROGRAM:

```
class Product < ApplicationRecord
  def check_reorder
    if quantity < reorder_threshold
      reorder_stock
    end
  end
  private
  def reorder_stock
    # Logic to generate purchase order
    puts "Reorder triggered for #{name}"
  end
end
```

3. Controller Logic:

a. Updating Inventory After Sales:

- Deduct stock levels automatically after a sale.

PROGRAM:

```
class SalesController < ApplicationController
  def create
    @sale = Sale.new(sale_params)
    if @sale.save
```

```
      product = Product.find(@sale.product_id)
      product.quantity -= @sale.quantity
      product.save
      product.check_reorder
      redirect_to sales_path, notice: "Sale recorded and inventory updated."
    else
      render :new, alert: "Failed to record sale."
    end
  end
end
```

b. Restocking Inventory:

- Update stock levels after new stock arrives.

PROGRAM:

```
class InventoryController < ApplicationController
  def restock
    product = Product.find(params[:product_id])
    product.quantity += params[:quantity].to_i
    product.save
    redirect_to inventory_path, notice: "Stock updated successfully."
  end
end
```

4. Frontend Integration:

a. Inventory Dashboard:

- Display current stock levels and low-stock alerts.

PROGRAM:

```
<h2>Inventory Dashboard</h2>
<table>
  <tr>
```

```
    <th>Product</th>
    <th>Quantity</th>
    <th>Reorder Threshold</th>
  </tr>
  <% @products.each do |product| %>
    <tr>
      <td><%= product.name %></td>
      <td><%= product.quantity %></td>
      <td><%= product.reorder_threshold %></td>
    </tr>
  <% end %>
</table>
```

b. Restock Form:

- Allow manual restocking.

PROGRAM:

```
<%= form_with url: restock_inventory_path, method: :post do |form| %>
  <%= form.text_field :product_id, placeholder: "Product ID" %>
  <%= form.number_field :quantity, placeholder: "Quantity" %>
  <%= form.submit "Restock" %>
<% end %>
```

5. Testing the Module:

Red Stage: Write test cases for missing functionalities.

PROGRAM:

```
require "test_helper"
class InventoryManagementTest < ActiveSupport::TestCase
  test "should deduct inventory after sale" do
    product = products(:one)
    sale = Sale.create(product: product, quantity: 5)
    assert_equal product.quantity, product.quantity - 5
  end

  test "should trigger reorder when quantity low" do
    product = products(:one)
    product.update(quantity: 2, reorder_threshold: 5)
    assert_output(/Reorder triggered for/) { product.check_reorder }
  end
end
```

Green Stage: Implement functionality and ensure tests pass.

Refactor Stage: Optimize code for readability and performance.

SPLIT UP:

Description	Allotted Marks	Obtained Marks
Preparation	20	
Design/Implementation	20	
Viva	15	
Output	10	
Record	10	
Total	75	

RESULT:

Thus, the implementation of Inventory Management Optimization using a retail domain deep dive has been successfully completed.