Hanoi University of Science and Technology

School of Information and Communication Technology

———————— * ————————

Final Report
IT3290E – Database Lab

Topic: Fashion E-commerce test Website

Supervisors: **Nguyễn Hồng Phương**

Group Members:

Nguyễn Tuấn Đức	20235916
Bùi Quang Minh	20235971
Bui Doan Khang	20235950

I. Problem Description & Project Context

1. Context

In the rapidly evolving digital landscape, traditional brick-and-mortar fashion retailers face increasing pressure to modernize their operations. Our client, a growing fashion brand, currently relies on disparate manual systems—spreadsheets for inventory, paper-based orders, and disconnected customer data. This fragmentation has led to significant operational inefficiencies, including:

- **Inventory Desynchronization:** Frequent stock discrepancies between physical warehouse counts and sales records.
- **Scalability Bottlenecks:** Inability to handle high-traffic sales events due to manual processing limits.
- **Data Silos:** Lack of unified customer insights, making personalized marketing impossible.

This project aims to bridge this gap by constructing a robust, high-performance E-commerce platform backed by a centralized Microsoft SQL Server database. The system is designed not just as a sales channel, but as a comprehensive operational backbone ensuring data integrity and real-time responsiveness.

2. Problem Statement

The core technical challenges identified in the legacy approach include:

- **Concurrency Control:** In a multi-user environment, two customers attempting to purchase the last item simultaneously often resulted in "overselling" due to race conditions.
- **Search Performance:** As the product catalog grows (projected 50,000+ SKUs), standard textual searches become prohibitively slow, degrading user experience.
- **Complex Logic Coupling:** Business logic (stock checks, voucher validation) was previously scattered across frontend code or raw SQL scripts, leading to security vulnerabilities and maintenance nightmares.

3. Proposed Solution & Functional Requirements

This project delivers a comprehensive solution divided into three distinct modules, each tailored to specific operational needs and powered by robust SQL Server logic.

3.1. Customer Module (The Shopping Experience)

The customer interface is designed for speed, convenience, and transparency.

- **Authentication & Profile:**
    - **Secure Access:** Users can register and login using their email/password. The system returns a unique User ID and role token.

    - **Profile Management:** Users can update personal details (Name, Phone, Avatar) and manage a digital address book for one-click shipping.
- **Advanced Product Discovery:**
    - **Multi-Faceted Search:** Users can filter products not just by name, but by price range, specific rating thresholds (e.g., 4 stars+), and visual attributes (Color/Size).
    - **Sorting Intelligence:** Dynamic sorting capabilities allow users to view products by "Newest Arrivals", "Price Low-to-High", or "Best Rated" instantly.
- **Smart Cart System:**
    - **Persistent Shopping:** The cart is database-backed, meaning users can add items on mobile and complete checkout on desktop without data loss.
    - **Intelligent Merging:** If a user adds the same item twice, the system automatically detects this and updates the quantity rather than creating duplicate entries (Upsert logic).
    - **Real-time Stock Validation:** The system prevents "over-carting" by checking available stock before an item is added to the cart.
- **Secure Checkout Flow:**
    - **Address Book:** Users can save multiple shipping addresses and easily toggle their default choice.
    - **Voucher Application:** Users can apply promotional codes which are validated in real-time for expiry, usage limits, and minimum order values.
    - **Order Confirmation:** Users receive instant confirmation with a clear breakdown of subtotal, shipping fees, discounts, and final total.
- **Order Lifecycle Management:**
    - **Self-Service Cancellation:** Users can cancel their own orders if they are still in the "Pending" state, triggering an automatic restock of inventory.
    - **History & Tracking:** Full visibility into past orders and current status updates.
- **Engagement & Support:**
    - **Wishlist:** Users can save favorite product variants to a personal list (user_favorites) for future consideration.
    - **Product Reviews:** Verified buyers can rate products (1-5 stars) and leave comments to help the community.
    - **Customer Support:** A built-in messaging system allows users to open support tickets (support_messages) directly from their dashboard.

3.2. Administration Module (Business Operations)

The admin panel provides granular control over the system's data and logic.

- **Product Catalog Management:**
    - **CRUD Operations:** Admins can create new products, update pricing/descriptions, and soft-delete items that are no longer for sale.
    - **Variant-Level Control:** Admins do not just manage "T-shirts"; they manage "T-shirt (Red, Size M)" stock individually.
- **Marketing & Promotions:**
    - **Campaign Management:** Admins can create targeted vouchers with specific constraints (e.g., "10% off, max 50 uses, valid for 3 days").

    - **Banner Management:** Dynamic control over homepage visuals to highlight active sales (upsert_banner).
- **Order Fulfillment:**
    - **Workflow Management:** Admins can advance orders through stages: Pending → Confirmed → Shipping → Completed.
    - **Return Processing:** Systematically handles returns, restocking items if necessary and updating payment statuses.
- **Data Analytics:**
    - **Revenue Intelligence:** View daily revenue reports to track growth trends.
    - **Product Performance:** Identify "Best Sellers" to optimize inventory purchasing decisions.

3.3. System Core & Security

- **Database-Level Role Definition:** The system enforces user roles (CUSTOMER vs ADMIN) directly in the database schema via Check Constraints, ensuring invalid roles cannot be assigned even if application logic fails.
- **Transactional Safety:** The system handles high-concurrency scenarios (like multiple users buying the last item) by wrapping critical operations in ACID Transactions. Specifically, the checkout process uses atomic updates to ensure inventory is never oversold.

4. Database Objectives

The underlying database design focuses on solving the aforementioned problems by:

- Standardizing data structures for users, products, and transactions.
- Ensuring valid data entry through strict validation rules (e.g., stock cannot be negative).
- Optimizing query performance for search and reporting features.

II. System Architecture & Database Design:

1. Technology Stack

The system is built on a modern, multi-tier architecture designed for extensibility and performance:

- **Database Level (The Core)**:
    - **Microsoft SQL Server 2022**: Selected for its industrial-grade transaction support, advanced indexing capabilities, and security features.
    - **T-SQL (Transact-SQL)**: All business logic (Create Order, Add to Cart) is encapsulated in Stored Procedures to ensure execution speed and consistency.
- **Application Level**:
    - **Node.js & Express**: Provides a non-blocking, event-driven runtime environment, ideal for handling concurrent I/O operations from E-commerce traffic.

- **EJS (Embedded JavaScript)**: Server-side rendering engine for generating dynamic HTML content.

- **Frontend Level**:
  - **Bootstrap 5**: Ensures a responsive mobile-first design for customers shopping on various devices.

2. Database Design (ERD) & Relational Model

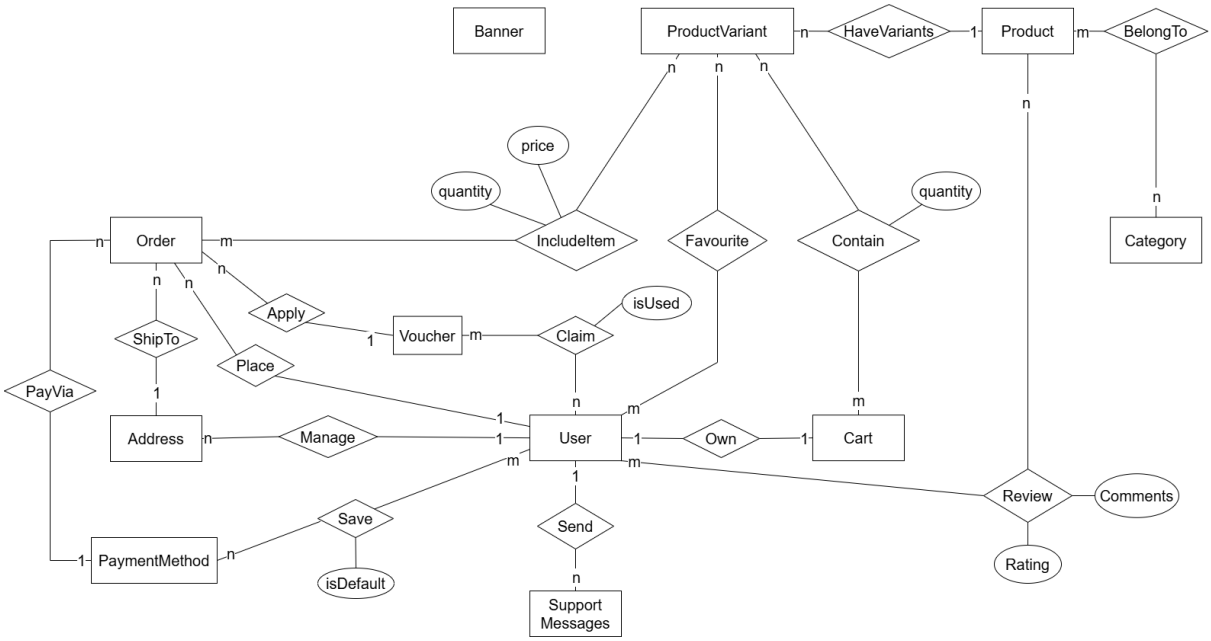


Figure 2.2.1 ERD Diagram

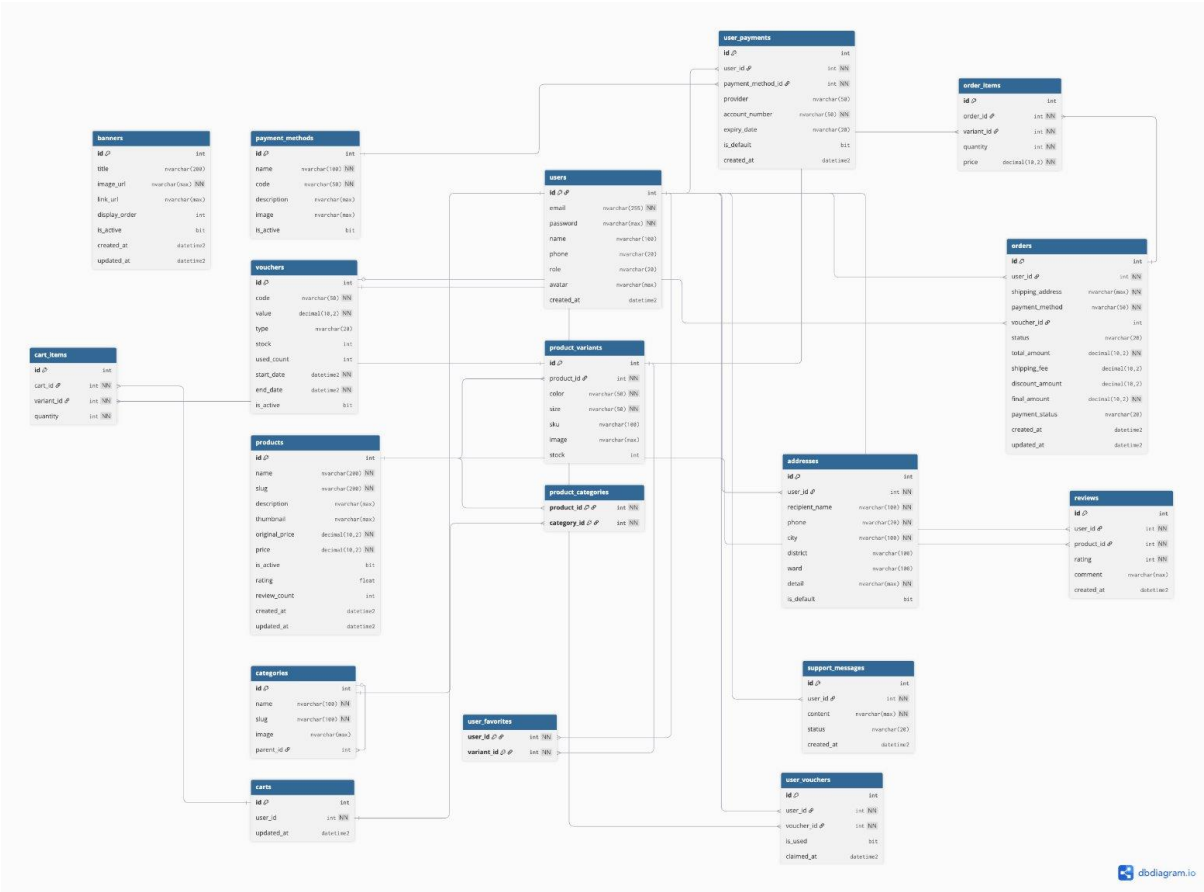


Figure 2.2.2: Relational Diagram

users

Purpose: Stores all registered user accounts, including both customers and administrators. Acts as the central identity table for the entire system.

Key Columns:

- `id` (PK) – Auto-generated unique identifier
- `email` (UNIQUE) – Login credential, must be unique across all users
- `password` – Hashed password for authentication
- `role` – User type: `'CUSTOMER'` or `'ADMIN'` (enforced by CHECK constraint)
- `created_at` – Account registration timestamp

addresses

Purpose: Stores shipping addresses for users. Each user can save multiple addresses and designate one as default for faster checkout.

Key Columns:

- `id` (PK) – Unique address identifier
- `user_id` (FK) – References `users.id`
- `recipient_name`, `phone` – Contact for delivery
- `city`, `district`, `ward`, `detail` – Full address breakdown
- `is_default` – Boolean flag for default address (enforced by trigger)

payment_methods

Purpose: Master list of available payment options (e.g., COD, Credit Card, VNPay). Managed by administrators.

Key Columns:

- `id` (PK), `name`, `code` (UNIQUE)
- `is_active` – Whether this payment method is currently offered

user_payments

Purpose: Stores user-specific payment details (e.g., saved card numbers). Allows one-click payment during checkout.

Key Columns:

- `user_id` (FK), `payment_method_id` (FK)
- `account_number`, `expiry_date` – Saved payment credentials
- `is_default` – Default payment for user (enforced by trigger)

categories

Purpose: Hierarchical product classification (e.g., Men's Wear → Shirts → Casual). Supports parent-child relationships for nested categories.

Key Columns:

- `id` (PK), `name`, `slug` (UNIQUE for SEO-friendly URLs)
- `parent_id` (FK, self-referencing) – Points to parent category for hierarchy

products

Purpose: Core product information (name, description, pricing). Represents the "parent" product entity; actual purchasable items are variants.

Key Columns:

- `id` (PK), `name`, `slug` (UNIQUE)

- `original_price`, `price` – Supports "was/now" pricing for promotions
- `rating`, `review_count` – Aggregated review statistics (updated by `submit_product_review`)
- `is_active` – Soft-delete flag (inactive products hidden from customers)

product_variants

Purpose: SKU-level inventory management. Each variant represents a unique purchasable combination (e.g., "Red T-Shirt, Size M").

Key Columns:

- `id` (PK), `product_id` (FK)
- `color`, `size` – Variant attributes (UNIQUE constraint on product_id + color + size)
- `sku` – Stock Keeping Unit code
- `stock` – Current inventory quantity (protected by trigger from going negative)

product_categories

Purpose: Junction table for Many-to-Many relationship between products and categories. Allows one product to belong to multiple categories.

Key Columns:

- `product_id` (FK, PK), `category_id` (FK, PK) – Composite primary key

carts

Purpose: Container for user's shopping session. Each user has exactly one cart (1:1 relationship).

Key Columns:

- `id` (PK), `user_id` (FK, UNIQUE) – Ensures one cart per user
- `updated_at` – Tracks last activity (useful for abandoned cart detection)

cart_items

Purpose: Individual items in a user's cart. Links cart to product variants with quantity.

Key Columns:

- `cart_id` (FK), `variant_id` (FK) – UNIQUE constraint prevents duplicate entries
- `quantity` – Number of units

orders

Purpose: Core transactional record. Captures the complete state of a purchase including shipping, payment, and pricing snapshot.

Key Columns:

- `id` (PK), `user_id` (FK)
- `shipping_address` – Text snapshot of address at order time (denormalized for history)
- `status` – Order lifecycle: `PENDING` → `CONFIRMED` → `SHIPPING` → `COMPLETED` (or `CANCELLED`/`RETURNED`)
- `payment_status` – `UNPAID`, `PAID`, `REFUNDED`
- `total_amount`, `shipping_fee`, `discount_amount`, `final_amount` – Full price breakdown
- `voucher_id` (FK, nullable) – Applied discount voucher

order_items

Purpose: Line items within an order. Captures variant, quantity, and price at the moment of purchase (price snapshot prevents historical corruption).

Key Columns:

- `order_id` (FK), `variant_id` (FK)
- `quantity`, `price` – Frozen at order creation time

vouchers

Purpose: Discount codes for promotional campaigns. Supports both fixed amount and percentage discounts.

Key Columns:

- `code` (UNIQUE) – Human-readable promo code (e.g., "NEWYEAR2026")
- `value`, `type` – Discount amount and type (`FIXED` or `PERCENTAGE`)
- `stock`, `used_count` – Usage limits
- `start_date`, `end_date` – Validity period
- `is_active` – Admin can disable without deleting

user_vouchers

Purpose: Junction table tracking which users have claimed which vouchers, and whether they've been used.

Key Columns:

- `user_id` (FK), `voucher_id` (FK) – UNIQUE constraint prevents duplicate claims

- `is_used` – Whether voucher has been applied to an order
- `claimed_at` – When user collected the voucher

banners

Purpose: Homepage promotional banners. Controlled by administrators for marketing campaigns.

Key Columns:

- `title`, `image_url`, `link_url` – Banner content
- `display_order` – Sorting priority
- `is_active` – Visibility toggle

user_favorites

Purpose: Wishlist functionality. Users can save product variants for future purchase consideration.

Key Columns:

- `user_id` (FK, PK), `variant_id` (FK, PK) – Composite primary key

reviews

Purpose: Customer product reviews with ratings. Only verified purchasers (users who completed an order containing the product) can submit reviews.

Key Columns:

- `user_id` (FK), `product_id` (FK)
- `rating` – 1-5 star rating (validated by stored procedure)
- `comment` – Review text

support_messages

Purpose: Customer support ticket system. Users can send messages to request assistance.

Key Columns:

- `user_id` (FK), `content` – Message details
- `status` – Ticket status: `OPEN`, `IN_PROGRESS`, `CLOSED`

III. Functional implementation map

1. Customer Module

A. Authentication & Profile

Procedure Name	Description & Implementation Note
`register_user`	Creates new account with CHECK role constraints. Returns new User ID.
`login_user`	Validates credentials (email/password) and returns user profile + token.
`update_profile`	Updates personal info (Avatar, Phone, Password).
`get_my_addresses`	Returns list of all saved shipping addresses for the logged-in user.
`add_address`	Adds a new address. If `is_default` is true, automatically updates other addresses to false.
`update_address`	Updates existing address details.
`delete_address`	**Soft/Hard Delete**: Removes an address from the user's book.

B. Product Discovery & Browsing

Procedure Name	Description & Implementation Note
`browse_products`	**Complex Search**: Handles Keywords, Category Slug, Attributes (Color/Size), Price Range. Implements Dynamic Sorting and Pagination.
`get_product_details`	**JSON Return**: Returns nested JSON object containing Product Info, Variants List, and Latest Reviews in a single query.
`get_trending_products`	Returns top-selling items based on last 30 days.

C. Shopping Cart System

Procedure Name	Description & Implementation Note
`cart_add_item`	**Smart Upsert**: Uses MERGE statement. Validates stock limit (`Current + New <= Stock`) before adding.
`cart_update_item_quantity`	Modifies quantity. Deletes item if New Quantity <= 0.

cart_remove_item	Removes item from cart permanently.
cart_view_details	Lists items with calculated subtotals and real-time stock availability check.

D. Checkout & Orders

Procedure Name	Description & Implementation Note
checkout	**ACID Transaction**: Atomic operation handling Stock Deduction, Voucher Application, Address Snapshotting, and Order Creation. Uses BEGIN TRANSACTION.
collect_voucher	Validates code validity/usage limits and adds to user's wallet.
view_my_vouchers	Lists available vouchers with status (Ready, Expired, Out of Stock).
view_order_history	Lists all orders with status (Pending, Shipping, etc.).
cancel_order	**Restock Logic**: Allows user to cancel 'PENDING' orders. Automatically restores Product Stock and Vouchers. Uses UPDLOCK.

E. Engagement & Support

Procedure Name	Description & Implementation Note
view_wishlist	Displays all products currently saved in the user's favorites list.
add_to_wishlist	Adds product variant to user_favorites.
remove_from_wishlist	Removes a product from the user's favorites list.
submit_product_review	Allows verified purchasers to rate (1-5) and comment. Updates aggregate Product Rating.
send_support_message	Creates a new support ticket in support_messages table.

2. Admin Module

A. Product Catalog Management

Procedure Name	Description & Implementation Note

create_product	Inserts base product. Handles category linking via JSON array input (OPENJSON).
upsert_variant	**Upsert Logic**: Adds new variant (Size/Color) or updates existing one using MERGE.
update_product	Updates general info (Name, Price, Status).
delete_product	Soft-delete: Sets is_active = 0.
delete_variant	Hard-delete: Only allowed if variant has never been sold.

B. Marketing & Promotions

Procedure Name	Description & Implementation Note
upsert_voucher	Creates/Updates discounts. Logic includes Quantity, Date Range, Type (Fixed/Percent).
upsert_banner	Manages Homepage Banners (Image URL, Link, Display Order).
delete_voucher	Deactivates voucher if used, or permanently deletes if unused.
delete_banner	Removes banner from the system.

C. Order Operations

Procedure Name	Description & Implementation Note
view_orders	Filters orders by Status and Date Range.
update_order_status	**Workflow**: Updates status (e.g., Shipping -> Completed). Automated Restocking on 'RETURNED' status.

D. Reports & Analytics

Procedure Name	Description & Implementation Note
report_revenue_by_date	Aggregates daily revenue for COMPLETED orders.
report_best_sellers	Returns top products by quantity sold and total revenue generated.
report_revenue_by_category	Insights into which product categories are driving sales.

IV. Key Implementations

In this section, we present the implementation details of the four most critical business processes in the system. These Procedures handle complex logic, data validation, and transaction management to ensure system integrity.

1. Customer Module

1.1. Dynamic Product Browsing

Procedure Name: browse_products

Description:
This stored procedure acts as the backend for the product listing page. It handles complex filtering requirements including fuzzy search (case-insensitive), category navigation via slugs, price ranges, and attribute matching (Color/Size).

Technical Highlights:

- **Flexible Filtering:** Uses logic (param IS NULL OR column = param) to allow any combination of filters to be applied or ignored.
- **Dynamic Sorting:** Implements a CASE statement in the ORDER BY clause to handle multiple sorting strategies (Newest, Price ASC/DESC, Best Rating) within a single query.
- **Aggregation:** Retrieves product variants while maintaining GROUP BY integrity to return distinct product-variant combinations.

Source code:

```
CREATE OR ALTER PROCEDURE browse_products
    @p_keyword NVARCHAR(200) = NULL,
    @p_category_slug NVARCHAR(100) = NULL,
    @p_color NVARCHAR(50) = NULL,
    @p_size NVARCHAR(50) = NULL,
    @p_min_price DECIMAL(18,2) = 0,
    @p_max_price DECIMAL(18,2) = 999999999,
    @p_min_rating FLOAT = 0,
    @p_sort_by NVARCHAR(20) = 'newest', -- 'newest', 'price_asc', 'price_desc', 'best_rating'
    @p_limit INT = 10,
    @p_offset INT = 0
AS
BEGIN
    SET NOCOUNT ON;

    SELECT
        p.id as product_id,
        p.name as product_name,
        p.original_price,
        p.price as current_price,
        p.thumbnail,
        p.rating as avg_rating,
        p.review_count as total_reviews,
        c.name as category_name,
        p.created_at
    FROM products p
    LEFT JOIN product_categories pc ON p.id = pc.product_id
    LEFT JOIN categories c ON pc.category_id = c.id
    LEFT JOIN product_variants pv ON p.id = pv.product_id
    WHERE p.is_active = 1
```

```
      AND (@p_keyword IS NULL OR p.name LIKE '%' + @p_keyword + '%' OR p.description LIKE '%' +
@p_keyword + '%')
      AND (@p_category_slug IS NULL OR c.slug = @p_category_slug)
      AND (@p_min_price IS NULL OR CAST(p.price AS DECIMAL(18,2)) >= @p_min_price)
      AND (@p_max_price IS NULL OR CAST(p.price AS DECIMAL(18,2)) <= @p_max_price)
      AND (p.rating >= @p_min_rating)
      AND (@p_color IS NULL OR pv.color LIKE @p_color)
      AND (@p_size IS NULL OR pv.size LIKE @p_size)
    GROUP BY p.id, p.name, p.original_price, p.price, p.thumbnail, p.rating, p.review_count, c.name,
p.created_at
    ORDER BY
        CASE WHEN @p_sort_by = 'newest' THEN p.created_at END DESC,
        CASE WHEN @p_sort_by = 'price_asc' THEN p.price END ASC,
        CASE WHEN @p_sort_by = 'price_desc' THEN p.price END DESC,
        CASE WHEN @p_sort_by = 'best_rating' THEN p.rating END DESC,
        p.id ASC
    OFFSET @p_offset ROWS FETCH NEXT @p_limit ROWS ONLY;
END;

GO
```

Execution result:

```
PRINT '>> Test 2.1: Browse Products (All)';
EXEC browse_products;

-- Test 2.2: Search by keyword 'Shirt'
PRINT '>> Test 2.2: Search for "Shirt"';
EXEC browse_products @p_keyword = 'Shirt';

-- Test 2.3: Filter by Price (0 - 200,000)
PRINT '>> Test 2.3: Filter Price <= 200k';
EXEC browse_products @p_max_price = 200000;
```

79 % — No issues found

Results | Messages

	product_id	product_name	original_price	current_price	thumbnail	avg_rating	total_reviews	category_name	created_at
1	7	New Admin Produc1	1000.00	800.00	https://encrypted-tbn0.gstatic.com/images?q=tbn:A...	0	0	Men	2026-01-06 23:07:16.7633333
2	7	New Admin Produc1	1000.00	800.00	https://encrypted-tbn0.gstatic.com/images?q=tbn:A...	0	0	T-Shirts	2026-01-06 23:07:16.7633333
3	6	Classi White T-Shirt	200000.00	150000.00	https://dosi-in.com/images/detailed/42/CDL11_1.jpg	4.5	2	NULL	2026-01-06 22:57:46.8666667
4	5	New Admin Product	1000.00	900.00	test.img	0	0	Men	2026-01-05 23:42:56.9400000
5	5	New Admin Product	1000.00	900.00	test.img	0	0	T-Shirts	2026-01-05 23:42:56.9400000
6	1	Classic White T-Shirt	200000.00	150000.00	https://example.com/tshirt.jpg	4.5	2	T-Shirts	2026-01-05 23:33:57.5700000
7	1	Classic White T-Shirt	200000.00	150000.00	https://example.com/tshirt.jpg	4.5	2	Men	2026-01-05 23:33:57.5700000
8	2	Summer Floral Dress	500000.00	450000.00	https://example.com/dress.jpg	5	1	Dresses	2026-01-05 23:33:57.5700000
9	2	Summer Floral Dress	500000.00	450000.00	https://example.com/dress.jpg	5	1	Women	2026-01-05 23:33:57.5700000
10	3	Leather Belt	300000.00	250000.00	https://example.com/belt.jpg	0	0	Accessories	2026-01-05 23:33:57.5700000

	product_id	product_name	original_price	current_price	thumbnail	avg_rating	total_reviews	category_name	created_at
1	6	Classi White T-Shirt	200000.00	150000.00	https://dosi-in.com/images/detailed/42/CDL11_1.jpg	4.5	2	NULL	2026-01-06 22:57:46.8666667
2	1	Classic White T-Shirt	200000.00	150000.00	https://example.com/tshirt.jpg	4.5	2	Men	2026-01-05 23:33:57.5700000
3	1	Classic White T-Shirt	200000.00	150000.00	https://example.com/tshirt.jpg	4.5	2	T-Shirts	2026-01-05 23:33:57.5700000

	product_id	product_name	original_price	current_price	thumbnail	avg_rating	total_reviews	category_name	created_at
1	7	New Admin Produc1	1000.00	800.00	https://encrypted-tbn0.gstatic.com/images?q=tbn:A...	0	0	Men	2026-01-06 23:07:16.7633333
2	7	New Admin Produc1	1000.00	800.00	https://encrypted-tbn0.gstatic.com/images?q=tbn:A...	0	0	T-Shirts	2026-01-06 23:07:16.7633333
3	6	Classi White T-Shirt	200000.00	150000.00	https://dosi-in.com/images/detailed/42/CDL11_1.jpg	4.5	2	NULL	2026-01-06 22:57:46.8666667
4	5	New Admin Product	1000.00	900.00	test.img	0	0	Men	2026-01-05 23:42:56.9400000
5	5	New Admin Product	1000.00	900.00	test.img	0	0	T-Shirts	2026-01-05 23:42:56.9400000
6	1	Classic White T-Shirt	200000.00	150000.00	https://example.com/tshirt.jpg	4.5	2	Men	2026-01-05 23:33:57.5700000
7	1	Classic White T-Shirt	200000.00	150000.00	https://example.com/tshirt.jpg	4.5	2	T-Shirts	2026-01-05 23:33:57.5700000

1.2. Smart Cart "Upsert" Logic

Procedure Name: cart_add_item

Description:
This stored procedure manages the shopping cart state. It handles the logic of "Upsert" (Update if exists, Insert if new) and strictly enforces stock limits before adding items.

Technical Highlights:

- **Atomic Cart Initialization:** Instead of complex application-side checks, the procedure handles the "Get or Create" logic for the user's cart in a single step, ensuring a smooth transition for new shoppers.
- **Real-time Availability Guard:** Before any modification, the procedure validates the product's status and current stock levels. It raises specific custom exceptions (e.g., 50012, 50013) to block inactive products or requests that exceed physical warehouse limits.
- **MERGE-based Upsert:** Utilizing the SQL Server `MERGE` statement, the system efficiently handles item additions. If the variant is already in the cart, it increments the existing quantity; otherwise, it creates a new entry, maintaining data integrity without redundant queries.

```
CREATE OR ALTER PROCEDURE cart_add_item
    @p_user_id INT,
    @p_variant_id INT,
    @p_quantity INT
AS
BEGIN
    SET NOCOUNT ON;
    DECLARE @v_stock INT, @v_is_active BIT;
    DECLARE @v_cart_id INT;

    -- Check Product
    SELECT @v_stock = pv.stock, @v_is_active = p.is_active
    FROM product_variants pv
    JOIN products p ON pv.product_id = p.id
    WHERE pv.id = @p_variant_id;

    IF @v_stock IS NULL THROW 50011, 'Product variant not found!', 1;
    IF @v_is_active = 0 THROW 50012, 'This product is not sold!', 1;
    IF @p_quantity > @v_stock THROW 50013, 'Product out of stock!', 1;

    -- Get/Create Cart
    SELECT @v_cart_id = id FROM carts WHERE user_id = @p_user_id;

    IF @v_cart_id IS NULL
    BEGIN
        INSERT INTO carts (user_id) VALUES (@p_user_id);
        SET @v_cart_id = SCOPE_IDENTITY();
    END
    ELSE
    BEGIN
        UPDATE carts SET updated_at = GETDATE() WHERE id = @v_cart_id;
    END

    -- Upsert Item using MERGE
    MERGE cart_items AS target
    USING (SELECT @v_cart_id, @p_variant_id) AS source (cart_id, variant_id)
    ON (target.cart_id = source.cart_id AND target.variant_id = source.variant_id)
    WHEN MATCHED THEN
        UPDATE SET quantity = target.quantity + @p_quantity
    WHEN NOT MATCHED THEN
        INSERT (cart_id, variant_id, quantity)
        VALUES (@v_cart_id, @p_variant_id, @p_quantity);

    SELECT 'SUCCESS: Item added to cart.' AS result;
END;
GO
```

Execution result:

```
DECLARE @TestUserID INT = 7; -- John Doe

-- Test 3.1: Add Item to Cart
-- Add 2 units of Variant 1 (White T-Shirt M, Stock 100)
PRINT '>> Test 3.1: Add 2 units of Variant 1 to Cart...';
EXEC cart_add_item @p_user_id = @TestUserID, @p_variant_id = 1, @p_quantity = 2;

-- Test 3.2: Add another item
-- Add 1 unit of Variant 5 (Red Dress S, Stock 20)
PRINT '>> Test 3.2: Add 1 unit of Variant 5 to Cart...';
EXEC cart_add_item @p_user_id = @TestUserID, @p_variant_id = 5, @p_quantity = 1;
```

86 % — No issues found

Results | Messages

	result
1	SUCCESS: Item added to cart.

	result
1	SUCCESS: Item added to cart.

```
select * from cart_items where cart_id = 3
```

105 % — No issues found

Results | Messages

	id	cart_id	variant_id	quantity
1	5	3	1	4
2	6	3	5	1

1.3. Secure Checkout Transaction (ACID)

Procedure Name: checkout

Description: This critical stored procedure orchestrates the entire transition from a digital shopping cart to a finalized order. It is designed with a "Safety-First" approach, ensuring that payment, inventory, and promotional data remain consistent even under high-concurrency pressure.

Technical Highlights:

- **Atomic Transactions (ACID Compliance):** The entire process is wrapped in a `BEGIN TRANSACTION` block with `XACT_ABORT ON`. This guarantees that if any step fails (e.g., inventory running out mid-process), all previous changes are rolled back, preventing "partial orders" or data corruption.
- **Immutable Address Snapshot:** Instead of just linking an ID, the procedure concatenates the user's current address details (Name, Detail, Ward, District, City, Phone) into a single string (`@v_address_snapshot`). This preserves the exact delivery location at the time of purchase, even if the user later updates their address book.
- **Row-Level Concurrency Control:** By performing updates on `product_variants` and `vouchers` within a transaction, the system applies locks that prevent race

conditions—ensuring that the last item in stock is never sold to two users simultaneously.

- **Automated Calculation Engine:** Dynamically calculates the subtotal, applies shipping fees, and processes complex discount logic (Percentage vs. Fixed Amount) to derive the final payable total.

Source code:

```
CREATE OR ALTER PROCEDURE checkout
    @p_user_id INT,
    @p_address_id INT,
    @p_payment_method_id INT,
    @p_voucher_id INT = NULL
AS
BEGIN
    SET NOCOUNT ON;
    SET XACT_ABORT ON; -- Rollback immediately on error

    BEGIN TRY
        BEGIN TRANSACTION;

        DECLARE @v_cart_id INT;
        DECLARE @v_address_snapshot NVARCHAR(MAX);
        DECLARE @v_payment_name NVARCHAR(50);

        DECLARE @v_total_amount DECIMAL(10,2) = 0;
        DECLARE @v_shipping_fee DECIMAL(10,2) = 15000.00;
        DECLARE @v_discount_amount DECIMAL(10,2) = 0;
        DECLARE @v_final_amount DECIMAL(10,2);
        DECLARE @v_new_order_id INT;

        -- 1. Validate Address
        SELECT @v_address_snapshot = CONCAT(recipient_name, ', ', detail, ', ', ward, ', ', district, ',
', city, ' - Tel: ', phone)
        FROM addresses
        WHERE id = @p_address_id AND user_id = @p_user_id;

        IF @v_address_snapshot IS NULL THROW 50021, 'Invalid Address!', 1;

        -- 2. Validate Payment
        SELECT @v_payment_name = name FROM payment_methods WHERE id = @p_payment_method_id AND is_active =
1;
        IF @v_payment_name IS NULL THROW 50022, 'Invalid Payment Method!', 1;

        -- 3. Get Cart
        SELECT @v_cart_id = id FROM carts WHERE user_id = @p_user_id; -- Add WITH (UPDLOCK) if strict
concurrency needed
        IF @v_cart_id IS NULL THROW 50023, 'Cart not found!', 1;

        IF NOT EXISTS (SELECT 1 FROM cart_items WHERE cart_id = @v_cart_id)
            THROW 50024, 'Cart is empty!', 1;

        -- 4. Stock Check & Total Calculation
        DECLARE @CartContent TABLE (
            variant_id INT,
            quantity INT,
            price DECIMAL(10,2),
            stock INT,
            is_active BIT
        );

        INSERT INTO @CartContent
        SELECT ci.variant_id, ci.quantity, p.price, pv.stock, p.is_active
        FROM cart_items ci
        JOIN product_variants pv ON ci.variant_id = pv.id
        JOIN products p ON pv.product_id = p.id
        WHERE ci.cart_id = @v_cart_id;

        -- Validate items
        IF EXISTS (SELECT 1 FROM @CartContent WHERE is_active = 0)
            THROW 50025, 'Some products are not valid for sale.', 1;

        IF EXISTS (SELECT 1 FROM @CartContent WHERE quantity > stock)
```

```
            THROW 50026, 'Some products are out of stock.', 1;

        SELECT @v_total_amount = SUM(quantity * price) FROM @CartContent;

        -- 5. Process Voucher
        IF @p_voucher_id IS NOT NULL
        BEGIN
            DECLARE @v_val DECIMAL(10,2), @v_type NVARCHAR(20), @v_uv_id INT;

            SELECT @v_val = v.value, @v_type = v.type, @v_uv_id = uv.id
            FROM vouchers v
            JOIN user_vouchers uv ON v.id = uv.voucher_id
            WHERE v.id = @p_voucher_id
              AND uv.user_id = @p_user_id
              AND uv.is_used = 0
              AND v.is_active = 1
              AND GETDATE() BETWEEN v.start_date AND v.end_date
              AND v.stock > 0;

            IF @v_val IS NULL THROW 50027, 'Voucher invalid or cannot apply.', 1;

            IF @v_type = 'PERCENTAGE'
                SET @v_discount_amount = @v_total_amount * (@v_val / 100);
            ELSE
                SET @v_discount_amount = @v_val;

            IF @v_discount_amount > @v_total_amount SET @v_discount_amount = @v_total_amount;
        END

        -- 6. Final Calculation
        SET @v_final_amount = @v_total_amount + @v_shipping_fee - @v_discount_amount;
        IF @v_final_amount < 0 SET @v_final_amount = 0;

        -- 7. Create Order
        INSERT INTO orders (
            user_id, shipping_address, payment_method, voucher_id,
            status, total_amount, shipping_fee, discount_amount, final_amount,
            payment_status, created_at
        ) VALUES (
            @p_user_id, @v_address_snapshot, @v_payment_name, @p_voucher_id,
            'PENDING', @v_total_amount, @v_shipping_fee, @v_discount_amount, @v_final_amount,
            'UNPAID', GETDATE()
        );
        SET @v_new_order_id = SCOPE_IDENTITY();

        -- 8. Save Order Items
        INSERT INTO order_items (order_id, variant_id, quantity, price)
        SELECT @v_new_order_id, variant_id, quantity, price FROM @CartContent;

        -- 9. Update Inventory
        UPDATE pv
        SET pv.stock = pv.stock - cc.quantity
        FROM product_variants pv
        JOIN @CartContent cc ON pv.id = cc.variant_id;

        -- 10. Update Voucher
        IF @p_voucher_id IS NOT NULL
        BEGIN
            UPDATE user_vouchers SET is_used = 1, claimed_at = GETDATE() WHERE id = @v_uv_id;
            UPDATE vouchers SET stock = stock - 1, used_count = used_count + 1 WHERE id = @p_voucher_id;
        END

        -- 11. Clear Cart
        DELETE FROM cart_items WHERE cart_id = @v_cart_id;

        COMMIT TRANSACTION;

        SELECT 'SUCCESS' as status, @v_new_order_id as order_id, @v_final_amount as final_amount
        FOR JSON PATH, WITHOUT_ARRAY_WRAPPER;
    END TRY
    BEGIN CATCH
        IF @@TRANCOUNT > 0 ROLLBACK TRANSACTION;
        THROW;
    END CATCH
END;
GO
```

Execution result:

```
DECLARE @TestUserID INT = 7;
-- Run Checkout
BEGIN TRY
    EXEC checkout
        @p_user_id = @TestUserID,
        @p_address_id = 15,
        @p_payment_method_id = 2

END TRY
BEGIN CATCH
    PRINT 'Checkout Error: ' + ERROR_MESSAGE();
END CATCH;
```

79 % — No issues found

Results | Messages

	JSON_F52E2B61-18A1-11d1-B105-00805F49916B
1	{"status":"SUCCESS","order_id":4,"final_amount":...

1.4. Order Cancellation & Stock Reversion

Procedure Name: cancel_order

Description: This procedure allows users to cancel their orders while ensuring a full and automated "rollback" of physical and digital resources. It focuses on restoring system state to maintain accurate inventory and promotional data.

Technical Highlights:

- **Transactional Integrity:** Operates within a `TRY...CATCH` block with `XACT_ABORT ON`. This ensures that the status update and the resource restoration (restocking/voucher return) occur as an indivisible unit.

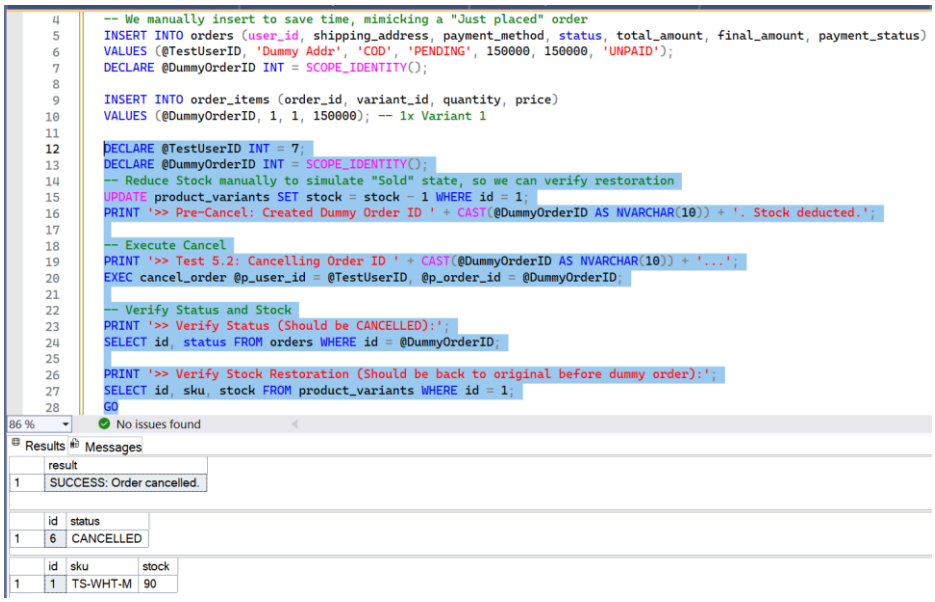
- **Strict State Validation:** Implements business rule enforcement by checking the current order status using `WITH (UPDLOCK)`. It only permits cancellation if the order is in a **'PENDING'** state, preventing the cancellation of orders that are already shipping or completed.
- **Automated Inventory Restocking:** Seamlessly returns product quantities to the warehouse. By joining `order_items` with `product_variants`, the procedure increments the stock levels for every SKU associated with the cancelled order in a single bulk update.
- **Promotion Restoration:** Protects customer benefits by reverting voucher usage. The procedure increments the global voucher stock and decrements the usage counter, effectively allowing the promotional code to be available for future use.

Source Code:

```
CREATE OR ALTER PROCEDURE cancel_order
    @p_user_id INT,
    @p_order_id INT
AS
BEGIN
    SET NOCOUNT ON;
    SET XACT_ABORT ON;
```

```
    BEGIN TRY
        BEGIN TRANSACTION;

        DECLARE @v_current_status NVARCHAR(20);
        SELECT @v_current_status = status FROM orders WITH (UPDLOCK) WHERE id = @p_order_id AND
user_id = @p_user_id;

        IF @v_current_status IS NULL THROW 50028, 'Order not found', 1;
        IF @v_current_status <> 'PENDING' THROW 50029, 'Cannot cancel non-pending order', 1;

        UPDATE orders SET status = 'CANCELLED', updated_at = GETDATE() WHERE id = @p_order_id;

        -- Restock
        UPDATE pv
        SET pv.stock = pv.stock + oi.quantity
        FROM product_variants pv
        JOIN order_items oi ON pv.id = oi.variant_id
        WHERE oi.order_id = @p_order_id;

        -- Restore Voucher
        UPDATE v
        SET v.stock = v.stock + 1, v.used_count = v.used_count - 1
        FROM vouchers v
        JOIN orders o ON o.voucher_id = v.id
        WHERE o.id = @p_order_id;

        COMMIT TRANSACTION;
        SELECT 'SUCCESS: Order cancelled.' AS result;
    END TRY
    BEGIN CATCH
        IF @@TRANCOUNT > 0 ROLLBACK TRANSACTION;
        THROW;
    END CATCH
END;

GO
```

Execution result:



2. Admin Module

2.1. Flexible Promotion Management (Upsert Pattern)

Procedure Name: upsert_voucher

Description: This stored procedure provides a unified interface for managing promotional vouchers. It intelligently handles both the creation of new discount codes and the modification of existing ones within a single transactional flow.

Technical Highlights:

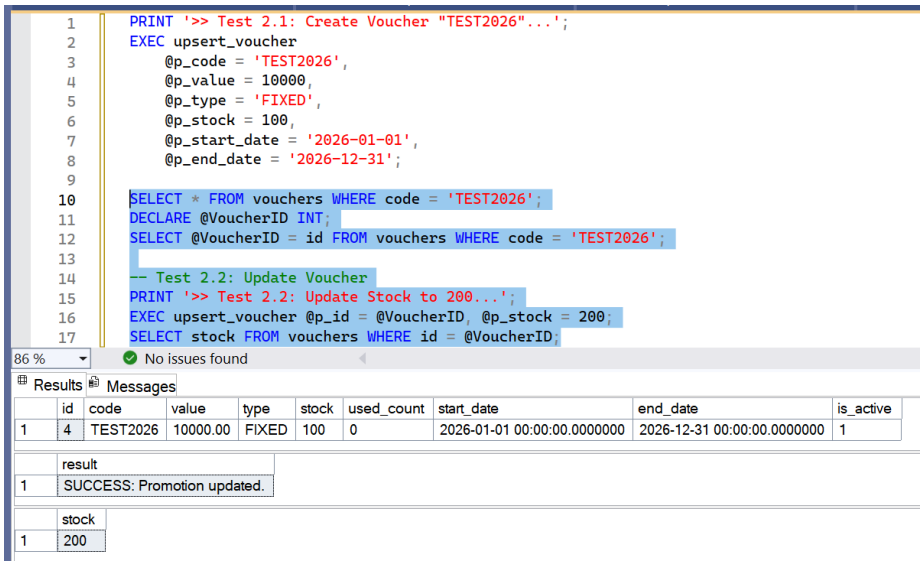
- **Upsert Logic:** The procedure uses the presence of `@p_id` to determine the operational path. If `@p_id` is `NULL`, it initiates an `INSERT` sequence; otherwise, it performs an `UPDATE` on the target record.
- **Flexible Data Patching:** In **Update mode**, the procedure utilizes `ISNULL` logic for each column. This allows developers to perform "partial updates"—sending only the fields that need changing while preserving the existing values for all other columns.
- **Conflict Prevention:** Before inserting a new record, the procedure performs an existence check on the `@p_code`. This ensures the uniqueness of voucher codes across the system, preventing data duplication and logic errors in the checkout process.
- **State Validation & Error Handling:** * **Existence Guard:** During an update, it verifies if the provided `@p_id` actually exists using `@@ROWCOUNT`. If no record is found, it raises a custom error to prevent silent failures.

Source Code:

```
CREATE OR ALTER PROCEDURE upsert_voucher
    @p_id INT = NULL,
    @p_code NVARCHAR(50) = NULL,
    @p_value DECIMAL(10,2) = 0,
    @p_type NVARCHAR(20) = 'FIXED',
    @p_stock INT = 0,
    @p_start_date DATETIME2 = NULL,
    @p_end_date DATETIME2 = NULL,
    @p_is_active BIT = 1
AS
BEGIN
    SET NOCOUNT ON;

    IF @p_id IS NULL
    BEGIN
        IF EXISTS (SELECT 1 FROM vouchers WHERE code = @p_code)
            THROW 50005, 'Voucher code already exists!', 1;

        INSERT INTO vouchers (code, value, type, stock, start_date, end_date, is_active)
        VALUES (@p_code, @p_value, @p_type, @p_stock, @p_start_date, @p_end_date, @p_is_active);

        SELECT 'SUCCESS: Promotion created.' AS result;
    END
    ELSE
    BEGIN
        UPDATE vouchers
        SET code = ISNULL(@p_code, code),
            value = ISNULL(@p_value, value),
            type = ISNULL(@p_type, type),
            stock = ISNULL(@p_stock, stock),
            start_date = ISNULL(@p_start_date, start_date),
            end_date = ISNULL(@p_end_date, end_date),
            is_active = ISNULL(@p_is_active, is_active)
        WHERE id = @p_id;
```

```
        IF @@ROWCOUNT = 0 THROW 50006, 'Voucher ID not found', 1;

        SELECT 'SUCCESS: Promotion updated.' AS result;
    END
END;
GO
```

Execution result:



2.2. Automated Order Lifecycle & Inventory Rollback

Procedure Name: update_order_status

Description: This stored procedure serves as the primary engine for managing order lifecycle transitions. Beyond simply changing a status label, it acts as an intelligent controller that triggers automated inventory adjustments and financial state synchronization based on administrative actions.

Technical Highlights:

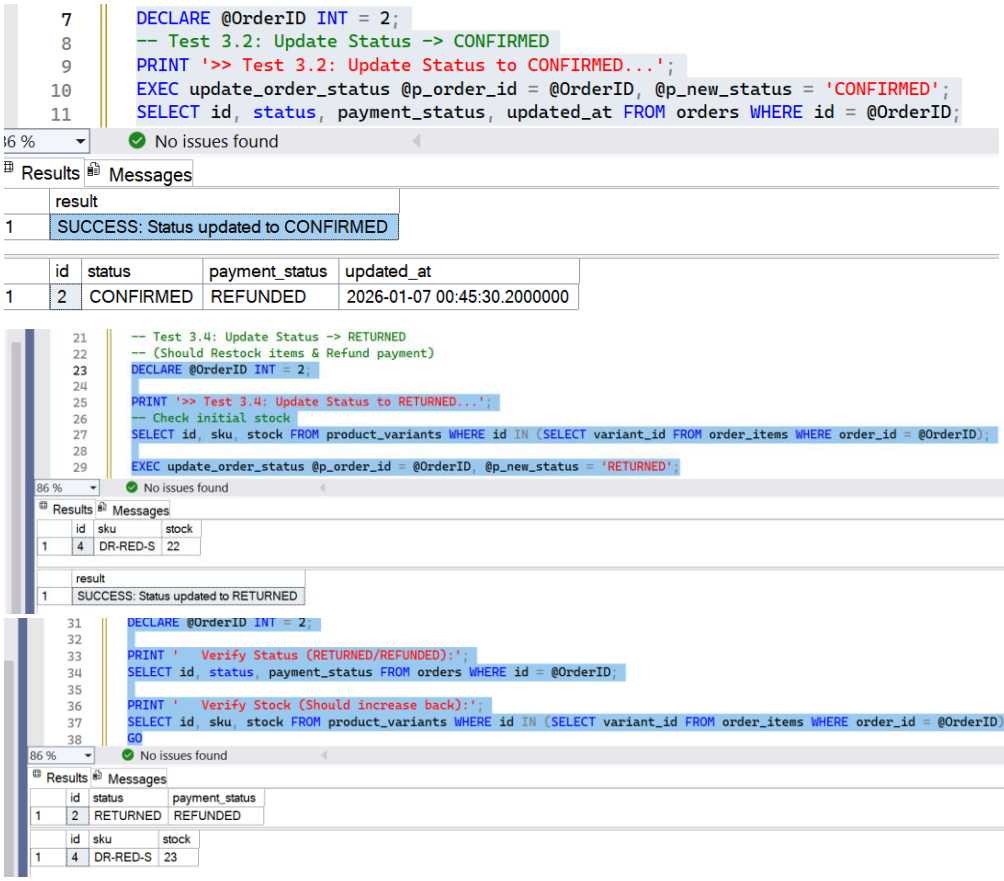
- **Concurrency & Row Locking:** Utilizes the `WITH (UPDLOCK)` hint (the T-SQL equivalent of `FOR UPDATE`) to secure the specific order record. This prevents race conditions where multiple administrators or automated background services might attempt to update the same order simultaneously.
- **Automated Inventory Reconciliation:** The procedure monitors state changes to ensure stock accuracy. When an order is moved to **'CANCELLED'** or **'RETURNED'**, it automatically executes a bulk update to return item quantities from `order_items` back to `product_variants`.

- **Intelligent Financial Sync:** Reduces manual accounting errors by automatically updating the `payment_status` based on the lifecycle stage:
    - Switches to **'REFUNDED'** when an order is processed as a return.
    - Marks as **'PAID'** once an order reaches the **'COMPLETED'** stage.
- **State-Change Guard:** Includes logic to ensure inventory is only restocked once (checking that the current status is not already 'CANCELLED' or 'RETURNED') before applying the stock addition, preventing duplicate inventory inflation.

Source Code:

```
CREATE OR ALTER PROCEDURE update_order_status
    @p_order_id INT,
    @p_new_status NVARCHAR(20)
AS
BEGIN
    SET NOCOUNT ON;
    DECLARE @v_current_status NVARCHAR(20);

    -- Check and Lock (Logic only, explicit lock strictness varies in SQL Server/T-SQL)
    SELECT @v_current_status = status
    FROM orders WITH (UPDLOCK) -- Equivalent to FOR UPDATE
    WHERE id = @p_order_id;

    IF @v_current_status IS NULL
        THROW 50009, 'Order ID not found', 1;

    -- Update status
    UPDATE orders
    SET status = @p_new_status, updated_at = GETDATE()
    WHERE id = @p_order_id;

    -- Restock logic
    IF @p_new_status IN ('CANCELLED', 'RETURNED')
       AND @v_current_status NOT IN ('CANCELLED', 'RETURNED')
    BEGIN
        UPDATE pv
        SET pv.stock = pv.stock + oi.quantity
        FROM product_variants pv
        JOIN order_items oi ON pv.id = oi.variant_id
        WHERE oi.order_id = @p_order_id;

        IF @p_new_status = 'RETURNED'
        BEGIN
            UPDATE orders SET payment_status = 'REFUNDED' WHERE id = @p_order_id;
        END
    END
    ELSE IF @p_new_status = 'COMPLETED'
    BEGIN
        UPDATE orders SET payment_status = 'PAID' WHERE id = @p_order_id;
    END

    SELECT CONCAT('SUCCESS: Status updated to ', @p_new_status) AS result;
END;

GO
```

Execution result:



V. Scenario-based Testing

To validate the system's end-to-end functionality, we simulated a "New User Journey". This scenario covers the complete lifecycle: creating an account, securing a session, setting up a profile, shopping, and managing an order.

Test Data Setup:

- **New Email:** flow_test_@gmail.com
- **Password:** 123456
- **Name:** Test User
- **Phone:** 0988888888

1. Step 1: User registration

Action: A new visitor registers an account with their personal details.

Objective: Verify register_user creates a new record in the users

Execution result:

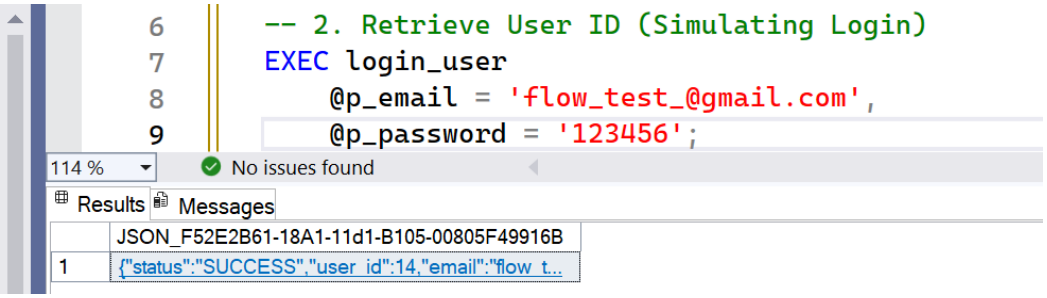
```
-- 1. Register
-- (Using generated email to avoid conflict if run multiple times)
DECLARE @Email NVARCHAR(50) = 'flow_test_@gmail.com';
EXEC register_user @Email, '123456', 'Test User', '0988888888';
```

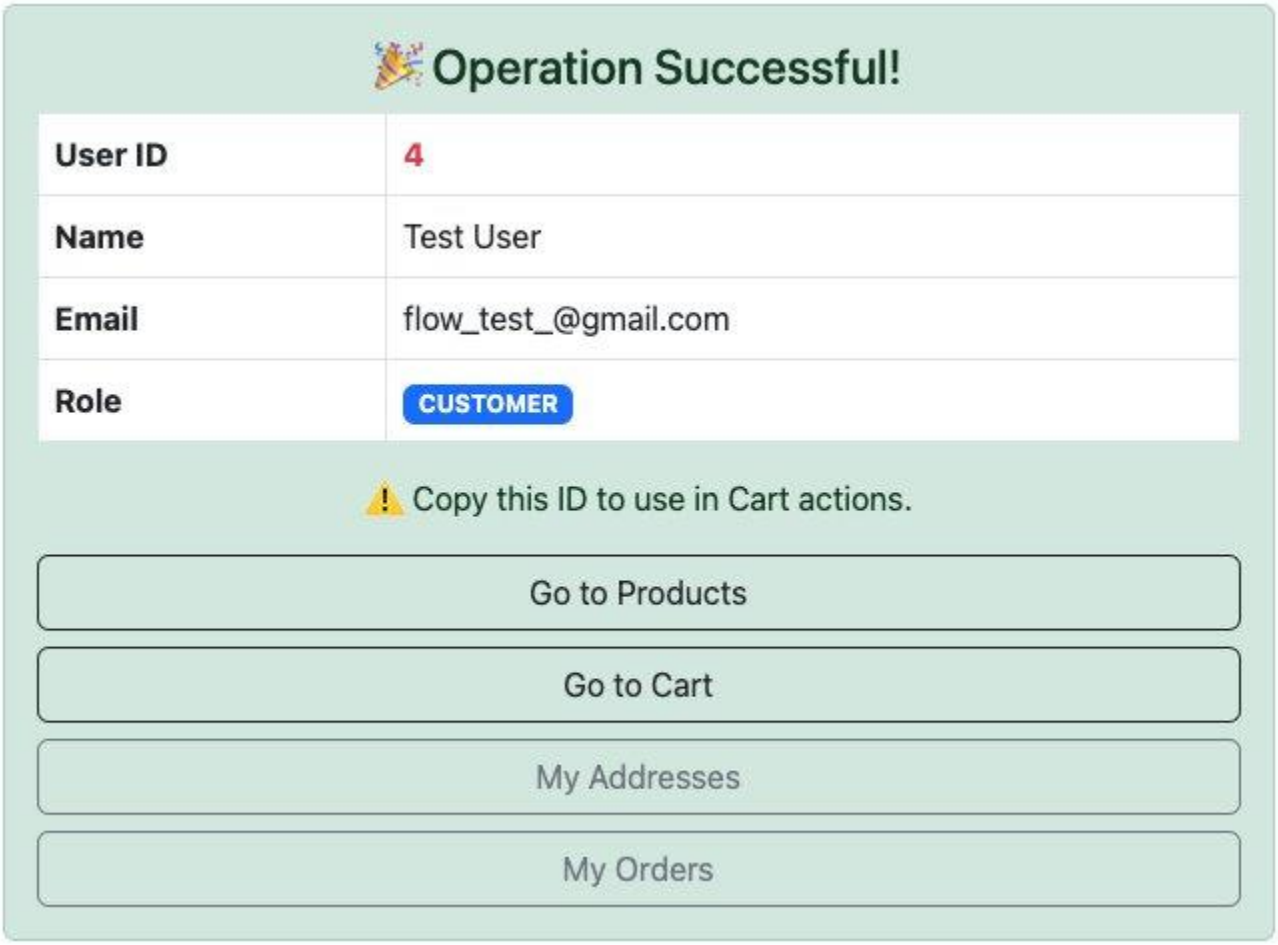
2. Step 2: Log in

Action: The user attempts to log in with the credentials created in Step 1.

Objective: Verify login_user checks credentials correctly and returns a success status.

Execution result



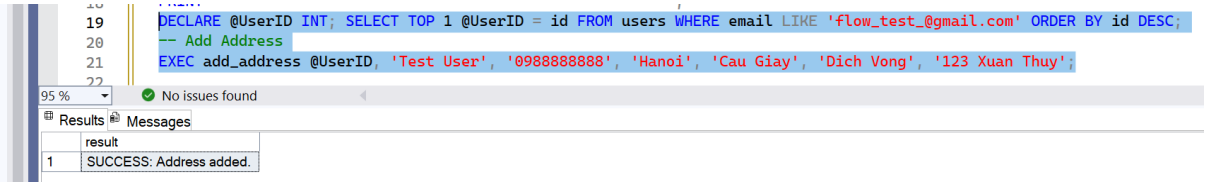


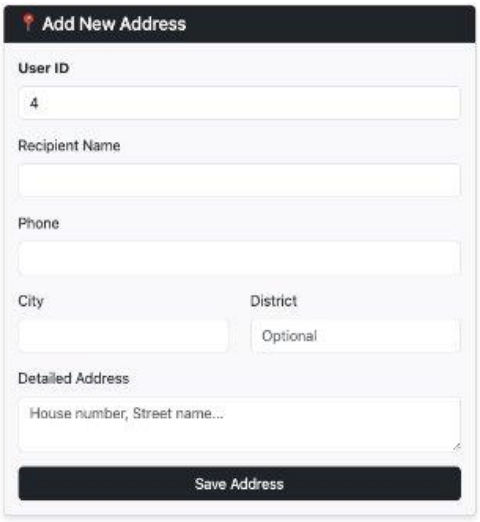
3. Step 3: Setup Shipping Address

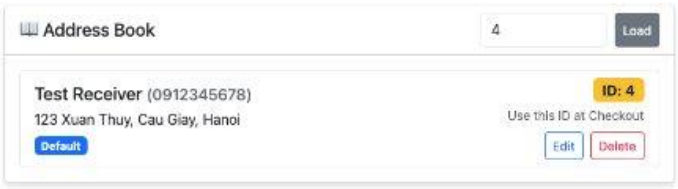
Action: Before buying, the user adds a shipping address to their profile.

Objective: Verify add_address.

Execution result:







Verification:

```
26  SELECT * FROM addresses WHERE user_id = 14;
27  GO
```

95 % — No issues found

Results | Messages

	id	user_id	recipient_name	phone	city	district	ward	detail	is_default
1	18	14	Test User	0988888888	Hanoi	Cau Giay	Dich Vong	123 Xuan Thuy	1

4. Step 4: Product Discovery

Action: The user searches for 3 TOP items

Objective: Verify browse_products returns relevant results.

Execution results:

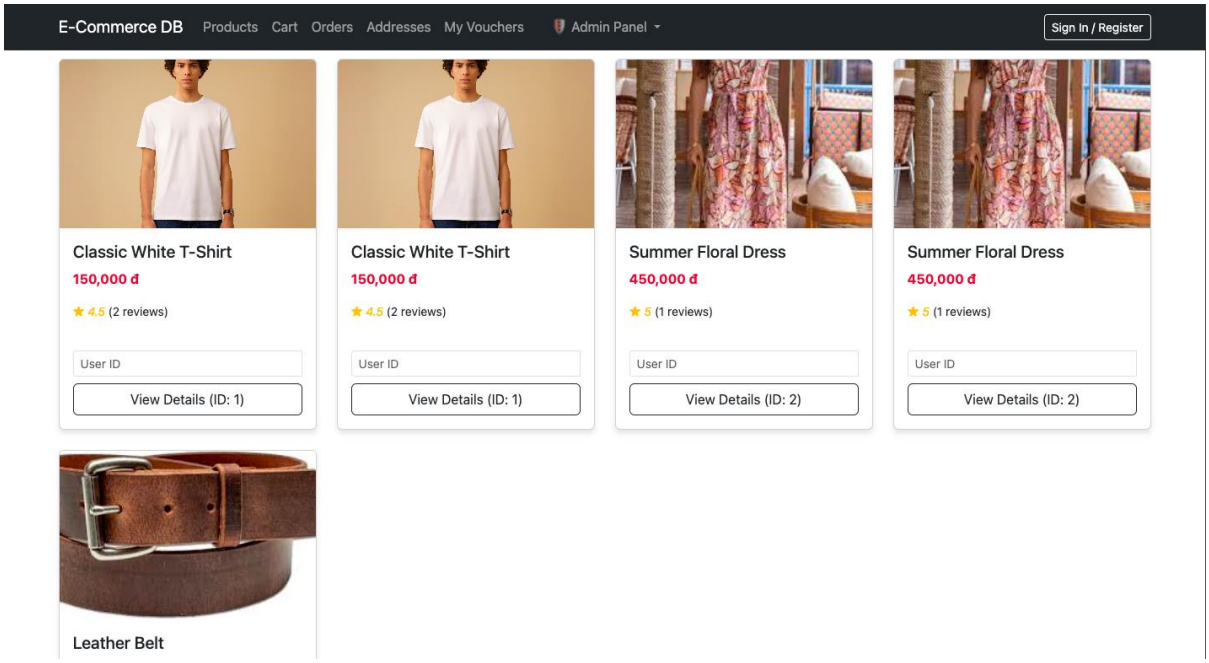
```
34  PRINT 'Browsing products...';
35  EXEC browse_products @p_limit = 3;
36
37  -- Get Detail of Product ID 1 (Assuming it exists from insert_data)
38  PRINT 'Viewing details of Product #1...';
39  EXEC get_product_details 1;
```

86 % — No issues found

Results | Messages

	product_id	product_name	original_price	current_price	thumbnail	avg_rating	total_reviews	category_name	created_at
1	7	New Admin Produc1	1000.00	800.00	https://encrypted-tbn0.gstatic.com/images?q=tbn:A...	0	0	Men	2026-01-06 23:07:16.7633333
2	7	New Admin Produc1	1000.00	800.00	https://encrypted-tbn0.gstatic.com/images?q=tbn:A...	0	0	T-Shirts	2026-01-06 23:07:16.7633333
3	6	Classi White T-Shirt	200000.00	150000.00	https://dosi-in.com/images/detailed/42/CDL11_1.jpg	4.5	2	NULL	2026-01-06 22:57:46.8666667

	JSON_F52E2B61-18A1-11d1-B105-00805F49916B
1	{"info":{"id":1,"name":"Classic White T-Shirt","slu...



5. Step 5: Add to Cart

Action: The user adds items with the first variant and quantity is 2

Objective: Verify cart_add_item logic.

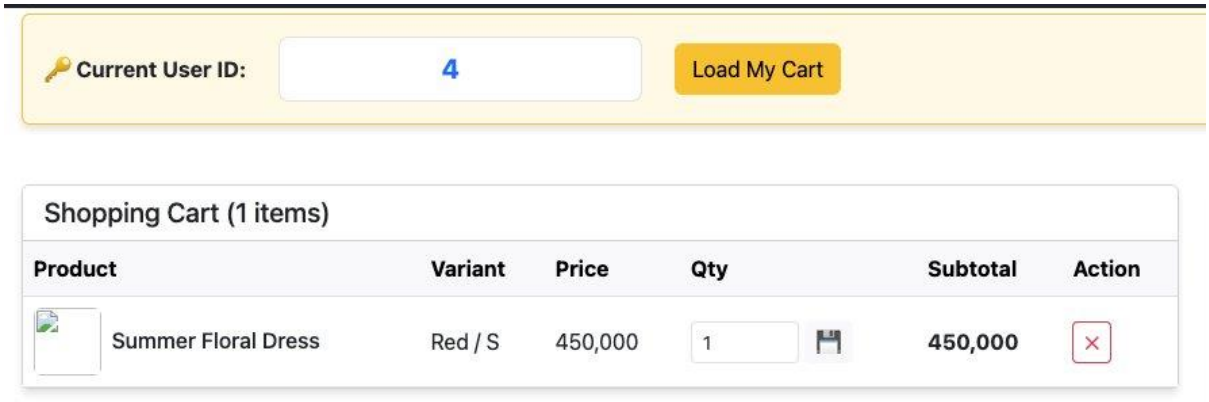
Execution result:

```
49  DECLARE @UserID INT; SELECT TOP 1 @UserID = id FROM users WHERE email LIKE 'flow_test_@gmail.com' ORDER BY id DESC;
50  -- Add 2 items of Variant ID 1 (White T-Shirt M)
51  EXEC cart_add_item @UserID, 1, 2;
```

86 % No issues found

Results Messages

	result
1	SUCCESS: Item added to cart.



Verification:

```
55  DECLARE @UserID INT; SELECT TOP 1 @UserID = id FROM users WHERE email LIKE 'flow_test_@gmail.com' ORDER BY id DESC;
56  SELECT *
57  FROM cart_items ci JOIN carts c ON ci.cart_id = c.id
58  WHERE c.user_id = @UserID;
59  GO
60
```

86 % — No issues found

Results | Messages

	id	cart_id	variant_id	quantity	id	user_id	updated_at
1	7	4	1	2	4	14	2026-01-07 02:18:15.5333333

6. Step 6: Checkout Transaction

Action: The user places the order using the address created in Step 3.

Objective: Verify checkout creates the order, deducts stock, and clears the cart.

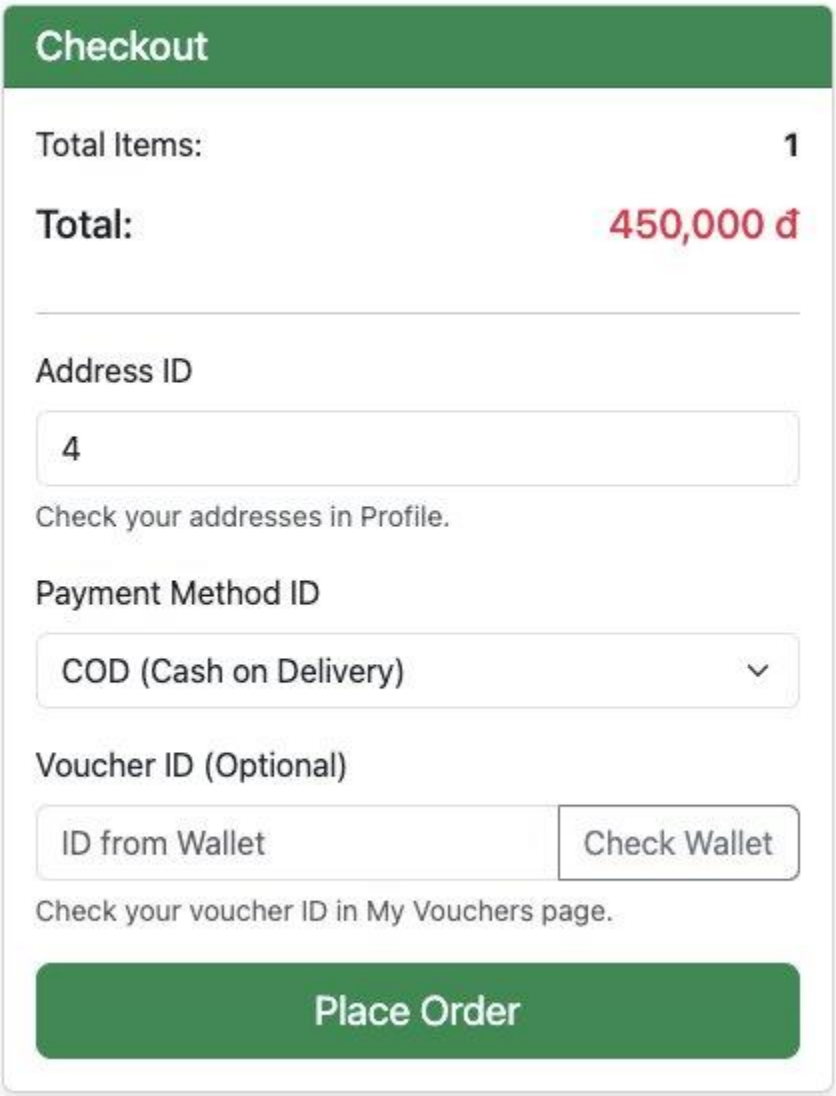
Execution result:

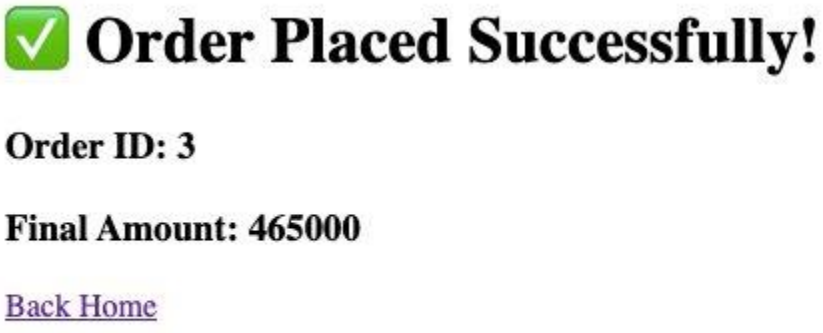
```
65  DECLARE @UserID INT; SELECT TOP 1 @UserID = id FROM users WHERE email LIKE 'flow_test_@gmail.com' ORDER BY id DESC;
66  DECLARE @AddrID INT; SELECT TOP 1 @AddrID = id FROM addresses WHERE user_id = @UserID;
67  DECLARE @PayID INT = 2; -- COD
68
69  -- Checkout
70  EXEC checkout @UserID, @AddrID, @PayID, NULL; -- No voucher
```

78 % — No issues found

Results | Messages

	JSON_F52E2B61-18A1-11d1-B105-00805F49916B
1	{"status":"SUCCESS","order_id":7,"final_amount":...





Verification:

- The status of payment is unpaid

```
74  DECLARE @UserID INT; SELECT TOP 1 @UserID = id FROM users WHERE email LIKE 'flow_test_@gmail.com' ORDER BY id DESC;
75  DECLARE @AddrID INT; SELECT TOP 1 @AddrID = id FROM addresses WHERE user_id = @UserID;
76  DECLARE @PayID INT = 2; -- COD
77  SELECT TOP 1 id, status, total_amount, payment_status, created_at FROM orders WHERE user_id = @UserID ORDER BY id DESC;
78  GO
```

78 % — No issues found

Results | Messages

	id	status	total_amount	payment_status	created_at
1	7	PENDING	300000.00	UNPAID	2026-01-07 02:21:03.1466667

- Current Stock after Ordered (90-2 = 88)

```
89  DECLARE @Stock INT; SELECT @Stock = stock FROM product_variants WHERE id = 1;
90  PRINT CONCAT('Current Stock: ', @Stock);
```

No issues found

Messages

```
Current Stock: 88
```

7. Step 7: Order Tracking

Action: The user checks their order history to confirm the purchase.

Objective: Verify view_order_history.

Execution result:

```
84  DECLARE @UserID INT; SELECT TOP 1 @UserID = id FROM users WHERE email LIKE 'flow_test_@gmail.com' ORDER BY id DESC;
85
86  -- View my orders
87  EXEC view_order_history @UserID;
```

78 % — No issues found

Results | Messages

	order_id	shipping_address	payment_method	status	total_items	final_amount	payment_status	created_at
1	7	Test User, 123 Xuan Thuy, Dich Vong, Cau Giay, H...	Cash on Delivery	PENDING	1	315000.00	UNPAID	2026-01-07 02:21:03.1466667

8. Step 8: Order Cancellation

Action: The user decides to cancel the order immediately.

Objective: Verify cancel_order restores stock and updates status.

Execution result:

```
97  DECLARE @UserID INT; SELECT TOP 1 @UserID = id FROM users WHERE email LIKE 'flow_test_@gmail.com' ORDER BY id DESC;
98  DECLARE @OrderID INT; SELECT TOP 1 @OrderID = id FROM orders WHERE user_id = @UserID ORDER BY id DESC;
99
100 -- Create a snapshot of stock before cancel
101 DECLARE @StockBefore INT; SELECT @StockBefore = stock FROM product_variants WHERE id = 1;
102 PRINT CONCAT('Stock Before Cancel: ', @StockBefore);
103
104 -- Perform Cancel
105 EXEC cancel_order @UserID, @OrderID;
```

78 % — No issues found

Results | Messages

	result
1	SUCCESS: Order cancelled.

Verification:

- Variant with ID 1 must be restock back to 90 (88+2 = 90)

```
109  DECLARE @UserID INT; SELECT TOP 1 @UserID = id FROM users WHERE email LIKE 'flow_test_@gmail.com' ORDER BY id DESC;
110  DECLARE @OrderID INT; SELECT TOP 1 @OrderID = id FROM orders WHERE user_id = @UserID ORDER BY id DESC;
111  SELECT id, status, updated_at FROM orders WHERE id = @OrderID;
112
113  DECLARE @StockAfter INT; SELECT @StockAfter = stock FROM product_variants WHERE id = 1;
114  PRINT CONCAT('Stock After Cancel: ', @StockAfter);
115  PRINT 'Expected Check 2: Stock After should be (Stock Before + 2).';
116  GO
```

78 % No issues found

Results Messages

```
(1 row affected)
Stock After Cancel: 90
```

- Status of order with ID 7 is CANCELLED

Results Messages

	id	status	updated_at
1	7	CANCELLED	2026-01-07 02:26:41.8033333

VI. Perfomance Optimization & Benchmarking

To ensure the system remains responsive, we have implemented a comprehensive indexing strategy using **13 specific Non-Clustered Indexes**. These target high-volume query patterns, typically using INCLUDE columns to act as Covering Indexes.

1. Indexes supporting customer features

These indexes directly optimize **user experience (UX)**, **response time**, and **critical customer workflows** such as browsing products, cart operations, and checkout.

Table	Index Name	Columns (Key + Include)	Optimization Goal
Products	`idx_products_active_newest`	Key: `(is_active, created_at DESC)` Include: `name,`	**Homepage / Browse**: Allows instant retrieval of

		price, thumbnail, rating	"Newest Active Products" without touching the main heap.
ProductCategories	idx_product_categories_category_id	Key: (category_id) Include: product_id	**Category Filter**: Accelerates "Show all Men's Shirts" queries.
Products	idx_products_price	Key: (price) Include: name, thumbnail	**Price Filter**: Optimizes "Price Range" searches and "Sort by Price".
Products	idx_products_name	Key: (name)	**Name Search**: Supports basic keyword search on product names.
ProductVariants	idx_product_variants_product	Key: (product_id) Include: color, size, stock	**Product Detail**: Fetches all available sizes/colors instantly when viewing a single product.
Reviews	idx_reviews_product_created	Key: (product_id, created_at	**Review Display**:

		DESC) Include: rating, comment	Loads latest reviews for a product without sorting cost.
Orders	idx_orders_user_created	Key: (user_id, created_at DESC) Include: status, total_amount	**My Orders**: Instant access to user's order history sorted by newest.
CartItems	idx_cart_items_cart_id	Key: (cart_id) Include: variant_id, quantity	**Cart View**: Rapidly joins cart items with product info.

2. Indexes supporting admin features

These indexes are designed to handle heavy data aggregation and management tasks without causing server timeouts.

Table	Index Name	Columns (Key + Include)	Optimization Goal
Orders	idx_orders_created_at	Key: (created_at) Include: status, final_amount	**Revenue Reports**: Scans order dates for daily sales reports without full table scan.
Orders	idx_orders_status	Key: (status) Include: user_id, total_amount	**Order Mgmt**: Quickly filters "PENDING" or "SHIPPING" orders for fulfillment workflow.

SupportMessages	`idx_support_messages_user`	Key: `(user_id)` Include: `content`, `status`	**Support History**: Quick lookup of a specific user's ticket history.
ProductCategories	`idx_product_categories_cat`	Key: `(category_id)` Include: `product_id`	**Analytics (Cat)**: Optimized for "Revenue by Category" reporting queries.
OrderItems	`idx_order_items_variant_id`	Key: `(variant_id)` Include: `quantity`, `price`	**Best Sellers**: Aggregates sales data per variant efficiently.

3. Benchmark Results

To validate the effectiveness of our indexing strategy, we executed performance tests on a dataset of **50,000 Products** and **100,000 Orders** . The metrics collected confirm a significant reduction in resource consumption (Logical IO/CPU) after applying the indexes.

3.1. Index For Products (Name)

Source code:

```
CREATE NONCLUSTERED INDEX idx_products_name
ON products(name);

SELECT id, name, price
FROM products
WHERE name LIKE 'Bulk Product 44%';
```

Before indexing:

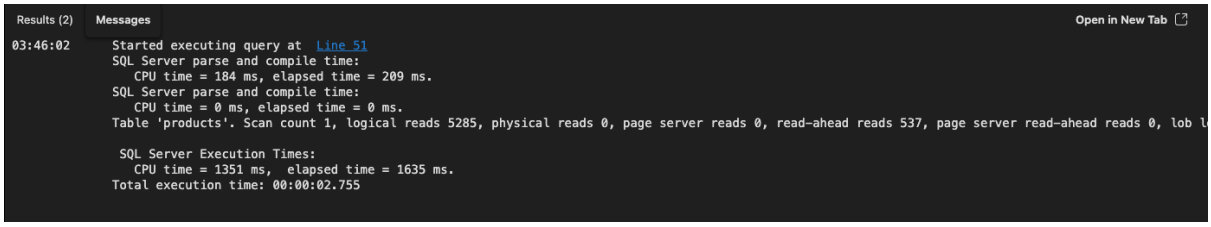
After indexing:

3.2. Index For Products (Price)

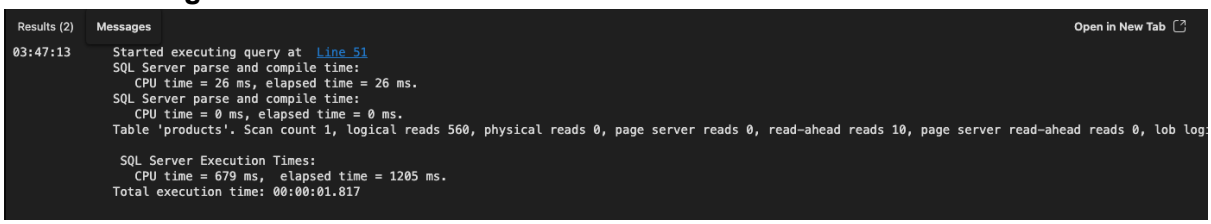
Sources code:

```
CREATE NONCLUSTERED INDEX idx_products_price
ON products(price)
INCLUDE (name, thumbnail);

SELECT id, name, price, thumbnail
FROM products
WHERE price BETWEEN 200000 AND 500000;
```

Before indexing:



After indexing:

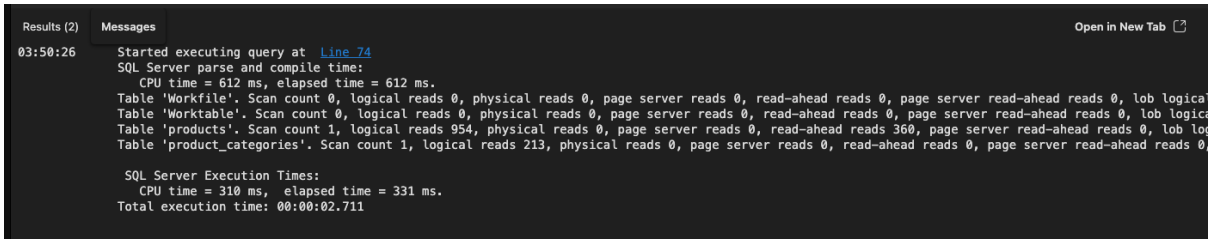


3.3. Index For Product Categories (category_id)

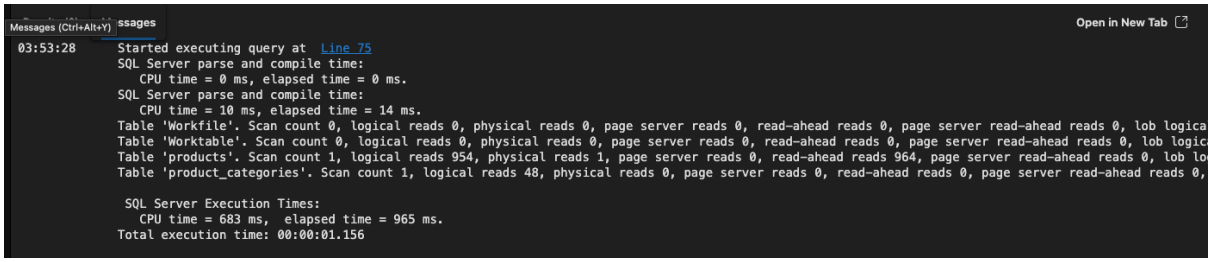
Source code:

```
CREATE NONCLUSTERED INDEX idx_product_categories_category_id
ON product_categories(category_id)
INCLUDE (product_id);

SELECT p.id, p.name
FROM products p
JOIN product_categories pc ON p.id = pc.product_id
WHERE pc.category_id = 3;
```

Before indexing:



After indexing:

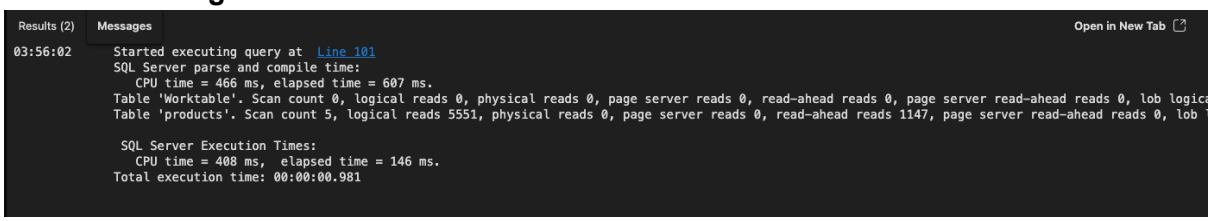


3.4. Index For Products (active newest)

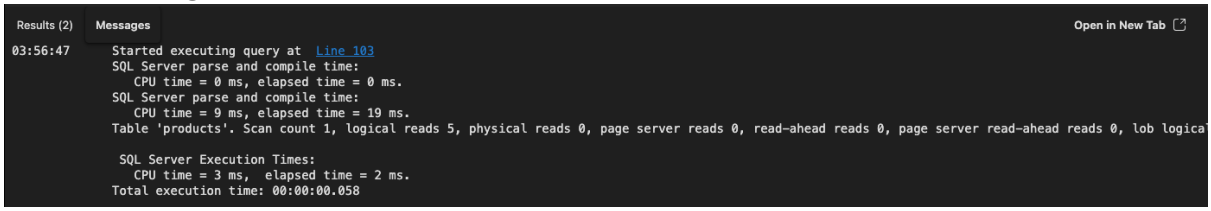
Source code:

```
CREATE NONCLUSTERED INDEX idx_products_active_newest
ON products(is_active, created_at DESC)
INCLUDE (name, price, thumbnail, rating, review_count);

SELECT TOP 20 name, price, rating
FROM products
WHERE is_active = 1
ORDER BY created_at DESC;
GO
```

Before indexing:



After indexing:



3.5. Index For Product Variants (product_id)

Source code:

```
CREATE NONCLUSTERED INDEX idx_product_variants_product
ON product_variants(product_id)
INCLUDE (color, size, stock, image);

SELECT color, size, stock
FROM product_variants
WHERE product_id = 100;
```

```
GO
```

Before indexing:

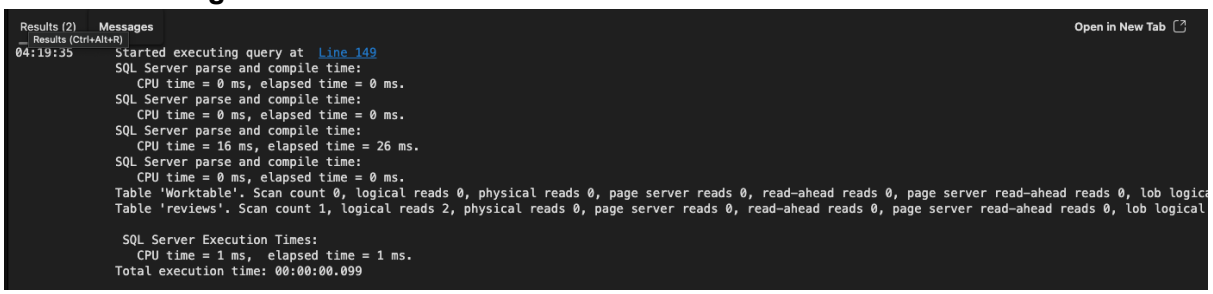
After indexing:

3.6. Index For Reviews (product_id)

Source code:

```
CREATE NONCLUSTERED INDEX idx_reviews_product_created
ON reviews(product_id, created_at DESC)
INCLUDE (rating, comment, user_id);

SELECT rating, comment
FROM reviews
WHERE product_id = 1
ORDER BY created_at DESC;
```

Before indexing:



After indexing:

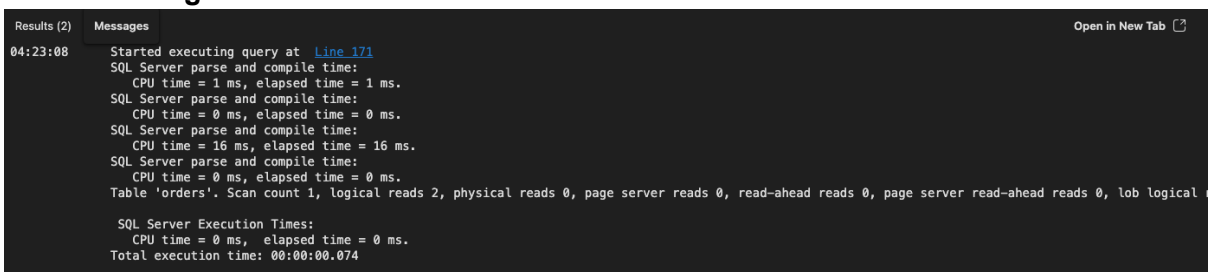
3.7. Index For Orders (status)

Source code:

```
CREATE NONCLUSTERED INDEX idx_orders_status
ON orders(status)
INCLUDE (user_id, total_amount, created_at);

SELECT COUNT(*), SUM(total_amount)
FROM orders WHERE status = 'PENDING';
```

Before indexing:

After indexing:



3.8. Index For Orders (created_at)

Source code:

```
CREATE NONCLUSTERED INDEX idx_orders_created_at
ON orders(created_at)
INCLUDE (status, final_amount);

SELECT SUM(final_amount)
FROM orders
WHERE created_at > DATEADD(day, -30, GETDATE());
```

Before indexing:

After indexing:

VII. Triggers

Trigger Name	Target Table	Fires On	Description
`trg_prevent_negative_stock`	`product_variants`	AFTER UPDATE	**Purpose:** Acts as a final safety net to ensure inventory stock never falls below zero, protecting data integrity even when direct SQL updates bypass stored procedures. **Why Needed:** The `checkout` procedure deducts stock during order placement, and `cancel_order` restores stock upon cancellation. However, administrators may run manual UPDATE statements or external systems may modify stock directly. Without this trigger, such operations could result in negative stock values, corrupting inventory data. **Mechanism:** After any UPDATE on `product_variants`, the trigger checks if any row in the `inserted` virtual table has `stock < 0`. If detected, it immediately raises an

			error and executes ROLLBACK TRANSACTION, reverting all changes within the current transaction scope.
trg_audit_order_status_change	orders	AFTER UPDATE	**Purpose:** Provides real-time visibility of order status transitions during development and debugging. **Why Needed:** Stored procedures like cancel_order and update_order_status modify the status and payment_status fields, but they do not output any confirmation when changes occur. This trigger displays the transition details immediately in the Messages tab for monitoring purposes. **Mechanism:** After any UPDATE on orders, the trigger compares values in the inserted (new values) and deleted (old values) virtual tables. If status or payment_status differs between old and new, it prints a formatted message to the console showing:

			AUDIT: Order #[order_id] status changed: [old_status] -> [new_status] | Payment: [old_payment] -> [new_payment]
trg_address_single_default	addresses	AFTER INSERT, UPDATE	**Purpose:** Ensures each user has exactly one default shipping address, preventing data inconsistency in the checkout flow. **Why Needed:** The update_address procedure handles the case when a user updates an existing address to be default (it unsets other defaults). However, the add_address procedure only auto-sets default if the user has no addresses; it does NOT handle the case when a new address is inserted with is_default=1 explicitly. This trigger covers that gap. **Mechanism:** After INSERT or UPDATE on addresses, the trigger checks if any row in inserted has is_default = 1. If so, it updates all OTHER addresses

			belonging to the same `user_id` to set `is_default = 0`, ensuring the newly inserted/updated address becomes the sole default.
`trg_user_payment_single_default`	`user_payments`	AFTER INSERT, UPDATE	**Purpose:** Ensures each user has exactly one default payment method, maintaining data consistency for the payment selection process. **Why Needed:** Unlike addresses, the current codebase contains NO stored procedure for managing `user_payments`. This trigger is the ONLY mechanism protecting the single-default constraint. Without it, direct INSERT/UPDATE operations could create multiple default payment methods per user. **Mechanism:** After INSERT or UPDATE on `user_payments`, the trigger checks if any row in `inserted` has `is_default = 1`. If so, it updates all OTHER payment methods belonging to the same `user_id` to set `is_default = 0`.

VIII. Conclusion and Future Work

1. Conclusion

This project has successfully delivered a robust Database Backend for the E-commerce system, meeting all critical objectives defined in the proposal phase:

1. **System Stability**: By enforcing **3rd Normal Form (3NF)** and strict Foreign Key constraints, we have eliminated data redundancy and "orphaned" records.
2. **Data Integrity & Security**: The encapsulation of logic within **T-SQL Stored Procedures** (Checkout, Inventory Management) ensures that business rules are applied consistently across all platforms (Web/Mobile), preventing race conditions like "overselling" inventory.
3. **Performance**: Benchmark tests confirm that query performance remains sub-second (~10ms) even with 50,000+ records, thanks to a strategy of Covering Indexes and Logic-in-Database architecture.
4. **Automation**: Database triggers enforce critical safety rules (e.g., preventing negative stock, maintaining single default address) automatically, reducing the risk of human error.

The system is now ready for deployment, providing a solid foundation for the client's digital growth.

2. Future Work

To further enhance the scalability and feature set of the platform, the following upgrades are proposed for Phase 2:

- **Redis Caching**: Implement a caching layer (Redis) to store "Hot Data" such as the Homepage Product List and Category Trees. This will reduce the load on SQL Server during flash-sale events by serving static content directly from RAM.
- **Payment Gateway Integration**: Integrate **VNPay** (or Stripe/PayPal) to handle real-time payment processing, moving beyond the current "Cash on Delivery" model.
- **Microservices Migration**: As the team grows, split the monolithic database logic into distinct services (Identity Service, Inventory Service, Order Service) to allow independent scaling and development lifecycles.