**Order Module**

The Order Module is a fundamental component of an E-Commerce Application, responsible for managing the entire lifecycle of customer orders from creation to completion. This module ensures accurate processing, status tracking, financial management, and inventory adjustments based on customer purchases. By handling the complexities of order management, it enhances operational efficiency, improves customer satisfaction, and provides valuable insights into sales performance.

**Key Features of the Order Module:**

The following are the Key Features of the Order Module that we will implement in our Ecommerce application:

* **Order Creation:** Facilitates the initiation of a new order by capturing essential details such as customer information, billing and shipping addresses, and the list of products being purchased.
* **Order Retrieval:** Allows administrators and customers to retrieve detailed information about specific orders.
* **Order Status Management:** Enables the updating and tracking of order statuses throughout their lifecycle. Allows transitions between predefined order statuses (e.g., Pending → Processing → Shipped → Delivered). Ensures that status changes follow logical and allowed pathways.
* **Financial Management:** Manages all financial aspects related to orders, including calculations of totals, discounts, and shipping costs.
* **Inventory Management:** Integrates with the inventory system to adjust stock levels based on order placements. Automatically deducts purchased quantities from product stock levels. Prevents orders from being placed if insufficient stock is available.

**Data Transfer Objects (DTOs)**

DTOs (Data Transfer Objects) play a critical role in structuring and validating the data exchanged between the client and server. They ensure that only the necessary and validated data is processed, enhancing security and maintaining a clear separation between internal data models and external API contracts. First, create a folder named **OrderDTOs** within the DTOs folder where we will create all the Required Request and Response DTOs for managing the Orders.

**OrderCreateDTO**

Create a class file named **OrderCreateDTO.cs** within the **DTOs\OrderDTOs** folder and then copy and paste the following code. The OrderCreateDTO Facilitates the creation of new orders by encapsulating essential order-related data, including customer ID, billing and shipping addresses, and a list of order items. It is used when a customer finalizes their cart and initiates an order, ensuring all required fields are provided and validated before processing.

**using System.ComponentModel.DataAnnotations;**

**namespace ECommerceApp.DTOs.OrderDTOs**

**{**

**// DTO for creating a new order**

**public class OrderCreateDTO**

**{**

**[Required(ErrorMessage = "Customer ID is required.")]**

**public int CustomerId { get; set; }**

**[Required(ErrorMessage = "Billing Address ID is required.")]**

**public int BillingAddressId { get; set; }**

**[Required(ErrorMessage = "Shipping Address ID is required.")]**

**public int ShippingAddressId { get; set; }**

**[Required(ErrorMessage = "At least one order item is required.")]**

**[MinLength(1, ErrorMessage = "At least one order item is required.")]**

**public List<OrderItemCreateDTO> OrderItems { get; set; }**

**}**

**}**

**OrderItemCreateDTO**

Create a class file named **OrderItemCreateDTO.cs** within the **DTOs\OrderDTOs** folder and then copy and paste the following code. The OrderItemCreateDTO Represents individual items within an order creation request, capturing necessary details such as product ID and quantity. It is used as part of the OrderCreateDTO to specify which products are being ordered and in what quantities, ensuring each item meets validation rules before inclusion in the order.

**using System.ComponentModel.DataAnnotations;**

**namespace ECommerceApp.DTOs.OrderDTOs**

**{**

**// DTO for individual order items**

**public class OrderItemCreateDTO**

**{**

**[Required(ErrorMessage = "Product ID is required.")]**

**public int ProductId { get; set; }**

**[Range(1, 100, ErrorMessage = "Quantity must be between 1 and 100.")]**

**public int Quantity { get; set; }**

**}**

**}**

**OrderStatusUpdateDTO**

Create a class file named **OrderStatusUpdateDTO.cs** within the **DTOs\OrderDTOs** folder and then copy and paste the following code. The OrderStatusUpdateDTO Facilitates the updating of an existing order's status by capturing the order ID and the new desired status. It is used by administrators to update the status of orders as they progress through different stages (e.g., from Pending to Processing).

**using ECommerceApp.Models;**

**using System.ComponentModel.DataAnnotations;**

**namespace ECommerceApp.DTOs.OrderDTOs**

**{**

**// DTO for updating order status**

**public class OrderStatusUpdateDTO**

**{**

**[Required(ErrorMessage = "OrderId is Required")]**

**public int OrderId { get; set; }**

**[Required]**

**[EnumDataType(typeof(OrderStatus), ErrorMessage = "Invalid Order Status.")]**

**public OrderStatus OrderStatus { get; set; }**

**}**

**}**

**OrderResponseDTO**

Create a class file named **OrderResponseDTO.cs** within the **DTOs\OrderDTOs** folder and then copy and paste the following code. The OrderResponseDTO Provides a comprehensive view of an order in API responses, including customer details, addresses, financials, status, and order items. It is used when retrieving order details to present a complete and organized overview to the client, ensuring all relevant information is available for display and further processing.

**using ECommerceApp.Models;**

**namespace ECommerceApp.DTOs.OrderDTOs**

**{**

**// DTO for returning complete order details.**

**public class OrderResponseDTO**

**{**

**public int Id { get; set; }**

**public string OrderNumber { get; set; }**

**public DateTime OrderDate { get; set; }**

**public int CustomerId { get; set; }**

**public int BillingAddressId { get; set; }**

**public int ShippingAddressId { get; set; }**

**public decimal TotalBaseAmount { get; set; }**

**public decimal TotalDiscountAmount { get; set; }**

**public decimal ShippingCost { get; set; }**

**public decimal TotalAmount { get; set; }**

**public OrderStatus OrderStatus { get; set; }**

**public List<OrderItemResponseDTO> OrderItems { get; set; }**

**}**

**}**

**OrderItemResponseDTO**

Create a class file named **OrderItemResponseDTO.cs** within the **DTOs\OrderDTOs** folder and then copy and paste the following code. The OrderItemResponseDTO Represents individual order item details in API responses, optionally including the product name for enhanced readability. It is used as part of the OrderResponseDTO to provide detailed information about each product included in an order, ensuring clients have access to all necessary item-specific data.

**namespace ECommerceApp.DTOs.OrderDTOs**

**{**

**// DTO for returning individual order item details.**

**public class OrderItemResponseDTO**

**{**

**public int Id { get; set; }**

**public int ProductId { get; set; }**

**public int Quantity { get; set; }**

**public decimal UnitPrice { get; set; }**

**public decimal Discount { get; set; }**

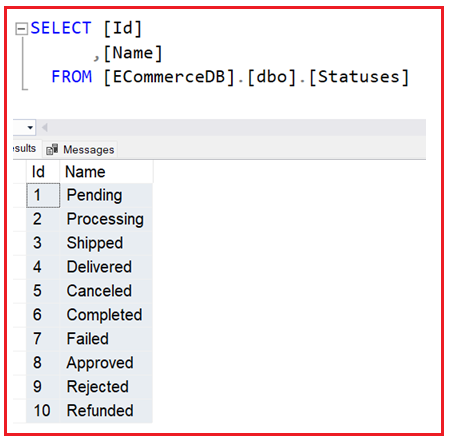
**public decimal TotalPrice { get; set; }**

**}**

**}**

**Modify the OrderStatus Enum as follows:**

Now, if you look at the **Statuses** Master table in the database, you will see the following data.



Now, we want ensure the Id whatever stored in the database for each status should and must be same as our Order Status enum named constants. By default, Enum named constants are start from 0 and increment by 1. So, please modify the OrderStatus enum as follows to sync to values with the database. This is important to ensure the Order Status are reflected correctly in the database.

**using System.Text.Json.Serialization;**

**namespace ECommerceApp.Models**

**{**

**// Enum to represent the status of an order**

**// The JsonStringEnumConverter ensures that enums are serialized as their string names**

**// instead of integer values, enhancing readability.**

**[JsonConverter(typeof(JsonStringEnumConverter))]**

**public enum OrderStatus**

**{**

**Pending = 1,**

**Processing = 2,**

**Shipped = 3,**

**Delivered = 4,**

**Canceled = 5**

**}**

**}**

**Creating Order Service:**

The OrderService class encapsulates all business logic for managing orders. It handles order creation, retrieval, status updates, inventory adjustments, and financial calculations. Its responsibilities include:

* **Order Creation:** Validates input data, manages inventory adjustments, generates unique order numbers, calculates financials (base amount, discounts, shipping cost, and final total), and marks the customer's cart as checked out.
* **Order Retrieval:** Fetches orders by ID, all orders, or orders for a specific customer with complete details.
* **Order Status Management:** Updates the order status by ensuring allowed status transitions.

By isolating order logic within this service, we achieve better maintainability, and consistency in how orders are processed across the application. So, create a class file named **OrderService.cs** within the **Services** folder and then copy and paste the following code:

**using Microsoft.EntityFrameworkCore;**

**using ECommerceApp.Data;**

**using ECommerceApp.DTOs;**

**using ECommerceApp.DTOs.OrderDTOs;**

**using ECommerceApp.Models;**

**using System.Security.Cryptography;**

**namespace ECommerceApp.Services**

**{**

**public class OrderService**

**{**

**private readonly ApplicationDbContext \_context;**

**// Allowed order status transitions for validating status changes.**

**private static readonly Dictionary<OrderStatus, List<OrderStatus>> AllowedStatusTransitions = new()**

**{**

**{ OrderStatus.Pending, new List<OrderStatus> { OrderStatus.Processing, OrderStatus.Canceled } },**

**{ OrderStatus.Processing, new List<OrderStatus> { OrderStatus.Shipped, OrderStatus.Canceled } },**

**{ OrderStatus.Shipped, new List<OrderStatus> { OrderStatus.Delivered } },**

**{ OrderStatus.Delivered, new List<OrderStatus>() }, // Terminal state**

**{ OrderStatus.Canceled, new List<OrderStatus>() } // Terminal state**

**};**

**// Inject the ApplicationDbContext.**

**public OrderService(ApplicationDbContext context)**

**{**

**\_context = context;**

**}**

**// Creates a new order from the provided OrderCreateDTO.**

**// This method validates customer and address data, checks product stock,**

**// calculates financial totals, deducts product stock, and marks any active cart as checked out.**

**public async Task<ApiResponse<OrderResponseDTO>> CreateOrderAsync(OrderCreateDTO orderDto)**

**{**

**try**

**{**

**// Validate that the customer exists.**

**var customer = await \_context.Customers.AsNoTracking().FirstOrDefaultAsync(cust => cust.Id == orderDto.CustomerId);**

**if (customer == null)**

**{**

**return new ApiResponse<OrderResponseDTO>(404, "Customer does not exist.");**

**}**

**// Validate that the billing address exists and belongs to the customer.**

**var billingAddress = await \_context.Addresses.AsNoTracking().FirstOrDefaultAsync(add => add.Id == orderDto.BillingAddressId);**

**if (billingAddress == null || billingAddress.CustomerId != orderDto.CustomerId)**

**{**

**return new ApiResponse<OrderResponseDTO>(400, "Billing Address is invalid or does not belong to the customer.");**

**}**

**// Validate that the shipping address exists and belongs to the customer.**

**var shippingAddress = await \_context.Addresses.AsNoTracking().FirstOrDefaultAsync(add => add.Id == orderDto.ShippingAddressId);**

**if (shippingAddress == null || shippingAddress.CustomerId != orderDto.CustomerId)**

**{**

**return new ApiResponse<OrderResponseDTO>(400, "Shipping Address is invalid or does not belong to the customer.");**

**}**

**// Initialize financial tracking.**

**decimal totalBaseAmount = 0;**

**decimal totalDiscountAmount = 0;**

**decimal shippingCost = 10.00m; // Example fixed shipping cost.**

**decimal totalAmount = 0;**

**// List to hold order items.**

**var orderItems = new List<OrderItem>();**

**// Process each order item from the DTO.**

**foreach (var itemDto in orderDto.OrderItems)**

**{**

**// Check if the product exists.**

**var product = await \_context.Products.FindAsync(itemDto.ProductId);**

**if (product == null)**

**{**

**return new ApiResponse<OrderResponseDTO>(404, $"Product with ID {itemDto.ProductId} does not exist.");**

**}**

**// Check if sufficient stock is available.**

**if (itemDto.Quantity > product.StockQuantity)**

**{**

**return new ApiResponse<OrderResponseDTO>(400, $"Insufficient stock for product {product.Name}.");**

**}**

**// Calculate base price, discount, and total price for the order item.**

**decimal basePrice = itemDto.Quantity \* product.Price;**

**decimal discount = (product.DiscountPercentage / 100.0m) \* basePrice;**

**decimal totalPrice = basePrice - discount;**

**// Create a new OrderItem.**

**var orderItem = new OrderItem**

**{**

**ProductId = product.Id,**

**Quantity = itemDto.Quantity,**

**UnitPrice = product.Price,**

**Discount = discount,**

**TotalPrice = totalPrice**

**};**

**// Add the order item to the list.**

**orderItems.Add(orderItem);**

**// Update the running totals.**

**totalBaseAmount += basePrice;**

**totalDiscountAmount += discount;**

**// Deduct the purchased quantity from the product’s stock.**

**product.StockQuantity -= itemDto.Quantity;**

**\_context.Products.Update(product);**

**}**

**// Calculate the final total amount.**

**totalAmount = totalBaseAmount - totalDiscountAmount + shippingCost;**

**// Manually map from DTO to Order model.**

**var order = new Order**

**{**

**OrderNumber = GenerateOrderNumber(),**

**CustomerId = orderDto.CustomerId,**

**OrderDate = DateTime.UtcNow,**

**BillingAddressId = orderDto.BillingAddressId,**

**ShippingAddressId = orderDto.ShippingAddressId,**

**TotalBaseAmount = totalBaseAmount,**

**TotalDiscountAmount = totalDiscountAmount,**

**ShippingCost = shippingCost,**

**TotalAmount = totalAmount,**

**OrderStatus = OrderStatus.Pending,**

**OrderItems = orderItems**

**};**

**// Add the order to the database.**

**\_context.Orders.Add(order);**

**// Mark the customer's active cart as checked out (if it exists).**

**var cart = await \_context.Carts.FirstOrDefaultAsync(c => c.CustomerId == orderDto.CustomerId && !c.IsCheckedOut);**

**if (cart != null)**

**{**

**cart.IsCheckedOut = true;**

**cart.UpdatedAt = DateTime.UtcNow;**

**}**

**// Save all changes.**

**await \_context.SaveChangesAsync();**

**// Map the saved order to OrderResponseDTO.**

**var orderResponse = MapOrderToDTO(order);**

**return new ApiResponse<OrderResponseDTO>(200, orderResponse);**

**}**

**catch (Exception ex)**

**{**

**return new ApiResponse<OrderResponseDTO>(500, $"An unexpected error occurred while processing your request, Error: {ex.Message}");**

**}**

**}**

**// Retrieves an order by its ID along with related entities.**

**public async Task<ApiResponse<OrderResponseDTO>> GetOrderByIdAsync(int orderId)**

**{**

**try**

**{**

**// Retrieve the order with its items, customer, and addresses details.**

**var order = await \_context.Orders**

**.Include(o => o.OrderItems)**

**.FirstOrDefaultAsync(o => o.Id == orderId);**

**if (order == null)**

**{**

**return new ApiResponse<OrderResponseDTO>(404, "Order not found.");**

**}**

**// Map the order to a DTO.**

**var orderResponse = MapOrderToDTO(order);**

**return new ApiResponse<OrderResponseDTO>(200, orderResponse);**

**}**

**catch (Exception ex)**

**{**

**return new ApiResponse<OrderResponseDTO>(500, $"An unexpected error occurred while processing your request, Error: {ex.Message}");**

**}**

**}**

**// Updates the status of an existing order.**

**// Validates allowed status transitions before applying the update.**

**public async Task<ApiResponse<ConfirmationResponseDTO>> UpdateOrderStatusAsync(OrderStatusUpdateDTO statusDto)**

**{**

**try**

**{**

**// Retrieve the order.**

**var order = await \_context.Orders.FirstOrDefaultAsync(o => o.Id == statusDto.OrderId);**

**if (order == null)**

**{**

**return new ApiResponse<ConfirmationResponseDTO>(404, "Order not found.");**

**}**

**var currentStatus = order.OrderStatus;**

**var newStatus = statusDto.OrderStatus;**

**// Validate the status transition.**

**if (!AllowedStatusTransitions.TryGetValue(currentStatus, out var allowedStatuses))**

**{**

**return new ApiResponse<ConfirmationResponseDTO>(500, "Current order status is invalid.");**

**}**

**if (!allowedStatuses.Contains(newStatus))**

**{**

**return new ApiResponse<ConfirmationResponseDTO>(400, $"Cannot change order status from {currentStatus} to {newStatus}.");**

**}**

**// Update the order status.**

**order.OrderStatus = newStatus;**

**await \_context.SaveChangesAsync();**

**// Prepare a confirmation message.**

**var confirmation = new ConfirmationResponseDTO**

**{**

**Message = $"Order Status with Id {statusDto.OrderId} updated successfully."**

**};**

**return new ApiResponse<ConfirmationResponseDTO>(200, confirmation);**

**}**

**catch (Exception ex)**

**{**

**return new ApiResponse<ConfirmationResponseDTO>(500, $"An unexpected error occurred while processing your request, Error: {ex.Message}");**

**}**

**}**

**// Retrieves all orders in the system.**

**public async Task<ApiResponse<List<OrderResponseDTO>>> GetAllOrdersAsync()**

**{**

**try**

**{**

**// Retrieve all orders with related entities.**

**var orders = await \_context.Orders**

**.Include(o => o.OrderItems)**

**.AsNoTracking()**

**.ToListAsync();**

**// Map each order to its corresponding DTO.**

**var orderList = orders.Select(order => MapOrderToDTO(order)).ToList();**

**return new ApiResponse<List<OrderResponseDTO>>(200, orderList);**

**}**

**catch (Exception ex)**

**{**

**return new ApiResponse<List<OrderResponseDTO>>(500, $"An unexpected error occurred while processing your request, Error: {ex.Message}");**

**}**

**}**

**// Retrieves all orders associated with a specific customer.**

**public async Task<ApiResponse<List<OrderResponseDTO>>> GetOrdersByCustomerAsync(int customerId)**

**{**

**try**

**{**

**// Retrieve the customer along with their orders and related data.**

**var customer = await \_context.Customers**

**.Include(c => c.Orders)**

**.ThenInclude(o => o.OrderItems)**

**.AsNoTracking()**

**.FirstOrDefaultAsync(c => c.Id == customerId);**

**if (customer == null)**

**{**

**return new ApiResponse<List<OrderResponseDTO>>(404, "Customer not found.");**

**}**

**// Map each order to a DTO.**

**var orders = customer.Orders.Select(order => MapOrderToDTO(order)).ToList();**

**return new ApiResponse<List<OrderResponseDTO>>(200, orders);**

**}**

**catch (Exception ex)**

**{**

**return new ApiResponse<List<OrderResponseDTO>>(500, $"An unexpected error occurred while processing your request, Error: {ex.Message}");**

**}**

**}**

**#region Helper Methods**

**// Maps an Order entity to an OrderResponseDTO.**

**private OrderResponseDTO MapOrderToDTO(Order order)**

**{**

**// Map order items.**

**var orderItemsDto = order.OrderItems.Select(oi => new OrderItemResponseDTO**

**{**

**Id = oi.Id,**

**ProductId = oi.ProductId,**

**Quantity = oi.Quantity,**

**UnitPrice = oi.UnitPrice,**

**Discount = oi.Discount,**

**TotalPrice = oi.TotalPrice**

**}).ToList();**

**// Create and return the DTO.**

**return new OrderResponseDTO**

**{**

**Id = order.Id,**

**OrderNumber = order.OrderNumber,**

**OrderDate = order.OrderDate,**

**CustomerId = order.CustomerId,**

**BillingAddressId = order.BillingAddressId,**

**ShippingAddressId = order.ShippingAddressId,**

**TotalBaseAmount = order.TotalBaseAmount,**

**TotalDiscountAmount = order.TotalDiscountAmount,**

**ShippingCost = order.ShippingCost,**

**TotalAmount = order.TotalAmount,**

**OrderStatus = order.OrderStatus,**

**OrderItems = orderItemsDto**

**};**

**}**

**// Generates a unique order number using the current UTC date/time and a random number.**

**// Format: ORD-yyyyMMdd-HHmmss-XXXX**

**private string GenerateOrderNumber()**

**{**

**return $"ORD-{DateTime.UtcNow.ToString("yyyyMMdd-HHmmss")}-{RandomNumber(1000, 9999)}";**

**}**

**// Generates a random number between min and max.**

**private int RandomNumber(int min, int max)**

**{**

**using (var rng = RandomNumberGenerator.Create())**

**{**

**var bytes = new byte[4];**

**rng.GetBytes(bytes);**

**return Math.Abs(BitConverter.ToInt32(bytes, 0) % (max - min + 1)) + min;**

**}**

**}**

**#endregion**

**}**

**}**

**Registering Order Service in Dependency Injection Container**

To enable dependency injection of the Order Service, we need to register the service within the dependency injection container. So, please modify the **Program.cs** class file as follows:

**using ECommerceApp.Data;**

**using ECommerceApp.Services;**

**using Microsoft.EntityFrameworkCore;**

**namespace ECommerceApp**

**{**

**public class Program**

**{**

**public static void Main(string[] args)**

**{**

**var builder = WebApplication.CreateBuilder(args);**

**// Add services to the container.**

**builder.Services.AddControllers()**

**.AddJsonOptions(options =>**

**{**

**// This will use the property names as defined in the C# model**

**options.JsonSerializerOptions.PropertyNamingPolicy = null;**

**});**

**// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle**

**builder.Services.AddEndpointsApiExplorer();**

**builder.Services.AddSwaggerGen();**

**// Configure EF Core with SQL Server**

**builder.Services.AddDbContext<ApplicationDbContext>(options =>**

**options.UseSqlServer(builder.Configuration.GetConnectionString("EFCoreDBConnection")));**

**// Registering the CustomerService**

**builder.Services.AddScoped<CustomerService>();**

**// Registering the AddressService**

**builder.Services.AddScoped<AddressService>();**

**// Registering the CategoryService**

**builder.Services.AddScoped<CategoryService>();**

**// Registering the ProductService**

**builder.Services.AddScoped<ProductService>();**

**// Registering the ShoppingCartService**

**builder.Services.AddScoped<ShoppingCartService>();**

**// Registering the OrderService**

**builder.Services.AddScoped<OrderService>();**

**var app = builder.Build();**

**// Configure the HTTP request pipeline.**

**if (app.Environment.IsDevelopment())**

**{**

**app.UseSwagger();**

**app.UseSwaggerUI();**

**}**

**app.UseHttpsRedirection();**

**app.UseAuthorization();**

**app.MapControllers();**

**app.Run();**

**}**

**}**

**}**

**Creating Orders Controller**

The OrdersController manages all operations related to orders, ensuring that administrators can efficiently create, retrieve, and update orders. Each action method is designed to handle specific aspects of order management, enforcing validation and maintaining data integrity. So, create a new API Empty Controller named **OrdersController** within the Controllers folder and then copy and paste the following code.

**using Microsoft.AspNetCore.Mvc;**

**using ECommerceApp.DTOs;**

**using ECommerceApp.DTOs.OrderDTOs;**

**using ECommerceApp.Services;**

**namespace ECommerceApp.Controllers**

**{**

**[ApiController]**

**[Route("api/[controller]")]**

**public class OrdersController : ControllerBase**

**{**

**private readonly OrderService \_orderService;**

**// Inject the OrderService.**

**public OrdersController(OrderService orderService)**

**{**

**\_orderService = orderService;**

**}**

**// Creates a new order.**

**// POST: api/Orders/CreateOrder**

**[HttpPost("CreateOrder")]**

**public async Task<ActionResult<ApiResponse<OrderResponseDTO>>> CreateOrder([FromBody] OrderCreateDTO orderDto)**

**{**

**var response = await \_orderService.CreateOrderAsync(orderDto);**

**if (response.StatusCode != 200)**

**{**

**return StatusCode(response.StatusCode, response);**

**}**

**return Ok(response);**

**}**

**// Retrieves an order by its ID.**

**// GET: api/Orders/GetOrderById/{id}**

**[HttpGet("GetOrderById/{id}")]**

**public async Task<ActionResult<ApiResponse<OrderResponseDTO>>> GetOrderById(int id)**

**{**

**var response = await \_orderService.GetOrderByIdAsync(id);**

**if (response.StatusCode != 200)**

**{**

**return StatusCode(response.StatusCode, response);**

**}**

**return Ok(response);**

**}**

**// Updates the status of an existing order.**

**// PUT: api/Orders/UpdateOrderStatus**

**[HttpPut("UpdateOrderStatus")]**

**public async Task<ActionResult<ApiResponse<ConfirmationResponseDTO>>> UpdateOrderStatus([FromBody] OrderStatusUpdateDTO statusDto)**

**{**

**var response = await \_orderService.UpdateOrderStatusAsync(statusDto);**

**if (response.StatusCode != 200)**

**{**

**return StatusCode(response.StatusCode, response);**

**}**

**return Ok(response);**

**}**

**// Retrieves all orders.**

**// GET: api/Orders/GetAllOrders**

**[HttpGet("GetAllOrders")]**

**public async Task<ActionResult<ApiResponse<List<OrderResponseDTO>>>> GetAllOrders()**

**{**

**var response = await \_orderService.GetAllOrdersAsync();**

**if (response.StatusCode != 200)**

**{**

**return StatusCode(response.StatusCode, response);**

**}**

**return Ok(response);**

**}**

**// Retrieves all orders for a specific customer.**

**// GET: api/Orders/GetOrdersByCustomer/{customerId}**

**[HttpGet("GetOrdersByCustomer/{customerId}")]**

**public async Task<ActionResult<ApiResponse<List<OrderResponseDTO>>>> GetOrdersByCustomer(int customerId)**

**{**

**var response = await \_orderService.GetOrdersByCustomerAsync(customerId);**

**if (response.StatusCode != 200)**

**{**

**return StatusCode(response.StatusCode, response);**

**}**

**return Ok(response);**

**}**

**}**

**}**

**Testing the Order Endpoints:**

Let us test each order related endpoints of our Ecommerce Application. Please replace {port} with the port number your application is running on (e.g., 5000 or 443).

**CreateOrder:**

Handles the creation of new orders by processing order details, validating data, managing inventory, and calculating financials. It is invoked when a customer finalizes their cart and initiates an order, requiring all necessary details to be captured and validated. Assuming the Customer is having two addresses having the Address Id 1 and 2.

**Method: POST**

**URL: http://localhost:{port}/api/Orders/CreateOrder**

**Headers: Content-Type: application/json**

**Body (raw JSON):**

**{**

**"CustomerId": 1,**

**"BillingAddressId": 1,**

**"ShippingAddressId": 2,**

**"OrderItems": [**

**{**

**"ProductId": 1,**

**"Quantity": 2**

**},**

**{**

**"ProductId": 2,**

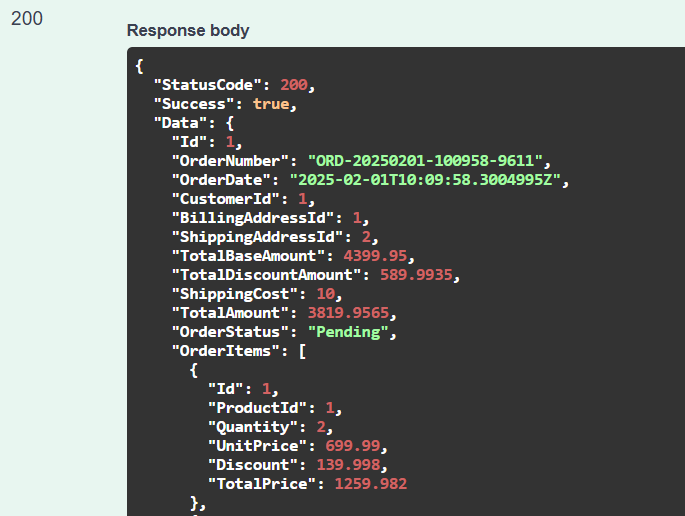
**"Quantity": 3**

**}**

**]**

**}**

**Response in Swagger:**

**’**

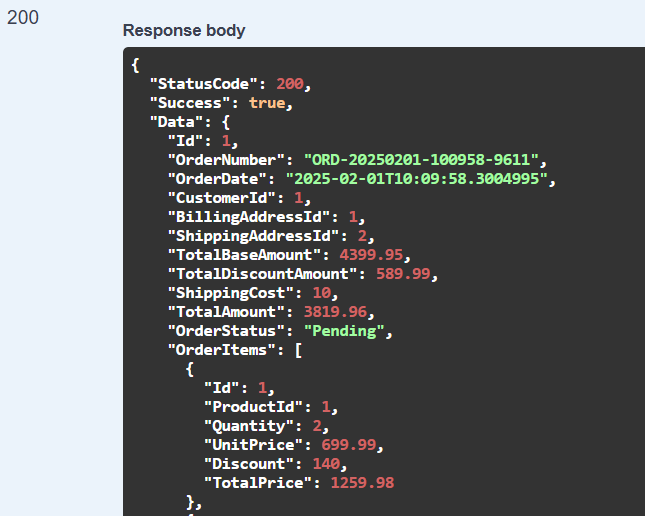
**Get Order by Id:**

Retrieves detailed information about a specific order based on its unique identifier. It is used by administrators or customers to view comprehensive details of a particular order, including customer information, addresses, financials, order items, and payment details.

**Method: GET**

**URL: http://localhost:{port}/api/Orders/GetOrderById/1**

**Response in Swagger: Retrieves detailed information for the order with ID 1.**

****

**Update Order Status:**

Enables the updating of an existing order's status, ensuring that status transitions follow predefined and allowed pathways. It is used by administrators to change the status of an order as it progresses through different stages (e.g., from Pending to Processing, then to Shipped).

**Method: PUT**

**URL: http://localhost:{port}/api/Orders/UpdateOrderStatus**

**Headers: Content-Type: application/json**

**Body (raw JSON):**

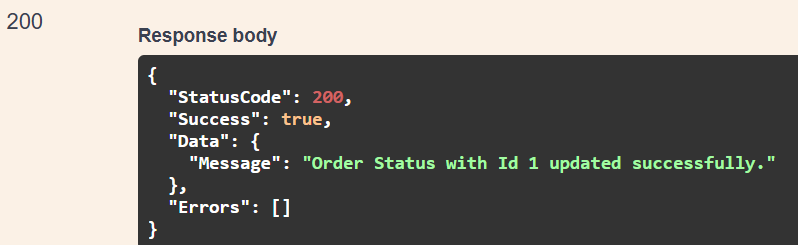
**{**

**"OrderId": 1,**

**"OrderStatus": "Processing"**

**}**

**Response in Swagger:** Updates the status of order 1 to "Processing" (ensure that the transition is allowed).



**Get All Orders:**

Retrieves a comprehensive list of all orders within the system, including detailed information about each order. It is used by administrators to view and manage all orders, facilitating oversight and analysis of sales performance.

**Method: GET**

**URL: http://localhost:{port}/api/Orders/GetAllOrders**

**Response: Retrieves all orders.**

**Get Orders by Customer:**

Retrieves all orders associated with a specific customer, providing a detailed overview of their purchase history. It is used by administrators to view a customer's orders or by customers to view their own order history.

**Method: GET**

**URL: http://localhost:{port}/api/Orders/GetOrdersByCustomer/1**

**Response: Retrieves all orders associated with customer ID 1.**

So, we have done with our Order Module Implementation. Next, we will discuss how to Implement the Payment Module of our Ecommerce Application.