**Shopping Cart Module**

The Shopping Cart Module serves as the intermediary between product browsing and order placement within the E-Commerce platform. It allows customers to add products they intend to purchase, review their selections, adjust quantities, and remove items before finalizing their orders. By maintaining an active cart for each customer, the module ensures a seamless and personalized shopping experience.

**Key Features of the Shopping Cart Module**

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

* **Add to Cart:** Enables customers to add products to their shopping cart from product listings or detail pages. Allows specification of product quantities while ensuring they do not exceed available stock. Automatically applies product-specific discounts to calculate the total price.
* **View Cart:** Provides customers with a detailed view of all items currently in their cart. Displays product names, descriptions, images, and prices. Shows individual item totals and the overall cart total.
* **Update Cart Items:** Allows customers to modify the quantity of each item in their cart. Ensures that updated quantities do not exceed available stock. Dynamically updates total prices based on quantity changes and applicable discounts.
* **Remove from Cart:** Enables customers to remove individual items or clear the entire cart. Allows deletion of specific products from the cart. Provides an option to clear all items in the cart at once.
* **Persisting Cart Data:** Associates carts with authenticated customers to retain cart contents across sessions and devices. Saves cart data in the database linked to the customer’s account. Allows access to the same cart from multiple devices when logged in.

**Data Transfer Objects (DTOs)**

DTOs (Data Transfer Objects) are crucial in structuring and validating the data exchanged between the client and server. They ensure that only necessary and validated data is processed, enhancing security and maintaining a clear separation between internal data models and external API contracts. Now, let us create a folder named **ShoppingCartDTOs** within the **DTOs** folder where we all add the Required Request and Response DTOs for managing the Shopping Cart functionalities.

**AddToCartDTO**

Create a class file named **AddToCartDTO.cs** within the **DTOs/ShoppingCartDTOs** folder and then copy and paste the following code. The AddToCartDTO Facilitates the addition of new items to a customer’s shopping cart by encapsulating necessary input data. It is used when a customer selects a product to add to their cart, ensuring that all required fields are provided and validated before processing.

**using System.ComponentModel.DataAnnotations;**

**namespace ECommerceApp.DTOs.ShoppingCartDTOs**

**{**

**// DTO for adding an item to the cart**

**public class AddToCartDTO**

**{**

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

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

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

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

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

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

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

**}**

**}**

**CartResponseDTO**

Create a class file named **CartResponseDTO.cs** within the **DTOs/ShoppingCartDTOs** folder and then copy and paste the following code. The **CartResponseDTO** Represents the entire shopping cart, including metadata and a list of cart items. It is used when retrieving or returning cart details to provide a comprehensive view of the customer’s current shopping cart.

**namespace ECommerceApp.DTOs.ShoppingCartDTOs**

**{**

**// DTO for returning cart details**

**public class CartResponseDTO**

**{**

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

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

**public bool IsCheckedOut { get; set; }**

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

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

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

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

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

**public List<CartItemResponseDTO>? CartItems { get; set; }**

**}**

**}**

**CartItemResponseDTO**

Create a class file named **CartItemResponseDTO.cs** within the **DTOs/ShoppingCartDTOs** folder and then copy and paste the following code. The CartItemResponseDTO Represents individual items within a shopping cart, encapsulating all relevant details needed for display and processing. It is used when retrieving cart details to present each item’s information to the client, ensuring clarity and completeness in the cart view.

**namespace ECommerceApp.DTOs.ShoppingCartDTOs**

**{**

**// DTO for returning cart item details**

**public class CartItemResponseDTO**

**{**

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

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

**public string? ProductName { get; set; }**

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

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

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

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

**}**

**}**

**UpdateCartItemDTO**

Create a class file named **UpdateCartItemDTO.cs** within the **DTOs/ShoppingCartDTOs** folder and then copy and paste the following code. The UpdateCartItemDTO Allows customers to update the quantity of an existing item in their cart. It is used when a customer adjusts the quantity of a product already in their cart, ensuring that the new quantity is within acceptable limits.

**using System.ComponentModel.DataAnnotations;**

**namespace ECommerceApp.DTOs.ShoppingCartDTOs**

**{**

**// DTO for updating a cart item's quantity**

**public class UpdateCartItemDTO**

**{**

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

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

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

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

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

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

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

**}**

**}**

**RemoveCartItemDTO**

Create a class file named **RemoveCartItemDTO.cs** within the **DTOs/ShoppingCartDTOs** folder and then copy and paste the following code. The RemoveCartItemDTO Facilitates the removal of specific items from a customer’s shopping cart. It is used when a customer decides to delete an item from their cart, ensuring that both the customer and the cart item identifiers are provided and validated.

**using System.ComponentModel.DataAnnotations;**

**namespace ECommerceApp.DTOs.ShoppingCartDTOs**

**{**

**// DTO for removing an item from the cart**

**public class RemoveCartItemDTO**

**{**

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

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

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

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

**}**

**}**

**Creating Shopping Cart Service**

The ShoppingCartService class encapsulates all the business logic and data access for managing a customer’s shopping cart. It handles operations such as adding items to the cart, updating quantities, removing items, clearing the cart, and checking out. Each method returns an ApiResponse<T> to standardize responses and error handling.

So, create a new class file named **ShoppingCartService.cs** within the **Services** folder and then copy and paste the following code:

**using Microsoft.EntityFrameworkCore;**

**using ECommerceApp.Data;**

**using ECommerceApp.DTOs.ShoppingCartDTOs;**

**using ECommerceApp.DTOs;**

**using ECommerceApp.Models;**

**namespace ECommerceApp.Services**

**{**

**public class ShoppingCartService**

**{**

**// Dependency injection of the ApplicationDbContext to interact with the database.**

**private readonly ApplicationDbContext \_context;**

**public ShoppingCartService(ApplicationDbContext context)**

**{**

**\_context = context;**

**}**

**// Retrieves the active (non-checked-out) cart for a given customer.**

**// If no active cart exists, an empty cart (with price details set to 0) is returned.**

**public async Task<ApiResponse<CartResponseDTO>> GetCartByCustomerIdAsync(int customerId)**

**{**

**try**

**{**

**// Query the database for a cart that belongs to the specified customer and is not checked out.**

**var cart = await \_context.Carts**

**.Include(c => c.CartItems) // Include the cart items in the query**

**.ThenInclude(ci => ci.Product) // Also include the product details for each cart item**

**.FirstOrDefaultAsync(c => c.CustomerId == customerId && !c.IsCheckedOut);**

**// If no active cart is found, create an empty DTO with default values.**

**if (cart == null)**

**{**

**var emptyCartDTO = new CartResponseDTO**

**{**

**CustomerId = customerId,**

**IsCheckedOut = false,**

**CreatedAt = DateTime.UtcNow,**

**UpdatedAt = DateTime.UtcNow,**

**CartItems = new List<CartItemResponseDTO>(),**

**TotalBasePrice = 0,**

**TotalDiscount = 0,**

**TotalAmount = 0**

**};**

**// Return the empty cart wrapped in an ApiResponse with status code 200 (OK).**

**return new ApiResponse<CartResponseDTO>(200, emptyCartDTO);**

**}**

**// Map the cart entity to its corresponding DTO (includes price calculations).**

**var cartDTO = MapCartToDTO(cart);**

**return new ApiResponse<CartResponseDTO>(200, cartDTO);**

**}**

**catch (Exception ex)**

**{**

**// In case of an exception, return a 500 status code with an error message.**

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

**}**

**}**

**// Adds a product to the customer's cart.**

**// Creates an active cart if one does not exist.**

**// If the product already exists in the cart, its quantity is updated.**

**public async Task<ApiResponse<CartResponseDTO>> AddToCartAsync(AddToCartDTO addToCartDTO)**

**{**

**try**

**{**

**// Retrieve the product from the database using the provided ProductId.**

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

**if (product == null)**

**{**

**// Return 404 if the product is not found.**

**return new ApiResponse<CartResponseDTO>(404, "Product not found.");**

**}**

**// Check if the requested quantity exceeds the available stock.**

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

**{**

**return new ApiResponse<CartResponseDTO>(400, $"Only {product.StockQuantity} units of {product.Name} are available.");**

**}**

**// Retrieve an active cart for the customer (include related CartItems and Products).**

**var cart = await \_context.Carts**

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

**.ThenInclude(ci => ci.Product)**

**.FirstOrDefaultAsync(c => c.CustomerId == addToCartDTO.CustomerId && !c.IsCheckedOut);**

**// If no active cart exists, create a new cart.**

**if (cart == null)**

**{**

**cart = new Cart**

**{**

**CustomerId = addToCartDTO.CustomerId,**

**IsCheckedOut = false,**

**CreatedAt = DateTime.UtcNow,**

**UpdatedAt = DateTime.UtcNow,**

**CartItems = new List<CartItem>()**

**};**

**// Add the new cart to the database and save changes to generate an Id.**

**\_context.Carts.Add(cart);**

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

**}**

**// Check if the product is already in the cart.**

**var existingCartItem = cart.CartItems.FirstOrDefault(ci => ci.ProductId == addToCartDTO.ProductId);**

**if (existingCartItem != null)**

**{**

**// If the new total quantity exceeds stock, return an error.**

**if (existingCartItem.Quantity + addToCartDTO.Quantity > product.StockQuantity)**

**{**

**return new ApiResponse<CartResponseDTO>(400, $"Adding {addToCartDTO.Quantity} exceeds available stock.");**

**}**

**// Update the quantity and recalculate the total price for this cart item.**

**existingCartItem.Quantity += addToCartDTO.Quantity;**

**existingCartItem.TotalPrice = (existingCartItem.UnitPrice - existingCartItem.Discount) \* existingCartItem.Quantity;**

**existingCartItem.UpdatedAt = DateTime.UtcNow;**

**// Mark the cart item as modified.**

**\_context.CartItems.Update(existingCartItem);**

**}**

**else**

**{**

**// Calculate discount per unit, if applicable.**

**var discount = product.DiscountPercentage > 0 ? product.Price \* product.DiscountPercentage / 100 : 0;**

**// Create a new CartItem with the product and quantity details.**

**var cartItem = new CartItem**

**{**

**CartId = cart.Id,**

**ProductId = product.Id,**

**Quantity = addToCartDTO.Quantity,**

**UnitPrice = product.Price,**

**Discount = discount,**

**TotalPrice = (product.Price - discount) \* addToCartDTO.Quantity,**

**CreatedAt = DateTime.UtcNow,**

**UpdatedAt = DateTime.UtcNow**

**};**

**// Add the new cart item to the database context.**

**\_context.CartItems.Add(cartItem);**

**}**

**// Update the cart's last updated timestamp.**

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

**\_context.Carts.Update(cart);**

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

**// Reload the cart with the latest details (including related CartItems and Products).**

**cart = await \_context.Carts**

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

**.ThenInclude(ci => ci.Product)**

**.FirstOrDefaultAsync(c => c.Id == cart.Id) ?? new Cart();**

**// Map the cart entity to the DTO, which includes price calculations.**

**var cartDTO = MapCartToDTO(cart);**

**return new ApiResponse<CartResponseDTO>(200, cartDTO);**

**}**

**catch (Exception ex)**

**{**

**// Return error response in case of exceptions.**

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

**}**

**}**

**// Updates the quantity of a specific item in the customer's cart.**

**public async Task<ApiResponse<CartResponseDTO>> UpdateCartItemAsync(UpdateCartItemDTO updateCartItemDTO)**

**{**

**try**

**{**

**// Retrieve the active cart for the customer along with cart items and product details.**

**var cart = await \_context.Carts**

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

**.ThenInclude(ci => ci.Product)**

**.FirstOrDefaultAsync(c => c.CustomerId == updateCartItemDTO.CustomerId && !c.IsCheckedOut);**

**// Return 404 if no active cart is found.**

**if (cart == null)**

**{**

**return new ApiResponse<CartResponseDTO>(404, "Active cart not found.");**

**}**

**// Find the specific cart item that needs to be updated.**

**var cartItem = cart.CartItems.FirstOrDefault(ci => ci.Id == updateCartItemDTO.CartItemId);**

**if (cartItem == null)**

**{**

**return new ApiResponse<CartResponseDTO>(404, "Cart item not found.");**

**}**

**// Ensure the updated quantity does not exceed the available product stock.**

**if (updateCartItemDTO.Quantity > cartItem.Product.StockQuantity)**

**{**

**return new ApiResponse<CartResponseDTO>(400, $"Only {cartItem.Product.StockQuantity} units of {cartItem.Product.Name} are available.");**

**}**

**// Update the cart item's quantity and recalculate its total price.**

**cartItem.Quantity = updateCartItemDTO.Quantity;**

**cartItem.TotalPrice = (cartItem.UnitPrice - cartItem.Discount) \* cartItem.Quantity;**

**cartItem.UpdatedAt = DateTime.UtcNow;**

**// Mark the cart item as updated.**

**\_context.CartItems.Update(cartItem);**

**// Update the cart's updated timestamp.**

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

**\_context.Carts.Update(cart);**

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

**// Reload the updated cart with its items.**

**cart = await \_context.Carts**

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

**.ThenInclude(ci => ci.Product)**

**.FirstOrDefaultAsync(c => c.Id == cart.Id) ?? new Cart();**

**// Map the updated cart to the DTO.**

**var cartDTO = MapCartToDTO(cart);**

**return new ApiResponse<CartResponseDTO>(200, cartDTO);**

**}**

**catch (Exception ex)**

**{**

**// Return error response if an exception occurs.**

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

**}**

**}**

**// Removes a specific item from the customer's cart.**

**public async Task<ApiResponse<CartResponseDTO>> RemoveCartItemAsync(RemoveCartItemDTO removeCartItemDTO)**

**{**

**try**

**{**

**// Retrieve the active cart along with its items and product details.**

**var cart = await \_context.Carts**

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

**.ThenInclude(ci => ci.Product)**

**.FirstOrDefaultAsync(c => c.CustomerId == removeCartItemDTO.CustomerId && !c.IsCheckedOut);**

**// Return 404 if no active cart is found.**

**if (cart == null)**

**{**

**return new ApiResponse<CartResponseDTO>(404, "Active cart not found.");**

**}**

**// Find the cart item to remove.**

**var cartItem = cart.CartItems.FirstOrDefault(ci => ci.Id == removeCartItemDTO.CartItemId);**

**if (cartItem == null)**

**{**

**return new ApiResponse<CartResponseDTO>(404, "Cart item not found.");**

**}**

**// Remove the cart item from the context.**

**\_context.CartItems.Remove(cartItem);**

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

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

**// Reload the updated cart after removal.**

**cart = await \_context.Carts**

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

**.ThenInclude(ci => ci.Product)**

**.FirstOrDefaultAsync(c => c.Id == cart.Id) ?? new Cart();**

**// Map the updated cart to the DTO.**

**var cartDTO = MapCartToDTO(cart);**

**return new ApiResponse<CartResponseDTO>(200, cartDTO);**

**}**

**catch (Exception ex)**

**{**

**// Return error response if an exception occurs.**

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

**}**

**}**

**// Clears all items from the customer's active cart.**

**public async Task<ApiResponse<ConfirmationResponseDTO>> ClearCartAsync(int customerId)**

**{**

**try**

**{**

**// Retrieve the active cart along with its items.**

**var cart = await \_context.Carts**

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

**.FirstOrDefaultAsync(c => c.CustomerId == customerId && !c.IsCheckedOut);**

**// Return 404 if no active cart is found.**

**if (cart == null)**

**{**

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

**}**

**// If there are any items in the cart, remove them.**

**if (cart.CartItems.Any())**

**{**

**\_context.CartItems.RemoveRange(cart.CartItems);**

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

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

**}**

**// Create a confirmation response DTO.**

**var confirmation = new ConfirmationResponseDTO**

**{**

**Message = "Cart has been cleared successfully."**

**};**

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

**}**

**catch (Exception ex)**

**{**

**// Return error response if an exception occurs.**

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

**}**

**}**

**// Helper method to manually map a Cart entity to a CartResponseDTO.**

**// This method also calculates the TotalBasePrice, TotalDiscount, and TotalAmount.**

**private CartResponseDTO MapCartToDTO(Cart cart)**

**{**

**// Map each CartItem entity to its corresponding CartItemResponseDTO.**

**var cartItemsDto = cart.CartItems?.Select(ci => new CartItemResponseDTO**

**{**

**Id = ci.Id,**

**ProductId = ci.ProductId,**

**ProductName = ci.Product?.Name,**

**Quantity = ci.Quantity,**

**UnitPrice = ci.UnitPrice,**

**Discount = ci.Discount,**

**TotalPrice = ci.TotalPrice**

**}).ToList() ?? new List<CartItemResponseDTO>();**

**// Initialize totals for base price, discount, and amount after discount.**

**decimal totalBasePrice = 0;**

**decimal totalDiscount = 0;**

**decimal totalAmount = 0;**

**// Iterate through each cart item DTO to accumulate the totals.**

**foreach (var item in cartItemsDto)**

**{**

**totalBasePrice += item.UnitPrice \* item.Quantity; // Sum of base prices (without discount)**

**totalDiscount += item.Discount \* item.Quantity; // Sum of discounts applied per unit**

**totalAmount += item.TotalPrice; // Sum of final prices after discount**

**}**

**// Create and return the final CartResponseDTO with all details and calculated totals.**

**return new CartResponseDTO**

**{**

**Id = cart.Id,**

**CustomerId = cart.CustomerId,**

**IsCheckedOut = cart.IsCheckedOut,**

**CreatedAt = cart.CreatedAt,**

**UpdatedAt = cart.UpdatedAt,**

**CartItems = cartItemsDto,**

**TotalBasePrice = totalBasePrice,**

**TotalDiscount = totalDiscount,**

**TotalAmount = totalAmount**

**};**

**}**

**}**

**}**

**Registering Shopping Cart Service in Dependency Injection Container**

To enable dependency injection of the Shopping Cart Service class, we need to register the service within the application's 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>();**

**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 Shopping Cart Controller**

Create a new API Empty Controller named **CartsController** within the Controllers folder and then copy and paste the following code. The CartsController manages all operations related to the shopping cart, ensuring that customers can efficiently add, view, update, and remove items from their carts. The controller simply delegates requests to the service and returns the appropriate ApiResponse<T>.

**using Microsoft.AspNetCore.Mvc;**

**using ECommerceApp.DTOs;**

**using ECommerceApp.DTOs.ShoppingCartDTOs;**

**using ECommerceApp.Services;**

**namespace ECommerceApp.Controllers**

**{**

**[ApiController]**

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

**public class CartsController : ControllerBase**

**{**

**private readonly ShoppingCartService \_shoppingCartService;**

**public CartsController(ShoppingCartService shoppingCartService)**

**{**

**\_shoppingCartService = shoppingCartService;**

**}**

**// Retrieves the active cart for a given customer.**

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

**public async Task<ActionResult<ApiResponse<CartResponseDTO>>> GetCartByCustomerId(int customerId)**

**{**

**var response = await \_shoppingCartService.GetCartByCustomerIdAsync(customerId);**

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

**{**

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

**}**

**return Ok(response);**

**}**

**// Adds an item to the customer's cart.**

**[HttpPost("AddToCart")]**

**public async Task<ActionResult<ApiResponse<CartResponseDTO>>> AddToCart([FromBody] AddToCartDTO addToCartDTO)**

**{**

**var response = await \_shoppingCartService.AddToCartAsync(addToCartDTO);**

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

**{**

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

**}**

**return Ok(response);**

**}**

**// Updates the quantity of an existing cart item.**

**[HttpPut("UpdateCartItem")]**

**public async Task<ActionResult<ApiResponse<CartResponseDTO>>> UpdateCartItem([FromBody] UpdateCartItemDTO updateCartItemDTO)**

**{**

**var response = await \_shoppingCartService.UpdateCartItemAsync(updateCartItemDTO);**

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

**{**

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

**}**

**return Ok(response);**

**}**

**// Removes a specific item from the cart.**

**[HttpDelete("RemoveCartItem")]**

**public async Task<ActionResult<ApiResponse<CartResponseDTO>>> RemoveCartItem([FromBody] RemoveCartItemDTO removeCartItemDTO)**

**{**

**var response = await \_shoppingCartService.RemoveCartItemAsync(removeCartItemDTO);**

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

**{**

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

**}**

**return Ok(response);**

**}**

**// Clears all items from the customer's active cart.**

**[HttpDelete("ClearCart")]**

**public async Task<ActionResult<ApiResponse<ConfirmationResponseDTO>>> ClearCart([FromQuery] int customerId)**

**{**

**var response = await \_shoppingCartService.ClearCartAsync(customerId);**

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

**{**

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

**}**

**return Ok(response);**

**}**

**}**

**}**

**Testing the Shopping Cart Endpoints:**

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

**Add To Cart:**

Adds a specified product to the customer’s active shopping cart. It is invoked when a customer selects a product to add to their cart, specifying the desired quantity.

**Method: POST**

**URL: http://localhost:{port}/api/Carts/AddToCart**

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

**Body (raw JSON):**

**{**

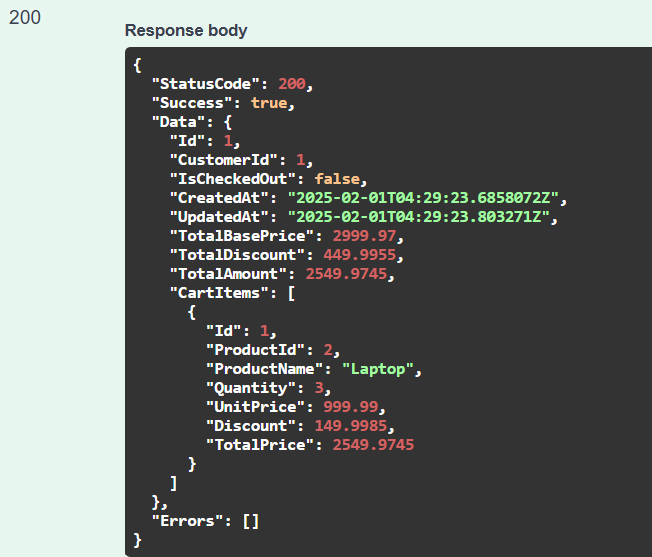
**"CustomerId": 1,**

**"ProductId": 2,**

**"Quantity": 3**

**}**

**Description:** Adds 3 units of product 2 to customer 1’s cart. You will get the following response in Swagger:

****

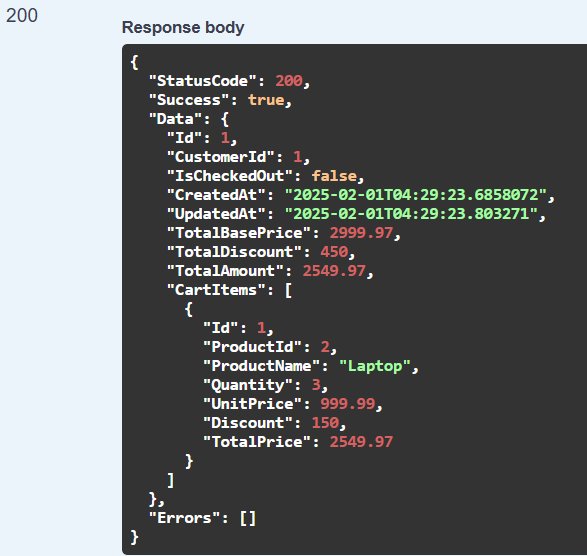
**Get Cart by CustomerId:**

Retrieves the current active shopping cart for a specified customer. It is used when a customer wants to view the contents of their cart, providing a comprehensive overview of all items selected for purchase.

**Method: GET**

**URL: http://localhost:{port}/api/Carts/GetCart/1**

**Response:** A JSON with the cart details (or an empty cart if none exists). You will get the following response in Swagger:

****

**Update Cart Item:**

Updates the quantity of a specific item within the customer’s active shopping cart. It is used when a customer decides to change the quantity of a product already in their cart, such as increasing or decreasing the amount.

**Method: PUT**

**URL: http://localhost:{port}/api/Carts/UpdateCartItem**

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

**Body (raw JSON):**

**{**

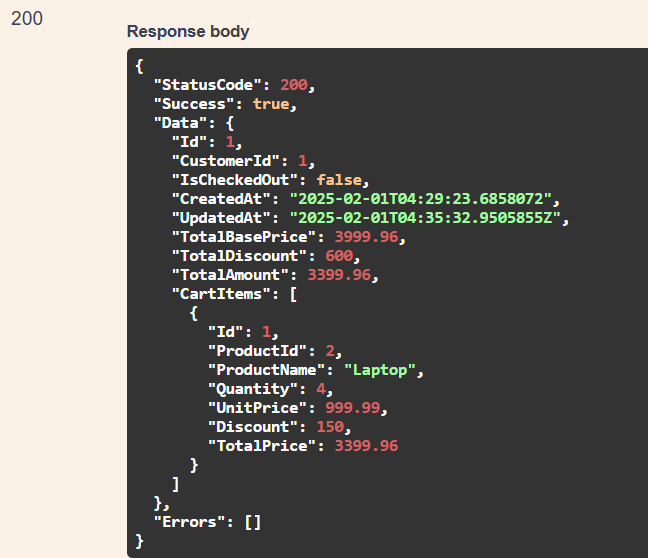
**"CustomerId": 1,**

**"CartItemId": 1,**

**"Quantity": 4**

**}**

**Description:** Updates the quantity of the cart item with ID 5 to 4 for customer 1. You will get the following response:

****

**Remove Cart Item:**

Removes a specific item from the customer’s active shopping cart. It is invoked when a customer decides to delete an item from their cart, either individually or as part of cart management. You will get the following response:

**Method: DELETE**

**URL: http://localhost:{port}/api/Carts/RemoveCartItem**

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

**Body (raw JSON):**

**{**

**"CustomerId": 1,**

**"CartItemId": 1**

**}**

**Description:** Removes the specified cart item from customer 1’s active cart. You will get the following response:



**Clear Cart:**

Clears all items from the customer’s active shopping cart. It is used when a customer wants to remove all items from their cart in a single action, such as starting a new shopping session or abandoning the current cart.

**Method: DELETE**

**URL: http://localhost:{port}/api/Carts/ClearCart?customerId=1**

**Expected Response:** Clears all items from customer 1’s active cart. You will get the following response:



So, we have done with our Shopping Cart Module Implementation in our Ecommerce Application. Next, we will Implement the Order Module of our Ecommerce Application.