Part 9: Payment Integration

Objective: Implement payment processing for orders in your e-commerce API using Stripe, ensuring data is managed with DTOs and circular references are avoided.

Step 1: Set Up a Stripe Account

* Register for a Stripe account.
* Obtain your API keys from the Stripe dashboard under Developers > API Keys.

Step 2: Install Stripe SDK

Add the Stripe .NET library to your project:

dotnet add package Stripe.net

Step 3: Add Payment Integration Settings

Create PaymentSettings class:

namespace AtirAPI.Helpers

{

public class PaymentSettings

{

public string SecretKey { get; set; }

}

}

Configure in appsettings.json:

"PaymentSettings": {

"SecretKey": "sk\_test\_your\_secret\_key"

}

Register in Program.cs:

using AtirAPI.Helpers;

builder.Services.Configure<PaymentSettings>(builder.Configuration.GetSection("PaymentSettings"));

Step 4: Update the Order Entity

Add PaymentStatus to Order.cs:

public class Order

{

// ... other properties ...

public string PaymentStatus { get; set; } = "Pending";

}

Run a migration to update the database:

dotnet ef migrations add AddPaymentStatusToOrder

dotnet ef database update

Step 5: Update DTOs

Define or Update DTOs:

OrderDTO:

public class OrderDTO

{

public int Id { get; set; }

public int CustomerId { get; set; }

public CustomerDTO Customer { get; set; }

public DateTime OrderDate { get; set; }

public decimal TotalAmount { get; set; }

public string PaymentStatus { get; set; }

public List<OrderItemDTO> OrderItems { get; set; }

}

OrderCreateDTO:

public class OrderCreateDTO

{

public int CustomerId { get; set; }

public string PaymentStatus { get; set; } = "Pending";

public ICollection<OrderItemCreateDTO> OrderItems { get; set; }

}

OrderItemCreateDTO:

public class OrderItemCreateDTO

{

public int ProductId { get; set; }

public int Quantity { get; set; }

}

OrderItemDTO:

public class OrderItemDTO

{

public int Id { get; set; }

public int ProductId { get; set; }

public ProductDTO Product { get; set; }

public int Quantity { get; set; }

public decimal Price { get; set; }

}

CustomerDTO:

public class CustomerDTO

{

public int Id { get; set; }

public string FirstName { get; set; }

public string LastName { get; set; }

public string Email { get; set; }

public string PhoneNumber { get; set; }

public string Address { get; set; }

}

ProductDTO:

public class ProductDTO

{

public int Id { get; set; }

public string Name { get; set; }

public string Description { get; set; }

public string? ImageUrl { get; set; }

public decimal Price { get; set; }

public int Stock { get; set; }

public int CategoryId { get; set; }

public CategoryDTO Category { get; set; }

}

CategoryDTO:

public class CategoryDTO

{

public int Id { get; set; }

public string Name { get; set; }

public string Description { get; set; }

}

Step 6: Configure AutoMapper

* Update or create an OrderProfile in your mapping configuration:

public class OrderProfile : Profile

{

public OrderProfile()

{

CreateMap<OrderCreateDTO, Order>()

.ForMember(dest => dest.Customer, opt => opt.Ignore())

.ForMember(dest => dest.OrderItems, opt => opt.MapFrom(src => src.OrderItems))

.ForMember(dest => dest.PaymentStatus, opt => opt.MapFrom(src => src.PaymentStatus));

CreateMap<OrderItemCreateDTO, OrderItem>()

.ForMember(dest => dest.Order, opt => opt.Ignore())

.ForMember(dest => dest.Product, opt => opt.Ignore());

CreateMap<Order, OrderDTO>()

.ForMember(dest => dest.Customer, opt => opt.MapFrom(src => src.Customer))

.ForMember(dest => dest.OrderItems, opt => opt.MapFrom(src => src.OrderItems));

CreateMap<OrderItem, OrderItemDTO>();

CreateMap<Customer, CustomerDTO>();

CreateMap<Product, ProductDTO>();

CreateMap<Category, CategoryDTO>();

}

}

Ensure AutoMapper is registered in Program.cs:

builder.Services.AddAutoMapper(AppDomain.CurrentDomain.GetAssemblies());

Step 7: Create a Payments Controller

* Add PaymentsController to handle payment processing:

using AtirAPI.Data;

using AtirAPI.Helpers;

using AtirAPI.Models;

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

using Microsoft.Extensions.Options;

using Stripe;

using System.Threading.Tasks;

namespace AtirAPI.Controllers

{

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

[ApiController]

public class PaymentsController : ControllerBase

{

private readonly AtirDbContext \_context;

private readonly PaymentSettings \_paymentSettings;

public PaymentsController(AtirDbContext context, IOptions<PaymentSettings> paymentSettings)

{

\_context = context;

\_paymentSettings = paymentSettings.Value;

StripeConfiguration.ApiKey = \_paymentSettings.SecretKey;

}

// POST: api/Payments/charge

[HttpPost("charge")]

public async Task<IActionResult> ProcessPayment(int orderId)

{

var order = await \_context.Orders.Include(o => o.OrderItems).FirstOrDefaultAsync(o => o.Id == orderId);

if (order == null)

{

return NotFound("Order not found.");

}

if (order.PaymentStatus == "Paid")

{

return BadRequest("Order is already paid.");

}

try

{

var paymentIntentService = new PaymentIntentService();

var paymentIntent = paymentIntentService.Create(new PaymentIntentCreateOptions

{

Amount = (long)(order.TotalAmount \* 100), // Convert to cents

Currency = "usd",

PaymentMethodTypes = new List<string> { "card" },

});

order.PaymentStatus = "Paid";

\_context.Entry(order).State = EntityState.Modified;

await \_context.SaveChangesAsync();

return Ok(new

{

Message = "Payment processed successfully.",

PaymentIntentId = paymentIntent.Id

});

}

catch (StripeException ex)

{

return BadRequest(new { Error = ex.Message });

}

}

}

}

Step 8: Update Orders Controller

* Modify GetOrders and GetOrder to return OrderDTO:

[HttpGet]

public async Task<ActionResult<IEnumerable<OrderDTO>>> GetOrders()

{

var orders = await \_context.Orders

.Include(o => o.Customer)

.Include(o => o.OrderItems)

.ThenInclude(oi => oi.Product)

.ThenInclude(p => p.Category)

.ToListAsync();

return Ok(\_mapper.Map<IEnumerable<OrderDTO>>(orders));

}

[HttpGet("{id}")]

public async Task<ActionResult<OrderDTO>> GetOrder(int id)

{

var order = await \_context.Orders

.Include(o => o.Customer)

.Include(o => o.OrderItems)

.ThenInclude(oi => oi.Product)

.ThenInclude(p => p.Category)

.FirstOrDefaultAsync(o => o.Id == id);

if (order == null)

{

return NotFound();

}

return Ok(\_mapper.Map<OrderDTO>(order));

}

Step 9: Test the Payment Endpoint

Use Postman or Swagger to test:

* Endpoint: POST /api/Payments/charge
* Body: { "orderId": 1 }
* Expected Response:

{

"message": "Payment processed successfully.",

"paymentIntentId": "pi\_1234567890"

}

Step 10: Enhancements

* Refunds: Implement a refund endpoint if needed.
* Webhooks: Add webhook support for real-time payment status updates.
* Multiple Payment Methods: Extend support for additional payment gateways or methods.

Last Changes with Order

* Ensure all GET operations for orders use OrderDTO