The Coupon Challenge Solution

Core project

1. Create a new Class for the Coupon (hint: Stripe already has a type for Coupon so use ‘AppCoupon’ to make life easier) that contains properties for the **Name**, **AmountOff**?, **PercentOff**?, **PromotionCode** and **CouponId.** Ensure the properties are the same types that Stripe uses.

```
namespace Core.Entities;

public class AppCoupon
{
    public required string Name { get; set; }
    public decimal? AmountOff { get; set; }
    public decimal? PercentOff { get; set; }
    public required string PromotionCode { get; set; }
    public required string CouponId { get; set; }
}

```

1. Add this as a new property into the **ShoppingCart**

```
namespace Core.Entities;

public class ShoppingCart
{
    public required string Id { get; set; }
    public List<CartItem> Items { get; set; } = [];
    public int? DeliveryMethodId { get; set; }
    public string? ClientSecret { get; set; }
    public string? PaymentIntentId { get; set; }
    public AppCoupon? Coupon { get; set; }
}

```

1. Create an interface called ‘ICouponService’ that has a single method called **GetCouponFromPromoCode(string code)** that returns **AppCoupon?**

```
using Core.Entities;

namespace Core.Interfaces;

public interface ICouponService
{
    Task<AppCoupon?> GetCouponFromPromoCode(string code);
}

```

1. Update the **Order** class and add a **Discount** property that is a decimal.

```
namespace Core.Entities.OrderAggregate;

public class Order : BaseEntity
{
        // omitted
    public decimal Discount { get; set; }
    public OrderStatus Status { get; set; } = OrderStatus.Pending;
    public required string PaymentIntentId { get; set; }
        // omitted
}
```

1. Update the **GetTotal** method in the Order class.

```
namespace Core.Entities.OrderAggregate;

public class Order : BaseEntity
{
    // omitted

    public decimal GetTotal()
    {
        return Subtotal - Discount + DeliveryMethod.Price;
    }
}

```

Infrastructure project

1. Create a new class in the services folder called **CouponService** that implements the **ICouponService** interface. Use the stripe PromotionCodeService() to get the coupon from the promotion code and use that to return the **AppCoupon**

```
using Core.Entities;
using Core.Interfaces;
using Microsoft.Extensions.Configuration;
using Stripe;

namespace Infrastructure.Services;

public class CouponService : ICouponService
{
    public CouponService(IConfiguration config)
    {
        StripeConfiguration.ApiKey = config["StripeSettings:SecretKey"];
    }

    public async Task<AppCoupon?> GetCouponFromPromoCode(string code)
    {
        var promotionService = new PromotionCodeService();

        var options = new PromotionCodeListOptions
        {
            Code = code
        };

        var promotionCodes = await promotionService.ListAsync(options);

        var promotionCode = promotionCodes.FirstOrDefault();

        if (promotionCode != null && promotionCode.Coupon != null)
        {
            return new AppCoupon
            {
                Name = promotionCode.Coupon.Name,
                AmountOff = promotionCode.Coupon.AmountOff,
                PercentOff = promotionCode.Coupon.PercentOff,
                CouponId = promotionCode.Coupon.Id,
                PromotionCode = promotionCode.Code
            };
        }

        return null;
    }
}

```

1. Update the PaymentIntent to accommodate the Discount. In the payment service we have quite a big method for the **CreateOrUpdatePaymentIntent**. This is a good opportunity to refactor this into small functions as we need to add extra here for the coupon functionality. Here is the shell of what would be an improved structure for this code. Note that none of these private methods return null so we need to use exceptions to avoid null ref warnings.

```
using Core.Entities;
using Core.Interfaces;
using Microsoft.Extensions.Configuration;
using Stripe;

namespace Infrastructure.Services;

public class PaymentService(IConfiguration config, ICartService cartService,
    IUnitOfWork unit) : IPaymentService
{
    public async Task<ShoppingCart?> CreateOrUpdatePaymentIntent(string cartId)
    {
        StripeConfiguration.ApiKey = config["StripeSettings:SecretKey"];

        var cart = await cartService.GetCartAsync(cartId)
            ?? throw new Exception("Cart unavailable");

        var shippingPrice = await GetShippingPriceAsync(cart) ?? 0;

        await ValidateCartItemsInCartAsync(cart);

        var subtotal = CalculateSubtotal(cart);

        if (cart.Coupon != null)
        {
            subtotal = await ApplyDiscountAsync(cart.Coupon, subtotal);
        }

        var total = subtotal + shippingPrice;

        await CreateUpdatePaymentIntentAsync(cart, total);

        await cartService.SetCartAsync(cart);

        return cart;
    }

    private async Task CreateUpdatePaymentIntentAsync(ShoppingCart cart,
        long total)
    {
        var service = new PaymentIntentService();

        if (string.IsNullOrEmpty(cart.PaymentIntentId))
        {
            var options = new PaymentIntentCreateOptions
            {
                Amount = total,
                Currency = "usd",
                PaymentMethodTypes = ["card"]
            };
            var intent = await service.CreateAsync(options);
            cart.PaymentIntentId = intent.Id;
            cart.ClientSecret = intent.ClientSecret;
        }
        else
        {
            var options = new PaymentIntentUpdateOptions
            {
```

```
                Amount = total
            };
            await service.UpdateAsync(cart.PaymentIntentId, options);
        }
    }

    private async Task<long> ApplyDiscountAsync(AppCoupon appCoupon,
        long amount)
    {
        // omitted
    }

    private long CalculateSubtotal(ShoppingCart cart)
    {
        var itemTotal = cart.Items.Sum(x => x.Quantity * x.Price * 100);
        return (long)itemTotal;
    }

    private async Task ValidateCartItemsInCartAsync(ShoppingCart cart)
    {
        foreach (var item in cart.Items)
        {
            var productItem = await unit.Repository<Core.Entities.Product>()
                .GetByIdAsync(item.ProductId)
                    ?? throw new Exception("Problem getting product in cart");

            if (item.Price != productItem.Price)
            {
                item.Price = productItem.Price;
            }
        }
    }

    private async Task<long?> GetShippingPriceAsync(ShoppingCart cart)
    {
        if (cart.DeliveryMethodId.HasValue)
        {
            var deliveryMethod = await unit.Repository<DeliveryMethod>()
                .GetByIdAsync((int)cart.DeliveryMethodId)
                    ?? throw new Exception("Problem with delivery method");

            return (long)deliveryMethod.Price * 100;
        }

        return null;
    }
}
```

1. Implement the **ApplyDiscountAsync** method to update the intent with the discounted price.

```
using Core.Entities;
using Core.Interfaces;
using Microsoft.Extensions.Configuration;
using Stripe;

namespace Infrastructure.Services;

public class PaymentService(IConfiguration config,
    ICartService cartService, IUnitOfWork unit) : IPaymentService
{

    // omitted

```

```
    private async Task<long> ApplyDiscountAsync(AppCoupon appCoupon,
        long amount)
    {
        var couponService = new Stripe.CouponService();

        var coupon = await couponService.GetAsync(appCoupon.CouponId);

        if (coupon.AmountOff.HasValue)
        {
            amount -= (long)coupon.AmountOff * 100;
        }

        if (coupon.PercentOff.HasValue)
        {
            var discount = amount * (coupon.PercentOff.Value / 100);
            amount -= (long)discount;
        }

        return amount;
    }

    // omitted

}

```

1. Update the **OrderConfiguration** to accommodate the new decimal property for the **Discount**

```
using Core.Entities.OrderAggregate;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace Infrastructure.Config;

public class OrderConfiguration : IEntityTypeConfiguration<Order>
{
    public void Configure(EntityTypeBuilder<Order> builder)
    {
                // omitted
        builder.Property(x => x.Subtotal).HasColumnType("decimal(18,2)");
        builder.Property(x => x.Discount).HasColumnType("decimal(18,2)");
                // omitted
    }
}

```

1. Create a new EF migration at this point to update the DB schema

```
# at the solution level
dotnet ef migrations add "CouponsAdded" -s API -p Infrastructure
```

API Project

1. Update the **Program.cs** to register the new **ICouponService** and its implementation class.

```
// using statements omitted

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

// omitted

```

```
builder.Services.AddScoped<ICouponService, CouponService>();

// omitted

```

1. Update the **OrderDto** to include the **Discount** property

```
using Core.Entities.OrderAggregate;

namespace API.DTOs;

public class OrderDto
{
    public int Id { get; set; }
    public DateTime OrderDate { get; set; }
    public required string BuyerEmail { get; set; }
    public required ShippingAddress ShippingAddress { get; set; }
    public required string DeliveryMethod { get; set; }
    public decimal ShippingPrice { get; set; }
    public required PaymentSummary PaymentSummary { get; set; }
    public required List<OrderItemDto> OrderItems { get; set; }
    public decimal Subtotal { get; set; }
    public decimal Discount { get; set; }
    public required string Status { get; set; }
    public decimal Total { get; set; }
    public required string PaymentIntentId { get; set; }
}

```

1. Update the **OrderMappingExtensions** to incorporate this Discount property.

```
    public static OrderDto ToDto(this Order order)
    {
        return new OrderDto
        {
            Id = order.Id,
            BuyerEmail = order.BuyerEmail,
            OrderDate = order.OrderDate,
            ShippingAddress = order.ShippingAddress,
            PaymentSummary = order.PaymentSummary,
            DeliveryMethod = order.DeliveryMethod.Description,
            ShippingPrice = order.DeliveryMethod.Price,
            OrderItems = order.OrderItems.Select(x => x.ToDto()).ToList(),
            Subtotal = order.Subtotal,
            Discount = order.Discount,
            Total = order.GetTotal(),
            Status = order.Status.ToString(),
            PaymentIntentId = order.PaymentIntentId
        };
    }
```

1. Create a CouponsController that has the following endpoint:

```
using Core.Entities;
using Core.Interfaces;
using Microsoft.AspNetCore.Mvc;

namespace API.Controllers;

public class CouponsController(ICouponService couponService) : BaseApiController
{
    [HttpGet("{code}")]
    public async Task<ActionResult<AppCoupon>> ValidateCoupon(string code)

```

```
    {
        var coupon = await couponService.GetCouponFromPromoCode(code);

        if (coupon == null) return BadRequest("Invalid voucher code");

        return coupon;
    }
}

```

1. Run the request in Postman to ensure you can validate and return the coupon.
2. Make sure you return a BadRequest for the second request:
3. Update the **CreateOrderDto** to take an optional Discount property that is a decimal.

```
using System.ComponentModel.DataAnnotations;
using Core.Entities.OrderAggregate;

namespace API.DTOs;

public class CreateOrderDto
{
    [Required]
    public string CartId { get; set; } = string.Empty;

    [Required]
    public int DeliveryMethodId { get; set; }

    [Required]
    public ShippingAddress ShippingAddress { get; set; } = null!;

    [Required]
    public PaymentSummary PaymentSummary { get; set; } = null!;
    public decimal Discount { get; set; }
}

```

1. Update the **OrdersController** to use this new property.

```
        var order = new Order
        {
            OrderItems = items,
            DeliveryMethod = deliveryMethod,
            ShippingAddress = orderDto.ShippingAddress,
            Subtotal = items.Sum(x => x.Price * x.Quantity),
            Discount = orderDto.Discount,
            PaymentSummary = orderDto.PaymentSummary,
            PaymentIntentId = cart.PaymentIntentId,
            BuyerEmail = email
        };
```

1. Certain order amounts may cause a rounding difference when comparing the Order amount to the Intent amount. This function if added to the webhook will ensure that the amounts are using the same rounding method:

```
// PaymentsController
private async Task HandlePaymentIntentSucceeded(PaymentIntent intent)
{
    if (intent.Status == "succeeded")
    {
        var spec = new OrderSpecification(intent.Id, true);

        var order = await unit.Repository<Order>().GetEntityWithSpec(spec)
            ?? throw new Exception("Order not found");
```

```
        var orderTotalInCents = (long)Math.Round(order.GetTotal() * 100,
            MidpointRounding.AwayFromZero);

        if (orderTotalInCents != intent.Amount)
        {
            order.Status = OrderStatus.PaymentMismatch;
        }
        else
        {
            order.Status = OrderStatus.PaymentReceived;
        }

        await unit.Complete();

        var connectionId = NotificationHub.GetConnectionIdByEmail(order.BuyerEmail);

        if (!string.IsNullOrEmpty(connectionId))
        {
            await hubContext.Clients.Client(connectionId)
                .SendAsync("OrderCompleteNotification", order.ToDto());
        }
    }
}
```

Client project

1. Create a type for the **Coupon** in the **cart.ts** file.

```
import {nanoid} from 'nanoid';

export type CartType = {
    id: string;
    items: CartItem[];
    deliveryMethodId?: number;
    paymentIntentId?: string;
    clientSecret?: string;
    coupon?: Coupon;
}

// cart item omitted

export class Cart implements CartType {
    id = nanoid();
    items: CartItem[] = [];
    deliveryMethodId?: number;
    paymentIntentId?: string;
    clientSecret?: string;
    coupon?: Coupon;
}

export type Coupon = {
    name: string;
    amountOff?: number;
    percentOff?: number;
    promotionCode: string;
    couponId: string;
}
```

1. In the **cart.service.ts** use the **coupon** from the **cart** to update the **totals** signal

```
export class CartService {
  baseUrl = environment.apiUrl;
  private http = inject(HttpClient);
```

```
  private location = inject(Location);
  cart = signal<Cart | null>(null);
  itemCount = computed(() => {
    return this.cart()?.items.reduce((sum, item) => sum + item.quantity, 0)
  });
  selectedDelivery = signal<DeliveryMethod | null>(null);
  totals = computed(() => {
    const cart = this.cart();
    const delivery = this.selectedDelivery();

    if (!cart) return null;
    const subtotal = cart.items.reduce((sum, item) =>
        sum + item.price * item.quantity, 0);

    let discountValue = 0;

    if (cart.coupon) {
      if (cart.coupon.amountOff) {
        discountValue = cart.coupon.amountOff;
      } else if (cart.coupon.percentOff) {
        discountValue = subtotal * (cart.coupon.percentOff / 100);
      }
    }

    const shipping = delivery ? delivery.price : 0;

    const total = subtotal + shipping - discountValue

    return {
      subtotal,
      shipping,
      discount: discountValue,
      total
    }
  })

```

1. In the **cart.service.ts** add an **applyDiscount(code: string): Observable** method to validate the **coupon.**

```
  applyDiscount(code: string) {
    return this.http.get<Coupon>(this.baseUrl + 'coupons/' + code);
  }

```

1. In the **order-summary.component.ts** create and implement the following 2 methods:

```

export class OrderSummaryComponent {
  cartService = inject(CartService);
  private stripeService = inject(StripeService);
  location = inject(Location);
  code?: string;

  applyCouponCode() {
    if (!this.code) return;
    this.cartService.applyDiscount(this.code).subscribe({
      next: async coupon => {
        const cart = this.cartService.cart();
        if (cart) {
          cart.coupon = coupon;
          await firstValueFrom(this.cartService.setCart(cart));
          this.code = undefined;
        }
        if (this.location.path() === '/checkout') {
```

```
            await firstValueFrom(this.stripeService.createOrUpdatePaymentIntent());
          }
        }
      });
    }

    async removeCouponCode() {
      const cart = this.cartService.cart();
      if (!cart) return;
      if (cart.coupon) cart.coupon = undefined;
      await firstValueFrom(this.cartService.setCart(cart));
      if (this.location.path() === '/checkout') {
        await firstValueFrom(this.stripeService.createOrUpdatePaymentIntent());
      }
    }
  }

```

1. Check the **createOrUpdatePaymentIntent** method in the **stripe.service.ts.** Will this cause a problem (hint: yes)? What can we do to prevent the cart being updated if we already have the clientSecret and paymentIntentId?

```
    createOrUpdatePaymentIntent() {
      const cart = this.cartService.cart();
      const hasClientSecret = !!cart?.clientSecret;
      if (!cart) throw new Error('Problem with cart');
      return this.http.post<Cart>(this.baseUrl + 'payments/' + cart.id, {}).pipe(
        map(cart => {
          if (!hasClientSecret) {
            await firstValueFrom(this.cartService.setCart(cart));
            return cart;
          }
          return cart;
        })
      )
    }
```

1. Use the FormsModule from Angular to use 2 way binding to update a **code** property in the **order-summary.component.ts**

```
// imports for the order-summary.component.ts

@Component({
  selector: 'app-order-summary',
  standalone: true,
  imports: [
    MatButton,
    RouterLink,
    MatFormField,
    MatLabel,
    MatInput,
    CurrencyPipe,
    FormsModule,
    NgIf,
    MatIcon
  ],
  templateUrl: './order-summary.component.html',
  styleUrl: './order-summary.component.scss'
})
```

1. Use the **ngSubmit** function from the FormsModule in the form to call the **applyDiscount** method from the template.
2. Disable the input if we have a coupon in the cart

3. Disable the button if we have a coupon in the cart
4. Display the name of the coupon that has been applied above the input. Provide an icon button to remove the applied coupon that calls the **removeCouponCode** method.

```
// code for steps 7, 8, 9 and 10
 <div class="space-y-4 rounded-lg border border-gray-200 bg-white shadow-sm">
    <form #form="ngForm" (ngSubmit)="applyCouponCode()"
        class="space-y-2 flex flex-col p-2">
      <label class="mb-2 block text-sm font-medium">Do you have a voucher
        code?</label>
      @if (cartService.cart()?.coupon; as coupon) {
      <div class="flex justify-between items-center">
        <span class="text-sm font-semibold">{{coupon.name}} applied</span>
        <button (click)="removeCouponCode()" mat-icon-button>
          <mat-icon color="warn">delete</mat-icon>
        </button>
      </div>
      }

      <mat-form-field appearance="outline">
        <mat-label>Voucher code</mat-label>
        <input [disabled]="!!cartService.cart()?.coupon" [(ngModel)]="code"
            name="code" type="text" matInput>
      </mat-form-field>

      <button
        [disabled]="!!cartService.cart()?.coupon"
        type="submit"
        mat-flat-button>Apply code</button>
    </form>
  </div>
```

1. Update the **Order** interface and the **OrderToCreate** in the **order.ts** to include the **discount**

```
export interface Order {
    id: number
    orderDate: string
    buyerEmail: string
    shippingAddress: ShippingAddress
    deliveryMethod: string
    shippingPrice: number
    paymentSummary: PaymentSummary
    orderItems: OrderItem[]
    subtotal: number
    discount?: number
    status: string
    total: number
    paymentIntentId: string
  }

  export interface OrderToCreate {
    cartId: string;
    deliveryMethodId: number;
    shippingAddress: ShippingAddress;
    paymentSummary: PaymentSummary;
    discount?: number;
  }
```

1. Update the **order-detailed.component.html** to display this

```
                        <dl class="flex items-center justify-between gap-4">
                            <dt class="font-medium text-gray-500">Discount</dt>
                            <dd class="font-medium text-green-500">
```

```
                                -{{order.discount | currency}}
                            </dd>
                        </dl>
```

1. In the **checkout.component.ts** update the **createOrderModel** method to include the discount when creating the order.

```
private async createOrderModel(): Promise<OrderToCreate> {
    const cart = this.cartService.cart();
    const shippingAddress = await this.getAddressFromStripeAddress() as ShippingAddress;
    const card = this.confirmationToken?.payment_method_preview.card;

    if (!cart?.id || !cart.deliveryMethodId || !card || !shippingAddress) {
      throw new Error('Problem creating order');
    }

    return {
      cartId: cart.id,
      paymentSummary: {
        last4: +card.last4,
        brand: card.brand,
        expMonth: card.exp_month,
        expYear: card.exp_year
      },
      deliveryMethodId: cart.deliveryMethodId,
      shippingAddress,
      discount: this.cartService.totals()?.discount
    }
  }
```

1. Test the new functionality.

Challenge complete! Publish changes to production