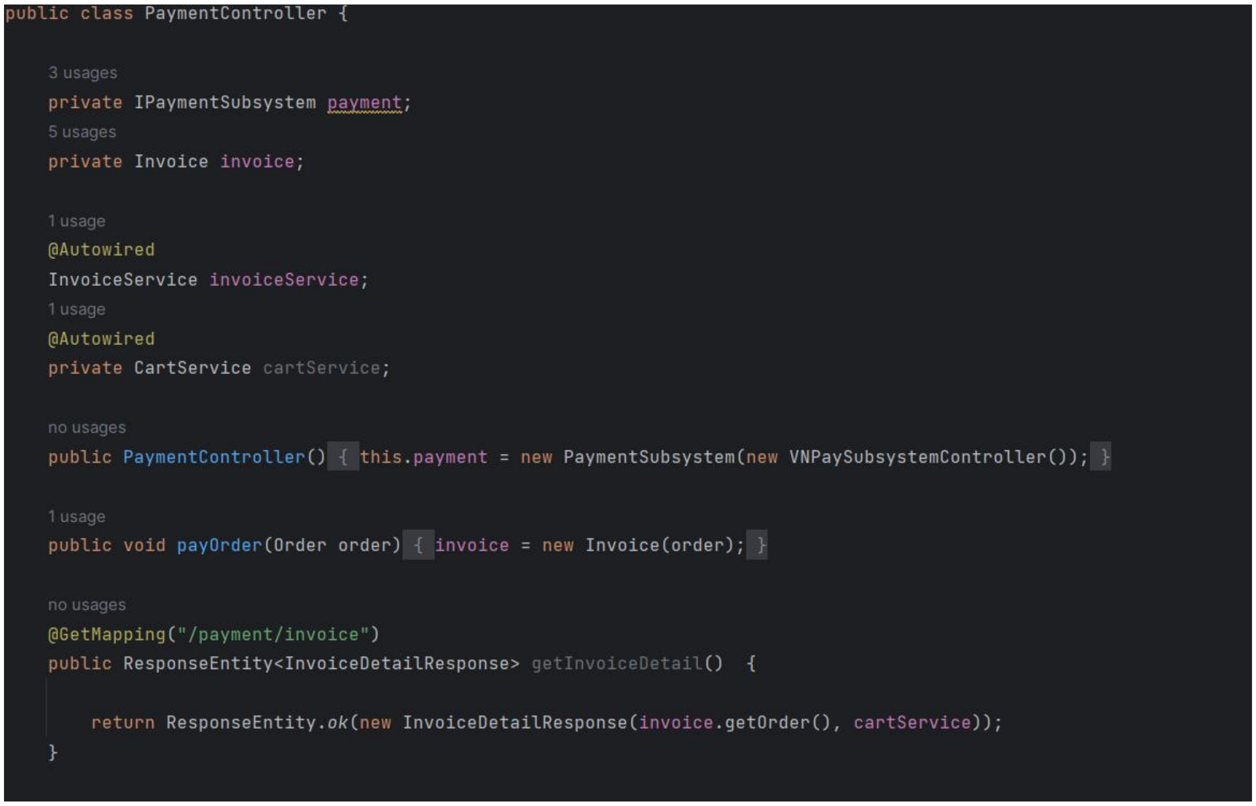
REPORT ITSS

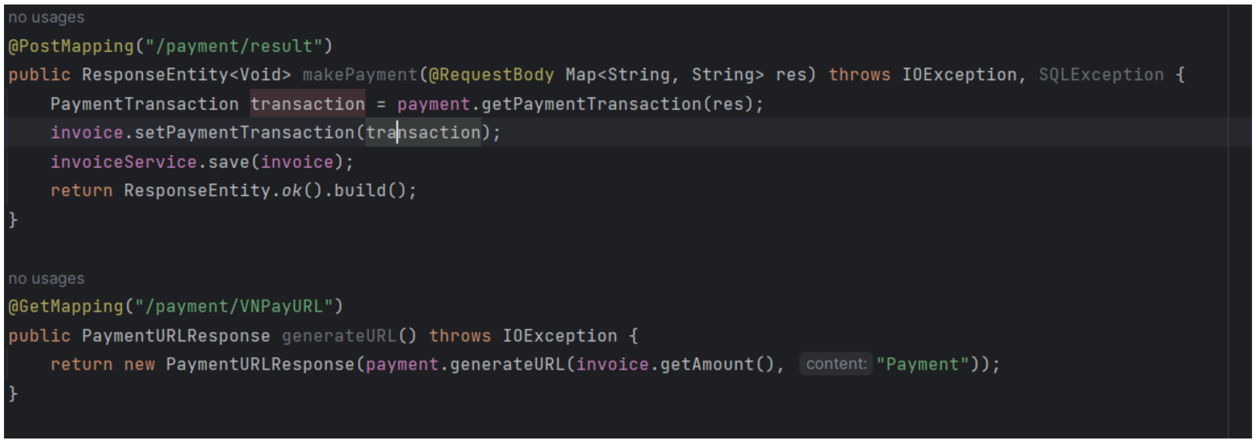
THE LISKOV SUBSTITUTION PRINCIPLE

To comply with the Liskov Substitution Principle (LSP) we need to ensure that subclasses can replace superclasses without changing the correctness of the program. An important part of LSP is ensuring that child objects can substitute for parent objects without causing errors or unexpected behavior.

1. ***At PaymentController class***



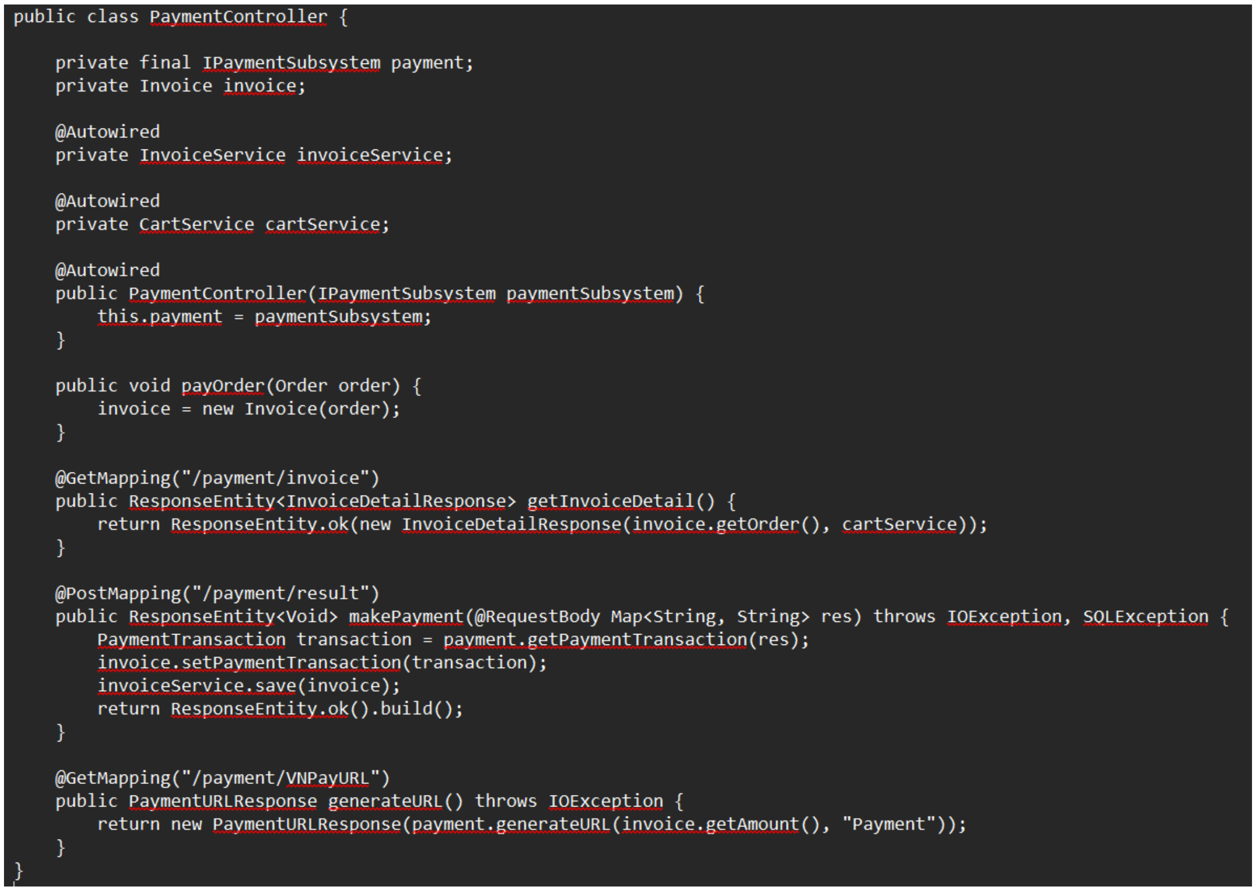
```
public class PaymentController {

    3 usages
    private IPaymentSubsystem payment;
    5 usages
    private Invoice invoice;

    1 usage
    @Autowired
    InvoiceService invoiceService;
    1 usage
    @Autowired
    private CartService cartService;

    no usages
    public PaymentController() { this.payment = new PaymentSubsystem(new VNPaySubsystemController()); }

    1 usage
    public void payOrder(Order order) { invoice = new Invoice(order); }

    no usages
    @GetMapping("/payment/invoice")
    public ResponseEntity<InvoiceDetailResponse> getInvoiceDetail()  {

        return ResponseEntity.ok(new InvoiceDetailResponse(invoice.getOrder(), cartService));
    }
```



```
no usages
@PostMapping("/payment/result")
public ResponseEntity<Void> makePayment(@RequestBody Map<String, String> res) throws IOException, SQLException {
    PaymentTransaction transaction = payment.getPaymentTransaction(res);
    invoice.setPaymentTransaction(transaction);
    invoiceService.save(invoice);
    return ResponseEntity.ok().build();
}

no usages
@GetMapping("/payment/VNPayURL")
public PaymentURLResponse generateURL() throws IOException {
    return new PaymentURLResponse(payment.generateURL(invoice.getAmount(),  content: "Payment"));
}
```

Some key point for improvement: In the above code, it has some potential problems with the use of subclasses and interfaces. I recommended the way to fix the code to ensure LSP principle.

- Using Dependency Injection to initialize objects: Currently, we are initializing the PaymentSubsystem object in the PaymentController constructor. This makes PaymentController directly dependent on PaymentSubsystem, violating LSP if we want to replace PaymentSubsystem with another payment system. We should use Dependency Injection to initialize these objects.
- Make sure subclasses can substitute for superclasses: We should ensure that subclasses of IPaymentSubsystem can substitute for IPaymentSubsystem without changing the behavior of the program.

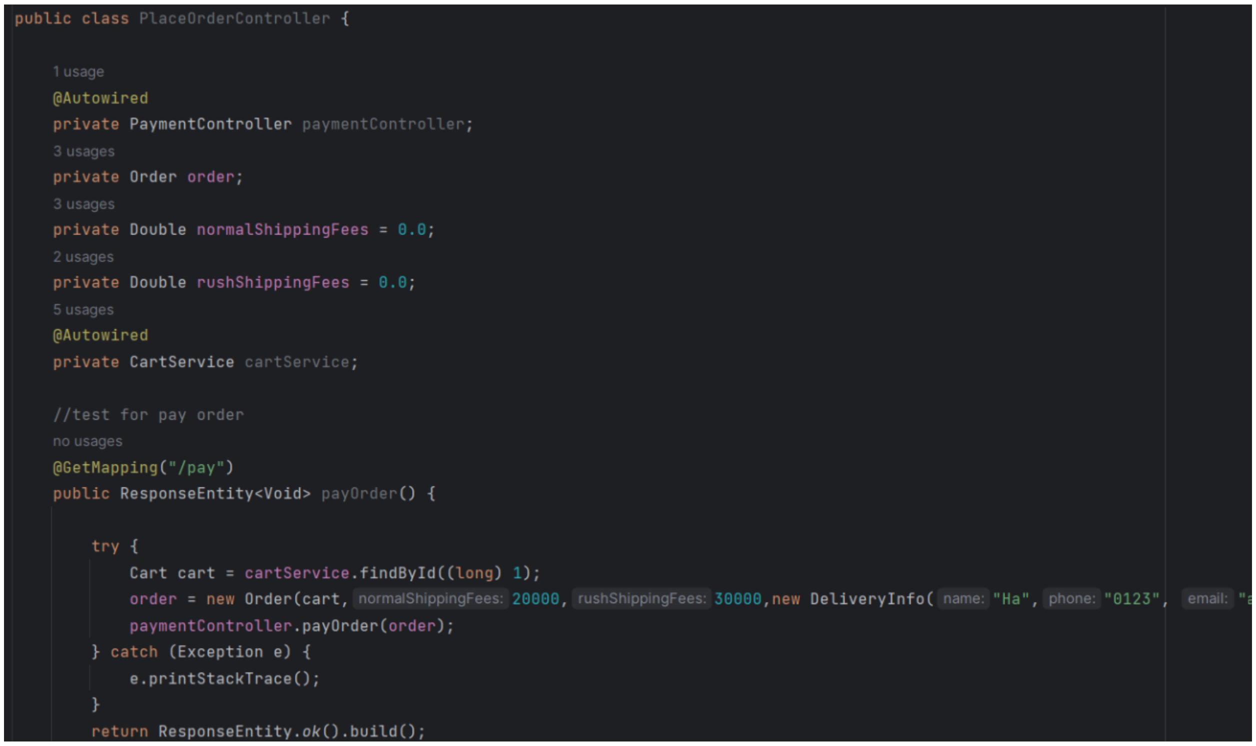


```
public class PaymentController {

    private final IPaymentSubsystem payment;
    private Invoice invoice;

    @Autowired
    private InvoiceService invoiceService;

    @Autowired
    private CartService cartService;

    @Autowired
    public PaymentController(IPaymentSubsystem paymentSubsystem) {
        this.payment = paymentSubsystem;
    }

    public void payOrder(Order order) {
        invoice = new Invoice(order);
    }

    @GetMapping("/payment/invoice")
    public ResponseEntity<InvoiceDetailResponse> getInvoiceDetail() {
        return ResponseEntity.ok(new InvoiceDetailResponse(invoice.getOrder(), cartService));
    }

    @PostMapping("/payment/result")
    public ResponseEntity<Void> makePayment(@RequestBody Map<String, String> res) throws IOException, SQLException {
        PaymentTransaction transaction = payment.getPaymentTransaction(res);
        invoice.setPaymentTransaction(transaction);
        invoiceService.save(invoice);
        return ResponseEntity.ok().build();
    }

    @GetMapping("/payment/VNPayURL")
    public PaymentURLResponse generateURL() throws IOException {
        return new PaymentURLResponse(payment.generateURL(invoice.getAmount(), "Payment"));
    }
}
```

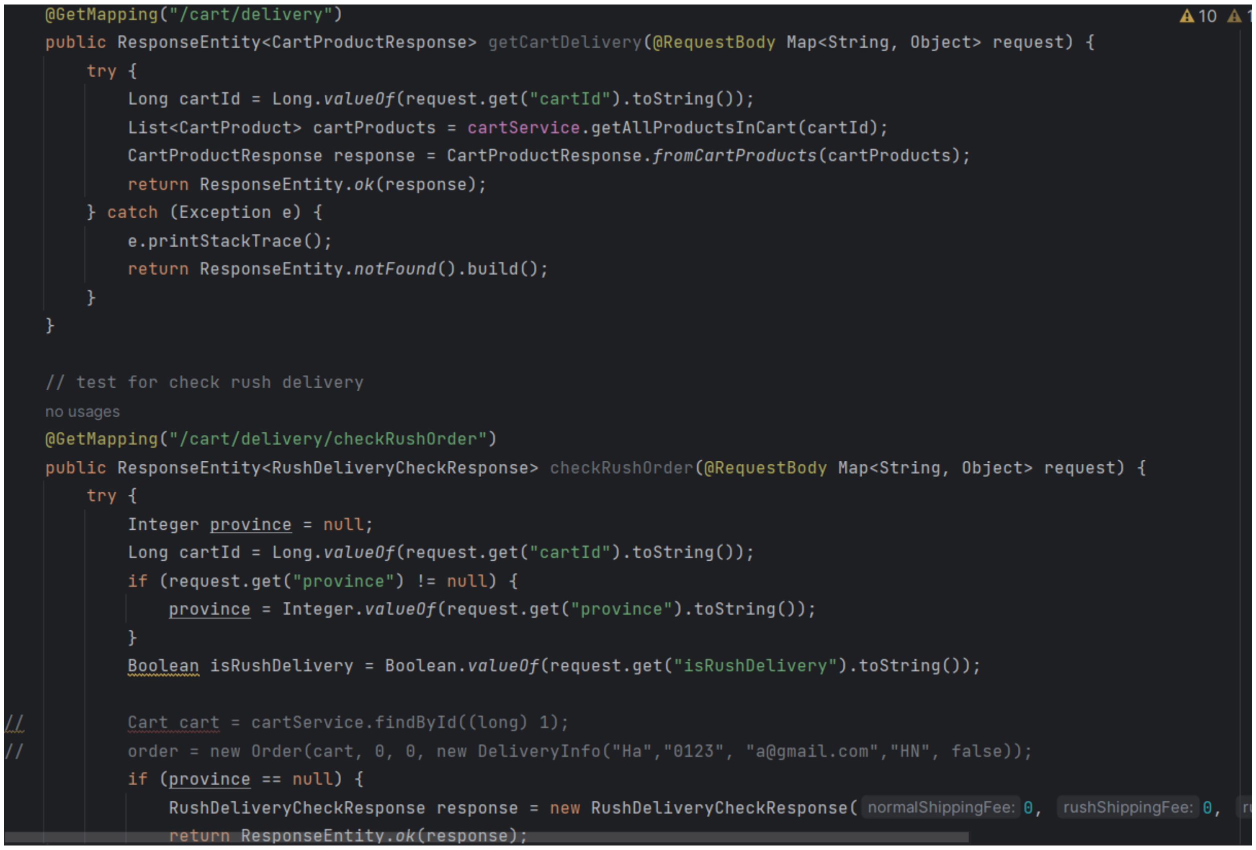
Some main changes:

- Dependency Injection: Instead of initializing PaymentSubsystem in the constructor, we use Dependency Injection to pass in an IPaymentSubsystem object. This makes it easy to replace IPaymentSubsystem with any other payment system without changing the source code of PaymentController.
- Initialization via constructor: The IPaymentSubsystem object is initialized via constructor and managed by Spring, ensuring flexibility and ease of replacement.

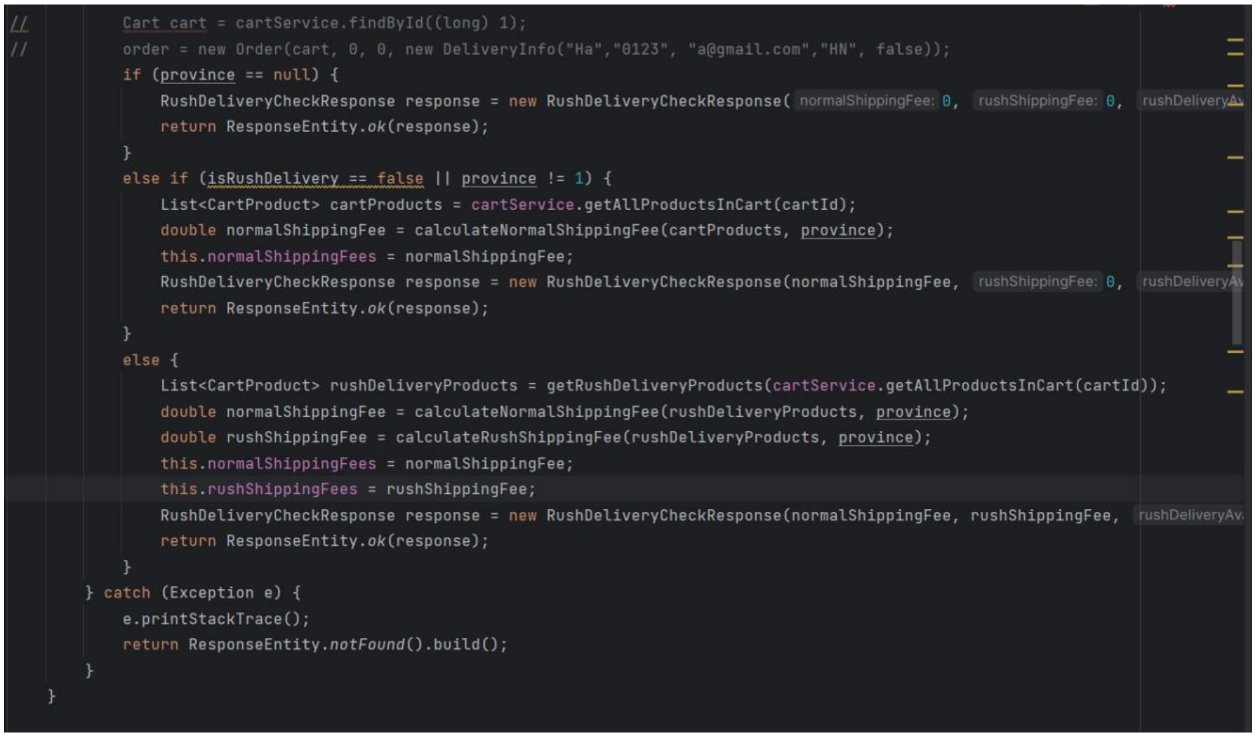
2. At file PlaceOrderController:



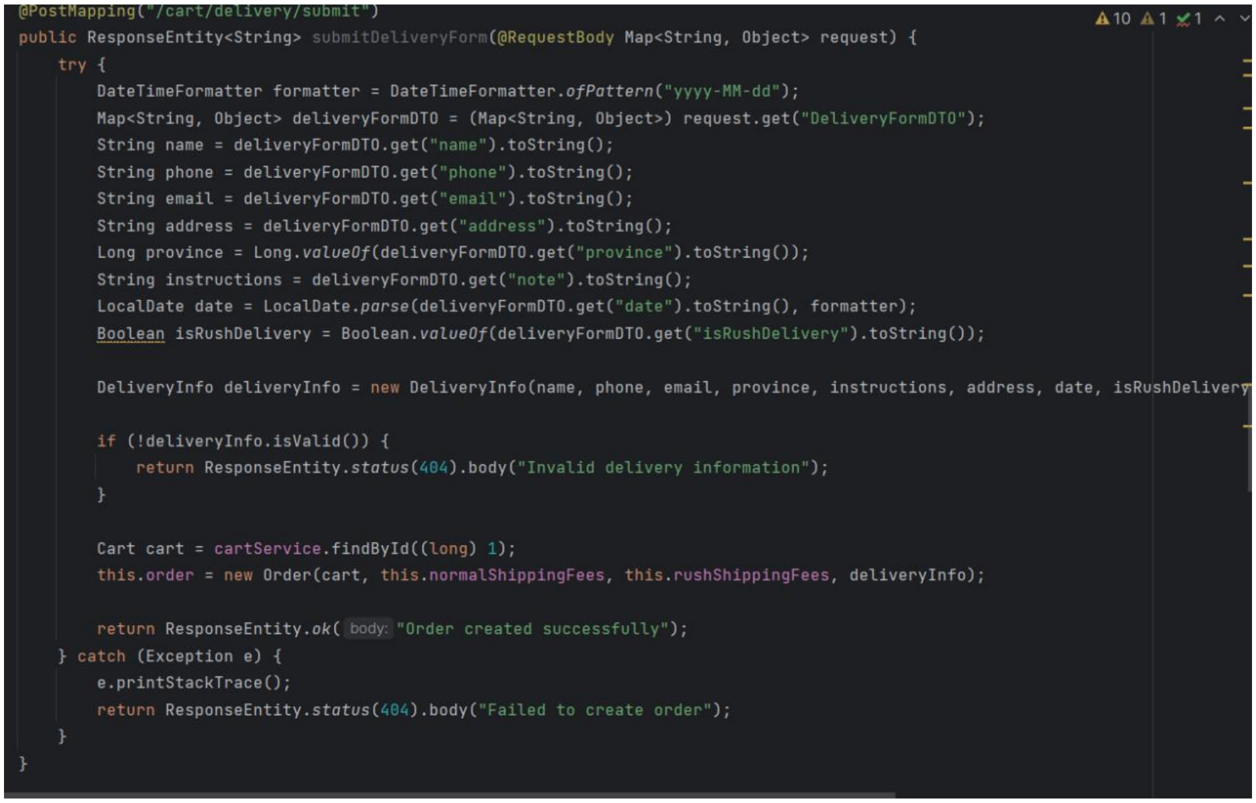
```java
public class PlaceOrderController {

    1 usage
    @Autowired
    private PaymentController paymentController;
    3 usages
    private Order order;
    3 usages
    private Double normalShippingFees = 0.0;
    2 usages
    private Double rushShippingFees = 0.0;
    5 usages
    @Autowired
    private CartService cartService;

    //test for pay order
    no usages
    @GetMapping("/pay")
    public ResponseEntity<Void> payOrder() {

        try {
            Cart cart = cartService.findById((long) 1);
            order = new Order(cart, normalShippingFees: 20000, rushShippingFees: 30000,new DeliveryInfo( name: "Ha", phone: "0123",  email: "a
            paymentController.payOrder(order);
        } catch (Exception e) {
            e.printStackTrace();
        }
        return ResponseEntity.ok().build();
```



```java
    @GetMapping("/cart/delivery")
    public ResponseEntity<CartProductResponse> getCartDelivery(@RequestBody Map<String, Object> request) {
        try {
            Long cartId = Long.valueOf(request.get("cartId").toString());
            List<CartProduct> cartProducts = cartService.getAllProductsInCart(cartId);
            CartProductResponse response = CartProductResponse.fromCartProducts(cartProducts);
            return ResponseEntity.ok(response);
        } catch (Exception e) {
            e.printStackTrace();
            return ResponseEntity.notFound().build();
        }
    }

    // test for check rush delivery
    no usages
    @GetMapping("/cart/delivery/checkRushOrder")
    public ResponseEntity<RushDeliveryCheckResponse> checkRushOrder(@RequestBody Map<String, Object> request) {
        try {
            Integer province = null;
            Long cartId = Long.valueOf(request.get("cartId").toString());
            if (request.get("province") != null) {
                province = Integer.valueOf(request.get("province").toString());
            }
            Boolean isRushDelivery = Boolean.valueOf(request.get("isRushDelivery").toString());

//          Cart cart = cartService.findById((long) 1);
//          order = new Order(cart, 0, 0, new DeliveryInfo("Ha","0123", "a@gmail.com","HN", false));
            if (province == null) {
                RushDeliveryCheckResponse response = new RushDeliveryCheckResponse( normalShippingFee: 0,  rushShippingFee: 0,  r
                return ResponseEntity.ok(response);
```



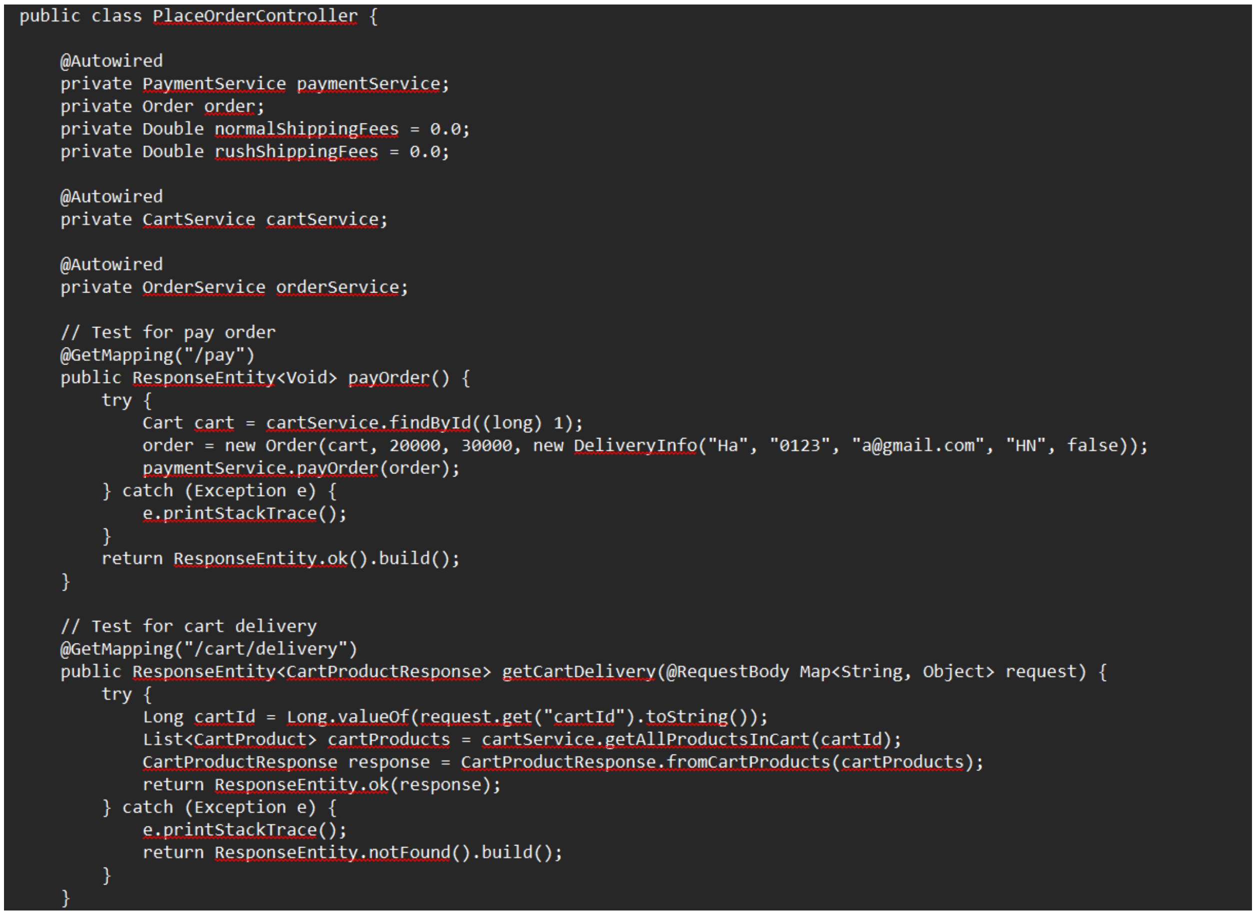
```java
//          Cart cart = cartService.findById((long) 1);
//          order = new Order(cart, 0, 0, new DeliveryInfo("Ha","0123", "a@gmail.com","HN", false));
            if (province == null) {
                RushDeliveryCheckResponse response = new RushDeliveryCheckResponse( normalShippingFee: 0,  rushShippingFee: 0,  rushDeliveryAv
                return ResponseEntity.ok(response);
            }
            else if (isRushDelivery == false || province != 1) {
                List<CartProduct> cartProducts = cartService.getAllProductsInCart(cartId);
                double normalShippingFee = calculateNormalShippingFee(cartProducts, province);
                this.normalShippingFees = normalShippingFee;
                RushDeliveryCheckResponse response = new RushDeliveryCheckResponse(normalShippingFee,  rushShippingFee: 0,  rushDeliveryAv
                return ResponseEntity.ok(response);
            }
            else {
                List<CartProduct> rushDeliveryProducts = getRushDeliveryProducts(cartService.getAllProductsInCart(cartId));
                double normalShippingFee = calculateNormalShippingFee(rushDeliveryProducts, province);
                double rushShippingFee = calculateRushShippingFee(rushDeliveryProducts, province);
                this.normalShippingFees = normalShippingFee;
                this.rushShippingFees = rushShippingFee;
                RushDeliveryCheckResponse response = new RushDeliveryCheckResponse(normalShippingFee, rushShippingFee,  rushDeliveryAv
                return ResponseEntity.ok(response);
            }
        } catch (Exception e) {
            e.printStackTrace();
            return ResponseEntity.notFound().build();
        }
    }
```



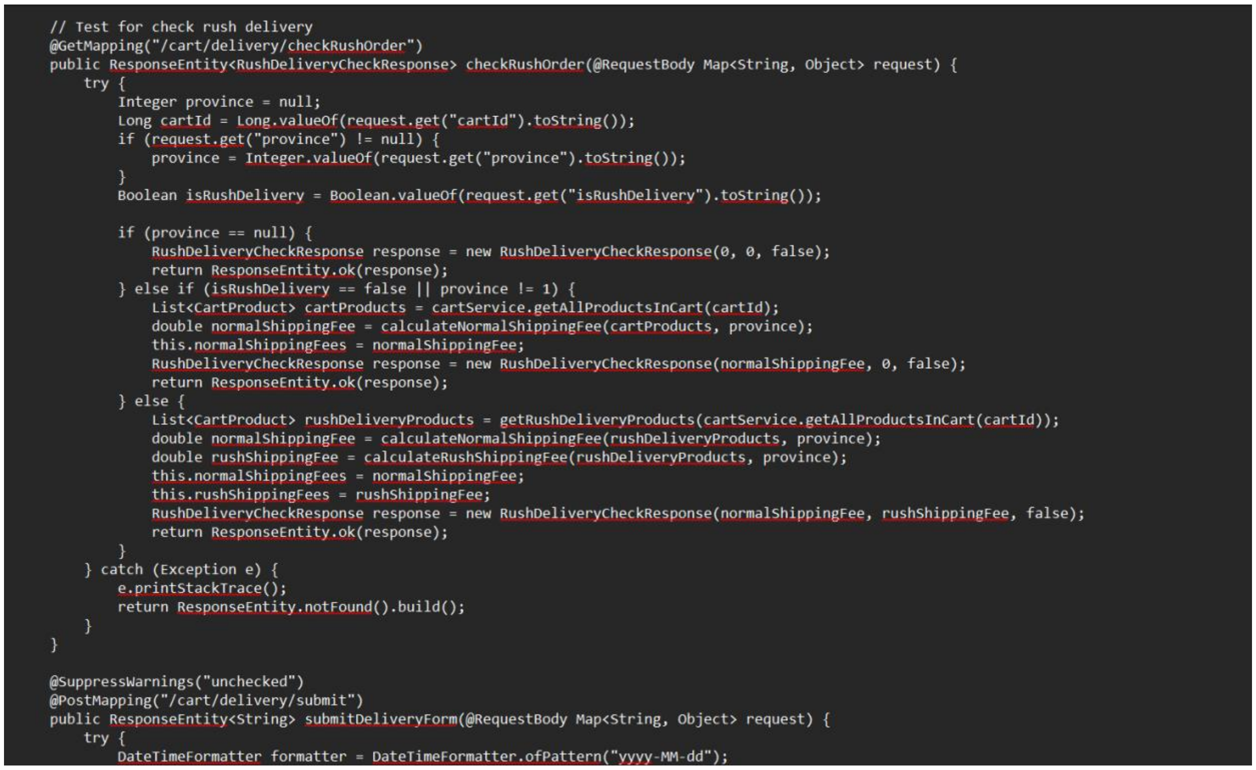
```java
@PostMapping("/cart/delivery/submit")
public ResponseEntity<String> submitDeliveryForm(@RequestBody Map<String, Object> request) {
    try {
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd");
        Map<String, Object> deliveryFormDTO = (Map<String, Object>) request.get("DeliveryFormDTO");
        String name = deliveryFormDTO.get("name").toString();
        String phone = deliveryFormDTO.get("phone").toString();
        String email = deliveryFormDTO.get("email").toString();
        String address = deliveryFormDTO.get("address").toString();
        Long province = Long.valueOf(deliveryFormDTO.get("province").toString());
        String instructions = deliveryFormDTO.get("note").toString();
        LocalDate date = LocalDate.parse(deliveryFormDTO.get("date").toString(), formatter);
        Boolean isRushDelivery = Boolean.valueOf(deliveryFormDTO.get("isRushDelivery").toString());

        DeliveryInfo deliveryInfo = new DeliveryInfo(name, phone, email, province, instructions, address, date, isRushDelivery

        if (!deliveryInfo.isValid()) {
            return ResponseEntity.status(404).body("Invalid delivery information");
        }

        Cart cart = cartService.findById((long) 1);
        this.order = new Order(cart, this.normalShippingFees, this.rushShippingFees, deliveryInfo);

        return ResponseEntity.ok( body: "Order created successfully");
    } catch (Exception e) {
        e.printStackTrace();
        return ResponseEntity.status(404).body("Failed to create order");
    }
}
```

Some key points for improvement:

- Reduced dependency on PaymentController: Instead of using PaymentController directly, we can separate the payment logic into a separate service for easy replacement and extension.
- Use Dependency Injection properly: Ensure that dependencies are properly managed by Spring.



```java
public class PlaceOrderController {

    @Autowired
    private PaymentService paymentService;
    private Order order;
    private Double normalShippingFees = 0.0;
    private Double rushShippingFees = 0.0;

    @Autowired
    private CartService cartService;

    @Autowired
    private OrderService orderService;

    // Test for pay order
    @GetMapping("/pay")
    public ResponseEntity<Void> payOrder() {
        try {
            Cart cart = cartService.findById((long) 1);
            order = new Order(cart, 20000, 30000, new DeliveryInfo("Ha", "0123", "a@gmail.com", "HN", false));
            paymentService.payOrder(order);
        } catch (Exception e) {
            e.printStackTrace();
        }
        return ResponseEntity.ok().build();
    }

    // Test for cart delivery
    @GetMapping("/cart/delivery")
    public ResponseEntity<CartProductResponse> getCartDelivery(@RequestBody Map<String, Object> request) {
        try {
            Long cartId = Long.valueOf(request.get("cartId").toString());
            List<CartProduct> cartProducts = cartService.getAllProductsInCart(cartId);
            CartProductResponse response = CartProductResponse.fromCartProducts(cartProducts);
            return ResponseEntity.ok(response);
        } catch (Exception e) {
            e.printStackTrace();
            return ResponseEntity.notFound().build();
        }
    }
```

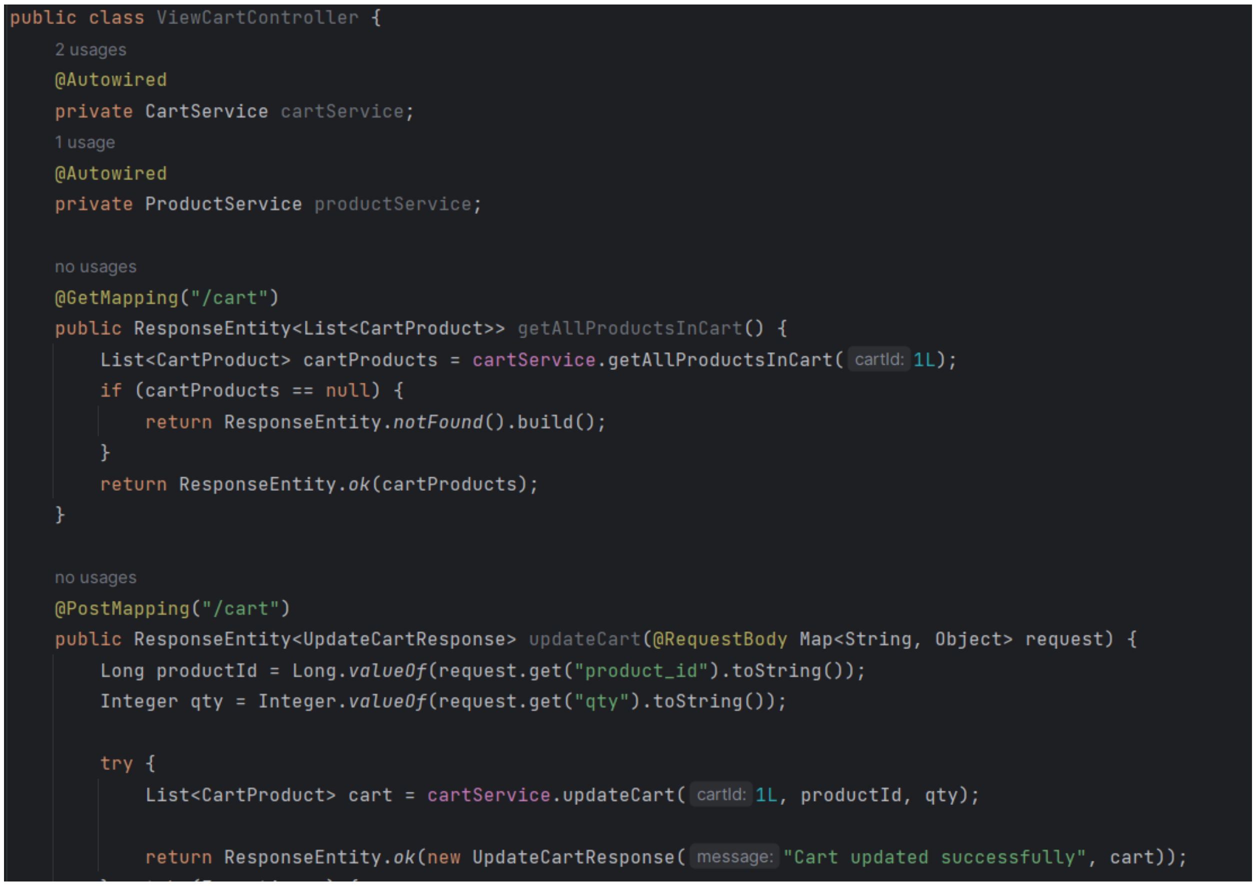


```
    // Test for check rush delivery
    @GetMapping("/cart/delivery/checkRushOrder")
    public ResponseEntity<RushDeliveryCheckResponse> checkRushOrder(@RequestBody Map<String, Object> request) {
        try {
            Integer province = null;
            Long cartId = Long.valueOf(request.get("cartId").toString());
            if (request.get("province") != null) {
                province = Integer.valueOf(request.get("province").toString());
            }
            Boolean isRushDelivery = Boolean.valueOf(request.get("isRushDelivery").toString());

            if (province == null) {
                RushDeliveryCheckResponse response = new RushDeliveryCheckResponse(0, 0, false);
                return ResponseEntity.ok(response);
            } else if (isRushDelivery == false || province != 1) {
                List<CartProduct> cartProducts = cartService.getAllProductsInCart(cartId);
                double normalShippingFee = calculateNormalShippingFee(cartProducts, province);
                this.normalShippingFees = normalShippingFee;
                RushDeliveryCheckResponse response = new RushDeliveryCheckResponse(normalShippingFee, 0, false);
                return ResponseEntity.ok(response);
            } else {
                List<CartProduct> rushDeliveryProducts = getRushDeliveryProducts(cartService.getAllProductsInCart(cartId));
                double normalShippingFee = calculateNormalShippingFee(rushDeliveryProducts, province);
                double rushShippingFee = calculateRushShippingFee(rushDeliveryProducts, province);
                this.normalShippingFees = normalShippingFee;
                this.rushShippingFees = rushShippingFee;
                RushDeliveryCheckResponse response = new RushDeliveryCheckResponse(normalShippingFee, rushShippingFee, false);
                return ResponseEntity.ok(response);
            }
        } catch (Exception e) {
            e.printStackTrace();
            return ResponseEntity.notFound().build();
        }
    }

    @SuppressWarnings("unchecked")
    @PostMapping("/cart/delivery/submit")
    public ResponseEntity<String> submitDeliveryForm(@RequestBody Map<String, Object> request) {
        try {
            DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd");
```

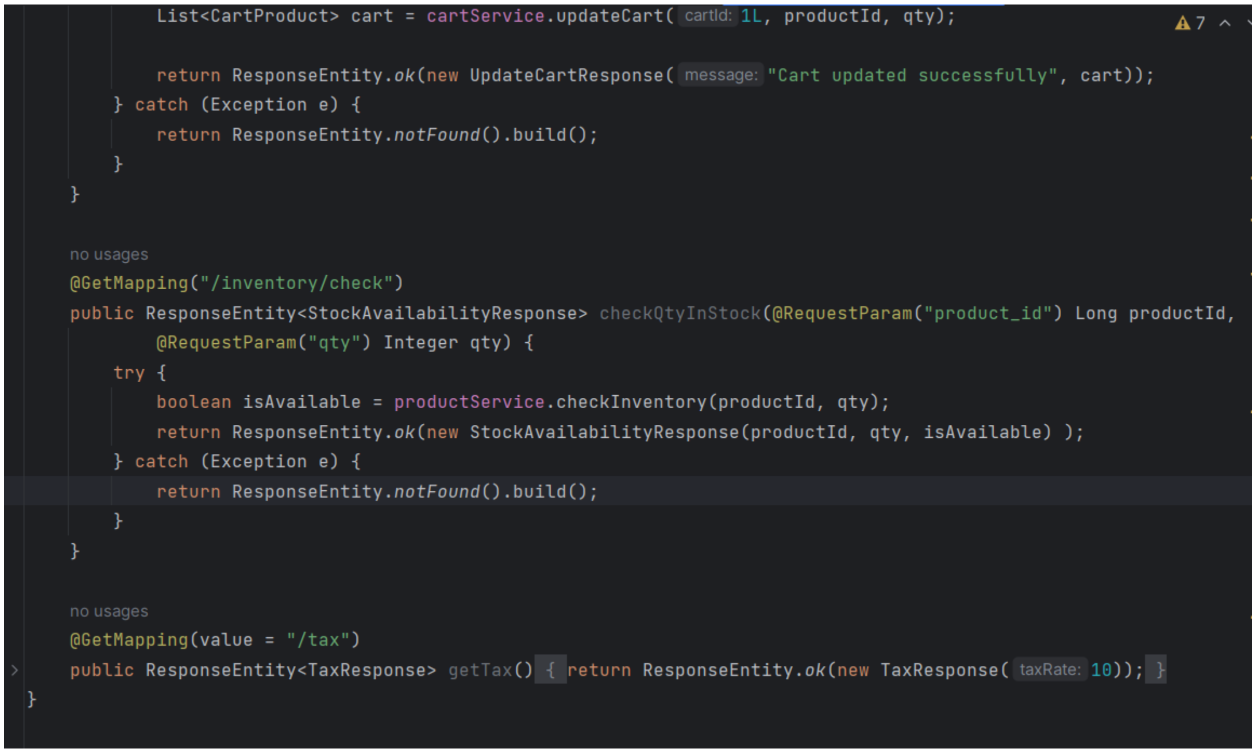
Main changes:

- Using PaymentService: Instead of depending on PaymentController directly, we use PaymentService. This service can be easily replaced or expanded

1. ***ViewCartController***



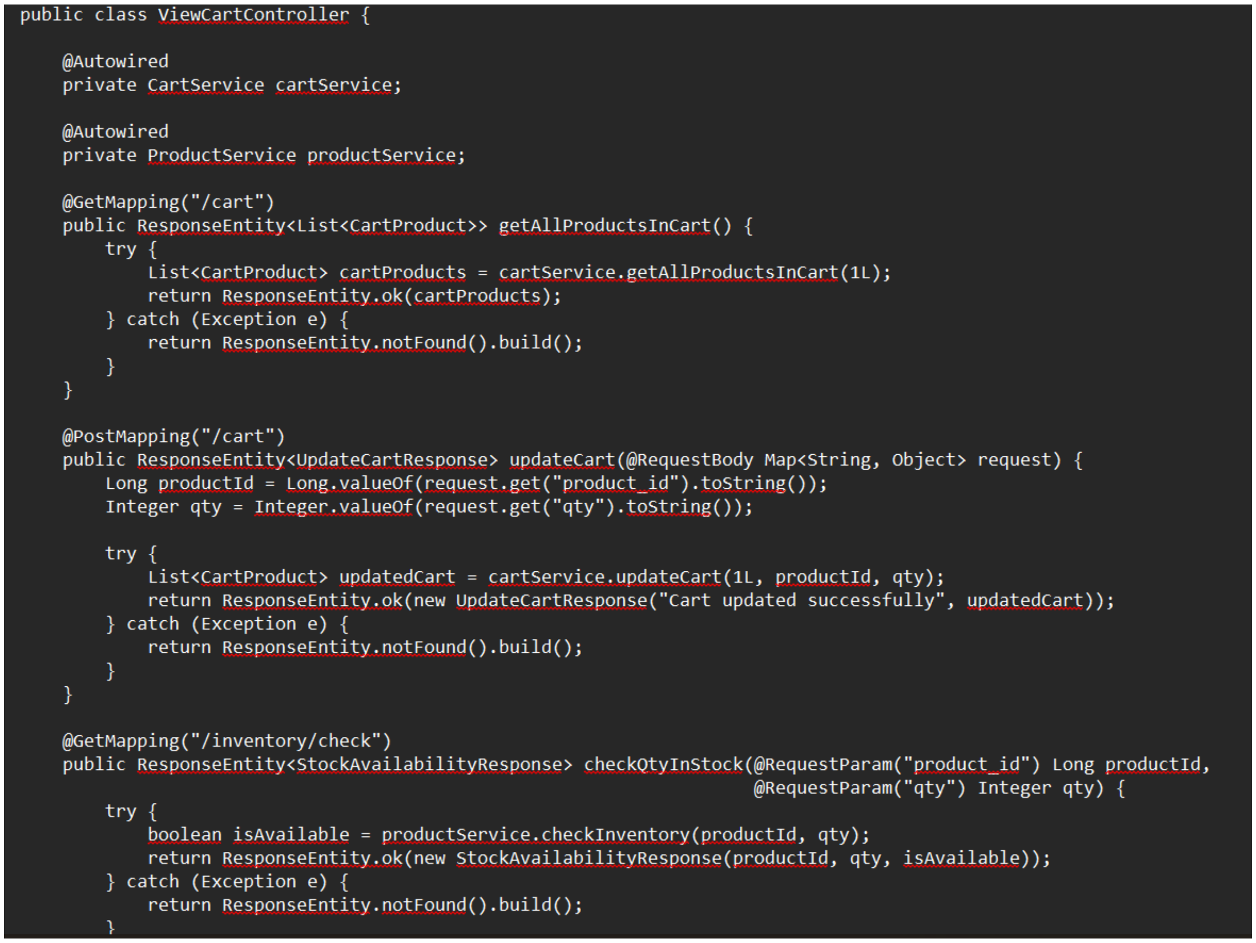
```
public class ViewCartController {
    2 usages
    @Autowired
    private CartService cartService;
    1 usage
    @Autowired
    private ProductService productService;

    no usages
    @GetMapping("/cart")
    public ResponseEntity<List<CartProduct>> getAllProductsInCart() {
        List<CartProduct> cartProducts = cartService.getAllProductsInCart( cartId: 1L);
        if (cartProducts == null) {
            return ResponseEntity.notFound().build();
        }
        return ResponseEntity.ok(cartProducts);
    }

    no usages
    @PostMapping("/cart")
    public ResponseEntity<UpdateCartResponse> updateCart(@RequestBody Map<String, Object> request) {
        Long productId = Long.valueOf(request.get("product_id").toString());
        Integer qty = Integer.valueOf(request.get("qty").toString());

        try {
            List<CartProduct> cart = cartService.updateCart( cartId: 1L, productId, qty);

            return ResponseEntity.ok(new UpdateCartResponse( message: "Cart updated successfully", cart));
```



```
            List<CartProduct> cart = cartService.updateCart( cartId: 1L, productId, qty);

            return ResponseEntity.ok(new UpdateCartResponse( message: "Cart updated successfully", cart));
        } catch (Exception e) {
            return ResponseEntity.notFound().build();
        }
    }

    no usages
    @GetMapping("/inventory/check")
    public ResponseEntity<StockAvailabilityResponse> checkQtyInStock(@RequestParam("product_id") Long productId,
            @RequestParam("qty") Integer qty) {
        try {
            boolean isAvailable = productService.checkInventory(productId, qty);
            return ResponseEntity.ok(new StockAvailabilityResponse(productId, qty, isAvailable) );
        } catch (Exception e) {
            return ResponseEntity.notFound().build();
        }
    }

    no usages
    @GetMapping(value = "/tax")
    public ResponseEntity<TaxResponse> getTax() { return ResponseEntity.ok(new TaxResponse( taxRate: 10)); }
}
```

Some key points to improvement:

- Separation of processing logic: Ensure that controller methods are only responsible for calling services and returning results, and do not contain complex processing logic.
- Use Dependency Injection properly: Ensure that services are managed by Spring.



```java
public class ViewCartController {

    @Autowired
    private CartService cartService;

    @Autowired
    private ProductService productService;

    @GetMapping("/cart")
    public ResponseEntity<List<CartProduct>> getAllProductsInCart() {
        try {
            List<CartProduct> cartProducts = cartService.getAllProductsInCart(1L);
            return ResponseEntity.ok(cartProducts);
        } catch (Exception e) {
            return ResponseEntity.notFound().build();
        }
    }

    @PostMapping("/cart")
    public ResponseEntity<UpdateCartResponse> updateCart(@RequestBody Map<String, Object> request) {
        Long productId = Long.valueOf(request.get("product_id").toString());
        Integer qty = Integer.valueOf(request.get("qty").toString());

        try {
            List<CartProduct> updatedCart = cartService.updateCart(1L, productId, qty);
            return ResponseEntity.ok(new UpdateCartResponse("Cart updated successfully", updatedCart));
        } catch (Exception e) {
            return ResponseEntity.notFound().build();
        }
    }

    @GetMapping("/inventory/check")
    public ResponseEntity<StockAvailabilityResponse> checkQtyInStock(@RequestParam("product_id") Long productId,
                                                                     @RequestParam("qty") Integer qty) {
        try {
            boolean isAvailable = productService.checkInventory(productId, qty);
            return ResponseEntity.ok(new StockAvailabilityResponse(productId, qty, isAvailable));
        } catch (Exception e) {
            return ResponseEntity.notFound().build();
        }
```

Main changes:

- Separation of processing logic: There are no major changes in this method, but it is important to ensure that the methods are only responsible for calling the service and returning results. Complex processing logic should be placed in service layers.
- Exception handling: Ensure that every method has exception handling that returns the appropriate response if an error occurs.
- Service dependency: Controller methods use the CartService and ProductService services to perform the necessary work, ensuring that these services can be replaced or extended without affecting correctness. of ViewCartController.