



E-Commerce Application

CS5722 – Software Architecture

TEAM MEMBERS

Prashant Agarwal	19089198
Mohammad Tauseef Khan	19140886
Rainy Mishal Dsouza	19064918
Tarun Kumar Sunkara	19116535

Table of Contents

1.Business Scenario.....	2
Overview	2
Business Requirements.....	2
2. USE CASE Diagram.....	3
USE CASE Descriptions	4
3.Quality Attributes/Tactics	5
4. Patterns.....	6
A. Architectural Pattern	6
The Interceptor	6
Implementation	7
Dispatcher	7
B. Design Patterns.....	8
Strategy Pattern	8
Bridge pattern	9
Builder pattern	10
Visitor pattern.....	11
Command pattern.....	14
Pluggable Adapter.....	15
C. Independently Researched Patterns.....	17
1.Microservices Architecture	17
Advantages.....	17
2. Layered Architecture.....	18
Implementation	18
3.Enterprise Pattern: Session Management	19
How it works?	19
Advantages.....	19
Disadvantages	19
Implementation	19
Code	20
5.Architectural view of the System.....	22
6.Structural and Behavioral diagrams.....	25
Class Diagram	25

Sequence Diagram	28
7. Coding Fragments	28
Strategy Pattern	28
Participant classes	28
Code	29
Bridge Pattern	30
Participant classes	30
Code	30
Builder	32
Participant Classes	32
Code	32
5. Added Value	34
Severity of type Major	37
Severity of type Info	38
Severity of type Minor #1	39
Severity of type Minor #2	40
Netflix Eureka Service Discovery	42
ZuulApi Gateway	43
7. Testing	44
8. Visualization	47
9. Discussion of Problems Encountered	51
10. Evaluation/ Critique of support for NFRs	52
11. Team Member Contribution Overview	52
12. References	59

List of Figures

Figure 1 - The Interceptor Pattern	6
Figure 2: Class diagram for Strategy Pattern	9
Figure 3: Class diagram for Bridge Pattern	10
Figure 4: Class diagram for Builder Pattern	11
Figure 5: Class diagram for Visitor pattern	12
Figure 6: Architecture Diagram of E-commerce system	18
Figure 7: JWT structure with header, payload and signature.....	20
Figure 8 : Architectural diagram for Cart microservice.....	22
Figure 9: Architectural Diagram for Checkout Microservice	23
Figure 10: Architectural Diagram for Payment Microservice	24
Figure 11- Class Diagram for Login microservice	25
Figure 12 - Class Diagram for Product microservice	25
Figure 13 - Class Diagram for Shopping-Cart microservice.....	26
Figure 14: Class Diagram for Checkout microservice.....	26
Figure 15 - Class Diagram for Payment microservice	27
Figure 16: Sequence diagram for Shopping Cart	28

This is a new project and **not** a continuation of the **CS5721** project.

The team for **CS5722** patterns project is a new team and not the same as **CS721** project.

Mohammed Tauseef Khan:

- 1) Scenario and key use cases:** Application Login and Sign up, Location Finding, Bookings.
- 2) Technologies and programming languages:** Java, Android, Dependency Injection.
- 3) Patterns:** Builder, Command.
- 4) Added Value:** Android Application.

Prashant Agarwal:

- 1) Scenario and key use cases:** Login security signup use case, Promotion use case.
- 2) Technologies and programming languages:** Spring MVC monolithic application, Java.
- 3) Patterns:** Factory, Mediator.
- 4) Added Value:** Multithreading.

Rainy Mishal Dsouza:

- 1) Scenario and key use cases:** Use case for admin activity i.e. Adding/deleting facility, instructor, location.
- 2) Technologies and programming languages:** Java and Spring MVC monolithic application.
- 3) Patterns:** Command Design Pattern.
- 4) Added Value:** RESTful Services.

Tarun Kumar Sunkara:

- 1) Scenario and key use cases:** Payment use case.
- 2) Technologies and programming languages:** Java and Spring MVC Monolithic application.
- 3) Patterns:** Decorator and State Design Patterns.
- 4) Added Value:** RESTful Services and Multithreading.

1. Business Scenario

Overview

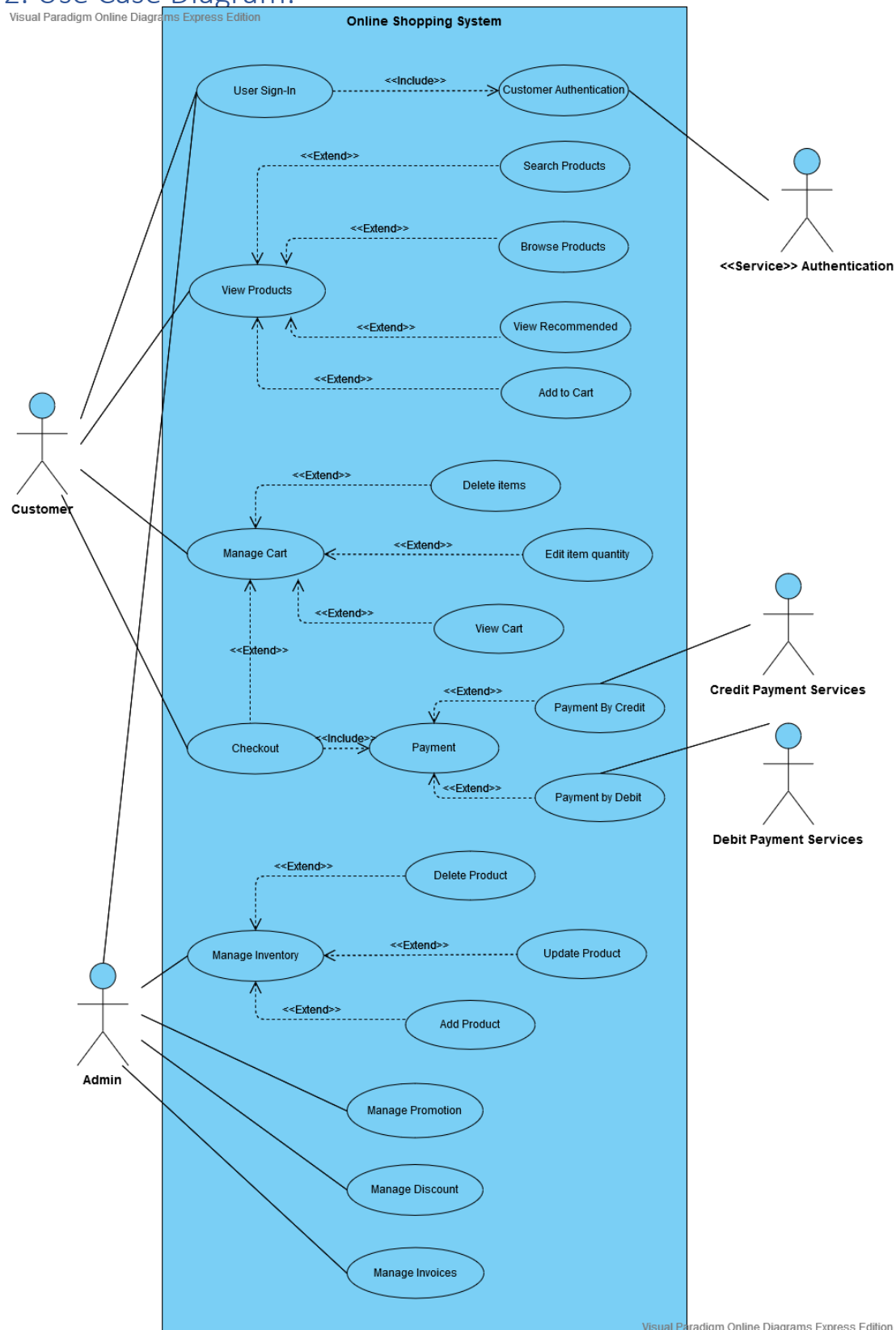
Our organization TECH Solutions has been commissioned to develop a Business-Consumer (B2C) application by GIANT Inc, an Organization based on operations based in Limerick. We have been asked to develop an online platform to sell their products and expand their operations across the country.

Business Requirements

TECH Solutions have a project from GIANT Inc- one of the top-rated retail chains in the country, to develop an E-commerce system for their organization. The proposal is to develop a scalable system that is future-oriented and with most of the features that are available in the market. The software has two main users. One is admin to manage the inventory and the other one is customers that can access the system to buy things.

2. Use Case Diagram:

Visual Paradigm Online Diagrams Express Edition



Visual Paradigm Online Diagrams Express Edition

USE CASE Descriptions

Use Case	Descriptions
Login	Previously registered users can sign-in to the website using the login page.
View Products	Customers can browse through various categories of products and clicking on product lands them on the product page.
Manage Cart	Customers with an account can add products to the shopping cart, remove items from the cart, and change the quantity of the items and can proceed to order them.
Search Products	Customers can search through the available products.
Manage Inventory	Admin can add new products to inventory, add new categories, change the product quantity, and remove products from the inventory.
Checkout	Once the items are added to the cart, customers can proceed to the payment.
Payment	Users can purchase the products using different payment methods.
Manage Invoices	Admin can manage the Invoices created.
Manage Promotions	Admin can manage promotions that are available on the products.
Manage Discounts	Admin manage the discounts offered.

3. Quality Attributes/Tactics

Quality attributes (QA) measure how well system attributes meet the objectives and satisfy the needs of the stakeholders. Quality tactics are simply the strategies to implement these attributes. Kiss solutions have been consistently delivering good quality applications for a long time now. In this application we aim to offer the following quality attributes.

- 1. Security:** Security and Integrity are high priority since application stores user private data. To overcome the security problem. We have implemented Token -based authentication for secure verification of requests from the Client-side using JWT Tokens and using Spring Security to secure user's access and their data. We have done a little research on existing e-commerce applications and their vulnerabilities which we had addressed through our design. We have also observed the most common threats to such applications and implementing measures to avoid such threats in this system.
- 2. Scalability:** We kept scalability of the system in mind during the design, so the system can be scaled independently for example number of users searching and browsing product will be much higher than users actually going from checkout to payment, so this way we can save cost than monolithic way where we have to scale whole application even if we don't need it. This approach can help us if in future our company decides to go for vertical scaling, once our user base and product will release, we will need to increase database infrastructure for login and product-catalog service, not for others. Our application can also achieves horizontal scaling very easily using Load-Balancer.
- 3. Performance:** In this project we have used the Spring Data JPA Framework, which will reduce the latency to fetch data from the database. JPA significantly improved the implementation of the data access layer by reducing the effort required. It removes the need to implement DAO, which consists of many boilerplate codes, by explicitly defining the JPA Repository extended from the Dao interface. Software built with microservices architecture has services that can be deployed and redeployed without other services. It also has better fault isolation since the failure of one service doesn't affect other services.
- 4. Testability:** The main goal of testability tactics is to allow the users to easily test the software increment. The tactics for testability are divided into two categories. The first one aims at adding controllability and observability to the system and the second one focuses on reducing the system's design complexity. In our project for performing unit testing, we followed a specific tactic from the first category known as *Executable assertions*. It can be placed at any location in the program mainly as a checkpoint to see if the specific step is pass and no faults exist. Assertions should be mainly placed where data values are modified. These can be referred to as pre and post conditions to validate a method. Observability is said to increase when assertion gets failed.

4. Patterns

A. Architectural Pattern

The Interceptor

Interceptor Architectural pattern is used in developing frameworks that can be extended transparently. It can be used to allow integration of additional services without needing to reconstruct the system architecture or impacting the existing services or applications of the framework. It can also be used to monitor and control the behavior framework platform.

The below diagram shows the structure of the Interceptor pattern :

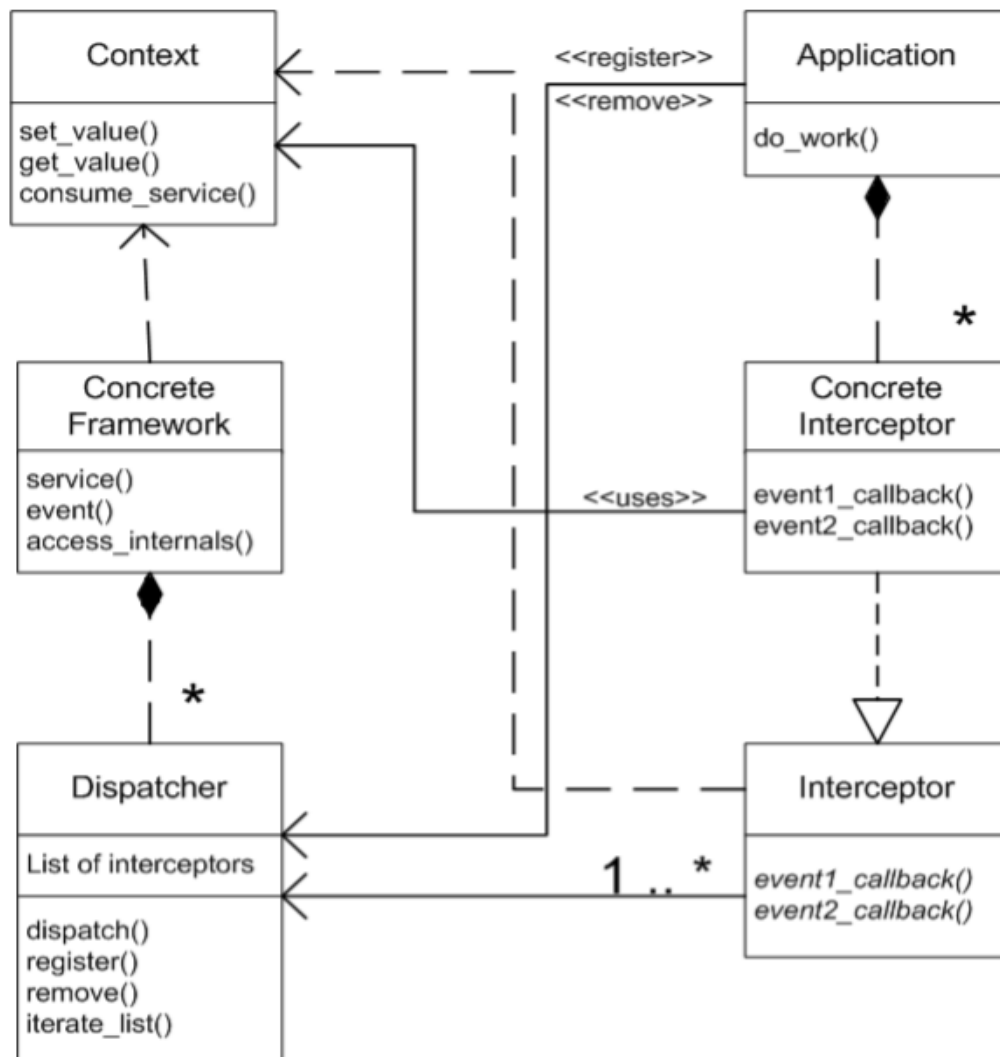


Figure 1 - The Interceptor Pattern

Implementation

In our project we have implemented an Interceptor pattern to check if the entry of payment is made to the database or not because it's a very critical task, business can't take that risk.

Concrete Framework

We have used observer design pattern as concrete framework. It will monitor state of payment status in PaymentDetailsModel class and as soon as status changes it calls dispatcher. As shown in pic below.

```
public class ConcreteFrameworkObserver {  
    public void update(PaymentDetailsModel paymentDetailsModel) {  
        Dispatcher dispatcher=new Dispatcher(paymentDetailsModel);  
        dispatcher.dispatch();  
    }  
}
```

Dispatcher

It is a service that allows us to register and remove concrete interceptors. We have made the dispatcher class which can add, remove and dispatch interceptors when its dispatch function is called from the concrete framework.

```
public class Dispatcher {  
    PaymentDetailsModel paymentDetailsModel;  
    public Dispatcher(PaymentDetailsModel paymentDetailsModel )  
    {  
        this.paymentDetailsModel=paymentDetailsModel;  
    }  
    Vector<Interceptor> interceptors_=new Vector<Interceptor>();  
  
    public void dispatch()  
    {  
        for(int i=0;i<interceptors_.size();i++)  
        {  
            Interceptor ic=(Interceptor) interceptors_.elementAt(i);  
            ic.doAction(paymentDetailsModel);  
        }  
    }  
    public void register(Interceptor i)
```

Concrete Interceptor

This class in our project calls the payment Dao class and it checks whether the payment is status is updated or not if not then it updates it in the database.

```
public class ConcreteInterceptor implements Interceptor {  
    @Override  
    public void doAction(PaymentDetailsModel paymentDetailsModel) {  
        PaymentDao paymentDao=new PaymentDao();  
        paymentDao.checkPaymentDetails(paymentDetailsModel);  
    }  
}
```

Context Objects

We have passed paymentDetailsModel as a context object as you can see in the above figure.

B. Design Patterns

Strategy Pattern

- Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it. Capture the abstraction in an interface, bury implementation details in derived classes.
- In a Strategy pattern, a class behavior or its algorithm can be changed at run time. This type of design pattern comes under behavior patterns.

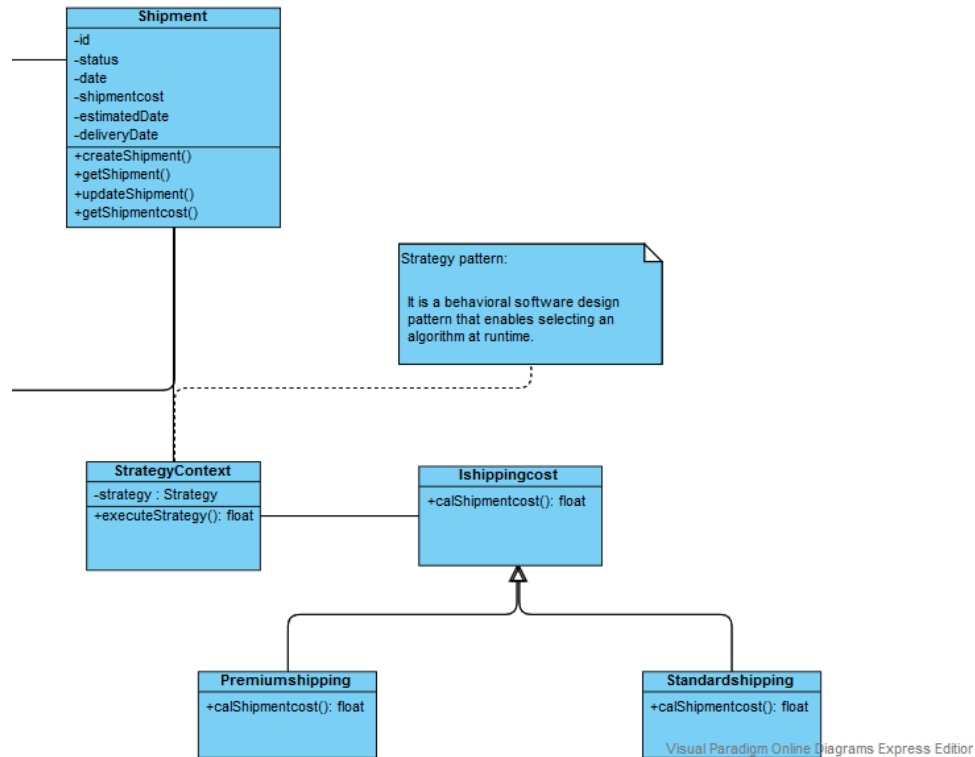


Figure 2: Class diagram for Strategy Pattern

Implementation

We have implemented a Strategy pattern in the shipment section of checkout microservice. We have multiple Shipment types customers can opt from. Depending on the shipment type selected by the user we calculate the shipping details like shipping charges and shipment time.

Bridge pattern

Bridge pattern is a structural design pattern. Bridge pattern follows the principle of decoupling the abstraction from the implementation so that the two can vary independently. The abstraction contains the reference to the implementation object so that it can refer to its methods. The client must just call the abstraction part without being concerned about the implementation part.

Implementation

We have implemented a Bridge pattern in the payment microservice. Users can select the payment as per his/her choice such as Credit Payment, Debit Payment, Net Banking, etc. and can process the payment using different payment gateways such as CCAvenue, DirectPay, etc.

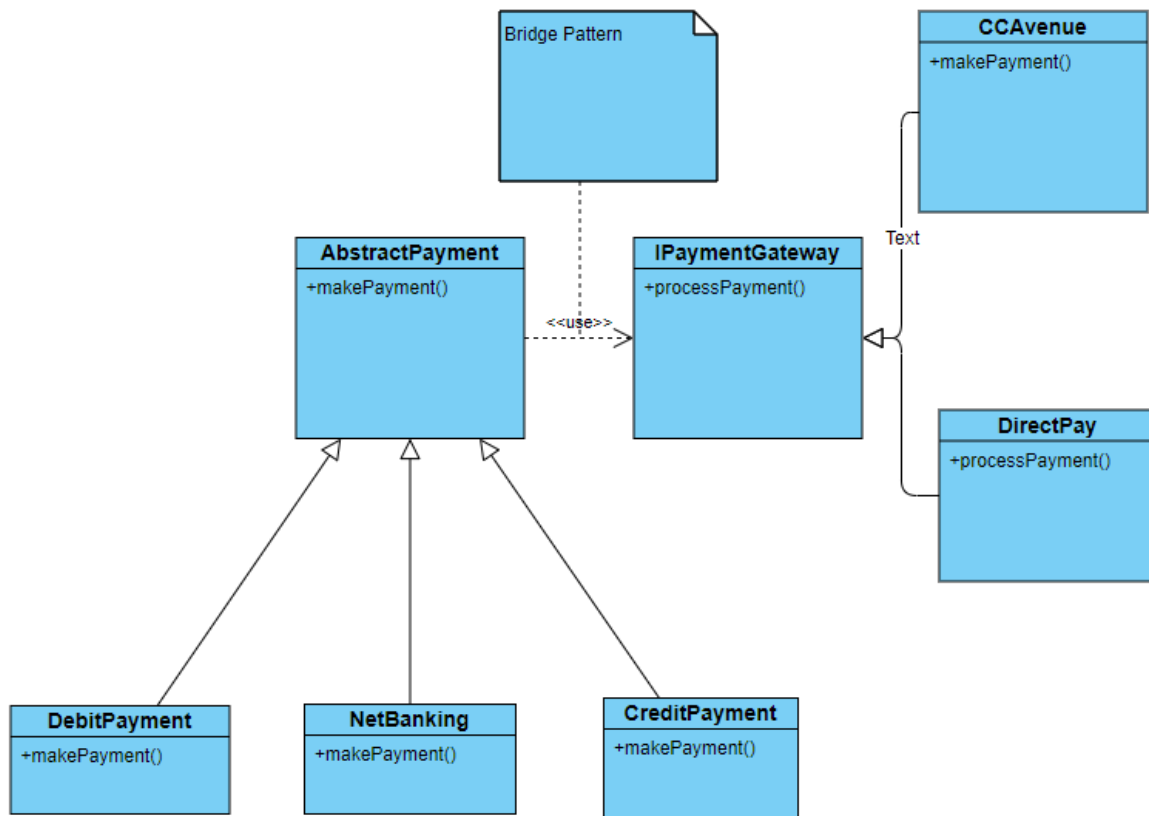


Figure 3: Class diagram for Bridge Pattern

Builder pattern

The Builder pattern is a creational design pattern. It builds a complex object from simple objects using a step-by-step approach. It is mostly used when an object can't be built in a single step like deserialization of a complex object. It provides a clear distinction between the construction and representation of the object. It also provides better control over the construction of an object.

Implementation

We have implemented Builder pattern in the login microservice.

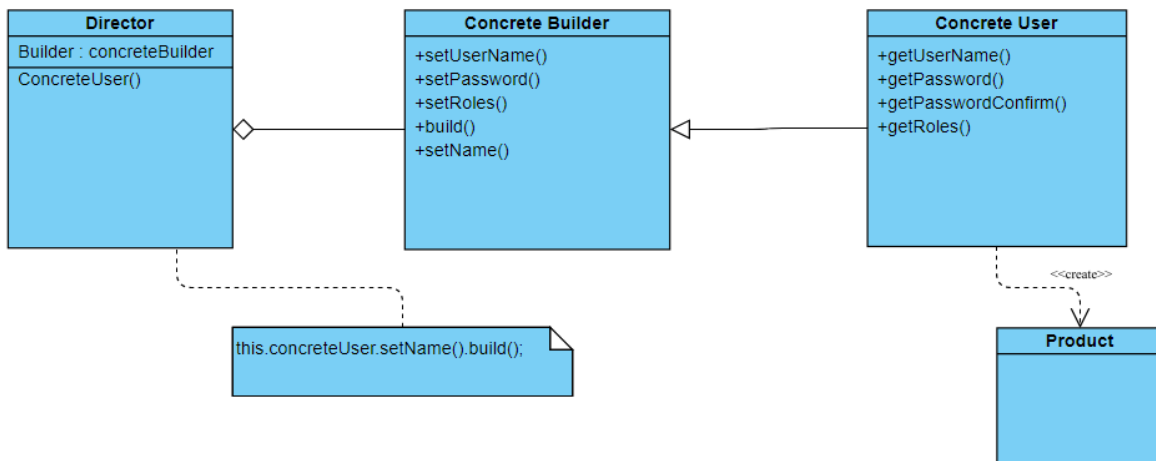


Figure 4: Class diagram for Builder Pattern

Visitor pattern

The Visitor design pattern is a type of behavioral design pattern. It is used when we must operate on a group of similar kinds of Objects. It helps in moving the operational logic from the objects to another class. In this pattern we use a visitor class that can change the executing algorithm of an element class. This helps in having different execution algorithms for different visitors.

The visitor pattern consists of two parts:

- a method called **Visit ()** which is implemented by the visitor and is called for every element in the data structure
- Visitor classes providing **Accept ()** methods that accept a visitor

Implementation

We have implemented the Visitor pattern in the Cart microservice to calculate the price of the line item. Price depends upon whether product price is based on weight or quantity. Some vegetables price is calculated according to its weight and some items like soap price is calculated by quantity. Let's have a look at the class diagram and code.

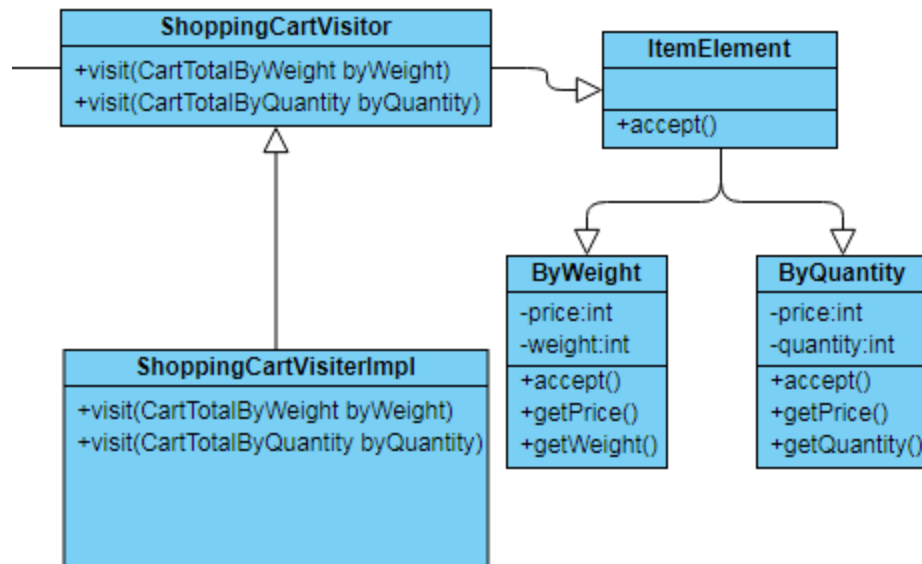


Figure 5: Class diagram for Visitor pattern

This is the snippet of code from `getCartTotal()` method in `CartServiceImpl` class. This method makes a list of items based on quantity or weight then calls `accept` method from `ItemElement` interface that gives the total amount of each line item.

```

List<ItemElement> listVisitor=new ArrayList<ItemElement>();

String cartId=shoppingCartDao.getCartId(userId);

List<LineItemModel> listLineItem=shoppingCartDao.getCartProducts(cartId);

for (LineItemModel item : listLineItem)
{
    if(item.getQuantity() !=0)
    {
        listVisitor.add(new CartTotalByQuantity(item.getPrice(),item.getQuantity()));
    }
    else
    {
        listVisitor.add(new CartTotalByWeight(item.getPrice(),item.getWeight()));
    }
}

int total = calculatePrice(listVisitor);
return total;

```

Code snippet of visitor class.


```

@Service
public class ShoppingCartVisitorImpl implements ShoppingCartVisitor{

    @Override
    public int visit(CartTotalByWeight byWeight) {
        int cost=0;
        cost=byWeight.getPrice()*byWeight.getWeight();

        return cost;
    }

    @Override
    public int visit(CartTotalByQuantity byQuantity) {
        int cost=0;
        cost=byQuantity.getPrice()*byQuantity.getQuantity();
        return cost;
    }
}

```

Code snippet CartTotalByQuantity class which implements accept method of ItemElement Interface.

```

public CartTotalByQuantity(int price, int quantity){
    this.price=price;
    this.quantity=quantity;
}

@Override
public int accept(ShoppingCartVisitor visitor) {
    return visitor.visit(this);
}

public int getPrice() {
    return price;
}

public int getQuantity() {
    return quantity;
}

```

Code snippet CartTotalByWeight class which implements accept method of ItemElement Interface.

```
public CartTotalByWeight(int price, int weight){
    this.price=price;
    this.weight=weight;
}

@Override
public int accept(ShoppingCartVisitor visitor) {
    // TODO Auto-generated method stub
    return visitor.visit(this);
}

public int getPrice() {
    return price;
}

public int getWeight() {
    return weight;
}
```

Command pattern

It's one of the data-driven design patterns. In this pattern a request is wrapped under an object as a command and passed to an invoker object. The Invoker object looks for the appropriate object which can handle this command and passes the command to the corresponding object which executes the command.

Implementation

We have implemented command for updating of inventory. Using command, we can update inventory for multiple products at once.

Below is the screenshot of Broker class wherein we have taken order method which adds new order to the order list, this way we can add multiple requests at a time which we are iterating in list format.

```

public class InventoryBroker {
    private List<IInventory> orderList = new ArrayList<IInventory>();

    public void takeOrder(IInventory order){
        orderList.add(order);
    }

    public void placeOrders(){
        for (IInventory order : orderList) {
            order.execute();
        }
        orderList.clear();
    }
}

```

Once all the orders are taken, placeOrder method is called which iterate over the orderList and executes the order based on whether we have to increase inventory or decrease.

Below are the screenshots of adding inventory and delete inventory code which implements execute method exposed by the IInventory.

```

public class AddInventory implements IInventory{
    private Inventory inventory;
    public AddInventory(Inventory inventory)
    {
        this.inventory=inventory;
    }
    public void execute(){
        //inventory.
        inventory.buy();
    }
}

public class DeleteInventory implements IInventory{
    private Inventory inventory;
    public DeleteInventory(Inventory inventory)
    {
        this.inventory=inventory;
    }
    public void execute(){
        inventory.sell();
    }
}

```

Pluggable Adapter

According to GOF, “this design pattern is used to translate the interface of one class into another class. This means that we can make classes work together that couldn’t otherwise work because of incompatible

interface". Pluggable adapter is better than normal adapter in a way that we don't need multiple adapters for multiple interfaces, one of the ways to achieve this is using delegate.

Implementation

We have implemented a Pluggable adapter design pattern to process payment from multiple payment gateways, in our case we have used CCAvenue and DirectPay. CCAvenue uses a different interface so we need an adapter in the middle to use the IPayment gateway interface for both types of payment gateways.

This piece of code is from a Pluggable adapter, it sets an object based on the interface type. If in future we need to integrate new payment gateways we will just add a new piece of code in a pluggable adapter and we are good to go, we don't need a new adapter, which is the main property of pluggable adapter.

```
import com.ecommnet.cc.payment.service.delegate.IPaymentGateway;
```

```
public class PluggableAdapter implements IPaymentGateway{
    IPaymentGateway directPay;
    ICCAvenue ccavenue;

    public PluggableAdapter(ICCAvenue ccavenue)
    {
        this.ccavenue=ccavenue;
    }
    public PluggableAdapter(IPaymentGateway directPay)
    {
        this.directPay=directPay;
    }
    public String processPayment(String paymentSystem) {
        String status;

        ...
    }
}
```

Below is the code for CCAvenue and DirectPay payment gateways.

```
public class CCAvenue implements ICCAvenue {

    @Override
    public String makePayment(String paymentSystem) {
```

```
public class DirectPay implements IPaymentGateway {

    @Override
    public String processPayment(String paymentSystem) {
        // TODO Auto-generated method stub
    }
}
```

Both payment gateway uses different interfaces, but the pluggable adapter takes care of it.

C. Independently Researched Patterns

1. Microservices Architecture

It is an architectural style where we divide the application into various services that can be developed and deployed independently without the other services. In Microservice Architecture, each service is self-contained and implements a single business capability.

Advantages

- Loosely coupled
- Independently deployed
- Easily maintainable and testable
- Fault isolation

We have developed the e-commerce application using this Microservice architecture. We have divided the application into the five services. They are:

Login Service:

This service validates the login details of registered customers and admin. It is also responsible for managing sessions of Users.

Product Service:

This service maintains the catalog of products and allows the admin to manage the inventory by adding or deleting products from stock.

Cart Service:

This service provides users with the functionality of managing carts like adding a product to cart, deleting a product from the cart, and changing the quantity of the product added.

Check-out Service:

This service manages the order creation and creation of shipment request and call the payment service.

Payment Service:

This service is responsible for the payment of the order amount. We've implemented the Bridge design pattern to manage various payment options.

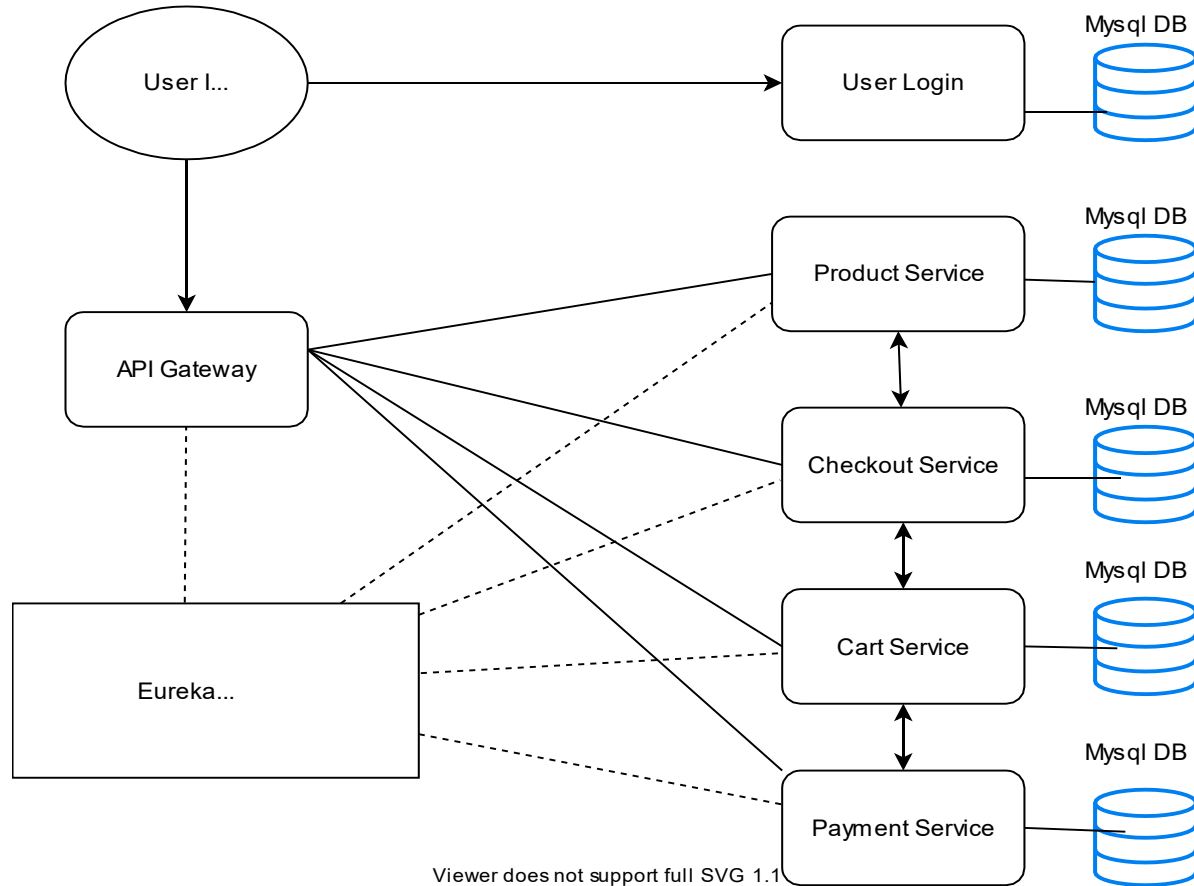


Figure 6: Architecture Diagram of E-commerce system

2. Layered Architecture

The N-tier or Multitier architectural pattern is a client-server architecture. The presentation, application processing, and data management functions are separated.

Implementation

E-commerce application implements a three-tier architectural pattern where layers are divided into

- Presentation layer
- Business layer

- DAO layer.

The Presentation layer provides basic user interfaces and access services to the application. The presentation layer is the topmost layer in the N-tier architecture pattern and consists of files that directly interact with the user. The Business layer contains classes that handle the application logic or business logic. Next, we have a DAO layer that is the Data access layer all database access from the business layer goes through the DAO layer class which is responsible for data fetching from database. By doing so application provides the database access without exposing details of the database.

3.Enterprise Pattern: Session Management

Martin Fowler has described the session management (Fowler, 2003), We have implemented this pattern with the use of JSON Web Token to add token in the header of every request in our microservices.

How it works?

In our Micro Services, we consider storing all session data on the client-side. The client sends the data in every request, which goes through our API server, gets validates, and then revert to the client as a response.

Data can be store on the client-side. For example, URL params, Cookies, session data to the client with response.

The Client will send the JWT token in the Authorization for all requests to access any protected resources.

Advantages

- **Stateless:** It is a stateless protocol to support the server.
- **Supportability:** It supports most of the architecture including Microservices.
- It allows access faster as it stores client data as a session in the form of cookies.

Disadvantages

- **Session Storage:** This requires attention, if the sessions are long then its transfer time and per session request handling requires concern. While most people tend to reject cookies when asking for permission to store the session.
- If the client is shut down, the session will be lost. However, this is acceptable as a failure.
- Data with the client request is vulnerable. It can be viewed as modifying by hackers. One way to avoid this situation is to enable end-to-end encryption then the server needs to store decrypted session data and its validation. The result will be impacting the performance of the server due to Encryption and Decryption.

Implementation

For the Session Management we will implement JWT Token based Authorization for each user request. JWT is a compact and self-contained method to transfer data between different services

as an object in the form of Java Web Tokens with security. These tokens are verified and trusted because it is encrypted by a digital signature. And these signatures can be created by a secret key (HMAC) or with RSA (public or private key).

JWT tokens are consist of three things: Header, Payload, and Signature and be understood as:

Header:

It contains the token type and the signing algorithm which uses (RSA or HMAC) and is encrypted by Base64.

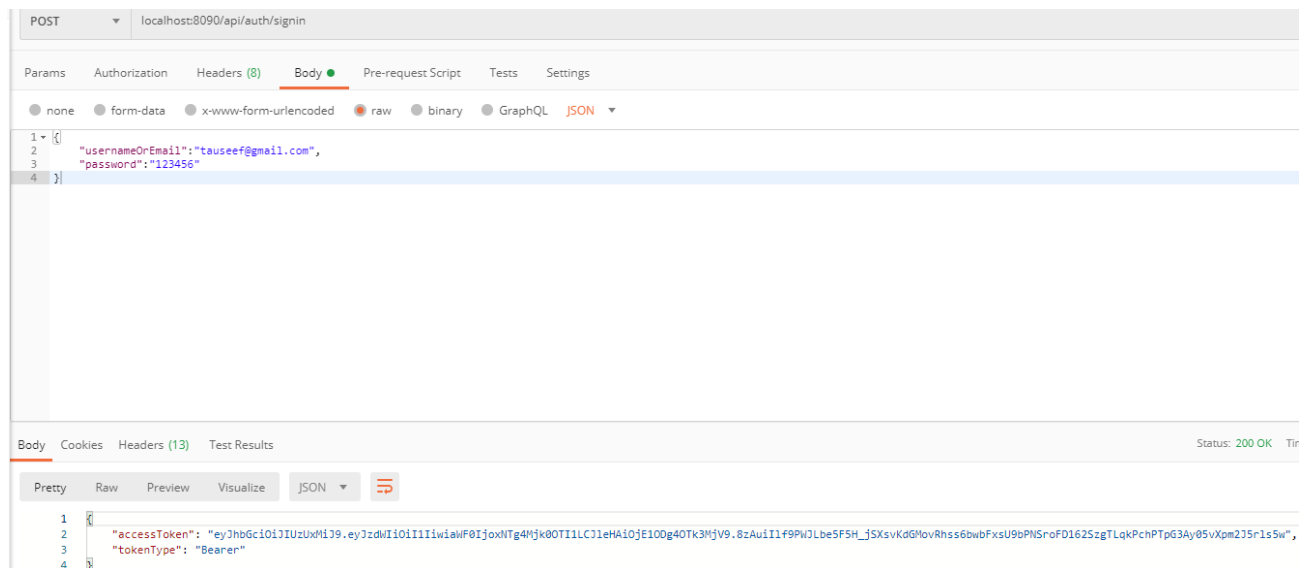
Payload:

It contains the claims – an entity and additional data like session time and expire time all together encrypted by Base64.

Signature:

We need to make sure the header is encoded; it includes a secret key to handle the payload and to sign the payload request.

Figure 7: JWT structure with header, payload and signature



Code

- **Token Generation:** Here we generate a JWT, set the claims, and signing with a secret key using the HS256 algorithm.


```

Date now = new Date();
Date expiryDate = new Date(now.getTime() + jwtExpirationInMs);

return Jwts.builder()
    .setSubject(Long.toString(userPrincipal.getId()))
    .setIssuedAt(new Date())
    .setExpiration(expiryDate)
    .signWith(SignatureAlgorithm.HS256, jwtSecret)
    .compact();
}

public Long getUserIdFromJWT(String token) {
    Claims claims = Jwts.parser()
        .setSigningKey(jwtSecret)
        .parseClaimsJws(token)
        .getBody();
}

```

- **Verification of Token:** We parse the token and then check its expiration and signature.

```

public boolean validateToken(String authToken) {
    try {
        Jwts.parser().setSigningKey(jwtSecret).parseClaimsJws(authToken);
        return true;
    } catch (SignatureException ex) {
        logger.error("Invalid JWT signature");
    } catch (MalformedJwtException ex) {
        logger.error("Invalid JWT token");
    } catch (ExpiredJwtException ex) {
        logger.error("Expired JWT token");
    } catch (UnsupportedJwtException ex) {
        logger.error("Unsupported JWT token");
    } catch (IllegalArgumentException ex) {
        logger.error("JWT claims string is empty.");
    }
    return false;
}
}

```

5. Architectural view of the System

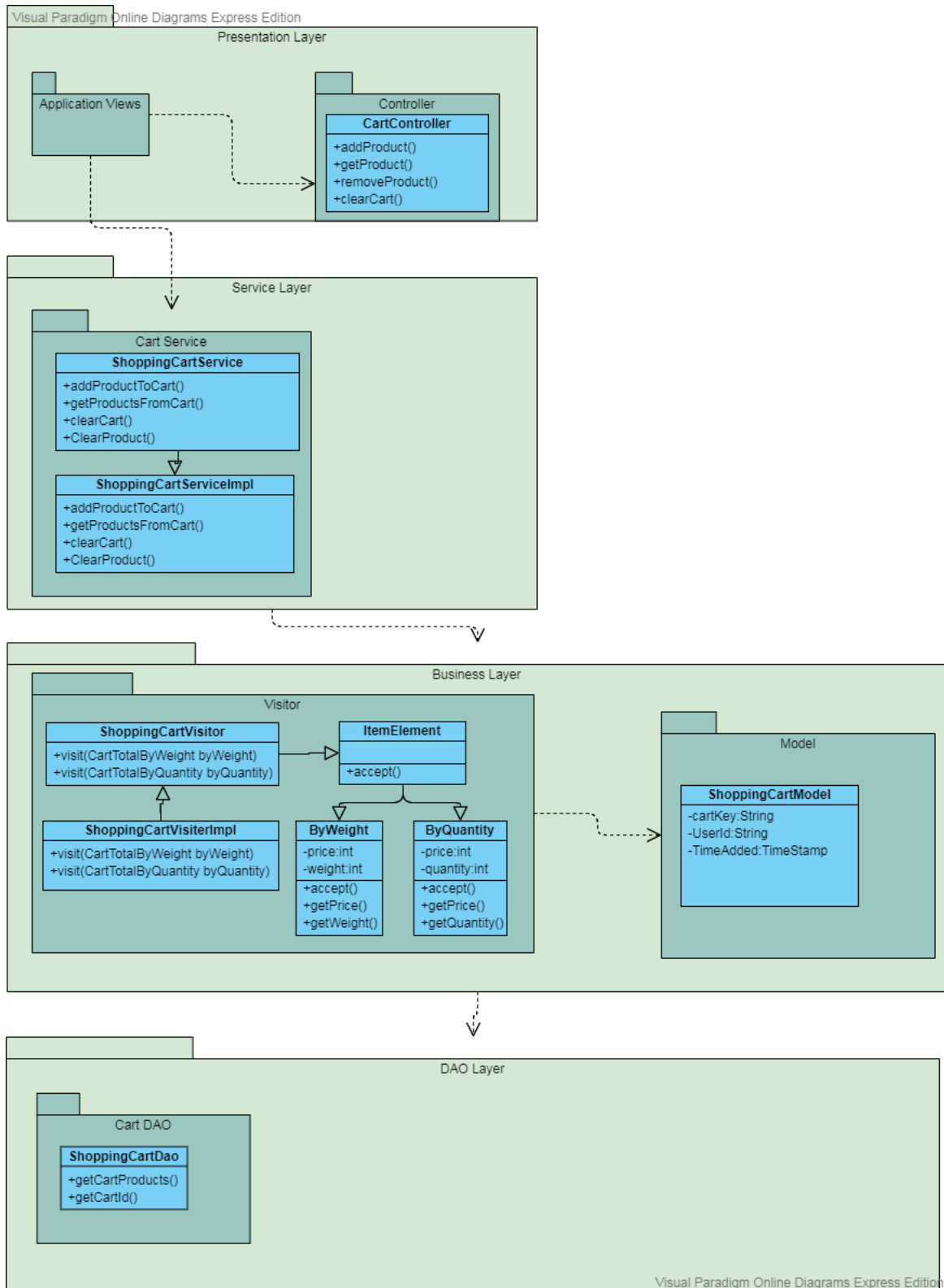


Figure 8 : Architectural diagram for Cart microservice

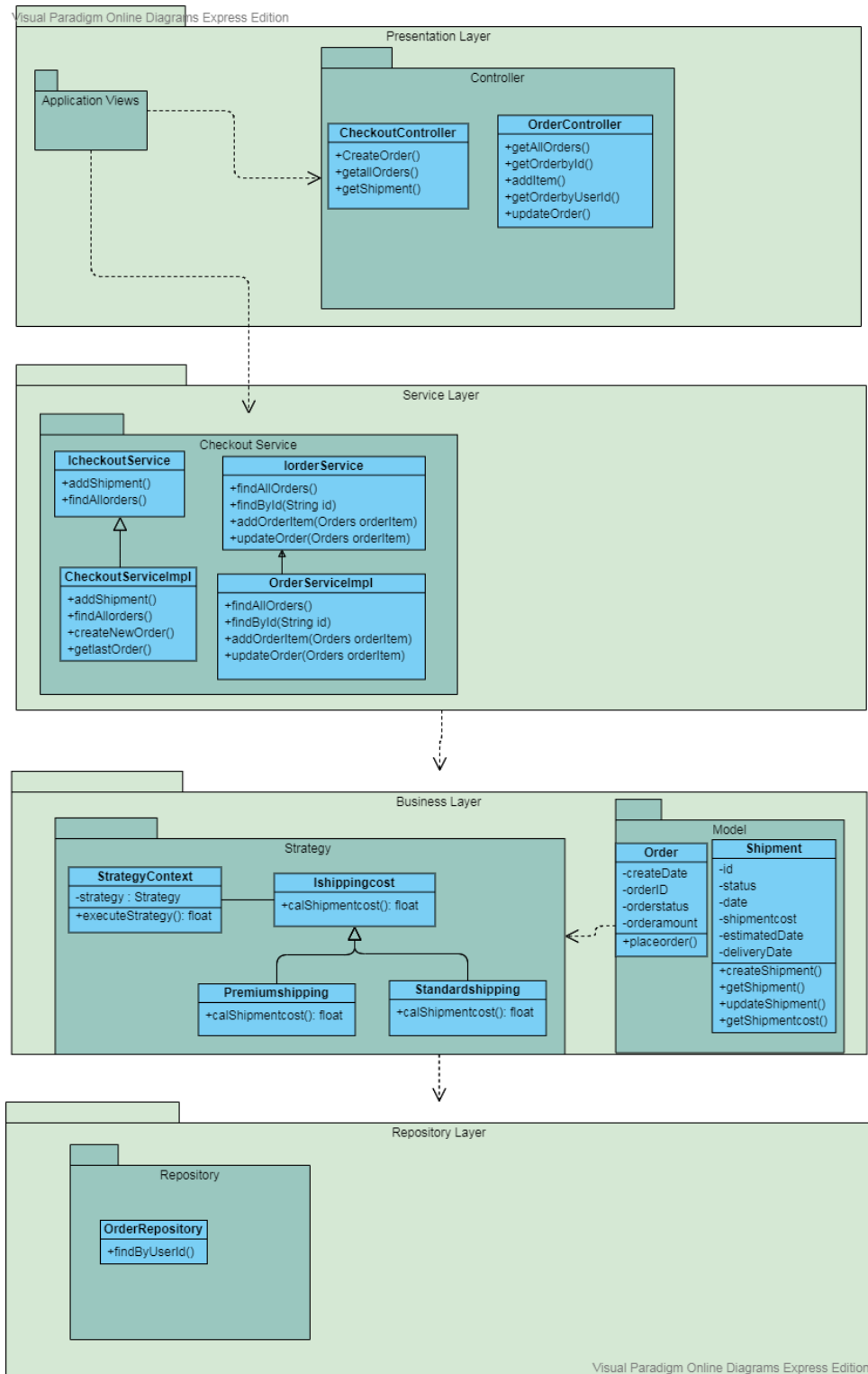


Figure 9: Architectural Diagram for Checkout Microservice

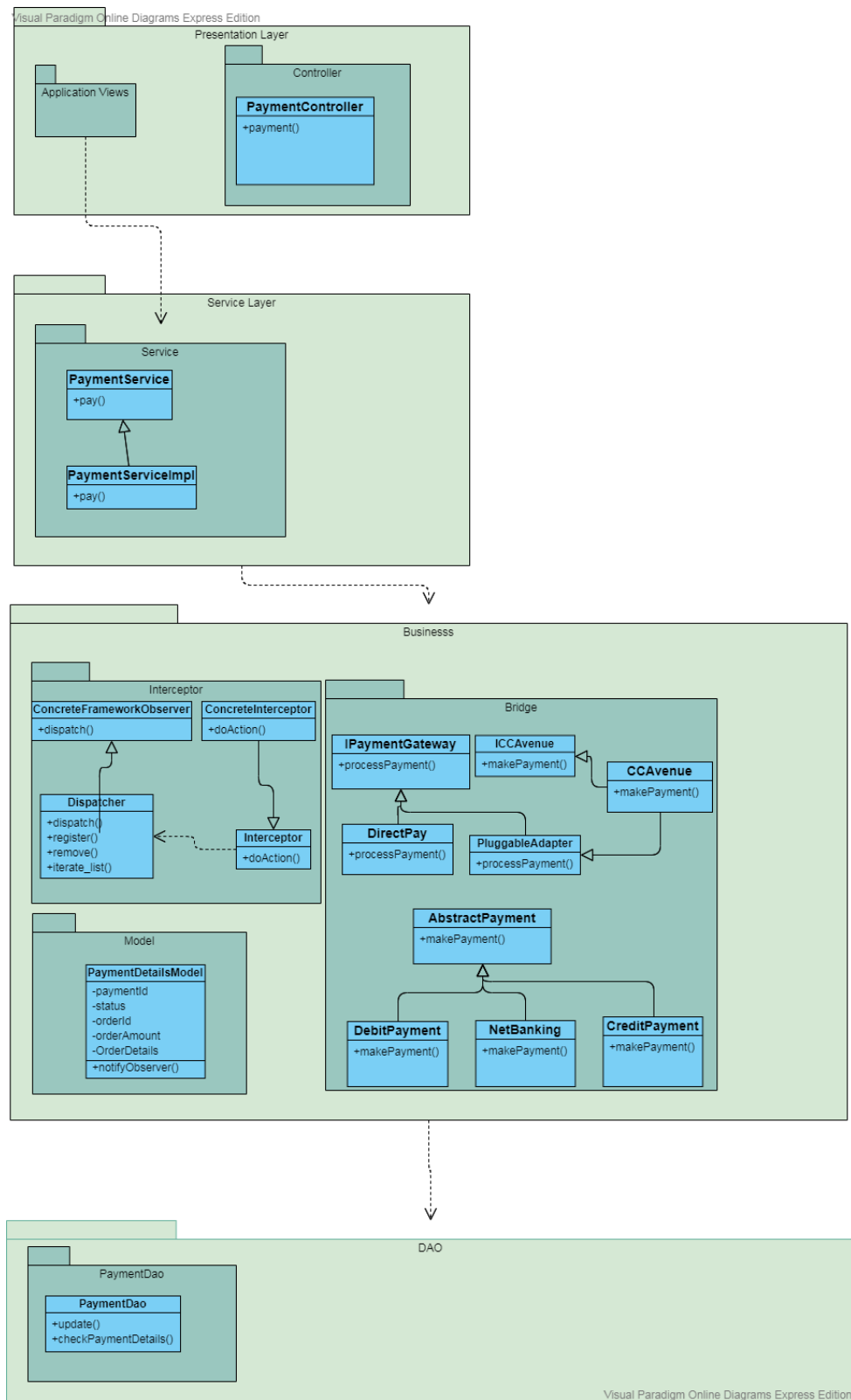


Figure 10: Architectural Diagram for Payment Microservice

6. Structural and Behavioral diagrams

Class Diagram

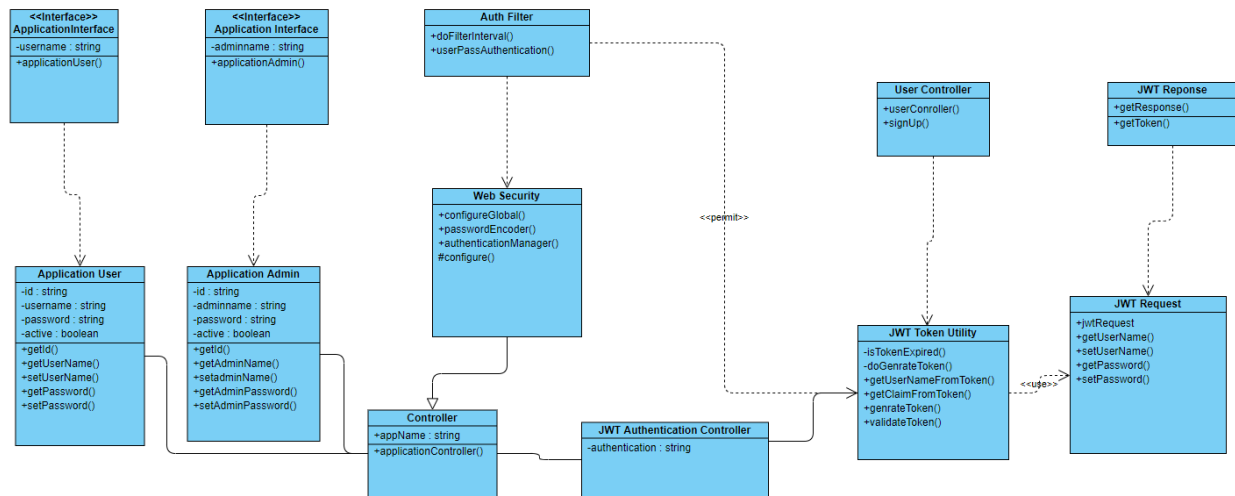


Figure 11- Class Diagram for Login microservice

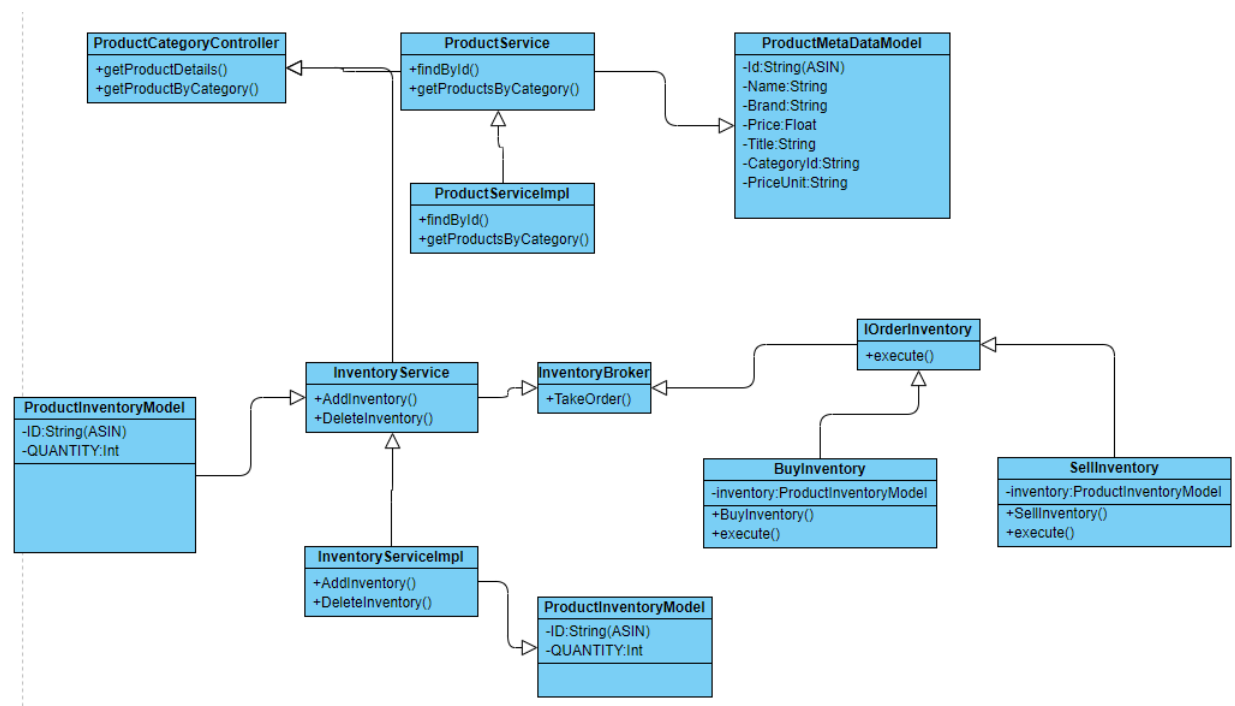


Figure 12 - Class Diagram for Product microservice

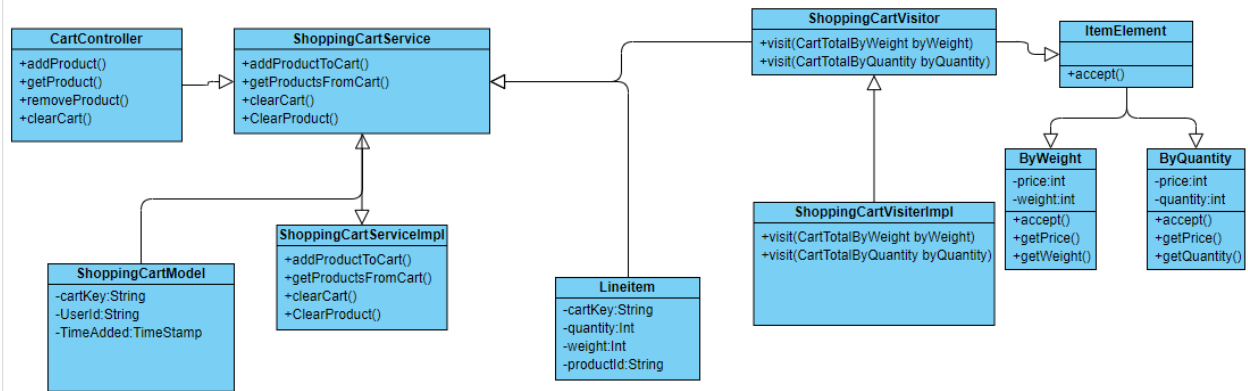


Figure 13 - Class Diagram for Shopping-Cart microservice

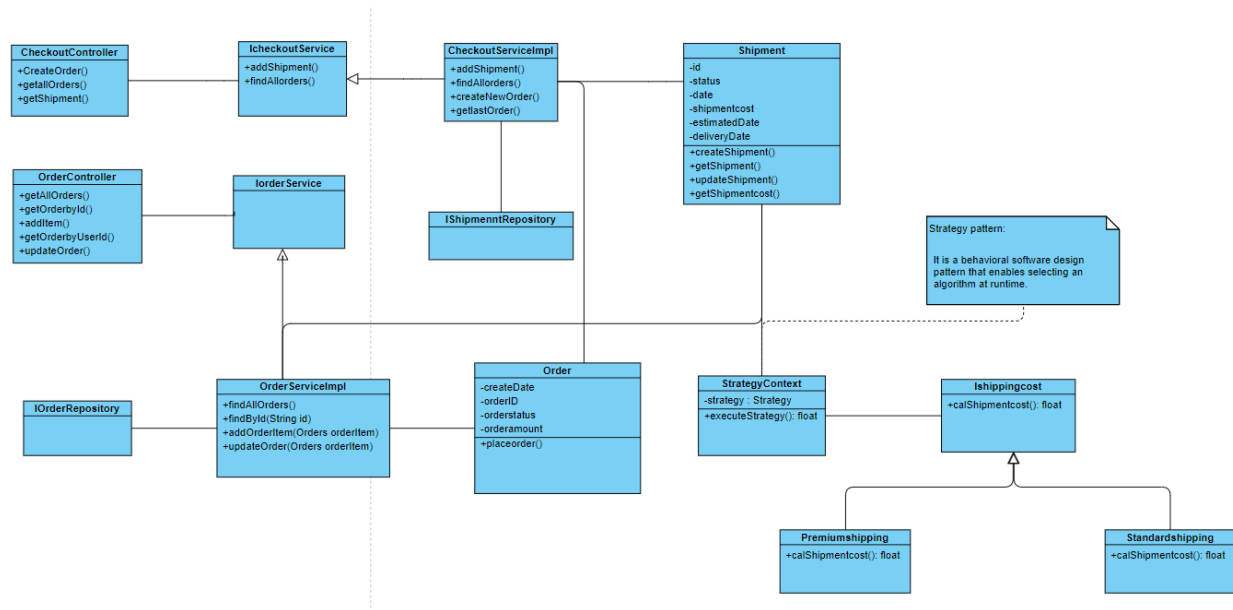


Figure 14: Class Diagram for Checkout microservice

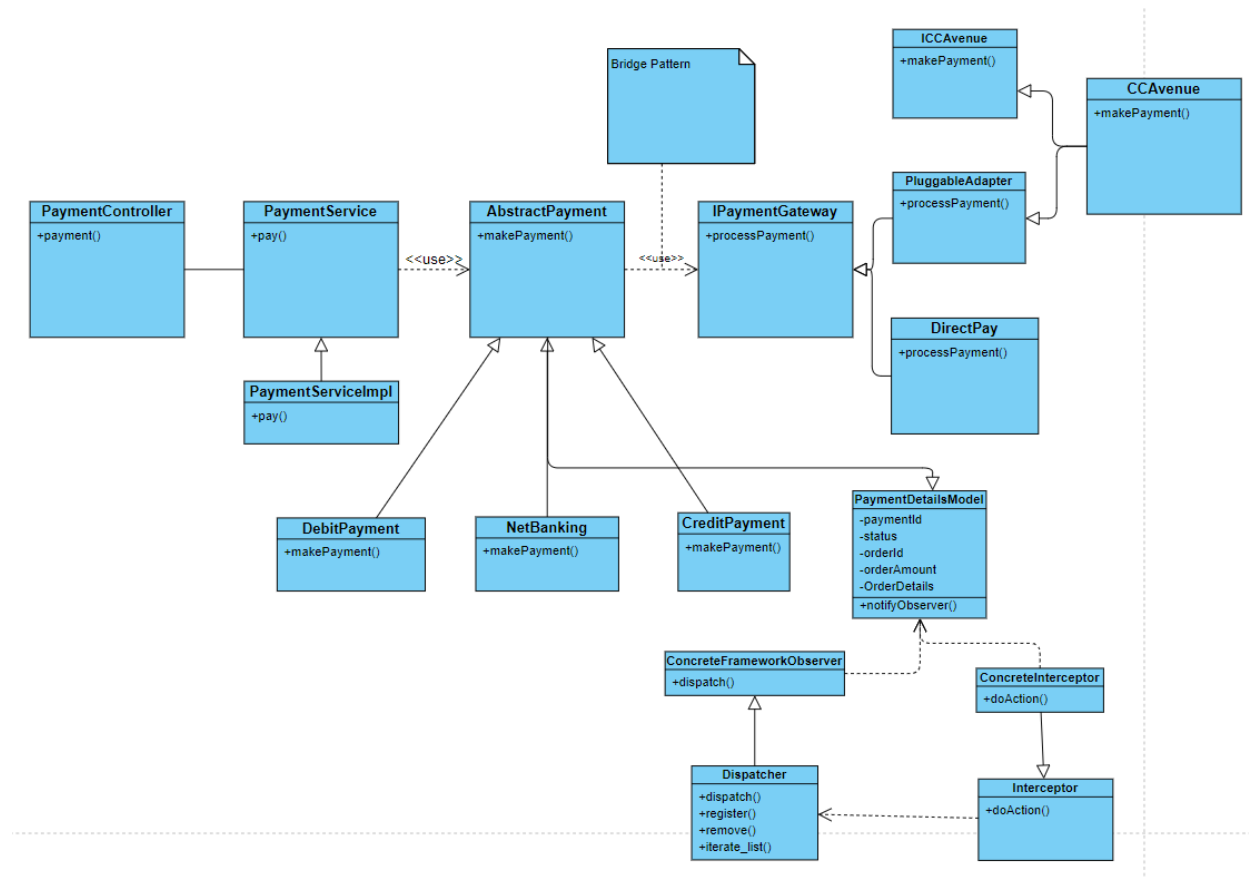


Figure 15 - Class Diagram for Payment microservice

Sequence Diagram

Sequence Diagram: Shopping Cart

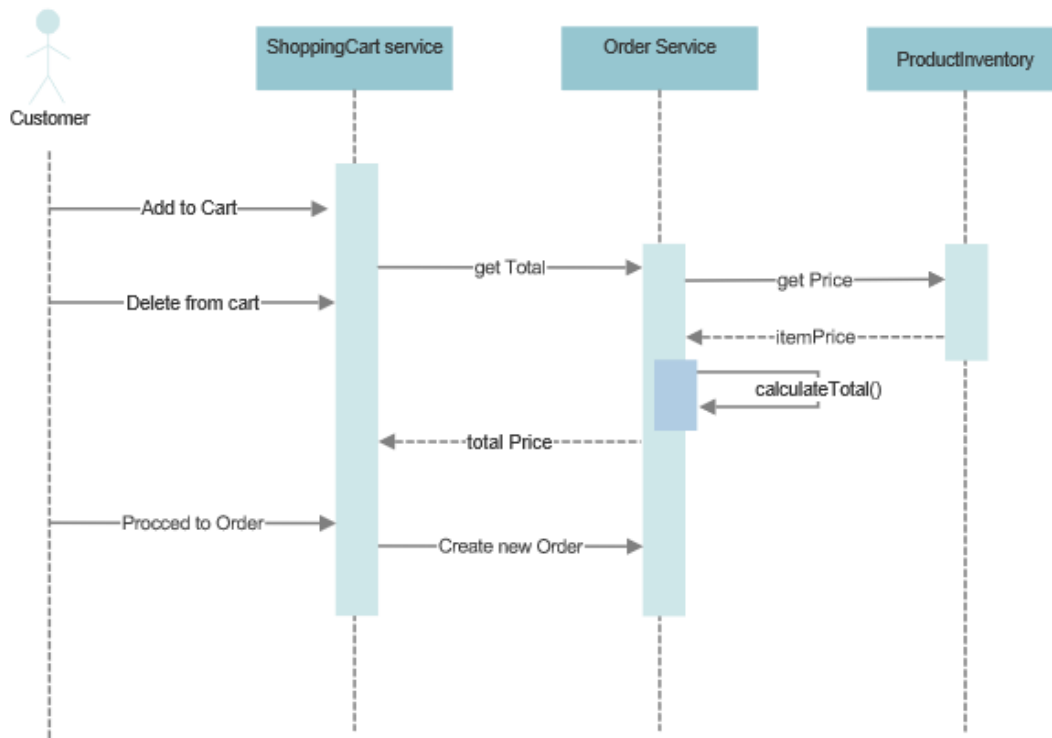


Figure 16: Sequence diagram for Shopping Cart

7. Coding Fragments

Strategy Pattern

Participant classes

- Istrategy
- StrategyContext
- PremiumShipping
- StandardShipping

Code

Strategy Interface :

```
public interface Strategy {  
  
    public double setShippingcharges(double carttotal);  
  
}
```

StrategyContext :

```
package com.ecommerce.Checkoutmicroservice.StrategyPattern;  
  
public class StrategyContext {  
  
    private Strategy strategy;  
  
    public StrategyContext(Strategy strategy){  
  
        this.strategy = strategy;  
    }  
  
    public double executeStrategy(double carttoal){  
        return strategy.setShippingcharges(carttoal);  
    }  
}
```

PremiumShipping :

```
public class PremiumShipping implements Strategy {  
  
    @Override  
    public double setShippingcharges(double carttotal) {  
        return carttotal * 0.1 ;  
    }  
}
```

StandardShipping :

```
public class StandardShipping implements Strategy {  
  
    @Override  
    public double setShippingcharges(double carttotal) {  
        return carttotal * 0.05 ;  
    }  
}
```

Bridge Pattern

Participant classes

- IPaymentGateway
- CCAvenue
- DirectPay
- AbstractPayment
- CreditPayment
- DebitPayment
- NetBanking

Code

```
package com.ecommerce.paymentservice.bridgepattern;  
  
public interface IPaymentGateway {  
    public String processPayment(String paymentSystem);  
}  
  
package com.ecommerce.paymentservice.bridgepattern;  
  
public class CCAvenue implements IPaymentGateway {  
  
    @Override  
    public String processPayment(String paymentSystem) {  
        System.out.println("Using CCAvenue gateway for: " + paymentSystem);  
        String status = "SUCCESS";  
        return status;  
    }  
}
```

```

package com.ecommerce.paymentservice.bridgepattern;

public class DirectPay implements IPaymentGateway {

    @Override
    public String processPayment(String paymentSystem) {

        System.out.println("Using DirectPay gateway for: " + paymentSystem);

        String status = "Success";

        return status;
    }
}

package com.ecommerce.paymentservice.bridgepattern;

import com.ecommerce.paymentservice.model.OrderModel;

public abstract class AbstractPayment {

    // Adding a reference to the payment gateway interface
    public IPaymentGateway payment_gateway;

    public abstract PaymentDetailsModel makePayment(OrderModel order);
}

package com.ecommerce.paymentservice.bridgepattern;

import com.ecommerce.paymentservice.model.OrderModel;

public class CreditPayment extends AbstractPayment {

    @Override
    public PaymentDetailsModel makePayment(OrderModel order) {

        PaymentDetailsModel paymentdetails = new PaymentDetailsModel();

        String processstatus = payment_gateway.processPayment("Credit Card");

        // Checking the payment status
        paymentdetails.setStatus(PaymentDetailsModel.SUCCESS);
        if (processstatus.contentEquals(paymentdetails.getStatus())) {
            System.out.println("Payment successful for order id: " + order.getId());
        }
        paymentdetails.setPayment_id(12345);
        return paymentdetails;
    }
}

package com.ecommerce.paymentservice.bridgepattern;

import com.ecommerce.paymentservice.model.OrderModel;

public class DebitPayment extends AbstractPayment {

    @Override
    public PaymentDetailsModel makePayment(OrderModel order) {

        PaymentDetailsModel paymentdetails = new PaymentDetailsModel();

        String processstatus = payment_gateway.processPayment("Debit Card");

        // Checking the payment status
        paymentdetails.setStatus(PaymentDetailsModel.SUCCESS);
        if (processstatus.contentEquals(paymentdetails.getStatus())) {
            System.out.println("Payment successful for order id: " + order.getId());
        }
        paymentdetails.setPayment_id(12345);
        return paymentdetails;
    }
}

```

```

package com.ecommerce.paymentservice.bridgepattern;

import com.ecommerce.paymentservice.model.OrderModel;

public class NetBanking extends AbstractPayment {

    @Override
    public PaymentDetailsModel makePayment(OrderModel order) {

        PaymentDetailsModel paymentdetails = new PaymentDetailsModel();

        String processstatus = payment_gateway.processPayment("Net Banking");

        // Checking the payment status
        paymentdetails.setStatus(PaymentDetailsModel.SUCCESS);
        if (processstatus.contentEquals(paymentdetails.getStatus())) {
            System.out.println("Payment successful for order id: " + order.getId());
        }
        paymentdetails.setPayment_id(12345);
        return paymentdetails;
    }
}

```

Builder

Participant Classes

- Concrete Builder
- Concrete User
- User Builder

Code

```

public ConcreteUser(String name, String username, String email, String password) {
    this.name = name;
    this.username = username;
    this.email = email;
    this.password = password;
}

public Long getId() { return id; }

public void setId(Long id) { this.id = id; }

public String getUsername() { return username; }

public void setUsername(String username) { this.username = username; }

public String getName() { return name; }

public void setName(String name) { this.name = name; }

public String getEmail() { return email; }

```

```

25 public class ConcreteUser extends DateAudit {
26     @ {...}
28     private Long id;
29
30     @ {...}
32     private String name;
33
34     @ {...}
36     private String username;
37
38     @ {...}
42     private String email;
43
44     @ {...}
46     private String password;
47
48     @ {...}
52     private Set<Role> roles = new HashSet<>();
53
54     public ConcreteUser(String name, String username, String email, String password) {
55         this.name = name;
56         this.username = username;

```

```

6 public class ConcreteBuilder {
7     Long id;
8     String name;
9     String username;
10    String email;
11    String password;
12
13    public ConcreteBuilder() {}
16
17    public ConcreteBuilder setName(String name) {...}
21
22    public ConcreteBuilder setUsername(String username) {...}
26
27    public ConcreteBuilder setPassword(String password) {...}
31
32    public ConcreteBuilder setEmail(String email) {
33        this.email = email;
34        return this;
35    }
36
37    public ConcreteUser build() { return new ConcreteUser(name, username, email, password); }
40
41 }

```

```

    }

    // Builder Patter Usage
    // Creating user's account
    ConcreteBuilder concreteBuilder = new ConcreteBuilder();
    ConcreteUser concreteUser = new ConcreteUser(signUpRequest.getName(), signUpRequest.getUsername(),
        signUpRequest.getEmail(), signUpRequest.getPassword());

    concreteBuilder.setPassword(passwordEncoder.encode(concreteUser.getPassword())).build();

    Role userRole = roleRepository.findByName(RoleName.ROLE_USER)
        .orElseThrow(() -> new ApplicationException("User Role not set."));

    concreteUser.setRoles(Collections.singleton(userRole));

    ConcreteUser result = userRepository.save(concreteUser);

    URI location = ServletUriComponentsBuilder
        .fromCurrentContextPath().path("/users/{username}")
        .buildAndExpand(result.getUsername()).toUri();

    return ResponseEntity.created(location).body(new ApiResponse( success: true, message: "User registered s
    }

```

5. Added Value

We have used SonarQube to identify bad code smells in our project. It is a static code analysis tool wherein the source code is examined. It's an open-source tool developed by SonarSource for continuous inspection of code quality.

1) Here is the first check that was performed on the ProductCatalogService Microservice code. It shows that we have 3 critical, 6 Major and 23 minor bad code smells.

The screenshot shows the SonarQube interface for the 'Product-Catalog-Service' project. The left sidebar has a 'Filters' section with 'Display Mode' set to 'Issues'. Under 'Type', 'Code Smell' is selected with 37 items. Under 'Severity', 'Critical' (3) and 'Major' (6) are selected. The main area shows a list of issues. The first issue is 'Rename this package name to match the regular expression' for 'src/.../ProductCatalogService/ProductCatalogServiceApplication.java', marked as 'Code Smell' (Minor), 'Open', 'Not assigned', and '10min effort'. The second issue is 'Remove this unused import' for 'src/.../ProductCatalogService/Service/ProductInventoryService.java', marked as 'Code Smell' (Minor), 'Open', 'Not assigned', and '2min effort'. The third issue is another 'Rename this package name' for 'src/.../ProductCatalogService/Service/ProductInventoryService.java', marked as 'Code Smell' (Minor), 'Open', 'Not assigned', and '10min effort'. The fourth issue is another 'Rename this package name' for 'src/.../ProductCatalogService/Service/ProductService.java', marked as 'Code Smell' (Minor), 'Open', 'Not assigned', and '10min effort'. The fifth issue is another 'Rename this package name' for 'src/.../Service/impl/ProductInventoryServiceImpl.java', marked as 'Code Smell' (Minor), 'Open', 'Not assigned', and '10min effort'.

Here is the first example of bad quality code where I was making the new object of productMetaDataModel but there is a way that I can directly return productService.findProductById(asin).

The screenshot shows a code snippet with a 'Code Smell' issue. The code is `ProductMetaDataModel productMetadata = productService.findProductById(asin);`. The issue message is 'Immediately return this expression instead of assigning it to the temporary variable "productMetadata".' The issue is marked as 'Code Smell' (Minor), 'Open', 'Not assigned', and '2min effort'. The issue is also marked as 'clumsy'.

Here is the piece of code after we made the required changes.

```
public ProductMetaDataModel getProductDetails(@PathVariable String asin) {  
    return productService.findProductById(asin);  
}
```

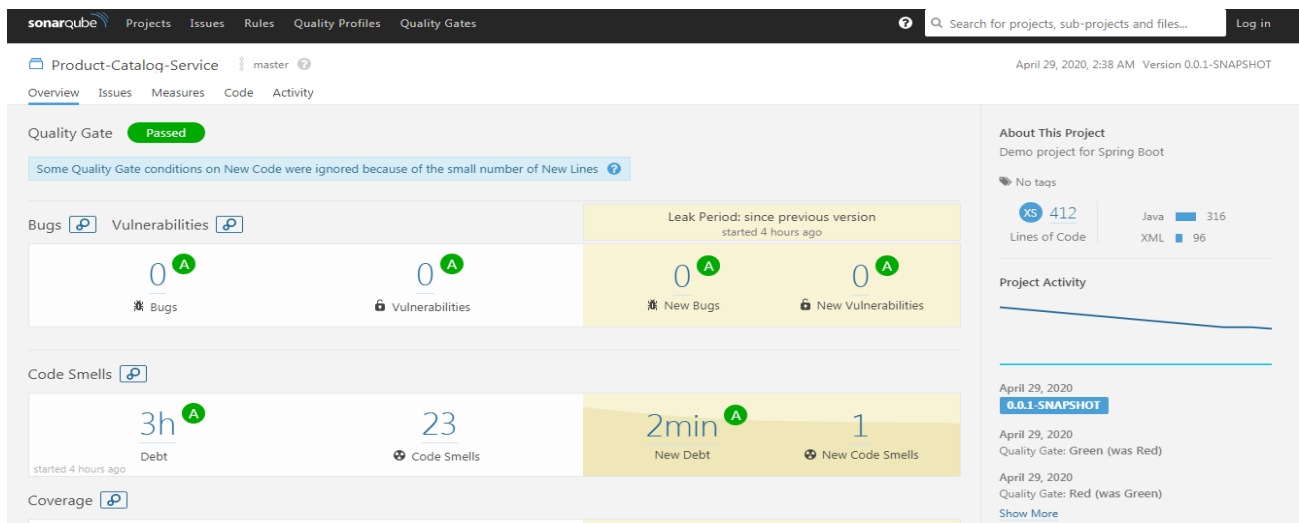
Here is the example of major severity bad code where we just left the line of code that we are not using.



Here is the piece of code where we removed that commented code.

```
@RequestMapping(method = RequestMethod.POST, value = "/product/{productMetaDataModel}", produces = "application/json")
public void addProductDetails(@PathVariable ProductMetaDataModel productMetaDataModel) {
    productService.addNewProduct(productMetaDataModel);
}
```

We did this for all major and critical issues and finally when we checked our code got passed.



2) The second check was performed on the PaymentService Microservice code. It had 1 Bug, 1 Vulnerability, 69 Code Smells, and 1 Security Hotspots initially when the project was analyzed.

[Projects](#)
[Issues](#)
[Rules](#)
[Quality Profiles](#)
[Quality Gates](#)

[Log in](#)

Continuous Code Quality

[Log in](#)
[Read documentation](#)

1

Projects Analyzed

1

Bugs

1

Vulnerabilities

69

Code Smells

1

Security Hotspots

[Projects](#)
[Issues](#)
[Rules](#)
[Quality Profiles](#)
[Quality Gates](#)

[Log in](#)

Filters

Quality Gate

Passed

1

Warning

0

Failed

0

Reliability (Bugs)

A

0

B

0

C

1

D

0

E

0

Security (Vulnerabilities)

A

0

B

1

C

0

D

0

E

0

Maintainability (Code Smells)

A

1

B

0

C

0

D

0

Perspective: Overall Status

Sort by: Last analysis date

1 projects

payment-service

Passed

1

C

Bugs

1

B

Vulnerabilities

69

A

Code Smells

0.0%

Coverage

0.0%

Duplications

Last analysis: April 28, 2020, 8:06 PM

446

XS

Java, XML

1 of 1 shown

Embedded database should be used for evaluation purposes only

The embedded database will not scale, it will not support upgrading to newer versions of SonarQube, and there is no support for migrating your data out of it into a different database engine.

SonarQube™ technology is powered by SonarSource SA

Community Edition - Version 7.8 (build 26217) - LGPL v3 - Community - Documentation - Get Support - Plugins - Web API - About

sonarq...jar

Later, we started solving all the Blocker, Critical, Major, Minor, and Info related Severity issues if any. Below are the few findings which captured after going through each issue.

Severity of type Major

Before resolving the issue:

Sonarqube interface showing a code review for `PaymentServiceImpl.java`. The issue is titled "Use the 'equals' method if value comparison was intended." and is classified as a Bug. The code snippet shows a comparison of `paymentdetails.getStatus()` with the string `"SUCCESS"` using the `==` operator.

```

18 @Service
19 public class PaymentServiceImpl implements PaymentService {
20
21     @Autowired
22     PaymentDao paymentdao;
23
24     @Override
25     public void pay(OrderModel order) {
26         // TODO Auto-generated method stub
27
28         // Implementing the bridge design pattern
29         AbstractPayment payment = new CreditPayment();
30         payment.payment_gateway = new PluggableAdapter(new CCAvenue());
31         PaymentDetailsModel paymentdetails = payment.makePayment(order);
32
33         System.out.println("Payment status is: " + paymentdetails.getStatus());
34
35         if (paymentdetails.getStatus() == "SUCCESS") {
36
37             paymentdao.update(paymentdetails, order);
38         }
39     }
40 }

```

The issue details indicate it was created 54 minutes ago, has a severity of Major, and is not assigned. The effort is 5 minutes.

Code for resolving the issue:

```

18 @Service
19 public class PaymentServiceImpl implements PaymentService {
20
21     @Autowired
22     PaymentDao paymentdao;
23
24     @Override
25     public void pay(OrderModel order) {
26         // TODO Auto-generated method stub
27
28         // Implementing the bridge design pattern
29         AbstractPayment payment = new CreditPayment();
30         payment.payment_gateway = new PluggableAdapter(new CCAvenue());
31         PaymentDetailsModel paymentdetails = payment.makePayment(order);
32
33         System.out.println("Payment status is: " + paymentdetails.getStatus());
34
35         if (paymentdetails.getStatus().equals("SUCCESS")) {
36             paymentdao.update(paymentdetails, order);
37         }
38     }
39 }

```

Severity of type Info

Before resolving the issue:

```
2      @Created by Prashant, Iauseef, Iarun and Rainy
3      */
4
5      package com.ecommerce.paymentservice.bridgepattern;
6
7      import com.ecommerce.paymentservice.model.OrderModel;
8      import com.ecommerce.paymentservice.model.PaymentDetailsModel;
9
10     public class NetBanking extends AbstractPayment {
11
12         @Override
13         public PaymentDetailsModel makePayment(OrderModel order) {
14             // TODO Auto-generated method stub
15
16             PaymentDetailsModel paymentdetails = new PaymentDetailsModel();
17
18             String processtatus = payment_gateway.processPayment("Net Banking");
19
15
16
17
18
19
```

Complete the task associated to this TODO comment. [See Rule](#) 1 hour ago ▾ L14 🔗

🐛 Code Smell ⓘ Info ○ Open Not assigned 🔍 cwe

Code for resolving the issue:

```
PaymentServi... PaymentDao.java AbstractPaym... NetBanking.java PluggableAda...
4
5 package com.ecommerce.paymentservice.bridgepattern;
6
7 import com.ecommerce.paymentservice.model.OrderModel;
8 import com.ecommerce.paymentservice.model.PaymentDetailsModel;
9
10 public class NetBanking extends AbstractPayment {
11
12     @Override
13     public PaymentDetailsModel makePayment(OrderModel order) {
14
15         PaymentDetailsModel paymentdetails = new PaymentDetailsModel();
16
```

Severity of type Minor #1

Before resolving the issue:

```
payment-service src/.../com/ecommerce/paymentservice/bridgepattern/DirectPay.java
1  ...  /*
2      @Created by Prashant,Tauseef,Tarun and Rainy
3  */
4  package com.ecommerce.paymentservice.bridgepattern;
5
6  public class DirectPay implements IPaymentGateway {
7
8      @Override
9      public String processPayment(String paymentSystem) {
10         // TODO Auto-generated method stub
11
12         System.out.println("Using DirectPay gateway for: " + paymentSystem);
13
14         String status = "Success";
15
16         return status;
17     }
}
```

Immediately return this expression instead of assigning it to the temporary variable "status". [See Rule](#) 1 hour ago ▾ L14 [🔗](#)

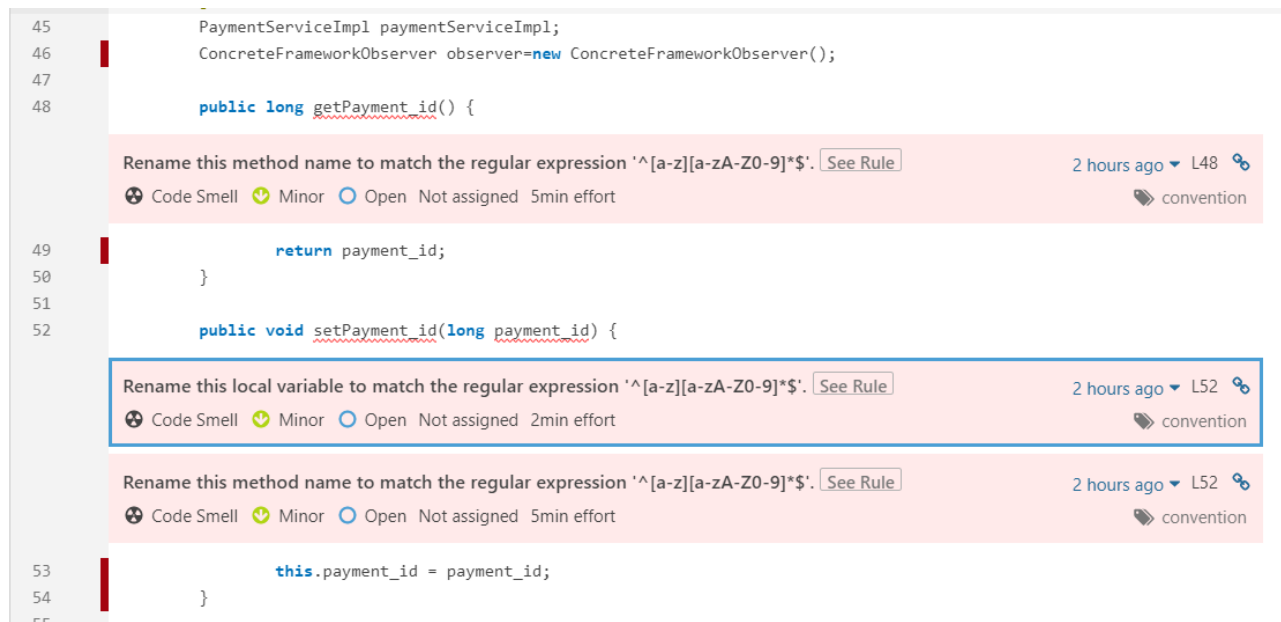
🚫 Code Smell 🟢 Minor 🔵 Open Not assigned 2min effort [👤 clumsy](#)

Code for resolving the issue:

```
2+ @Created by Prashant,Tauseef,Tarun and Rainy
4 package com.ecommerce.paymentservice.bridgepattern;
5
6 public class DirectPay implements IPaymentGateway {
7
8     @Override
9     public String processPayment(String paymentSystem) {
10
11         System.out.println("Using DirectPay gateway for: " + paymentSystem);
12
13         return "Success";
14     }
15
16 }
17 }
```

Severity of type Minor #2

Before resolving the issue:



Code for resolving the issue:



There was one interesting observation while resolving the issues. Initially, for the first run there were only 72 issues and on resolving some issues it resulted in bad code smells in some other parts of the code which lead to more issues. So finally, it led to 125 issues with 1 issue left unresolved and the details of that issue are mentioned in the critique section of our report.

sonarqube

[Projects](#)
[Issues](#)
[Rules](#)

Filters

Clear All Filters

Type

Bug

1

Vulnerability

4

Code Smell

118

Security Hotspot

1

Severity

Blocker

1

Minor

90

Critical

2

Info

7

Major

23

Resolution

Fixed

Reset

Unresolved

1

False Positive

0

Fixed

124

Removed

0

Won't Fix

0

sonarqube

[Projects](#)
[Issues](#)
[Rules](#)
[Quality Profiles](#)
[Quality Gates](#)

?

Search for projects and files...

Log in

Continuous Code Quality

Log in

Read documentation

1

Projects Analyzed

0

Bugs

0

Vulnerabilities

1

Code Smells

0

Security Hotspots

Netflix Eureka Service Discovery

For services able to communicate with each other without hard-coding hostname and port, it just requires each service should register to eureka. Using eureka client retrieves a list of all connected peers of a service registry and makes all further requests to any other service using a load balancer, we have used Zuul API gateway as load balancer.

Here is the screenshot of the registration of all services to eureka.

System Status

Environment	test	Current time	2020-05-01T00:32:09 +0530
Data center	default	Uptime	09:28
		Lease expiration enabled	true
		Renews threshold	11
		Renews (last min)	16

DS Replicas

localhost

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CART-SERVICE	n/a (1)	(1)	UP (1) - DELL-PC.fritz.box:cart-Service:8091
ECOMMERCE-ZUUL-SERVICE	n/a (1)	(1)	UP (1) - DELL-PC.fritz.box:ecommerce-zuul-service:8079
LOGIN-SERVICE	n/a (1)	(1)	UP (1) - DELL-PC.fritz.box:login-service:8090
ORDER-SERVICE	n/a (1)	(1)	UP (1) - DELL-PC.fritz.box:order-service:8084
PRODUCT-CATALOG-SERVICE	n/a (1)	(1)	UP (1) - DELL-PC.fritz.box:Product-Catalog-Service:8089

ZuulApi Gateway

It's a JVM based router and server-side load balancer by Netflix. It provides a single-entry point to microservices and doesn't expose microservices to clients, the client only knows about Zuul .

Here is the snippet of code which shows multiple services registered to zuul.

```
1 zuul.routes.productService.url=http://localhost:8089
2 zuul.routes.loginService.url=http://localhost:8090
3 zuul.routes.cartService.url=http://localhost:8091
4 zuul.routes.checkoutService.url=http://localhost:8084
5 zuul.routes.paymentService.url=http://localhost:8086
6
7 eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka/
8 server.port=8079
9 spring.application.name=ecommerce-zuul-service
10
```

If one service wants to call other service then it has to just find ZUUL-service instance from eureka service discovery using `discoveryClient.getInstances("ecommerce-zuul-service")` instead of getting instance of microservice.

Heroku CLI

The screenshot shows the Heroku CLI interface for the application 'e-commerce-product-service'. The top bar includes the user 'Personal', the application name, and buttons for 'Open app' and 'More'. Below the top bar is a navigation menu with 'Overview', 'Resources', 'Deploy', 'Metrics', 'Activity', 'Access', and 'Settings'. The 'Activity' tab is selected, showing the 'Build Log' for build ID '02c1954d-8a10-4335-bca0-3a5289a474e5'. The log content is as follows:

```
Catalog-Service-0.0.1-SNAPSHOT.pom
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 20.590 s
[INFO] Finished at: 2020-04-09T22:43:04Z
[INFO] -----
----> Discovering process types
Procfile declares types -> (none)
----> Compressing...
Done: 158.5M
----> Launching...
Released v3
https://e-commerce-product-service.herokuapp.com/ deployed to Heroku

Build finished
```

7. Testing

We have tested our application using the Junit Framework. Junit is a unit testing framework for Java programming language. We have written test cases to check the function of the application. Using Junit framework, we can write some test cases to test the code. We have provided a couple of sample test cases below. Test case-1 checks the integration of the service to the database by creating a new order request and checking its entry in the database. Test case-2 checks whether the order status got updated when is payment processing is finished. Test case-3 verifies the ProductInventory class whether it can add a new product request into the database. Test case-4 verifies whether payment status is updated after a successful payment request.

Test Case -1:

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class OrderRepositoryIntegrationTest {

    @Autowired
    OrderRepository orderRepository;

    @Test
    public void testorderindb() {

        //Creating a Order Request and saving to Database
        Orders orderexpected = new Orders("888",123,"details","12.3","Processing",20.3);
        orderRepository.save(orderexpected);

        //Retreiving Value from Database
```



```

Optional<Orders> orderactual = orderRepository.findById("888");

//Comparing both values
Assert.assertEquals(orderexpected.getOrderld() , orderactual.get().getOrderld());
//System.out.println("Test run Successful");
}
}

```

Test Case -2:

```

@RunWith(SpringRunner.class)
@SpringBootTest
public class Orderstatustest {

    @Autowired
    OrderService orderService;

    @Test
    public void testorderstatus() {

        // Creating a new Order Request
        Orders orderactual = new Orders("123",123,"details","12.3","Processing", 20.3);
        //Optional<Orders> orderactual = orderService.findById("123");
        System.out.println(orderactual.getStatus());
        String orderexpected = Orders.SUCCESS;

        //Updating the order status
        orderService.updateOrder(orderactual);

        //Verifying whether Order Status got updated
        Assert.assertEquals(orderexpected , orderactual.getStatus());
    }
}

```

Test Case-3:

```

@RunWith(SpringRunner.class)
@SpringBootTest
public class addInventoryTest {

    @Autowired
    Inventory inventory;

    @Autowired
    ProductService productService;

    @Test
    public void updateInvenotytest() {

```

```

//Adding a new Product to Inventory
ProductMetaDataModel productMetaDataModelexpected = new ProductMetaDataModel();
productService.addNewProduct(productMetaDataModelexpected );

//Requesting the Product from Database
ProductMetaDataModel productMetaDataModelactual =
productService.findProductById(productMetaDataModelexpected.getId());

//Comparing both the ProductId's
Assert.assertEquals(productMetaDataModelexpected.getId(), productMetaDataModelactual.getId());
}
}

```

Test Case-4:

```

@SpringBootTest
public class PaymentIntegrationtest {

    @Autowired
    PaymentDao paymentdao;

    @Test
    public void testpaymentindb() {

        //Creating a New payment request
        PaymentDetailsModel paymentDetailsModelexpected = new PaymentDetailsModel();
        OrderModel paymentDetailsModelactual = new OrderModel();

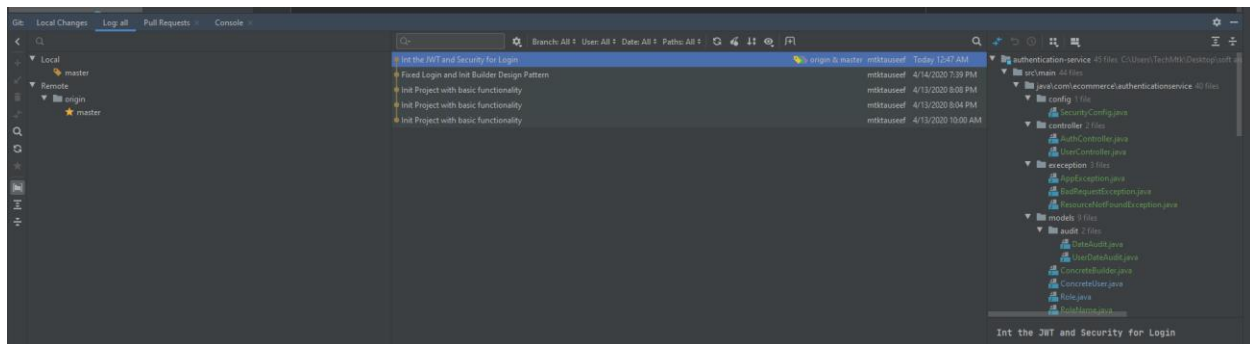
        //Updating the Database
        paymentdao.update(paymentDetailsModelexpected, paymentDetailsModelactual);

        //Testing the Update status
        Assert.assertEquals(paymentDetailsModelexpected.getOrder_details(),
paymentDetailsModelactual.getOrder_details());
        //System.out.println("Test run Successful");
    }
}

```

8. Visualization

We have used Version control (GitHub) for code sharing and working collaboratively over the semester.



Branch: master ▼

Commits on May 1, 2020

- Added Netflix Eureka Server and SonarQube Integration(Code Check)
mtktauseef committed 43 seconds ago [f5922e0](#) [↔](#)
- Int the JWT and Security for Login
mtktauseef committed 23 hours ago [95cca2a](#) [↔](#)

Commits on Apr 14, 2020























- Fixed Login and Init Builder Design Pattern
mtktauseef committed 17 days ago [1894eee](#) [↔](#)

Commits on Apr 13, 2020

- Init Project with basic functionality
mtktauseef committed 18 days ago [b705f8e](#) [↔](#)
- Init Project with basic functionality
mtktauseef committed 18 days ago [28699a5](#) [↔](#)
- Init Project with basic functionality
mtktauseef committed 19 days ago [f071b1c](#) [↔](#)

Branch: master

Commits on May 1, 2020

Init Concrete Interceptor for InterceptorPattern mtktauseef committed 2 minutes ago	Verified	 97b7cc8	
Concreteframework observer update prashanaga committed 2 minutes ago	Verified	 e39282a	
Dispatcher for Interceptor implementation done rainydsz committed 2 minutes ago	Verified	 e9e52c5	
Delete Interceptor.java rainydsz committed 5 minutes ago	Verified	 91283b1	
Interceptor and ConcreteInterceptor ... tarun-s7 committed 5 minutes ago	Verified	 ed1193f	
Delete ConcreteInterceptor.java rainydsz committed 5 minutes ago	Verified	 3f864b9	
Delete ConcreteFrameworkObserver.java rainydsz committed 5 minutes ago	Verified	 12fcaec	
Delete Dispatcher.java rainydsz committed 6 minutes ago	Verified	 ed4040a	
Dispatcher for Interceptor Design pattern completed rainydsz committed 8 minutes ago	Verified	 6dddf91	
PaymentService code after using SonarQube rainydsz committed 6 hours ago	Verified	 a0da3b4	
Updated Bridge design pattern rainydsz committed 7 hours ago	Verified	 d61ee53	





prashanaga / E-Commerce-CartService

Unwatch 1 Star 0 Fork 0

Code Issues 0 Pull requests 0 Actions Projects 0 Wiki Security 0 Insights Settings

Branch: master

Commits on Apr 14, 2020

14th apr commit prashanaga committed 17 days ago	 bece874	
14th apr commit prashanaga committed 17 days ago	 e1d262c	

More Older

Commits on May 1, 2020

Added Communication
 tarun-s7 committed 24 minutes ago
 Verified ee4df21

Commits on Apr 30, 2020

Added test cases
 tarun-s7 committed yesterday
 Verified dec01ed

Commits on Apr 18, 2020

// modifications to checkout controller
 tarun-s7 committed 13 days ago
 Verified 20d5de1

added client discovery
 tarun-s7 committed 13 days ago
 Verified 2ce9f21

Commits on Apr 14, 2020

Add files via upload
 tarun-s7 committed 17 days ago
 1 Verified f377975

rainydsz / PaymentMicroserviceRepo

Unwatch 3 Star 0 Fork 0

Code Issues 0 Pull requests 0 Actions Projects 0 Wiki Security 0 Insights Settings

Branch: master

Commits on May 1, 2020

Init Concrete Interceptor for InterceptorPattern
 mtktauseef committed 2 minutes ago
 Verified 97b7cc8

Concreteframework observer update
 prashanaga committed 2 minutes ago
 Verified e39282a

Dispatcher for Interceptor implementation done
 rainydsz committed 2 minutes ago
 Verified e9e52c5

Delete Interceptor.java
 rainydsz committed 5 minutes ago
 Verified 91283b1

Interceptor and ConcreteInterceptor ...
 tarun-s7 committed 5 minutes ago
 Verified ed1193f

Delete ConcreteInterceptor.java
 rainydsz committed 5 minutes ago
 Verified 3f864b9

Delete ConcreteFrameworkObserver.java
 rainydsz committed 5 minutes ago
 Verified 12fcaec

Delete Dispatcher.java
 rainydsz committed 5 minutes ago
 Verified ed4040a

rainydsz committed 5 minutes ago

Delete ConcreteFrameworkObserver.java
rainydsz committed 5 minutes ago

Verified

12fcae

<>

rainydsz committed 5 minutes ago

Delete Dispatcher.java
rainydsz committed 5 minutes ago

Verified

ed4040a

<>

rainydsz committed 8 minutes ago

Dispatcher for Interceptor Design pattern completed
rainydsz committed 8 minutes ago

Verified

6ddd91

<>

rainydsz committed 6 hours ago

PaymentService code after using SonarQube
rainydsz committed 6 hours ago

Verified

a0da3b4

<>

rainydsz committed 7 hours ago

Updated Bridge design pattern
rainydsz committed 7 hours ago

Verified

d61ee53

<>

Commits on Apr 30, 2020

rainydsz committed yesterday

Eureka Discovery Client implementation
rainydsz committed yesterday

Verified

610a316

<>

rainydsz committed yesterday

Add files via upload
rainydsz committed yesterday

Verified

14d4911

<>

Commits on Apr 13, 2020

mtktauseef committed 18 days ago

my second commit
mtktauseef committed 18 days ago

2

b5346cd

<>

Commits on Apr 12, 2020

mtktauseef committed 19 days ago

My first commit
mtktauseef committed 19 days ago

1

f40eeee

<>

prashanaga / E-Commerce-Microservice

Unwatch 1
Star 0
Fork 0

Code
Issues 0
Pull requests 0
Actions
Projects 0
Wiki
Security 0
Insights
Settings

Branch: master

Commits on Apr 14, 2020

prashanaga committed 18 days ago

13th march commit

2d9c8f3

<>

Commits on Apr 12, 2020

prashanaga committed 20 days ago

11th march commit

b433e40

<>

Commits on Apr 10, 2020

prashanaga committed 21 days ago

10th apr commit

a7b7c8c

<>

prashanaga committed 22 days ago

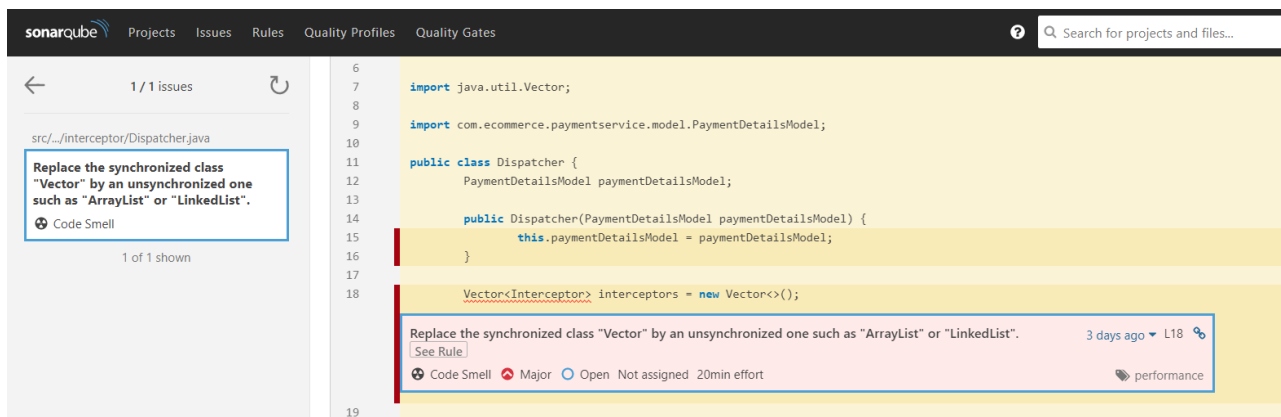
our first commit

6113d5b

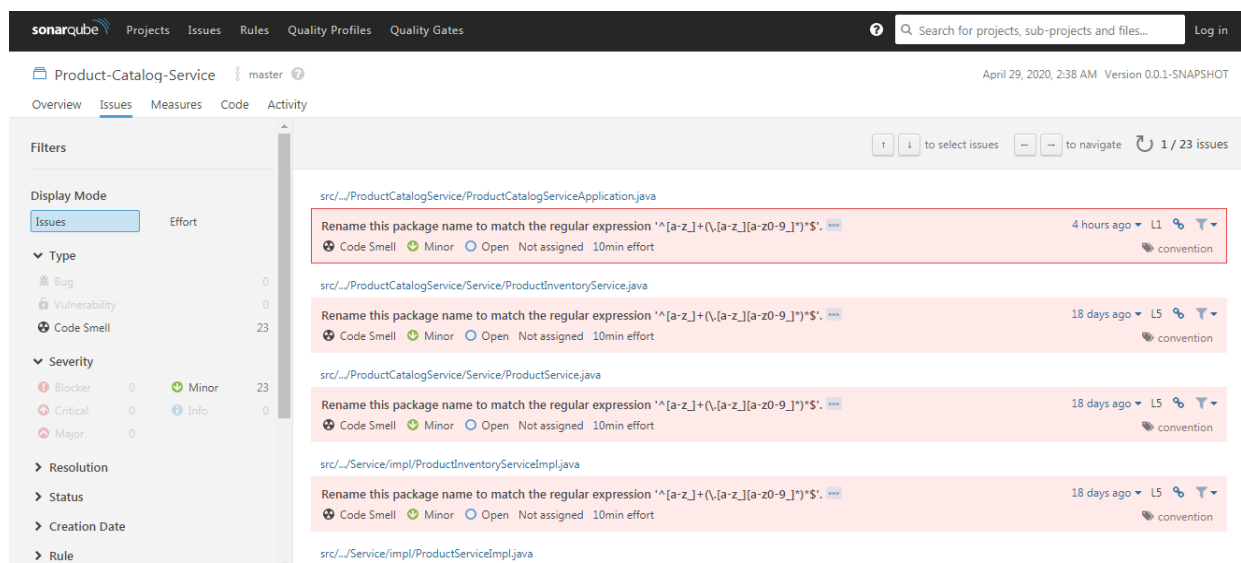
<>

9. Discussion of Problems Encountered

1. We have used SonarQube in our project to measure and analyze the quality of the source code. While analyzing payment service microservice we failed to resolve one **Major Severity** issue. According to SonarQube, the synchronized data type will impact performance but now if we change our code, we may have to redesign the interceptor pattern again which may impact functionality, so these small points should be kept in mind before setting our data type which is a new learning for us.



2. In Product-catalog service microservice we are getting 23 **Minor Severity** issues that are because we named our packages starting from capital letter i.e. "Com" not "com" but now changing package name can create issues with spring boot project. So, our basic understanding from this fact is that we should keep foremost concentration while designing and setting the meta-data of the project.



10. Evaluation/ Critique of support for NFRs

This project is based on Micro Services Architecture where we have different Services like Login, Catalog, Cart, Checkout, etc. We got enhanced performance results because of the synchronized and continuous effort by all the team members. This project is giving us significant learning experience, where the team had a chance to search, learn and explore the features of Spring framework, including Spring Boot, Spring Data JPA, Different patterns like Builder, Bridge, Pluggable Adapter, Strategy, etc. Due to time constraint and the ongoing pandemic situation the entire system was not developed, most of the key use cases were design and implemented. We have used Token-based Authentication and Spring security to secure the endpoints of the Application. We have used several architectural and design patterns, which makes this project more efficient. The main use of the design patterns was to minimize the complexity and to avoid boilerplate code.

The Spring Boot makes this Project more robust and provides endless options to enhance the project business logic. We have used JWT Token for session management, to secure the request from the client-side. Using the JPA Repository reduces the latency and improves the overall performance.

11. Team Member Contribution Overview

Team Member	Total Lines of Code
Mohammed Tauseef Khan	1315
Rainy Mishal Dsouza	682
Prashant Agarwal	967
Tarun Kumar Sunkara	888

Login Service (Authentication)			
Package	Class	LOC	Author
Com.ecommerce.authenticationservice.config	SecurityConfig	32	Tauseef
Com.ecommerce.authenticationservice.controller	AuthController	106	Tauseef
com.ecommerce.authenticationservice.controller	UserController	36	Tauseef
com.ecommerce.authenticationservice.exception	AppException	19	Tauseef
com.ecommerce.authenticationservice.exception	BadRequestException	21	Tauseef
com.ecommerce.authenticationservice.exception	ResourceNotFoundException	34	Tauseef
com.ecommerce.authenticationservice.models.Audit	DateAudit	47	Tauseef
com.ecommerce.authenticationservice.models.Audit	UserDateAudit	41	Tauseef
com.ecommerce.authenticationservice.model	ConcereteBuilder	44	Tauseef
com.ecommerce.authenticationservice.model	ConcereteUser	109	Tauseef
com.ecommerce.authenticationservice.model	Role	47	Tauseef
com.ecommerce.authenticationservice.model	RoleName	10	Tauseef
com.ecommerce.authenticationservice.model	User	112	Tauseef

com.ecommerce.authenticationservice.model	UserBuilder	34	Tauseef
com.ecommerce.authenticationservice.model .payload	ApiResponse	28	Tauseef
com.ecommerce.authenticationservice.model .payload	Request	20	Tauseef
com.ecommerce.authenticationservice.model .payload	Response	35	Tauseef
com.ecommerce.authenticationservice.model .payload	JwtAuthenticationResponse	30	Tauseef
com.ecommerce.authenticationservice.model .payload	LoginRequest	31	Tauseef
com.ecommerce.authenticationservice.model .payload28	SignUpRequest	60	Tauseef
com.ecommerce.authenticationservice.model .payload	UserIdentityAvailabilty	21	Tauseef
com.ecommerce.authenticationservice.reposi tory	RoleRepository	17	Tauseef
com.ecommerce.authenticationservice.reposi tory	UserRepository	27	Tauseef
com.ecommerce.authenticationservice.securit y	CustomUserDetailsService	43	Tauseef
com.ecommerce.authenticationservice.securit y	JwtAuthenticationEntryPoint	29	Tauseef
com.ecommerce.authenticationservice.securit y	JwtAuthenticationFilter	65	Tauseef
com.ecommerce.authenticationservice.securit y	JwtTokenProvider	68	Tauseef
com.ecommerce.authenticationservice.securit y	UserPrincipal	117	Tauseef
com.ecommerce.authenticationservice.	WebApplication	32	Tauseef
Product Service			

Package	Class	LOC	Author
Com.ecommerce.ProductCatalogService.broker	InventoryBroker	30	Rainy
Com.ecommerce.ProductCatalogService.command	AddInventory	20	Rainy
Com.ecommerce.ProductCatalogService.command	DeleteInventory	15	Rainy
Com.ecommerce.ProductCatalogService.command	Inventory	50	Rainy
Com.ecommerce.ProductCatalogService.controller	ProductCategoryController	60	Prashant
Com.ecommerce.ProductCatalogService.dao	ProductInventoryDao	40	Prashant
Com.ecommerce.ProductCatalogService.dao	ProductMetaDataDao	55	Prashant
Com.ecommerce.ProductCatalogService.model	ProductInventoryModel	30	Prashant
Com.ecommerce.ProductCatalogService.model	ProductMetaDataModel	65	Prashant
Com.ecommerce.ProductCatalogService.rowMapper	ProductMetaDataRowMapper	30	Prashant
Com.ecommerce.ProductCatalogService.Service	ProductInventoryServiceImpl	55	Prashant
Com.ecommerce.ProductCatalogService.Service	ProductServiceImpl	55	Prashant
Com.ecommerce.ProductCatalogService	addInventoryTest	40	Prashant
Cart Service			

Package	Class	LOC	Author
com.ecommerce.cartservice.controller	CartController	55	Prashant
com.ecommerce.cartservice.dao	ShoppingCartDao	35	Prashant
com.ecommerce.cartservice.model	CartModel	55	Prashant
com.ecommerce.cartservice.model	LineItemModel	60	Prashant
com.ecommerce.cartservice.rowMapper	CartItemRowMapper	25	Prashant
com.ecommerce.cartservice.serviceImpl	CartServiceImpl	80	Prashant
com.ecommerce.cartservice.visitor	CartTotalByQuantity	30	Prashant
com.ecommerce.cartservice.visitor	CartTotalByWeight	30	Prashant
com.ecommerce.cartservice.visitor	ShoppingCartVisitorImpl	40	Prashant
com.ecommerce.cartservice	AddToCartTest	40	Prashant
Checkout Service			
Package	Class	LOC	Author
Checkoutmicroservice	CheckoutMicroserviceApplication	36	Tarun
Checkoutmicroservice.controller	CheckoutController	110	Tarun
Checkoutmicroservice.controller	OrderController	55	Tarun

Checkoutmicroservice.model	Cart	69	Tarun
Checkoutmicroservice.model	Orders	92	Tarun
Checkoutmicroservice.model	Shipment	72	Tarun
Checkoutmicroservice.model	Users	76	Tarun
Checkoutmicroservice.Repository	IOrderRepository	15	Tarun
Checkoutmicroservice.Repository	IShipmentRepository	10	Tarun
Checkoutmicroservice.service	IcheckoutService	18	Tarun
Checkoutmicroservice.service	CheckoutServiceImpl	109	Tarun
Checkoutmicroservice.service	IOrderService	16	Tarun
Checkoutmicroservice.service	OrderServiceImpl	50	Tarun
Checkoutmicroservice.StrategyPattern	IStrategy	8	Tarun
Checkoutmicroservice.StrategyPattern	PremiumShipping	11	Tarun
Checkoutmicroservice.StrategyPattern	StandardShipping	11	Tarun
Checkoutmicroservice.StrategyPattern	StartegyContext	15	Tarun
Checkoutmicroservice.test	OrderRepositoryIntegrationtest	36	Tarun
Checkoutmicroservice.test	Orderstatustest	35	Tarun

Payment Service			
Package	Class	LOC	Author
com.ecommerce.paymentservice	PaymentServiceApplication	20	Rainy
com.ecommerce.paymentservice.bridgepattern	AbstractPayment	20	Rainy
com.ecommerce.paymentservice.bridgepattern	CreditPayment	30	Rainy
com.ecommerce.paymentservice.bridgepattern	DebitPayment	30	Rainy
com.ecommerce.paymentservice.bridgepattern	NetBanking	30	Rainy
com.ecommerce.paymentservice.bridgepattern	IPaymentGateway	10	Rainy
com.ecommerce.paymentservice.bridgepattern	CCAvenue	20	Rainy
com.ecommerce.paymentservice.bridgepattern	DirectPay	20	Rainy
com.ecommerce.paymentservice.bridgepattern	ICCAvenue	11	Rainy
com.ecommerce.paymentservice.controller	PaymentController	34	Rainy
com.ecommerce.paymentservice.dao	PaymentDao	32	Rainy
com.ecommerce.paymentservice.model	OrderModel	74	Rainy
com.ecommerce.paymentservice.model	PaymentDetailsModel	80	Rainy
com.ecommerce.paymentservice.service	PaymentService	15	Rainy

com.ecommerce.paymentservice.service.impl	PaymentServiceImpl	40	Rainy
com.ecommerce.paymentservice	PaymentServiceApplicationTests	35	Rainy
com.ecommerce.eureka.server	EurekaServerApplication	16	Rainy
com.ecommerce.paymentservice.service.adapter	PluggableAdapter	40	Prashant
com.ecommerce.paymentservice.service.interceptor	ConcreteFrameworkObserver	20	Prashant
com.ecommerce.paymentservice.service.interceptor	ConcreteInterceptor	25	Tarun
com.ecommerce.paymentservice.service.interceptor	Dispatcher	50	Rainy
com.ecommerce.paymentservice.service.interceptor	Interceptor	20	Tarun
ApiGatewayZuluService			
com.ecommerce.ApiGatewayZuluService	ApiGateWayZuluServiceApplication	15	Prashant
com.ecommerce.ApiGatewayZuluService	ServletInitializer	12	Prashant

12. References

- [1] Fowler, M., 2005. Patterns in enterprise software. *Retrieved January, 30*, p.2013.
- [2] Gamma, E., 1995. *Design patterns: elements of reusable object-oriented software*. Pearson Education India.
- [3] Spring Initializr. 2020. *Spring Initializr*. [online] Available at: <<https://start.spring.io/>> [Accessed 27 March 2020].

[4]Securing a Web Application [Online] Available at: <https://spring.io/guides/gs/securing-web/>

[5] Java Implementation [Online] Available at: <https://docs.oracle.com/javaee/6/tutorial/doc/gkhpa.html>

[6] Introduction to JSON Web Tokens [Online] Available at: <https://jwt.io/introduction/>