ASSIGNMENT REPORT :

GITHUB LINK :

1. Introduction

Application that allows user to be a vendor in an online store and add products. This project uses Java (PlayFrameWork), JavaScript & TailwindCss.

2. Architecture

- JAVA : High-level programming language for desktop and web applications.
- ReactJS : Used to build UI Components.
- TailwindCss : Utility-First CSS framework for building websites.
- MYSQL : Scalable RDBMS for web-based applications and data management.

I use ReactJS for my frontend, which is connected to my backend (JAVA) and Database (MYSQL). I chose this text stack has it allowed a very flexible approach to the requirements. React provided a fast and responsive user interface and java provides secure backend for handling business logic and database access. MySQL offers scalable database that has the ability to handle large amounts of data.

These technologies work well together to create contemporary, scalable, and high-performance online applications. Furthermore, there is a big developer community and tools for learning and debugging these technologies, making it easier to create and maintain your application over time.

3. Functionalities

The program is created with two unique user roles in mind: customer and administrator. Based on the project requirements, each job has varied powers and permissions.

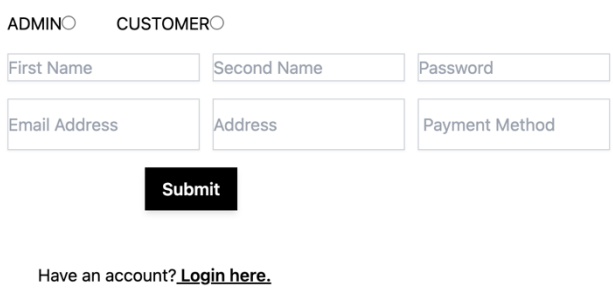
The program is created with two unique user roles in mind: customer and administrator. Based on the project needs, each job has varied powers and permissions.

An admin user has access to more features and functions than a consumer. An administrator has the ability to add new products, manage stocks, and get information about all customers, including their purchase history. These capabilities enable the administrator to manage the product catalogue and stocks, as well as track consumer behaviour and make smart business decisions.

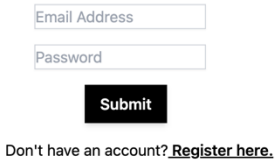
A client, on the other hand, has limited capabilities and can only engage with the application as a regular user. They may explore the product catalogue and add goods to their shopping and view their shopping cart.

```java
@Entity
public class User {

    @Id
    @GeneratedValue(generator = "uuid2")
    @GenericGenerator(name = "uuid2", strategy = "uuid2")
    @Column(name = "id", updatable = false, nullable = false, columnDefinition = "VARCHAR(36)")
    @Type(type = "uuid-char")
    public UUID id;
    4 usages
    public String firstname;
    4 usages
    public String lastname;
    6 usages
    public String password;
    6 usages
    public String email;
    4 usages
    public String address;

    4 usages
    @Enumerated(EnumType.STRING)
    private UserType userType;

    4 usages
    @OneToMany (fetch = FetchType.EAGER, cascade = {CascadeType.MERGE, CascadeType.REMOVE})
    public List<PurchaseHistory> purchaseHistory;
    Peter Solomon
```

- Register



- Login

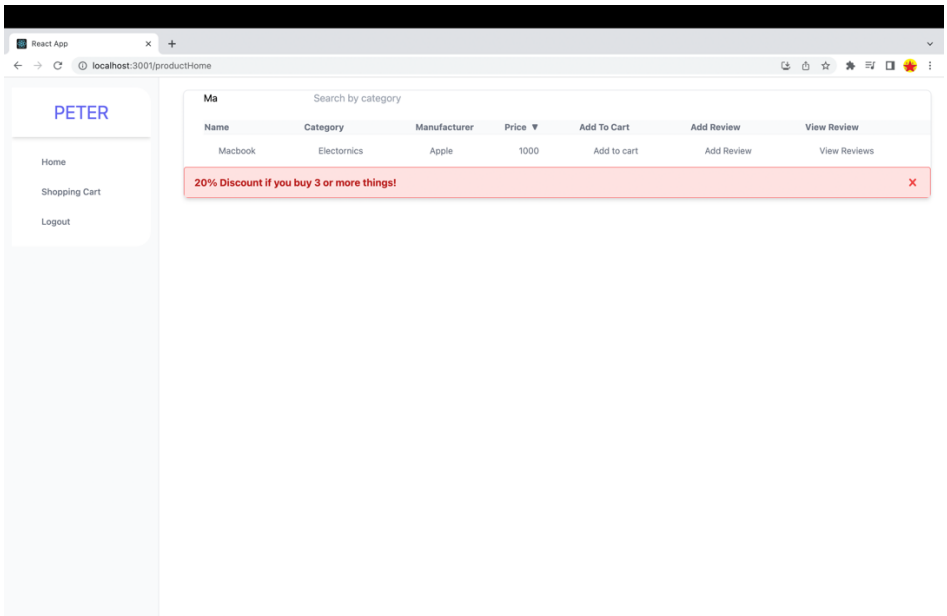


CUSTOMER

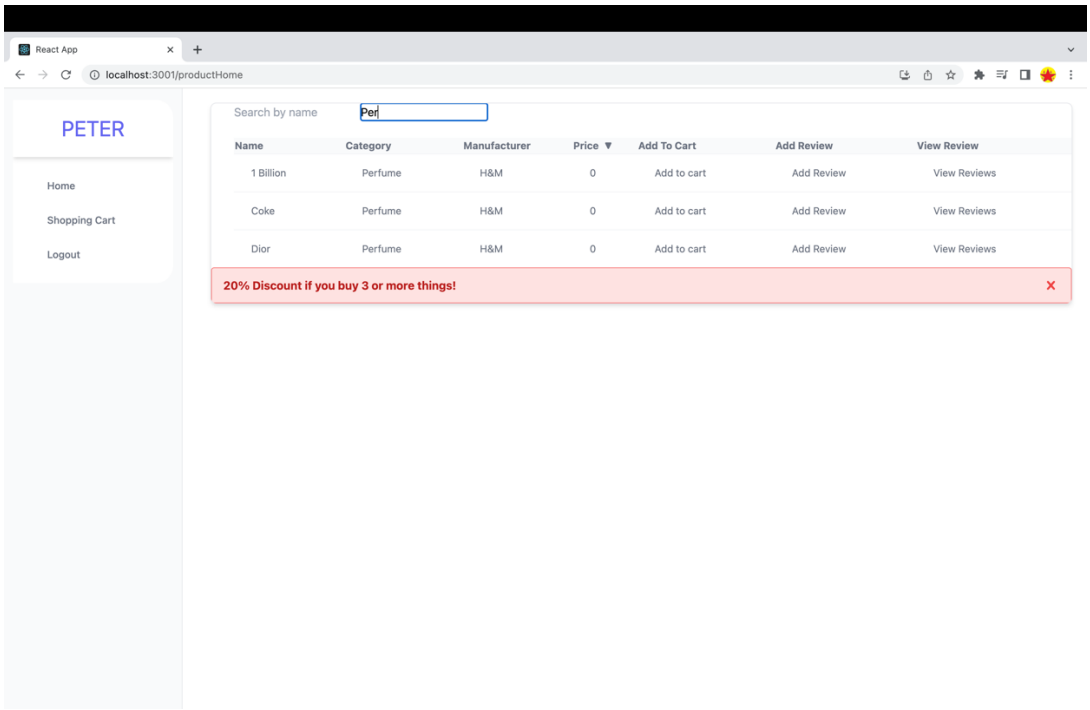
- HOME PAGE

Upon Signing in, customer is brought to home page where they can browse all the products available. Based on project requirements it was said that they should be able to filter by name, category and price which can be seen below. Note the loyalty system designed in the application is to offer 20% discount if user buys 3 or more items.

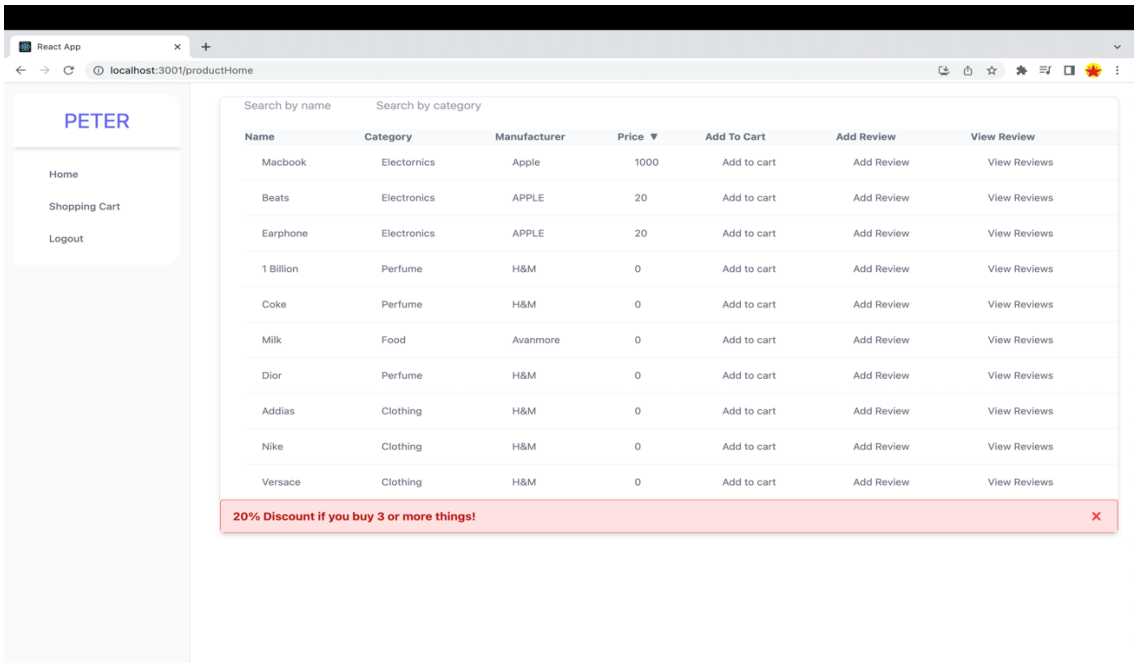
SEARCH BY NAME



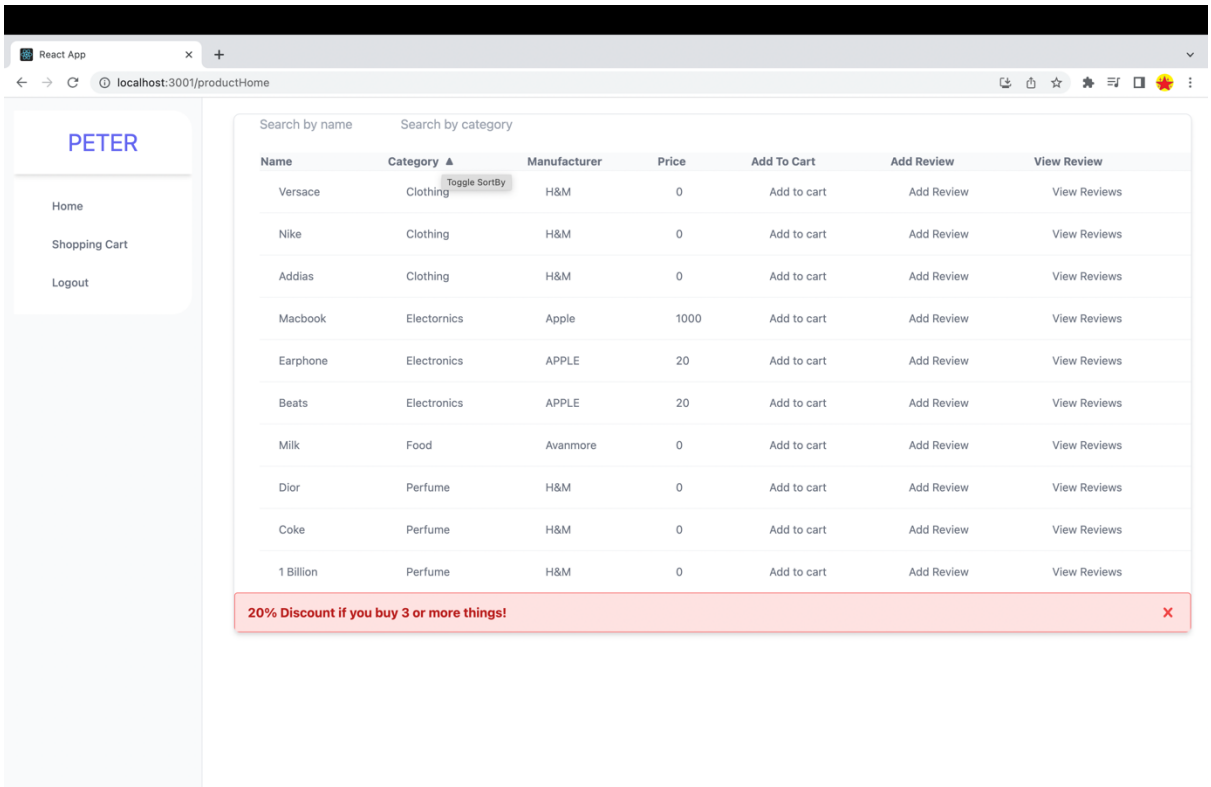
SEARCH BY CATEGORY



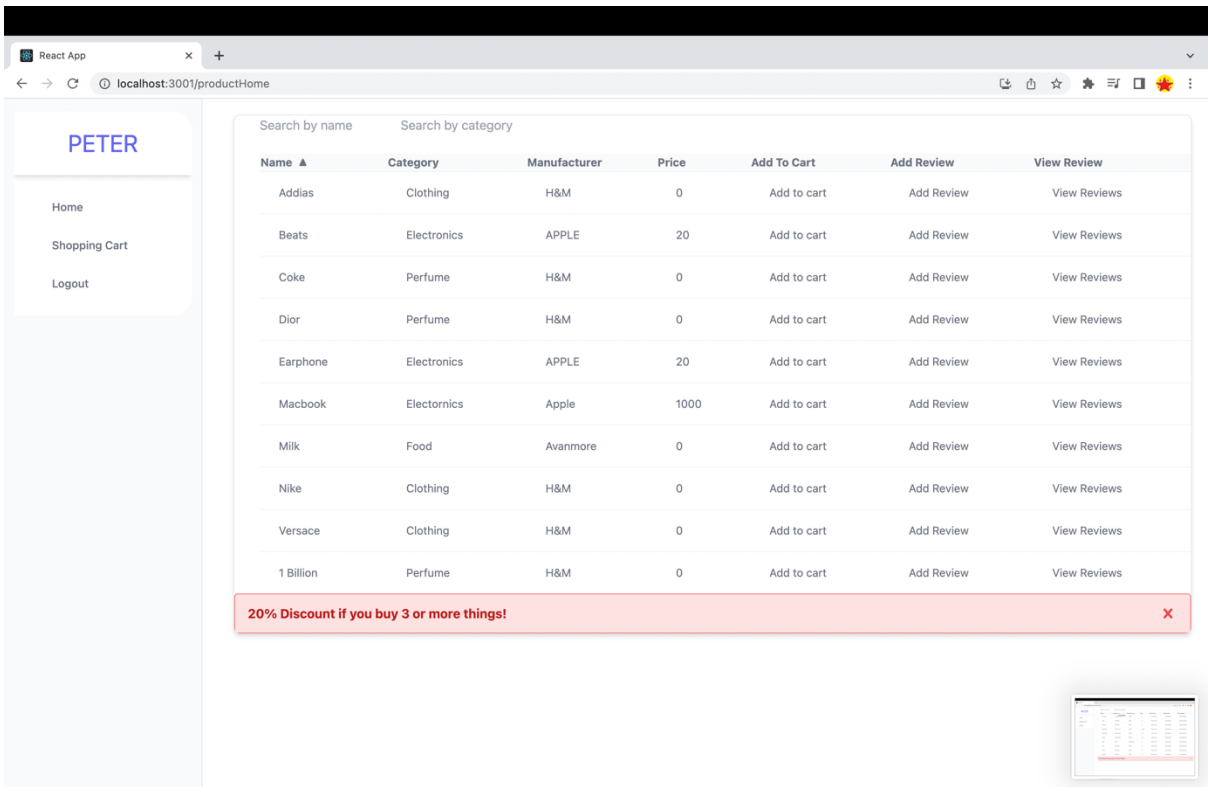
ASCENDING ORDER BY PRICE



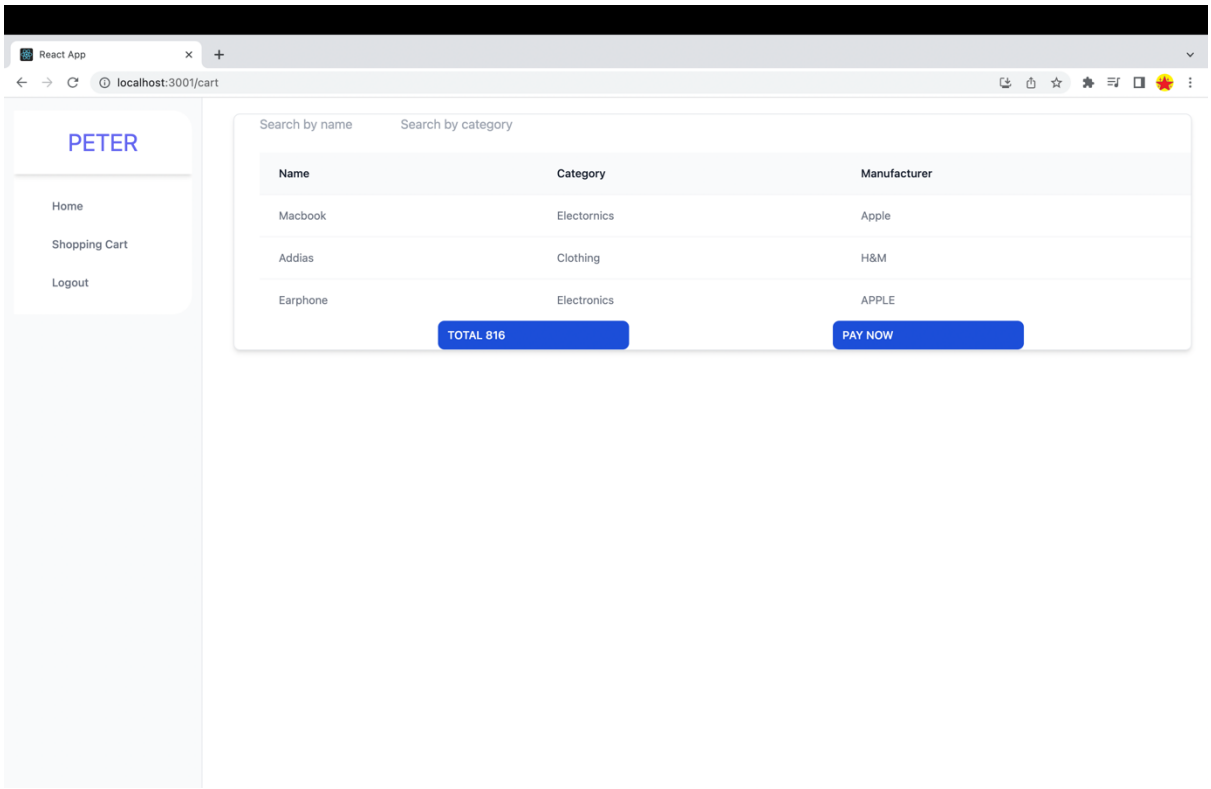
ASCENDING ORDER BY CATEGORY



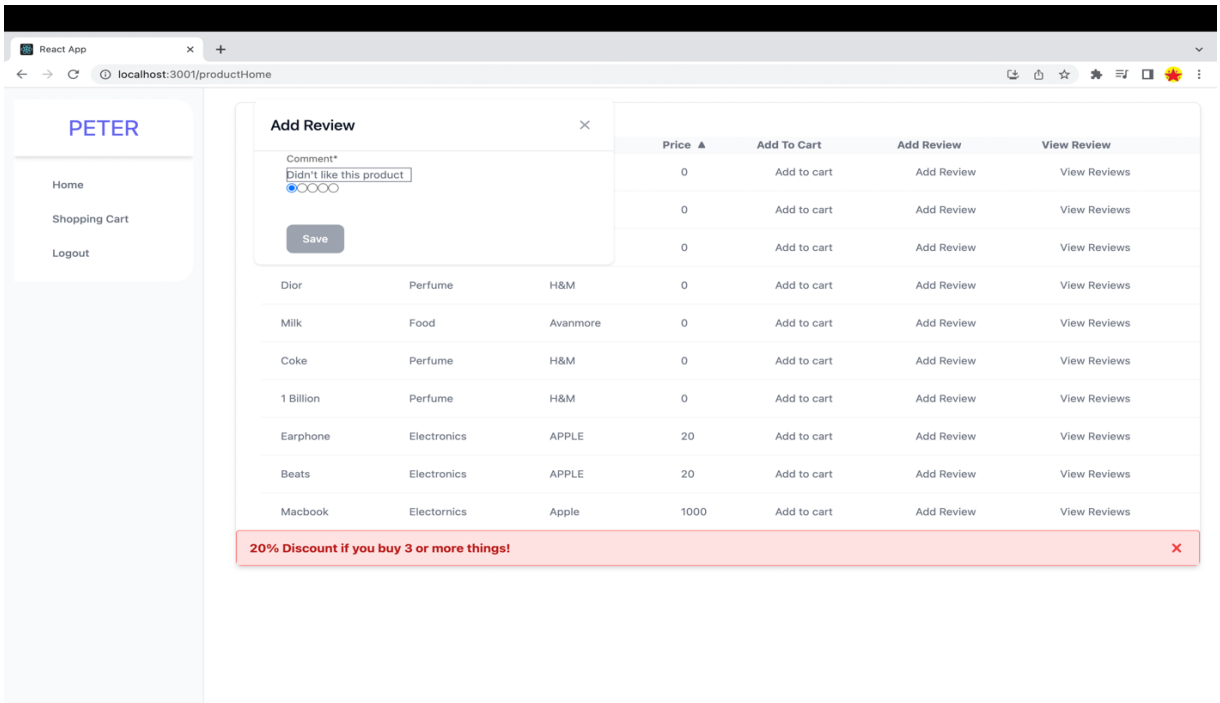
ASCENDING ORDER BY TITLE

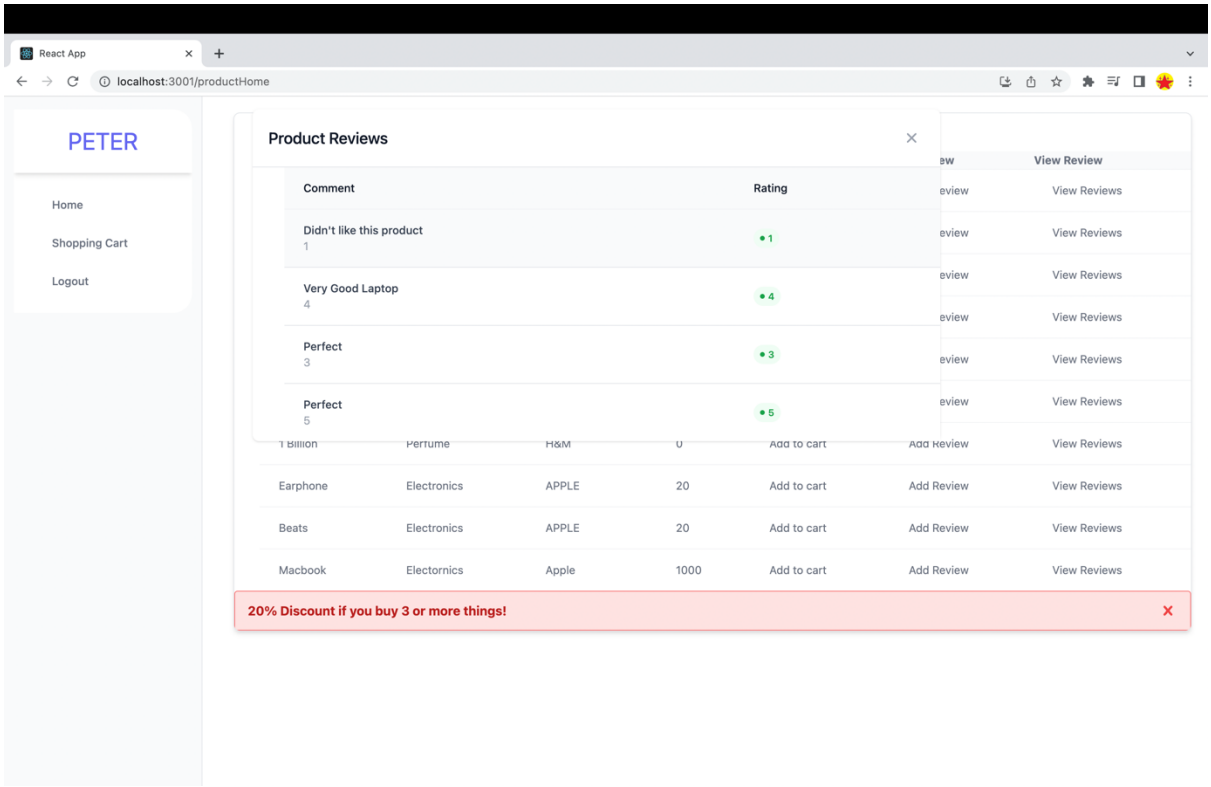


ITEMS ADDED TO CART AND PURCHASED



ADD REVIEW AND VIEW REVIEWS OF SPECIFIC PRODUCT

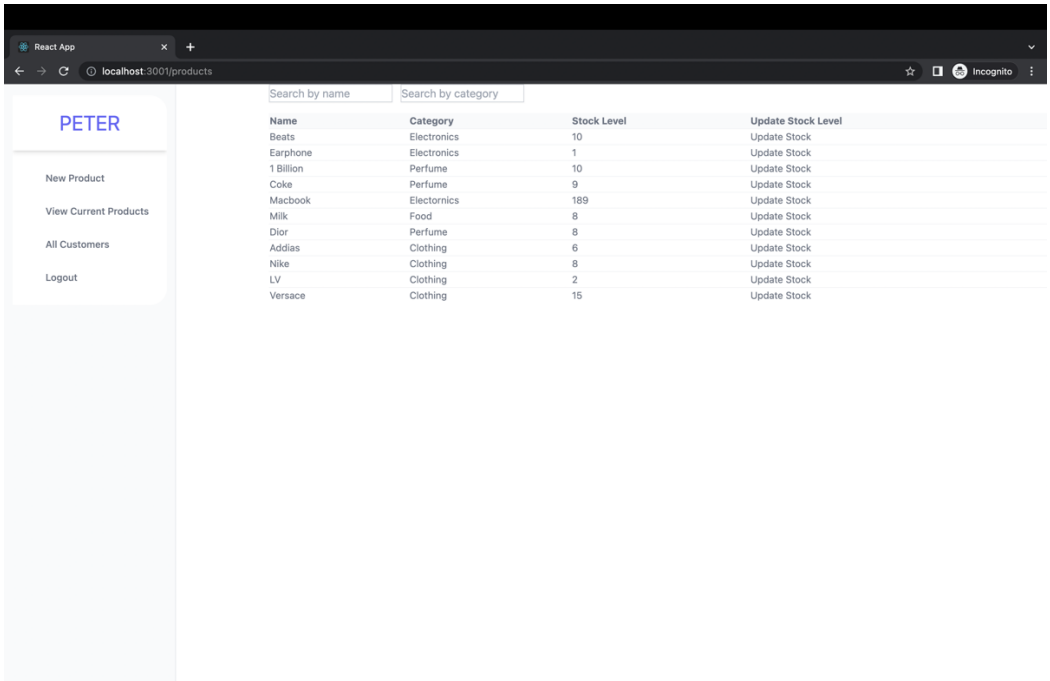




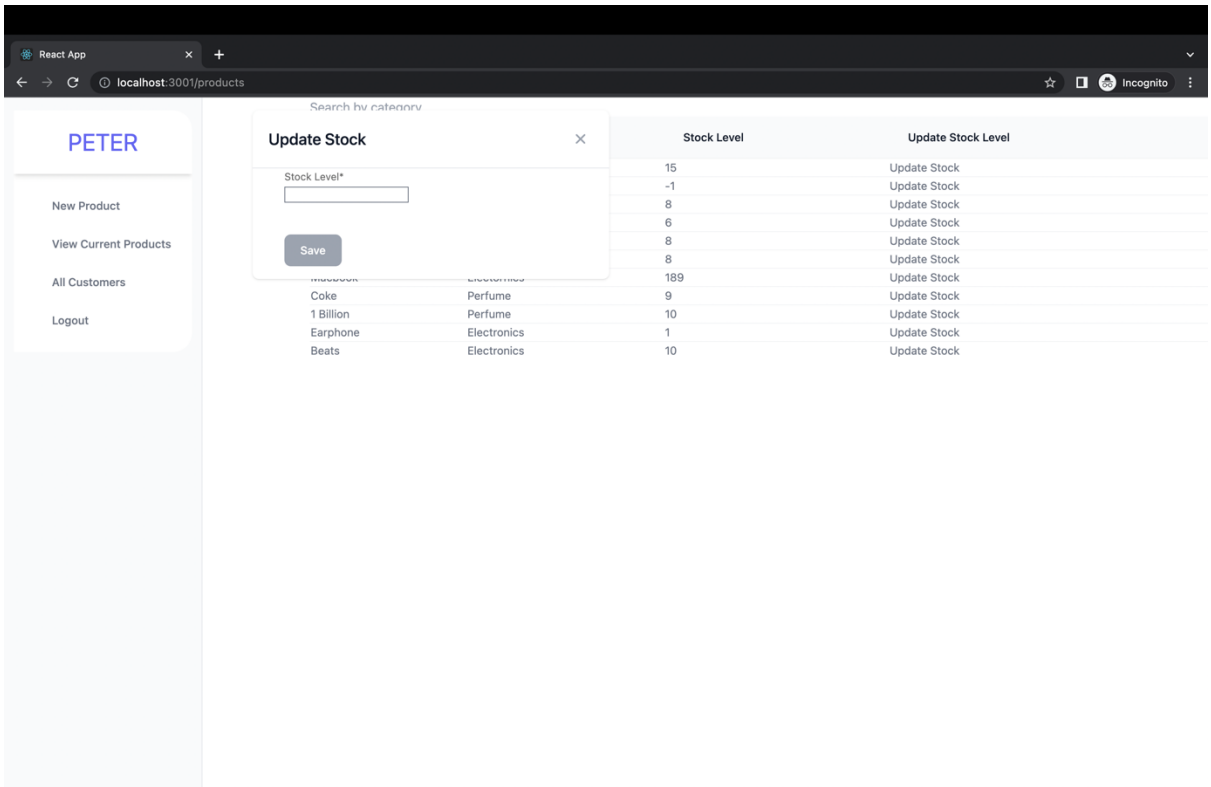
ADMIN

- Home Page

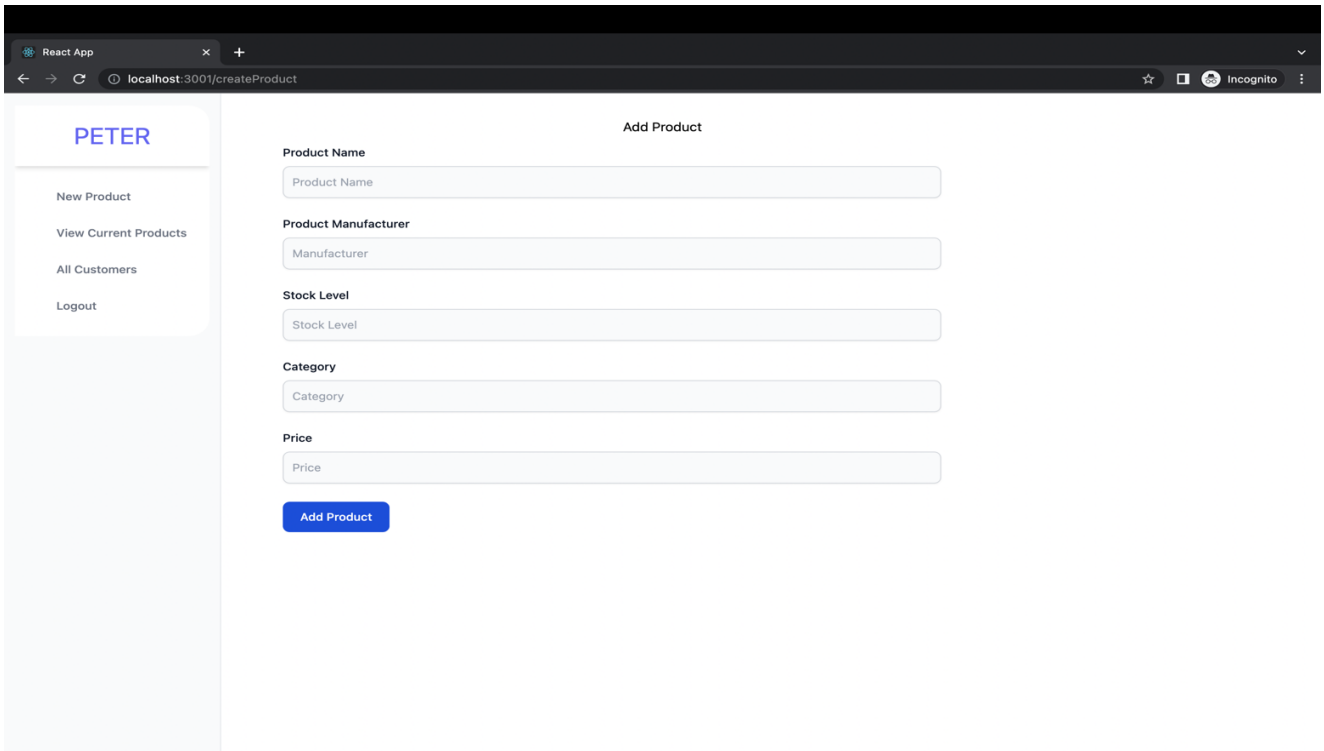
An admin is a more advanced user who has the ability to view all customers, add products, update stock level of their products. Note that their products only show up if it has stock available.



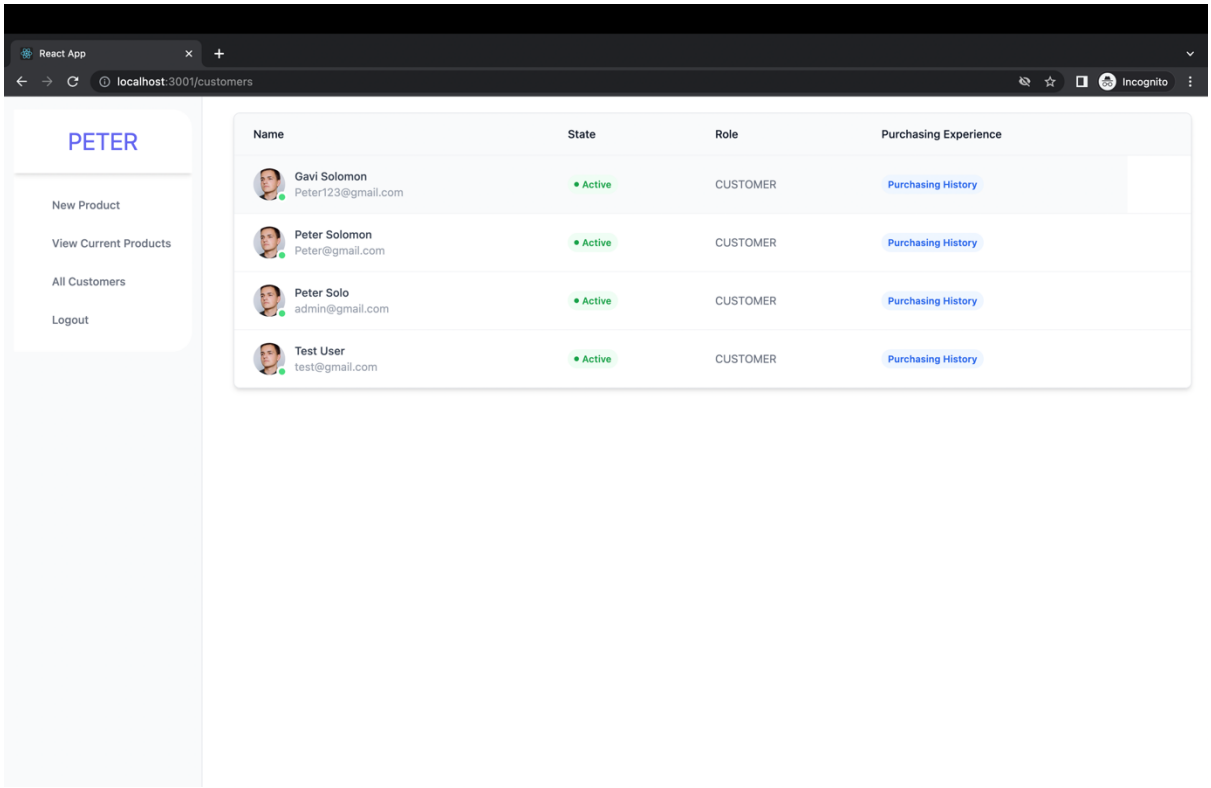
UPDATE STOCK LEVEL



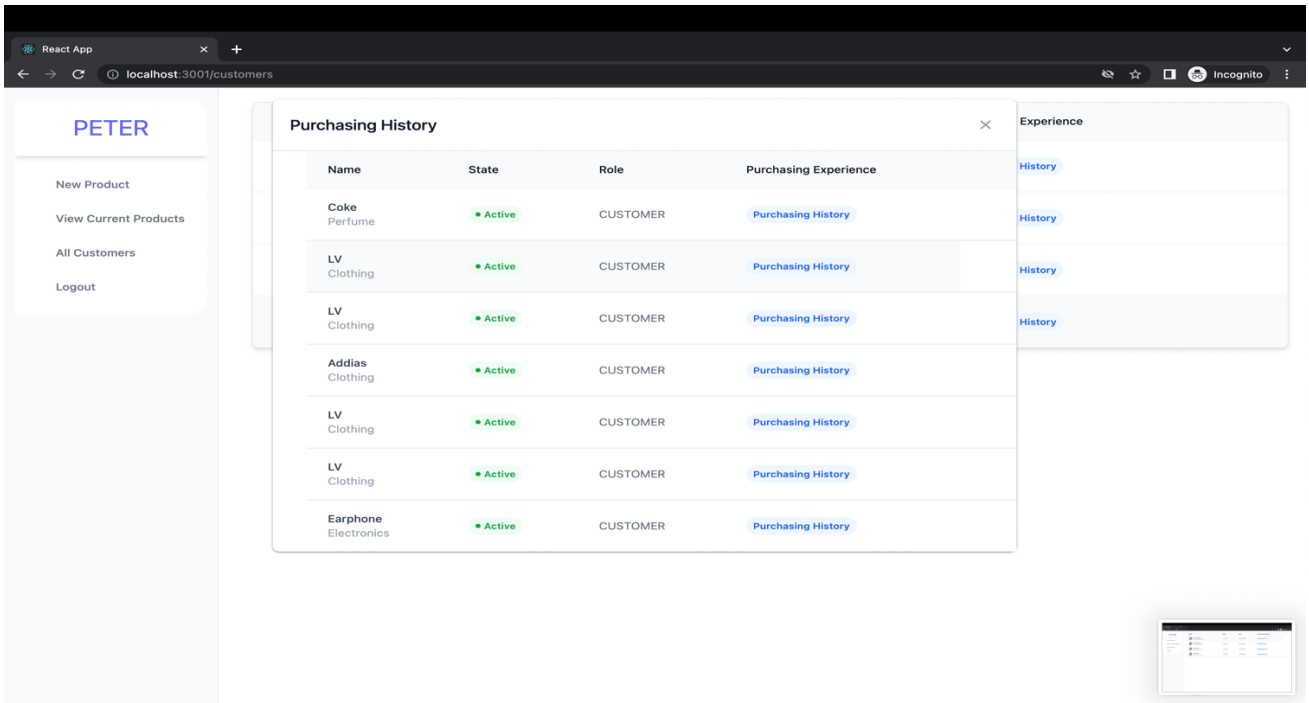
CREATE NEW PRODUCT



ALL CUSTOMERS



CUSTOMER PURCHASING HISTORY



DATABASE

```
+-------------------------+
| Tables_in_spca2         |
+-------------------------+
| Cart                    |
| Cart_Product            |
| Product                 |
| Product_Reviews         |
| PurchaseHistory         |
| PurchaseHistory_Product |
| Reviews                 |
| User                    |
| User_PurchaseHistory    |
+-------------------------+
9 rows in set (0.00 sec)

mysql>
```

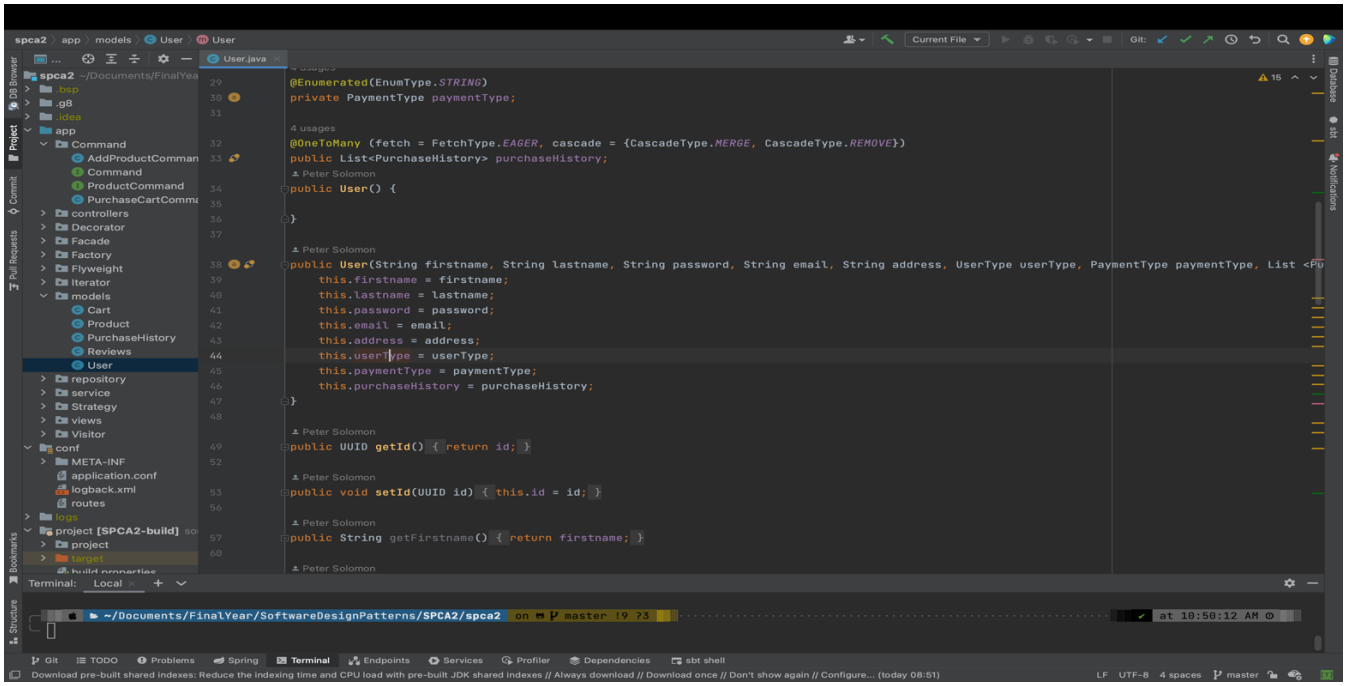
SOFTWARE PATTERNS

Before I began I made a conscious decision to first achieve all the project requirements using normal coding standards, e.g., clean code and after I had completed this go back and refractor for different coding patterns. This was to ensure that I am able to test that the code works after implementing different patterns.

1. MVC PATTERN

I was able to implement the (Model, View Controller Pattern) (Controller, Service, Model, View). It is a software architectural pattern used to design applications.

- Model handles the business logic and manages the state and behaviour. It receives requests from the controller.
- View is the applications UI and displays data for user and sends request to the controller. React Project serves as the view in this project.
- Controller this is the middle-man between model and view.



MVC makes the code more modular and easier to maintain. It breaks the application in different parts making bugs and defects easier to find. A change made in one part only affects the specific MVC.

2. Builder Pattern

Builder pattern helps to simplify and organise the creation of objects. I used it in the CartService.addItems() class to make the creation of Cart object and Product objects easier to maintain.

- In the beginning I created different objects and used formFactory but it seemed harder to maintain and read as there were different objects been created.
- After refactoring I created a CartBuilder class which handled the creating of Cart object. It made cart object easier to maintain thus reducing likelihood for errors. All I need to do is call the methods if I want to add extra properties which makes it easier.

```
//ADDED BUILDER PATTERN, STRATEGY PATTERN
1 usage  Peter Solomon
public Cart addItems(Http.Request cartRequest) {
    UUID uuid = jsonUuid.getUuid(cartRequest);

    CartBuilder cartBuilder = new CartBuilder()
            .setUser(userRepos.getUser(uuid))
            .addProductFromRequest(cartRequest);

    Cart cartObject = cartBuilder.build();

    if (cartRepos.getUserCart(cartObject.getUser()) == null) {
        cartRepos.insertCart(cartObject);

    } else {
        Cart existingCart = cartRepos.getUserCart(cartObject.getUser());
        for (Product cartProduct : cartObject.getProduct()) {
            existingCart.getProduct().add(cartProduct);
        }
        cartRepos.updateCart(existingCart);
    }

    return cartObject;
}
```

3. Iterator Pattern

Iterator pattern helps to iterate over a collection of objects without exposing the underlying data structure to the client code.

- Before refactoring, I was iterating to check for stock level which wasn't an issue as the criteria's were quite small but I had to make changes as a preventive measure in the event of business requirements changed it would be very hard to maintain at some point.
- The new class encapsulates the iteration and filtering of the product collection, the client only calls a line of code and the filtered product is returned making it easier to read and maintain. Thus making it very flexible.

```
//Added Iterator Pattern
1 usage   Peter Solomon
public List <Product> allProducts(Http.Request productRequest){
    AvailProductIterator productIterator = new AvailProductIterator(productRepos.allProducts());

    return productIterator.getActiveProducts();
}
```

```
Cart existingCart = cartRepos.getCart(cartObject);

Iterator<Product> iterator = existingCart.getProduct().iterator();
while (iterator.hasNext()) {
    Product existingProduct = iterator.next();
    if (existingProduct.getId().equals(product.getId())) {
        iterator.remove();
        break;
    }
}
```

4. Command Pattern

Command pattern helps to encapsulate a request as an object and allows for the request to be processed at a later time or by a different component.

- Before refactoring, I was preforming different updates all at once and making it harder to maintain or test.
- After refactoring, I was able to encapsulate the purchase method in the PurchaseCartCommand, making the code more modular. If I wanted to insert a different method for purchasing cart it can easily be added to PurchaseCartCommand and calling it making it more flexible.

```
//Added Command Pattern to purchaseItem, STRATEGY PATTERN
1 usage  Peter Solomon
public Cart purchaseItem(Http.Request cartRequest) throws Exception {
    Cart cartObject = formFactory.form(Cart.class).bindFromRequest(cartRequest).get();
    UUID uuid = jsonUuid.getUuid(cartRequest);
    cartObject.setUser(userRepos.getUser(uuid));
    PurchaseCartCommand purchaseCommand = new PurchaseCartCommand(cartObject, cartService: this, cartRepos, productRepos, purchaseHistoryService);
    purchaseCommand.execute();
    return cartObject;
}
```

This the same logic used with adding a product.

```
//Added Command Pattern to addProduct
    1 usage  Peter Solomon
    public Product addProduct(Http.Request productRequest) throws Exception {
    AddProductCommand addProductCommand = new AddProductCommand(productRequest, productRepos, userRepos, formFactory);

    return addProductCommand.execute();
}
```

5. Factory Pattern

Factory pattern helps to encapsulate the creation of objects and brings flexibility in the creation process of the object.

- After refactoring, I was able to make the code easier to maintain as the ProductFactory class has the ability to do necessary validations. Separates different creation logic making it easier to test. As this particular class, I was having issues with but when I implemented the factory method I was able to discover what the issue was.

```
//ADDED STRATEGY PATTERN, FACTORY PATTERN
1 usage  Peter Solomon
public Product searchProduct(Http.Request productRequest) {
    ProductFactory productFactory = new ProductFactory(userRepos, formFactory);
    Product productObject = productFactory.createProduct(productRequest);

    return productRepos.getProductByName(productObject.getName(), productObject.getUser());
}
```

6. Facade Pattern

Facade pattern is a structural design pattern that provides a simplified interface to a complex system of classes, library or framework.

- Façade pattern simplified the interface of the insertPurchase method, hiding the complexity of the system. It hides the different systems being interacted with. It provides a single point of entry to the subsystems by creating a façade “PurchasingHistoryFacade”.

```
//ADDED FACADE PATTERN
1 usage  Peter Solomon
public void insertPurchase(Cart cart){
    PurchasingHistoryFacade purchasingHistoryFacade = new PurchasingHistoryFacade(purchaseHistoryRepos, userRepos);
    purchasingHistoryFacade.insertPurchase(cart);
}
```

7. Singleton Pattern

Singleton pattern ensures that only one instance of this class is created making it easily accessible globally.

- Singleton pattern is useful here as there's only one instance of the class throughout the entire application, this eliminates the need for creating multiple instances of the class saving memory and synchronisation issues.

```
public class JsonUuid implements Uuid {
    3 usages
    private static JsonUuid instance;

    1 usage   Peter Solomon
    private JsonUuid() {}

    //Added Singleton to create one instance
    4 usages   Peter Solomon
    public static synchronized JsonUuid getInstance() {
        if (instance == null) {
            instance = new JsonUuid();
        }
        return instance;
    }

    7 usages   Peter Solomon
    @Override
    public UUID getUuid(Http.Request request) {
        JsonNode body = request.body().asJson();
        String id = body.get("uuid").asText();
        return UUID.fromString(id);
    }
```

8. Visitor Pattern

Visitor pattern is used when we have to perform an operation on a group of similar kind of objects.

- The reason why I added visitor pattern is that it allows for separation between product and review, in the long run it makes code more maintainable.

```
//ADDED STRATEGY PATTERN, VISITOR PATTERN
1 usage   Peter Solomon
public Reviews addReview(Http.Request reviewRequest){
    Reviews reviewObject = formFactory.form(Reviews.class).bindFromRequest(reviewRequest).get();

    UUID uuid = jsonUuid.getUuid(reviewRequest);
    Product product = new Product();
    product.setId(uuid);
    Product existingProduct = productService.getProduct(product);

    ReviewVisitor reviewVisitor = new ReviewVisitor(reviewObject, reviewRepos);

    existingProduct.accept(reviewVisitor);

    productRepos.updateProduct(existingProduct);

    return reviewVisitor.getPersistedReview();
}
```

9. Decorator Pattern

Decorator is a structural pattern that allows adding new behaviours to objects dynamically by placing them inside special wrapper objects, called decorators.

- I used decorator pattern to add validation functionalities to the user object without modifying the user class. Creating concrete decorators helps to validate functionalities.

```
//ADDED DECORATOR PATTERN
1 usage  Peter Solomon
public User addUser(Http.Request userRequest) throws Exception {
    User user = formFactory.form(User.class).bindFromRequest(userRequest).get();
    UserValidator emailValidator = new EmailValidator();
    UserValidator dupUserValidator = new DupUserValidator(userRepos);

    user = emailValidator.validateUser(user);
    user = dupUserValidator.validateUser(user);

    return userRepos.insertUser(user);
}
```

10. Flyweight Pattern

Flyweight pattern is used to reduce the memory footprint. It can also improve performance in applications where instantiation is expensive.

- Before refactoring, allCustomers created a new list of user each time which can be very resource-intensive, if the system has a large number of users.
- After refactoring, the system creates a pool of users that have been created by the flyweight and is reusable anytime.

```
public List<User> allCustomers() throws IllegalAccessException {
    Flyweight flyweight = new Flyweight(userRepos);
    return flyweight.getUsersByType(User.UserType.CUSTOMER);
}
```

11. Strategy Pattern

Strategy pattern is a behavioural design pattern that turns a set of behaviours into objects and makes them interchangeable inside original context object.

```
public class JsonUuid implements Uuid {
    3 usages
    private static JsonUuid instance;

    1 usage   Peter Solomon
    private JsonUuid() {}

    //Added Singleton to create one instance
    4 usages   Peter Solomon
    public static synchronized JsonUuid getInstance() {
        if (instance == null) {
            instance = new JsonUuid();
        }
        return instance;
    }

    7 usages   Peter Solomon
    @Override
    public UUID getUuid(Http.Request request) {
        JsonNode body = request.body().asJson();
        String id = body.get("uuid").asText();
        return UUID.fromString(id);
    }
```