Abstract:

For this assignment, a spring boot project has been developed. The project represents an e-commerce application that has four microservices, the Customer Account microservice, the Procurement microservice, the Sales microservice, and the Analytics microservice. Each microservice is implemented as a bounded context and uses several domain-driven design patterns including entities, value objects, events, aggregate, and domain service. This allows us to maintain control over our system as it grows and allows us to indicate the functions of each class clearly. The project also uses the event-driven architecture where some microservices publish events and others subscribe to those events using kafka. Stream processing is also used in the Analytics microservice to determine the total quantities of products being bought.

Table of Contents

1. Introduction:

This spring boot project represents an e-commerce application for a supermarket. It was designed to allow companies (that act as customers) and products to be created and updated using the POST and PUT requests in the Customer Account microservice and the Procurement microservice respectively. Contacts for those companies can also be created and updated as well as the details for the products by also using the POST and PUT requests in the Customer Account microservice and the Procurement microservice respectively. This allows the customers to update themselves and their contacts when needed to provide up-to-date information. It also allows the e-commerce application owner to add products to the list of products and add descriptions for them.

Customers can create orders and update them using the POST and PUT requests in the Sales microservice. They can also buy the product in which the status of the order will change from unpaid to paid. Orders, however, cannot be created nor updated if the customer id and product id do not exist. This is ensured by using REST Template that allows the Sales microservice to communicate with the CustomerAccount microservice and the Procurement microservice to ensure the existence of the customer id and product id respectively.

For administration purposes, the application also allows the listing of all orders, all products, or all customers using the GET request. It also supports getting a specific order's, product's, or customer's details using the same request. Additionally, the application allows the sorting of products and customers by price and country respectively. Furthermore, the system analyses the data using stream processing to allow the business to have a better understanding of the popularity of their products.

This report explains that spring project. It first begins by explaining the design methodology of the application. Secondly, it provides a description of the each microservices' architecture and how the packages aligns with it. Then it describes the domain classes and their patterns. It then moves to explaining the use cases that have been developed in the system. Furthermore, it explains the integration of microservices via the REST requests and messaging and lastly, it provides an explanation of the stream processing used in the application.

2. Design Methodology:

Domain Driven Design is an approach to Software Design that bases a systems implementation to an ever-evolving model, disregarding irrelevant details like languages and other unimportant factors (Fowler 2020; Martinez 2020). It consists of many well-documented and perfected principles, many of which we used in our assignment and focuses on creating clear logic to solve the business problems presented to it.

The Principles of Domain Driven Design can be clearly seen in this application, specifically Bounded Contexts, Aggregation, Value Objects and Layered Architecture.

2.1. Bounded Contexts:

A Bounded Context is a conceptual boundary separating parts of the application in regards to their role in the system (Fowler 2014). Bounded Contexts group related concepts, logic and components in order to minimise confusion and ambiguity within the system, to clarify their interrelationships and to provide context to each other (Fowler 2014).

In our System, Our Bounded Contexts are separated by concept and logic, with Procurement, Sale and Customer components being the Bounded Contexts. Their boundary separates their logic, defines their role within their system and illustrates their relationship with other bounded contexts and domains. They communicate with each other via RestTemplate in order to access information from one another.

2.2. Aggregates:

Aggregates in Domain Driven Design refer to a group of domain objects that can effectively and efficiently be treated as a single unit (Fowler 2013). Each Aggregate has an outward facing entity, the Aggregate Root, which is the only component of the aggregate that can be accessed from outside the Aggregate itself (Fowler 2013).

In our system, The Aggregate Roots are the “Product.java” and the “Customer.java” with “ProductDetails.java” and “Contact.java” being the Aggregated component respectively.

2.3. Value Objects:

Value Objects are objects that represent a descriptive aspect of the domain with no conceptual identity as a Value Object (Martinez 2020). They are instantiated to represent elements of the design that we care about only for what they are, not who or which they are (Martinez 2020).

In our system, the value objects are the “ProductDetailValueObject.java” and the “ContactValueObject.java”. The attributes of “ProductDetailValueObject.java” are “comment” and “description” while those of “ContactValueObject.java” are name, phone, email, and position. These classes were chosen to be value objects as they are integral to the overall model but have no need for a unique identity within the system.

2.4. Layered Architecture:

The essential principle of layered architecture is that any element of a layer depends solely on other elements in the same layer or elements of the layers beneath it (WordPress.com n.d.). Communication upward must pass through some indirect mechanism, which keeps each layer's purpose and identity clear and concise (WordPress.com n.d.).

We used Layered Architecture by separating each bounded context into its own package, with the logic and classes within also distributed into different folders (layers) based on their logic and concept. The controller package represents the API layer, the model package represents the domain model layer, the service package represents the service layer, and the repository represents the data access layer. The use of packages enhances the modularity of the project as well as keeping each bounded context's logic and concept clear and concise (WordPress.com n.d.).

Each layer of our system is entirely independent, relying solely on the classes within the package to carry out required business logic. This keeps the overall systems performance and inner logic clear and removes the need for complex delegation between layers and context (WordPress.com n.d.).

2.5. Entities:

An object defined primarily by its unique identity is an Entity (Martinez 2020). Entities have life cycles that can radically change their form and content, but their unique identity is their thread of continuity throughout changes (Martinez 2020). A unique identity is maintained via a unique key, or combination of attributes guaranteed to be unique (Martinez 2020).

Our systems Entities are Product, Customer and Order. Each entity's unique identity is maintained via a generated ID.

2.6. Domain Services:

A service is an operation with defined responsibilities, offered as an interface that stands alone in the model without encapsulating state, as entities and value objects do (Martinez 2020).

In our system, the Domain Service class is the “DomainService.java” class located in the “Sales” microservice. It resides in its respective bounded context, providing its services to components within its own domain.

2.7. Domain Events:

Domain Events capture and process stimuli that can change the state of an application (Fowler 2005). In our system, synchronous and transaction-based events were created. This includes the synchronous “sort customers by country name” event in the Customer Account microservice, the synchronous “sort products by prices” event in the Procurement microservice, the transactional “get product id and stock” event in the Procurement microservice, and the transactional “buy order” event in the Sales microservice. These events include the usage of event listeners with the synchronous events using event publishers as well. Three of those events are being published to kafka using the event listeners for the other microservices to subscribe to. This includes the synchronous “sort customers by country name” event in the Customer Account microservice, the transactional event “get product id and stock” in the Procurement microservice that is being subscribed to by the Sales microservice, and the transactional “buy order” event in the Sales microservice that is being subscribed to by the Customer Account microservice.

2.8. Event Driven Architecture:

Event Driven Architecture is a software design pattern that uses the triggering of events to communicate information between decoupled layers and across separate microservices. Applications that utilise Event Driven Architecture consist of three key components : Event Producers, Event Routers and Event Consumers. Each component has clear responsibilities and scopes in regards to their function and purpose (AWS Amazon 2023).

A Producer is a segment of code or a class that is responsible for spawning or creating an event. The created event is then passed onto the Event Router, which sends the triggered events and the events data to the Event Consumer, which uses the passed data to perform required business logic.

Event Driven Architecture can be seen clearly in our application through the various events within the project, for example the OrderEvent event.

The Order Event has a producer, being itself (OrderEvent.java), which is capable of creating instances of an OrderEvent.

The OrderEvent has a corresponding Event Router(EventSource.java), which specifies the final destination of the created event, which in this instance is a Kafka Topic.

The OrderEvent is then routed to an Event Consumer,(StreamProcessor.java), which consumes the routed data and performs the required business logic with it.

2.9. Stream Processing:

Stream Processing is a programming paradigm that involves continuously ingesting large amounts of data in order to be quickly analysed, filtered, enhanced, transformed or otherwise processed. Once data is processed, it is stored or delivered to other components within the application (AWS Amazon 2023).

The paradigm of Stream Processing can be seen within the Analytics microservice (analytics) of our application. The Analytics Microservice is continuously routed instances of an Order Event via the Event Router, which the main Stream Processing

class (StreamProcessor.java) continuously consumes. The Stream Processor class then filters, analyses and otherwise processes this continuous stream of data and outputs it to the terminal.

3. Comprehensive Design Description:

3.1. Spring Project's Architecture:

3.1.1. Architecture of the Customer Account, Procurement, and Sales Microservices:

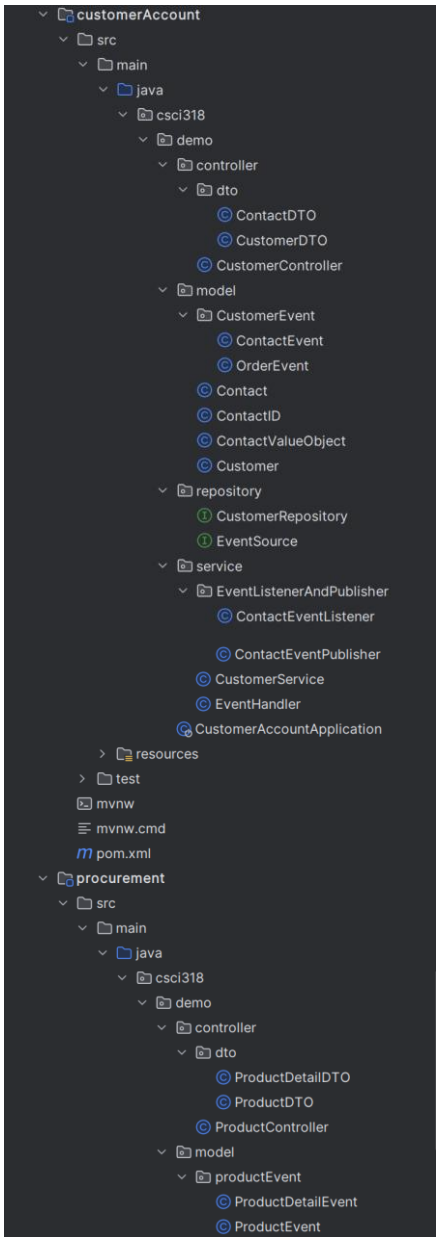
Each microservice was created as a package and inside each one of those packages, four packages were created in order to clearly indicate that the layered architecture specified in the assignment is being followed. Each package includes classes whose functions align with those layers as shown in Figure 1:

- The "customerAccount" package represents the "Customer Account" microservice where the "Customer" domain is defined.
- The "procurement" package represents the "Procurement" microservice where the "Product" domain is defined.
- The "sales" package represents the "Sales" microservice where the "Order" domain is defined.
- The "controller" package contains the controller classes and they represent the "API" layer.
- The "services" package contains the service classes that represent the "Service" layer.
- The "model" package contains the entity classes that have getters and setters which enable them to access the data and therefore they represent the "domain model" layer.
- The "repository" package contains the repository interface classes that can access the "H2 Database". It represents the "Data Access layer".

Inside some of the child packages of the microservices, packages were made in order to clearly indicate and differentiate between the functions of the classes inside those specific packages and the classes in the same directory as those packages. Example:

- In the "model" packages inside the "procurement", "sales", and "customerAccount" packages, event packages were created in order to separate the main entity classes from the event classes.

- In the “service” packages inside the “procurement” and the “customerAccount” packages, packages were created to hold the event listener classes in order to separate it from the main domain service classes.
- In the “controller” packages inside the “procurement”, “sales”, and “customerAccount” packages, dto packages were created in order to separate the controller classes from from the dto classes.



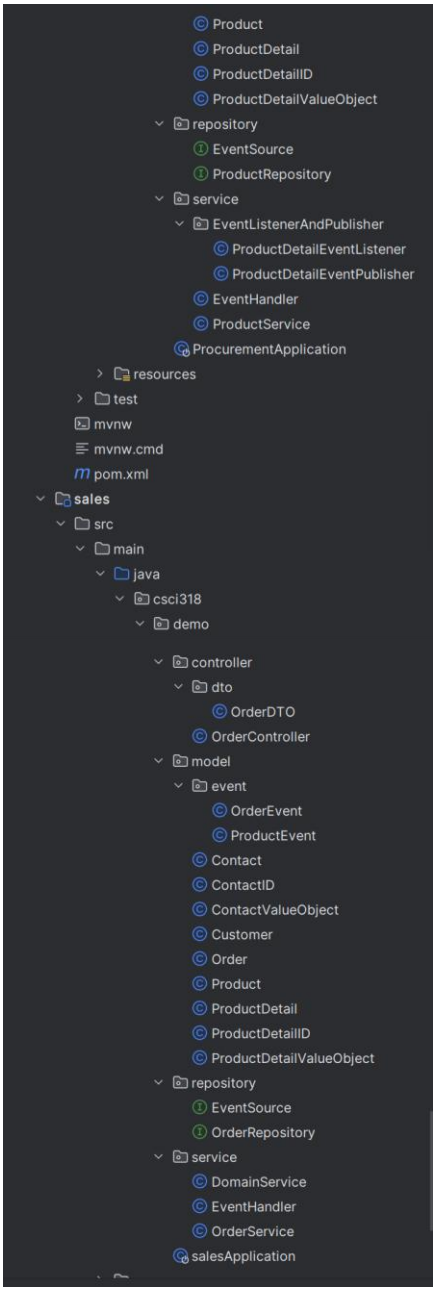


Figure 1: *Structure of the Spring Project 1*

3.1.2. Architecture of the Analytics Microservice:

The Analytics microservice is simple in comparison to the other microservices in this application. The Analytics microservice consists of two packages and two folder :

- The ‘Analytics’ package that contains:
    - The ‘Application Service’ folder, which contains the stream processing class responsible for using stream processing and accepting SQL-style queries.
    - The SpringBootApplication file required to run the spring boot.

- The ‘Events’ package, which contains a copy of the event classes that pertain to the Analytics, in this case the ‘OrderEvent’ class. A copy of this event is needed to serialise and deserialize data for stream processing functions.
- The ‘Resources Folder’, which contains the metadata files necessary to perform stream processing operations

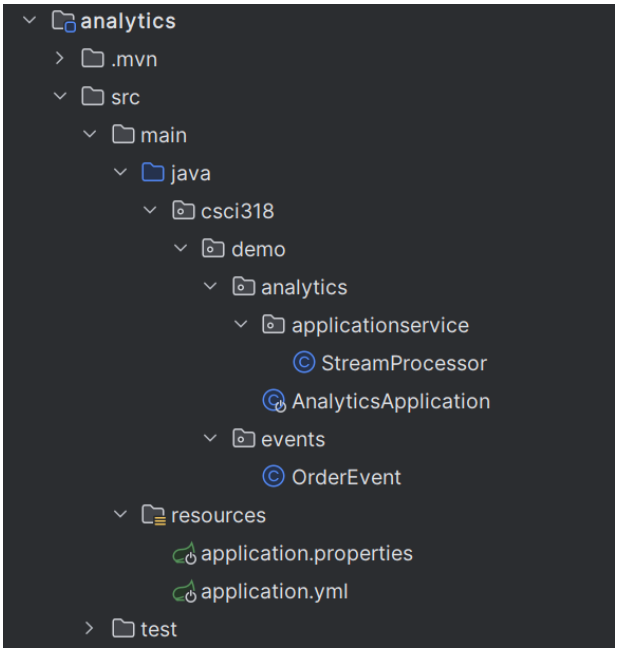


Figure 2: *Structure of the Spring Project 2*

3.2. Description of Domain Classes:

In this project several domain classes have been created including the following classes:

- Customer.java
- Contact.java
- ContactID.java
- ContactValueObject.java
- ContactEvent.java
- Product.java
- ProductDetail.java
- ProductDetailID.java
- ProductDetailValueObject.java
- ProductDetailEvent.java
- ProductEvent.java
- Order.java
- OrderEvent.java
- DomainService.java

These classes have several patterns such as entities, value objects, aggregates, events, and domain services.

3.2.1. Customer.java:

This is an entity class that is part of the “model” package in the project’s structure. It uses the entity pattern and acts as an aggregate root to control the access of its aggregate members (DevIQ n.d.), which are the value object classes. It is one of the classes that belong to the “Data Access” layer in the layered architecture whose role is to connect the “Service” layer and the “Database”, particularly the “CustomerService.java” and the “CustomerRepository.java”. It initiates the attributes that represent the customer, such as the company name, address, and country, and contains the getters as well as the setters that return and set the values of those attributes.

3.2.2. Contact.java:

This is an entity class that is part of the “model” package in the project’s structure and is one of the “Customer” domain’s entity classes. It uses the entity pattern and is being aggregated into the “Customer.java” via the aggregate identifier “ContactID.java” that is defined in “Customer.java” as an object. “Customer.java” is one of the classes that belong to the “Data Access” layer in the layered architecture whose role is to connect the “Service” layer and the “Database”, particularly the “CustomerService.java” and the “CustomerRepository.java”. It has an object of “ContactValueObject.java” in order access its attributes and contains the getters as well as the setters that return and set the value of that object and the “ContactID.java” object.

3.2.3. ContactID.java:

This is an entity class that is part of the “model” package in the project’s structure and is one of the “Customer” domain’s entity classes. It uses the entity pattern and acts as the

aggregate identifier for “Contact.java” and contains the getters as well as the setters that return and set the value of its “contactId”.

3.2.4. ContactValueObject.java:

This is a value object class that is part of the “model” package in the project’s structure. It uses the value object pattern and contains the getters as well as the setters that return and set the value of its attributes.

3.2.5. ContactEvent.java:

This class is part of the “event” package inside the “model” package in the project’s structure. It uses the event pattern and is used to create the “sort” event that is responsible for sorting the customers according to the alphabetical order of their country names. This class is later called by the “CustomerController.java” in order to create an event. The “ContactEventListener.java” listens to that event using the ApplicationListener and the “ContactEventPublisher.java” publishes the event. The “ContactEvent.java” class initiates some attributes that represent the event, such as its eventMessage. It also contains the getters as well as the setters that return and set the values of those attributes.

3.2.6. Product.java:

This is an entity class that is part of the “model” package in the project’s structure. It uses the entity pattern and acts as an aggregate root to control the access of its aggregate members (DevIQ n.d.), which are the value object classes. It is one of the classes that belong to the “Data Access” layer in the layered architecture whose role is to connect the “Service” layer and the “Database”, particularly the “ProductService.java” and the “ProductRepository.java”. It initiates the attributes that represent the product, such as its name, category, and price, and contains the getters as well as the setters that return and set the values of those attributes.

3.2.7. ProductDetail.java:

This is an entity class that is part of the “model” package in the project’s structure and is one of the “Product” domain’s entity classes. It uses the entity pattern and is being aggregated into the “Product.java” via the aggregate identifier “ProductDetailID.java” that is defined in “Product.java” as an object. “Product.java” is one of the classes that belong to the “Data Access” layer in the layered architecture whose role is to connect the “Service” layer and

the “Database”, particularly the “ProductService.java” and the “ProductRepository.java”. It has an object of “ProductDetailValueObject.java” in order access its attributes and contains the getters as well as the setters that return and set the value of that object and the “ProductDetailID.java” object.

3.2.8. ProductDetailID.java:

This is an entity class that is part of the “model” package in the project’s structure and is one of the “Product” domain’s entity classes. It uses the entity pattern and acts as the aggregate identifier for “ProductDetail.java” and contains the getters as well as the setters that return and set the value of its “productDetailId”.

3.2.0. ProductDetailValueObject.java:

This is a value object class that is part of the “model” package in the project’s structure. It uses the value object pattern and contains the getters as well as the setters that return and set the value of its attributes.

3.2.10. ProductDetailEvent.java:

This class is part of the “event” package inside the “model” package in the project’s structure. It uses the event pattern and is used to create the “sort” event that is responsible for sorting the products according to the ascending order of their prices. This class is later called by the “ProductController.java” in order to create an event. The “ProductDetailEventListener.java” listens to that event using the ApplicationListener and the “ProductDetailEventPublisher.java” publishes the event. The “ProductDetailEvent.java” class initiates some attributes that represent the event, such as its eventMessage. It also contains the getters as well as the setters that return and set the values of those attributes.

3.2.11. ProductEvent.java:

This class is part of the “event” package inside the “model” package in the project’s structure. It uses the event pattern and is one of the classes that belong to the “Data Access” layer in the layered architecture. Its object is used by the “Product.java” entity class in order to create the “find product and its” event in the “stock(long productId)” method. This method is later called by the “ProductService.java” class and the “ProductController.java” then calls the method in the “ProductService.java” that called that method. The “EventHandler.java” listens to that event using the TransactionalEventListener and publishes the event to kafka on the

channel specified in the interface “EventSource.java”. The “ProductEvent.java” class initiates some attributes, such the product’s category, name, and price. It also contains the getters as well as the setters that return and set the values of those attributes.

3.2.12. Order.java:

This is an entity class that is part of the “model” package in the project’s structure. It uses the entity pattern and acts as an aggregate root in order to add events to the list of events related to it. It is one of the classes that belong to the “Data Access” layer in the layered architecture whose role is to connect the “Service” layer and the “Database”, particularly the “OrderService.java” and the “OrderRepository.java”. It initiates the attributes that represent the order, such as its supplier, product, and quantity, and contains the getters as well as the setters that return and set the values of those attributes.

3.2.13. OrderEvent.java:

This class is part of the “event” package inside the “model” package in the project’s structure. It uses the event pattern and is one of the classes that belong to the “Data Access” layer in the layered architecture. Its object is used by the “Order.java” entity class in order to create the “buy” event in the “buy(long orderId)” method. This method is later called by the “OrderService.java” class and the “OrderController.java” then calls the method in the “OrderService.java” that called that method. The “EventHandler.java” listens to that event using the TransactionalEventListener and publishes the event to kafka on the channel specified in the interface “EventSource.java”. The “OrderEvent.java” class initiates some attributes that represent the event, such as its name and the order’s supplier, product, and quantity. It also contains the getters as well as the setters that return and set the values of those attributes. Additionally, it is being consumed by the Analytics microservice to determine the sum of quantities of each productId.

3.2.14. DomainService.java:

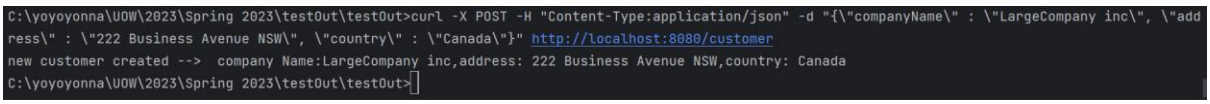
This is a service class that is part of the “services” package in the project’s structure inside the “sales” package. It is one of the classes that belong to the “domain model” layer in the layered architecture it contains the business logic. It also uses the domain service pattern where it operates on the “sales”’s domain objects to check whether a product has been bought or not then return a boolean value that is then used by the “Order.java” to get that value and operate with it in the “buy(long orderId)” method to change the value of the status of an order to “paid” if it is not. The method “buy(long orderId)” in “Order.java” is then called

by the “OrderService.java” in a method which is later called by the “OrderController.java” which prints the updated order details or a message stating the order is already paid for.

3.3 Use Cases:

3.3.1. Use Case 1 - Create Customer:

- **Description:** This use case involves the creation of a new customer using the POST request.
- **Request:** POST
- **API:** http://localhost:8080/customer
- **Input Example (Windows):** curl -X POST -H "Content-Type:application/json " -d "{ \"companyName\" : \"LargeCompany inc\", \"address\" : \"222 Business Avenue NSW\",\"country\" : \"Canada\" }" http://localhost:8080/customer
- **Input Example (Linux):** curl --location 'http://localhost:8080/customer' --header 'Content-Type: application/json' --data '{"companyName": "LargeCompany inc ", "address": "222 Business Avenue NSW", "country": "Canada"}'
- **Output:**



3.3.2. Use Case 2 - Create Contact:

- **Description:** This use case involves the creation of a new contact to an existing customer using the POST request. If the customer does not exist or if it has a contact, the the request will not go through. If a contact is not created for a customer, then its value will be null.
- **Request:** POST
- **API:** http://localhost:8080/customer/{customerId}/contact
- **Input Example (Windows):**curl -X POST -H "Content-Type:application/json" -d "{\"name\" : \"John Boss\", \"phone\" : \"123 456 990\", \"email\" : \"ceo.business@mail.com\", \"position\" : \"CEO\"}" http://localhost:8080/customer/1/contact

- **Input Example (Linux):** curl --location 'http://localhost:8080/customer/1/contact' --header 'Content-Type: application/json' --data '{"name": "John Boss", "phone": "123 456 990", "email": "ceo.business@mail.com", "position": "CEO"}'

- **Output:**

```
C:\yoyoyonna\UOW\2023\Spring 2023\testOut\testOut>curl -X POST -H "Content-Type:application/json" -d "{\"name\" : \"John Boss\", \"phone\" : \"123 4
56 990\", \"email\" : \"ceo.business@mail.com\", \"position\" : \"CEO\"}" http://localhost:8080/customer/1/contact
Contact created for John Boss(1) --> phone: 123 456 990,email: ceo.business@mail.com,position: CEO
C:\yoyoyonna\UOW\2023\Spring 2023\testOut\testOut>
```

3.3.3. Use Case 3 - Create Product:

- **Description:** This use case involves the creation of a new product using the POST request.

- **Request:** POST

- **API:** http://localhost:8081/product

- **Input Example (Windows):** curl -X POST -H "Content-Type:application/json" -d "{\"productCategory\" : \"Dairy\", \"name\" : \"Milk\", \"price\" : \"4.20\"}" http://localhost:8081/product

- **Input Example (Linux):** curl --location 'http://localhost:8081/product' --header 'Content-Type: application/json' --data '{"productCategory": "Dairy","name": "Milk","price": 4.20}'

- **Output:**

```
C:\yoyoyonna\UOW\2023\Spring 2023\CSCI318\Group Project\Part C\Software-Pracs-And-Princips-Assignment-partC-Youmna>curl -X POST -H "Content-Type:app
lication/json" -d "{\"productCategory\" : \"Dairy\", \"name\" : \"Milk\", \"price\" : \"4.20\", \"stock\" : \"20\"}" http://localhost:8081/product
new product created --> category: Dairy, name: Milk, price: 4.2, stock: 20
```

3.3.4. Use Case 4 - Create Product Detail:

- **Description:** This use case involves the creation of a new product detail to an existing product using the POST request. If the product does not exist or if it has a product detail, the the request will not go through. If a product detail is not created for a product, then its value will be null.

- **Request:** POST

- **API:** http://localhost:8081/product/{productID}/detail

- **Input Example (Windows):** curl -X POST -H "Content-Type:application/json" -d "{\"description\" : \"500ml of Chocolate Milk\", \"comment\" : \"Very yummy\"}" http://localhost:8081/product/1/detail

- **Input Example (Linux):** curl --location 'http://localhost:8081/product/1/detail' --header 'Content-Type: application/json' --data '{"description": "500ml of Chocolate Milk","comment": "Very yummy"}'

- **Output:**

3.3.5. Use Case 5 - Create Order:

- **Description:** This use case involves the creation of a new order using the POST request. The implementation of this use case involves using a REST template to check the product's and customer's availability by communicating with the "Procurement" and the "Customer Account" microservices respectively. It does so by checking if the product ID and the customer ID provided in the REST request exist or not in the "Procurement" and the "Customer Account" microservices respectively. If they do then the request is successful. Otherwise an error will be displayed. The "Procurement" and "Customer Account" microservices must be running in order for this use case to work, otherwise an error will appear.

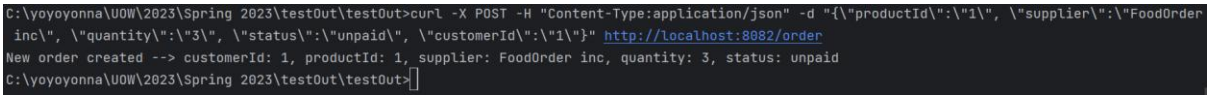
- **Request:** POST

- **API:** http://localhost:8082/order

- **Input Example (Windows):** curl -X POST -H "Content-Type:application/json" -d "{\"productId\":\"1\", \"supplier\":\"FoodOrder inc\", \"quantity\":\"3\", \"status\":\"unpaid\", \"customerId\":\"1\"}" http://localhost:8082/order

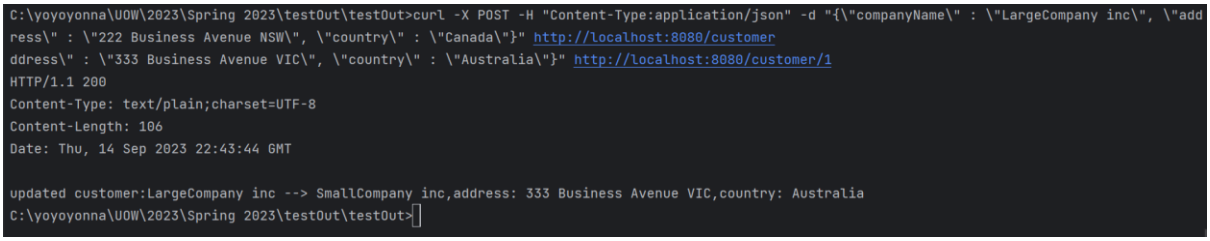
- **Input Example (Linux):** curl --location 'http://localhost:8082/order' \ --header 'Content-Type:application/json' \ --data '{"productId" : "1", "supplier" : "FoodOrder inc", "quantity" : "3", "status" : "unpaid", "customerId" : "1"}'

- **Output:**



3.3.6. Use Case 6 - Update Customer:

- **Description:** This use case involves updating an existing customer using the PUT request. If the customer does not exist the request will not go through.
- **Request:** PUT
- **API:** http://localhost:8080/customer/{customerId}
- **Input Example (Windows):** curl -i -X PUT -H "Content-Type:application/json" -d "{\"companyName\" : \"SmallCompany inc\", \"address\" : \"333 Business Avenue VIC\", \"country\" : \"Australia\"}" http://localhost:8080/customer/1
- **Input Example (Linux):** curl --location --request PUT 'http://localhost:8080/customer/1' --header 'Content-Type: application/json' --data '{"companyName": "SmallCompany inc", "address": "333 Business Avenue VIC", "country": "Australia"}'
- **Output:**



3.3.7. Use Case 7 - Update Contact:

- **Description:** This use case involves updating an existing customer's contactl using the PUT request. If the customer does not exist the request will not go through.
- **Request:** http://localhost:8080/customer/{customerId}/contact
- **API:** PUT
- **Input Example (Windows):** curl -i -X PUT -H "Content-Type:application/json" -d "{\"name\" : \"allan smith\", \"phone\" : \"102020202\", \"email\" :

\"smithy.allan@mail.com\", \"position\" : \"CFO\"}" http://localhost:8080/customer/1/contact

- **Input Example (Linux):** curl --location --request PUT 'http://localhost:8080/customer/1/contact' --header 'Content-Type: application/json' --data '{"name": "allan smith", "phone": "102020202", “email": "smithy.allan@mail.com", "position": CFO"}'

- **Output:**

3.3.8. Use Case 8 - Update Product:

- **Description:** This use case involves updating an existing product using the PUT request. If the product does not exist the request will not go through.

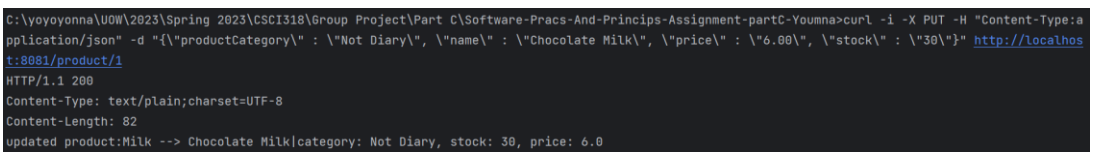
- **Request:** PUT

- **API:** http://localhost:8081/product/{productId}

- **Input Example (Windows):** curl -i -X PUT -H "Content-Type:application/json" -d "{\"productCategory\" : \"Not Diary\", \"name\" : \"Chocolate Milk\", \"price\" : \"6.00\"}" http://localhost:8081/product/1

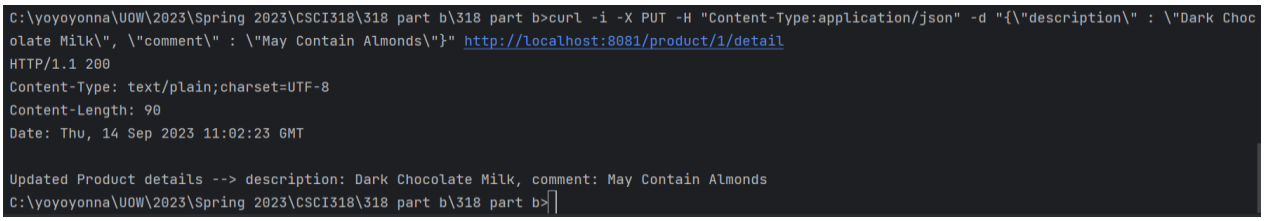
- **Input Example (Linux):** curl --location --request PUT 'http://localhost:8081/product/1' --header 'Content-Type: application/json' --data '{"productCategory": "Not Dairy","name": "Chocolate Milk", "price": 6.00}'

- **Output:**



3.3.9. Use Case 9 - Update Product Detail:

- **Description:** This use case involves updating an existing product's product detail using the PUT request. If the product does not exist the request will not go through.
- **Request:** PUT
- **API:** http://localhost:8081/product/{productId}/detail
- **Input Example (Windows):** curl -i -X PUT -H "Content-Type:application/json" -d "{\"description\" : \"Dark Chocolate Milk\", \"comment\" : \"May Contain Almonds\"}" http://localhost:8081/product/1/detail
- **Input Example (Linux):** curl --location --request PUT 'http://localhost:8081/product/1/detail' --header 'Content-Type: application/json' --data '{"description": "Dark Chcocolate Milk","comment": "May cointain Almonds"}'
- **Output:**



3.3.10. Use Case 10 - Update Order:

- **Description:** This use case involves updating an existing order using the PUT request. If the order does not exist the request will not go through. The implementation of this use case involves using a REST template to check the product's and customer's availability by communicating with the "Procurement" and the "Customer Account" microservices respectively. It does so by checking if the product ID and the customer ID provided in the REST request exist or not in the "Procurement" and the "Customer Account" microservices respectively. If they do then the request is successful. Otherwise an error will be displayed. The "Procurement" and "Customer Account" microservices must be running in order for this use case to work, otherwise an error will appear.

- **Request:** PUT
- **API:** http://localhost:8082/order/{orderId}
- **Input Example (Windows):** curl -i -X PUT -H "Content-Type:application/json" -d "{\"productId\" : \"1\", \"supplier\" : \"FoodDelivery\", \"quantity\" : \"5\", \"status\":\"unpaid\", \"customerId\":\"1\"}" http://localhost:8082/order/1
- **Input Example (Linux):** curl --location --request PUT 'http://localhost:8080/order/1' \ --header 'Content-Type:application/json' \ --data '{"productId" : 1, "supplier" : "FoodDelivery", "quantity" : 5, "status" : "unpaid", "customerId" : "1"}'
- **Output:**

3.3.11. Use Case 11 - Buy Order:

- **Description:** This use case involves updating an existing order using the PUT request. If the order does not exist the request will not go through. The implementation of this use case involves the usage of a transactional event where the status of the order will change from “unpaid” to “paid” indicating that the user has “bought” the order. It also involves the usage of a domain service to check if the order has already been “bought” or not. Additionally, once this event is run, it is published to kafka and the “Customer Account” microservice subscribes to it.
- **Request:** PUT
- **API:** http://localhost:8082/order/buy/{orderId}
- **Input Example (Windows):** curl -X PUT http://localhost:8082/order/buy/1
- **Input Example (Linux):** curl --location --request PUT 'http://localhost:8082/order/buy/1'

- **Output:**

```
C:\yoyoyonna\UOW\2023\Spring 2023\testOut\testOut>curl -X PUT http://localhost:8082/order/buy/1
{"id":1,"supplier":"FoodDelivery","productId":1,"quantity":5,"status":"paid","customerId":1}
C:\yoyoyonna\UOW\2023\Spring 2023\testOut\testOut>
```

3.3.12. Use Case 12 - Get All Customers and their Contacts:

- **Description:** This use case involves getting an arraylist of all the customers and their contacts using the GET request.
- **Request:** GET
- **API:** http://localhost:8080/customers
- **Input Example (Windows):**curl -X GET http://localhost:8080/customers
- **Input Example (Linux):** curl --location 'http://localhost:8080/customers'
- **Output:**

```
C:\yoyoyonna\UOW\2023\Spring 2023\Group8_Source_Code>curl -X GET http://localhost:8080/customers
[{"id":1,"companyName":"SmallCompany inc","address":"333 Business Avenue VIC","country":"Australia","contact":{"contactID":{"contactIdId":1},"contac
tValueObject":{"name":"allan smith","phone":"102020202","email":"smithy.allan@mail.com","position":"CFO"}}},{"id":2,"companyName":"LargeCompany inc"
,"address":"222 Business Avenue NSW","country":"Canada","contact":{"contactID":{"contactIdId":2},"contactValueObject":{"name":"John Boss","phone":"1
23 456 990","email":"ceo.business@mail.com","position":"CEO"}}}]
C:\yoyoyonna\UOW\2023\Spring 2023\Group8_Source_Code>
```

3.3.13. Use Case 13 - Find a Specific Customer and its Contacts:

- **Description:** This use case involves getting a specific customer and its contact using the GET request. If the customer does not exist the request will not go through.
- **Request:** GET
- **API:** http://localhost:8080/customer/{customerId}
- **Input Example (Windows):**curl -X GET http://localhost:8080/customer/1
- **Input Example (Linux):** curl --location 'http://localhost:8080/customer/1'
- **Output:**

```
C:\yoyoyonna\UOW\2023\Spring 2023\Group8_Source_Code>curl -X GET http://localhost:8080/customer/1
{"id":1,"companyName":"SmallCompany inc","address":"333 Business Avenue VIC","country":"Australia","contact":{"contactID":{"contactIdId":1},"contact
ValueObject":{"name":"allan smith","phone":"102020202","email":"smithy.allan@mail.com","position":"CFO"}}}
C:\yoyoyonna\UOW\2023\Spring 2023\Group8_Source_Code>
```

3.3.14. Use Case 14 - Get All Products and their Details:

- **Description:** This use case involves getting an arraylist of all the products and their details using the GET request.
- **Request:** GET
- **API:** http://localhost:8081/products
- **Input Example (Windows):** curl -X GET http://localhost:8081/products
- **Input Example (Linux):** curl --location 'http://localhost:8081/products'
- **Output:**

```
C:\yoyoyonna\UOW\2023\Spring 2023\CSCI318\Group Project\Part C\Software-Pracs-And-Princips-Assignment-partC-Youmna>curl -X GET http://localhost:8081
/products
[{"id":1,"productCategory":"Not Diary","name":"Chocolate Milk","price":6.0,"stock":30,"productDetails":{"productDetailID":{"productDetailId":1},"pro
ductDetailValueObject":{"comment":"May Contain Almonds","description":"Dark Chocolate Milk"}}},{"id":2,"productCategory":"Dairy","name":"Milk","pric
e":4.2,"stock":20,"productDetails":null}]
```

3.3.15. Use Case 15 - Find a Specific Product and its Details:

- **Description:** This use case involves getting a specific product and its details using the GET request. If the product does not exist the request will not go through.
- **Request:** GET
- **API:** http://localhost:8081/product/{productId}
- **Input Example (Windows):** curl -X GET http://localhost:8081/product/1
- **Input Example (Linux):** curl --location 'http://localhost:8081/product/1'
- **Output:**

```
C:\yoyoyonna\UOW\2023\Spring 2023\CSCI318\Group Project\Part C\Software-Pracs-And-Princips-Assignment-partC-Youmna>curl -X GET http://localhost:8081
/product/1
{"id":1,"productCategory":"Not Diary","name":"Chocolate Milk","price":6.0,"stock":30,"productDetails":{"productDetailID":{"productDetailId":1},"prod
uctDetailValueObject":{"comment":"May Contain Almonds","description":"Dark Chocolate Milk"}}}
```

3.3.16. Use Case 16 - Find a Specific Product and its Stock:

- **Description:** This use case involves getting a specific product and its stock using the GET request. If the product does not exist the request will not go through. The implementation of this use case involves the usage of a

transactional event. Additionally, once this event is run, it is published to kafka and the “Sales” microservice subscribes to it.

- **Request:** GET
- **API:** http://localhost:8081/product/{productId}/stock
- **Input Example (Windows):** curl -X GET http://localhost:8081/product/1/stock
- **Input Example (Linux):** curl --location 'http://localhost:8081/product/1/stock'
- **Output:**

```
C:\yoyoyonna\UOW\2023\Spring 2023\CSCI318\Group Project\Part C\Software-Pracs-And-Princips-Assignment-partC-Youmna>curl -X GET http://localhost:8081
/product/1/stock
Product ID: 1, Stock: 30
```

3.3.17. Use Case 17 - Find All Orders:

- **Description:** This use case involves getting an arraylist of all the orders using the GET request.
- **Request:** GET
- **API:** http://localhost:8082/order
- **Input Example (Windows):** curl -X GET http://localhost:8082/order
- **Input Example (Linux):** curl --location 'http://localhost:8082/order'
- **Output:**

```
C:\yoyoyonna\UOW\2023\Spring 2023\testOut\testOut>curl -X GET http://localhost:8082/order
[{"id":1,"supplier":"FoodDelivery","productId":1,"quantity":5,"status":"paid","customerId":1},{"id":2,"supplier":"FoodOrder inc","productId":1,"quan
tity":3,"status":"unpaid","customerId":1}]
C:\yoyoyonna\UOW\2023\Spring 2023\testOut\testOut>
```

3.3.18. Use Case 18 - Find a Specific Order:

- **Description:** This use case involves getting a specific order using the GET request. If the order does not exist the request will not go through.
- **Request:** GET
- **API:** http://localhost:8082/order/{orderId}
- **Input Example (Windows):** curl -X GET http://localhost:8082/order/1

- **Input Example (Linux):** curl --location 'http://localhost:8082/order/1'

- **Output:**

```
C:\yoyoyonna\UOW\2023\Spring 2023\testOut\testOut>curl -X GET http://localhost:8082/order/1
{"id":1,"supplier":"FoodDelivery","productId":1,"quantity":5,"status":"paid","customerId":1}
C:\yoyoyonna\UOW\2023\Spring 2023\testOut\testOut>
```

3.3.19. Use Case 19 - Get All Customers Sorted By Country Name Alphabetically:

- **Description:** This use case involves getting a sorted list of all the customers in alphabetical order of their country names using the GET request. The implementation of this use case involves the usage of a synchronous event (Paraschiv 2021) where the country names of all customers are compared then the customers are sorted according to the alphabetical order of the country names. It also involves the publishing of the results to kafka.

- **Request:** GET

- **API:** http://localhost:8080/customers/sort

- **Input Example (Windows):** curl -X GET http://localhost:8080/customers/sort

- **Input Example (Linux):** curl --location 'http://localhost:8080/customers/sort'

- **Output:**

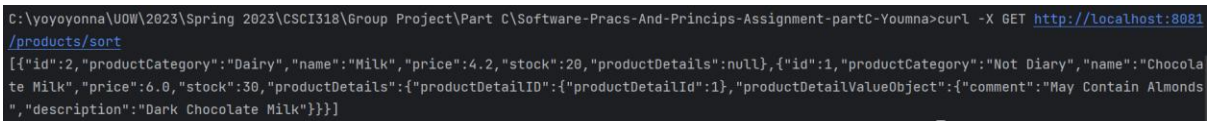
```
C:\yoyoyonna\UOW\2023\Spring 2023\Group8_Source_Code>curl -X GET http://localhost:8080/customers/sort
[{"id":1,"companyName":"SmallCompany inc","address":"333 Business Avenue VIC","country":"Australia","contact":{"contactID":{"contactIdId":1},"contac
tValueObject":{"name":"allan smith","phone":"102020202","email":"smithy.allan@mail.com","position":"CFO"}}},{"id":3,"companyName":"LargeCompany inc"
,"address":"222 Business Avenue NSW","country":"Brazil","contact":null},{"id":2,"companyName":"LargeCompany inc","address":"222 Business Avenue NSW"
,"country":"Canada","contact":{"contactID":{"contactIdId":2},"contactValueObject":{"name":"John Boss","phone":"123 456 990","email":"ceo.business@ma
il.com","position":"CEO"}}}]
C:\yoyoyonna\UOW\2023\Spring 2023\Group8_Source_Code>
```

3.3.20. Use Case 20 - Get All Products Sorted By Price:

- **Description:** This use case involves getting a sorted list of all the products in ascending order of their prices using the GET request. The implementation of this use case involves the usage of a synchronous event (Paraschiv 2021) where the prices of all products are compared then the products are sorted according to the ascending order of the prices.

- **Request:** GET

- **API:** http://localhost:8081/products/sort
- **Input Example (Windows):** curl -X GET http://localhost:8081/products/sort
- **Input Example (Linux):** curl --location 'http://localhost:8081/products/sort'
- **Output:**



3.4. REST Template and Kafka Integration:

3.4.1. Customer Account Microservice Integration:

The “Customer Account” microservice does not use the REST Template. However, it uses kafka and spring cloud stream to publish and subscribe to an event.

When the “ContactEvent.java” is executed, the customers are sorted according to the ascending order of their countries and a list of sorted customers is published to the kafka topic “sortedcustomers”. There are two classes in the “Customer Account” microservice that are responsible for publishing the event to kafka and subscribing to another; the “EventSource.java” class and the “EventHandler.java” class.

3.4.1.1. EventSource.java Class:

The “EventSource.java” class uses Spring Cloud Stream to define the binding messaging channels which are used for publishing (output) and subscription (input) purposes. In this class, there are two main elements:

- *Output Channel (sortedCustomers()):* The "sortedCustomers()" output channel is used to publish the "ContactEvent" event to the kafka topic "sortedcustomers".
- *Input Channel (ordersChannel()):* The "ordersChannel()" input channel is used to subscribe to events coming from a Kafka topic named "orderplaced". When events are published to this Kafka topic, they can be consumed by components that subscribe to the "ordersChannel()" channel.

3.4.1.2. EventHandler.java Class (EventHandler):

The “EventHandler.java” class is a service component in the Spring Boot application that receives and processes events published to Kafka. It is annotated with "@Service" and "@EnableBinding(EventSource.class)" to establish a binding to the EventSource defined in the Event Source class.

- *EventListener for ContactEvent:* The "@EventListener" annotation is used to listen for events of type "ContactEvent." When a "ContactEvent" occurs, this event handler method, "handleSortingEvent(ContactEvent contactEvent)," is

invoked. It sends the "ContactEvent" as a message to the "sortedCustomers()" output channel, effectively publishing it to the corresponding Kafka topic, "sortedcustomers".

- *StreamListener for OrderEvent:* The "@StreamListener" annotation is used to subscribe to events from the "orderPlacedChannel" Kafka topic. When an "OrderEvent" is received from this topic, the "receiveEvent(OrderEvent orderEvent)" method is called. It then processes and logs information from the received event, such as the order ID and status.

3.4.2. Procurement Microservice Integration:

The “Procurement” microservice does not use the REST Template. However, it uses kafka and spring cloud stream to publish an event.

When the “ProductEvent.java” is executed, the product’s stock is published to the kafka topic “productstock”. There are two classes in the “Procurement” microservice that are responsible for publishing the event to kafka; the “EventSource.java” class and the “EventHandler.java” class.

3.4.2.1. EventSource.java Class:

The “EventSource.java” class uses Spring Cloud Stream to define the binding messaging channels which are used for publishing (output) purposes. In this class, there are is a main element:

- *Output Channel (productStock()):* The "productStock()" output channel is used to publish the "ProductEvent" event to the kafka topic "productStockChannel".

3.4.2.2. EventHandler.java Class (EventHandler):

The “EventHandler.java” class is a service component in the Spring Boot application that receives and processes events published to Kafka. It is annotated with "@Service" and "@EnableBinding(EventSource.class)" to establish a binding to the EventSource defined in the Event Source class.

- *TransactionalEventListener for ContactEvent:* The "@TransactionalEventListener" annotation is used to listen for events of type

"ProductEvent." When a "ProductEvent" occurs, this event handler method, "handleProductStock(ProductEvent productEvent)," is invoked. It sends the "ProductEvent" as a message to the "productStock()" output channel, effectively publishing it to the corresponding Kafka topic, "productStockChannel".

3.4.3. Sales Microservice Integration:

The “Customer Account” microservice uses the REST Template. It also uses kafka and spring cloud stream to publish and subscribe to an event.

When creating or updating an order, the relevant creating or updating methods in the “OrderService.java” are called respectively. These methods involves the usage of a REST template to check the product’s and customer’s existence by communicating with the “Procurement” and the “Customer Account” microservices respectively. It does so by checking if the product ID and the customer ID provided in the REST request exist or not in the “Procurement” and the “Customer Account” microservices respectively. If they do then the request is successful. Otherwise an error will be displayed. The “Procurement” and “Customer Account” microservices must be running in order for this use case to work, otherwise an error will also appear.

When the “OrderEvent.java” is executed, the status of the order will change from “unpaid” to “paid” if it is not already labelled as “paid”. There are two classes in the “Sales” microservice that are responsible for publishing the event to kafka and subscribing to another; the “EventSource.java” class and the “EventHandler.java” class.

3.4.3.1. EventSource.java Class:

The “EventSource.java” class uses Spring Cloud Stream to define the binding messaging channels which are used for publishing (output) and subscription (input) purposes. In this class, there are two main elements:

- *Output Channel (orderPlaced()):* The "orderPlaced()" output channel is used to publish the "OrderEvent" event to the kafka topic "orderplaced".

- *Input Channel (stockChannel()):* The "stockChannel()" input channel is used to subscribe to events coming from a Kafka topic named "productstock". When events are published to this Kafka topic, they can be consumed by components that subscribe to the "stockChannel()" channel.

3.4.3.2. EventHandler.java Class (EventHandler):

The “EventHandler.java” class is a service component in the Spring Boot application that receives and processes events published to Kafka. It is annotated with "@Service" and "@EnableBinding(EventSource.class)" to establish a binding to the EventSource defined in the Event Source class.

- *TranscationalEventListener for OrderEvent:* The "@TransactionalEventListener" annotation is used to listen for events of type "OrderEvent." When a "OrderEvent" occurs, this event handler method, "handlePlacingOrder(OrderEvent orderEvent)," is invoked. It sends the "OrderEvent" as a message to the "orderPlaced()" output channel, effectively publishing it to the corresponding Kafka topic, "orderplaced".
- *StreamListener for OrderEvent:* The "@StreamListener" annotation is used to subscribe to events from the "productStockChannel" Kafka topic. When an "OrderEvent" is received from this topic, the "receiveEvent(ProductEvent productEvent)" method is called. It then processes and logs information from the received event, such as the product ID and stock.

3.5. Stream Processing:

The Stream Processing functionality in this application takes place in the StreamProcessor class. This class consumes Order Events from the Kafka Topic “orderplaced” and uses the consumed data to perform analytical functionalities needed for business logic.

Once an Order Event is spawned in another layer or microservice of the application, the Order Event object is outputted to the “orderplaced” topic, which the Stream Processing class is a subscribed listener of.

Once a new entry is outputted to the channel, the stream processing class consumes it. Data is consumed as a JSON string, which is then Deserialized upon entry into the StreamProcessing class, using the Serdes Serializer and Deserialiser specified in the resources of the stream processing class.

Once the stream processing class has enough consumed Order Event instances, it begins the analysis functionalities.

The Processor streams the consumed, now deserialized Order Event instance into a KStream, which can then perform a variety of streaming functionalities, like filtering and mapping. Once the data in the KStream has been altered, it is transformed into a KTable. KTables are capable of performing more complex functionalities, like aggregation and reduction. Completed Queries are returned via the command terminal.

4. References:

AWS Amazon 2023, *What is a modern data streaming architecture?,* viewed 14 October 2023, <https://docs.aws.amazon.com/whitepapers/latest/build-modern-data-streaming-analytics-architectures/what-is-a-modern-streaming-data-architecture.html>.

AWS Amazon 2023, *What is an Event-Driven Architecture?*, viewed 14 October 2023, <https://aws.amazon.com/event-driven-architecture/>.

DevIQ n.d., *Aggregate Pattern*, viewed 15 September 2023, <https://deviq.com/domain-driven-design/aggregate-pattern>.

Fowler 2014, *BoundedContext*, viewed 15 September 2023, <https://martinfowler.com/bliki/BoundedContext.html>.

Fowler 2013, *DDD_Aggregate*, viewed 15 September 2023, <https://martinfowler.com/bliki/DDD_Aggregate.html>.

Fowler 2020, *DomainDrivenDesign*, viewed 15 September 2023, <https://martinfowler.com/bliki/DomainDrivenDesign.html>.

Fowler 2005, *Domain Event*, viewed 15 September 2023, <https://www.martinfowler.com/eaaDev/DomainEvent.html>.

HazelCast 2023, *What is Stream Processing?*, viewed 14 October 2023, <https://hazelcast.com/glossary/stream-processing/>.

Martinez, P 2020, *Domain-Driven Design: Everything You Always Wanted to Know About it, But Were Afraid to Ask*, viewed 14 September 2023, <https://medium.com/ssense-tech/domain-driven-design-everything-you-always-wanted-to-know-about-it-but-were-afraid-to-ask-a85e7b74497a>.

Paraschiv, E 2021, *Spring Events*, viewed 14 September 2023, <https://www.baeldung.com/spring-events>.

WordPress.com n.d., *Layered Architecture*, viewed 15 September 2023, <https://ddd-practitioners.com/home/glossary/layered-architecture/>.