# Java Microservices – S2 – Practice Exercise

## Practice Exercise

This document provides a list of exercises to be practiced by learners. Please raise feedback in Talent Next, should you have any queries.

| | |
|---|---|
| Skill | Java Microservices |
| Proficiency | S2 |
| Document Type | Lab Practice Exercises |
| Author | L & D |
| Current Version | 3.0 |
| Current Version Date | 10-July-2023 |
| Status | Active |

## Document Control

| Version | Change Date | Change Description | Changed By |
|---|---|---|---|
| 1.0 | 30-Oct-2019 | Baseline version | Manpreet Singh Bindra |
| 2.0 | 25-Sep-2022 | Added the problem statements for the topics like creating table, multiple profiles, Actuator, AWS Parameter Store, Microservice with MySQL.<br>Modified the detailed description to the existing problem statements. | Manpreet Singh Bindra |
| 3.0 | 10-July-2023 | Added the Spring Initializer (start.spring.io) to all the problem statements. Added the new problem statements for the topics like Spring Cloud Eureka Server, Spring Cloud Eureka Client, Spring Cloud Gateway, Axon Server, CQRS, Event Sourcing, Bean Validation, Message Dispatch Interceptor, Set Based Consistency, Handling Errors and Rollback Transaction, Orchestration based Saga, and Compensating Transaction. | Manpreet Singh Bindra |

## Contents

# Java Microservices – S2 – Practice Exercise

**Note: Every Problem Statement start on a new page**

## Problem Statement 1: Basic Docker Commands

Try out some docker basic commands:

1) Write a command to pull an openjdk:8-apline and mysql image from registry to local machine.
2) Write a command to show all the images.
3) Write a command to run both the container and link the mysql server to openjdk:8-apline instance in interactive mode.
4) Write a command to run both the container and link the mysql server to openjdk:8-apline instance in detached mode.
5) Write a command to list all the registered containers that are running.
6) Write a command to show all the logs of the containers.
7) Write a command to interact with the containers.
8) Write a command to stop and start the container.
9) Write a command to create your own registry for faster performance.
10) Write a command to create your own image and list all images.
11) Write a command to push, pull and remove the image from your own registry.
12) Write a command to remove the stopped containers.
13) Write a command to list information about one or more networks.
14) Write a Dockerfile for the “Hello World” Java program and build the customize image.
15) Write a command to run the customize image and push the image to hub.docker.com.

## Problem Statement 2: Create Spring Boot Microservice for ProductService

Mphasis got a requirement to create an eStoreApplication portal, wherein multiple users access the product details. We need to design the API for the application so we can achieve the interoperability feature in the application.

**Technology stack:**

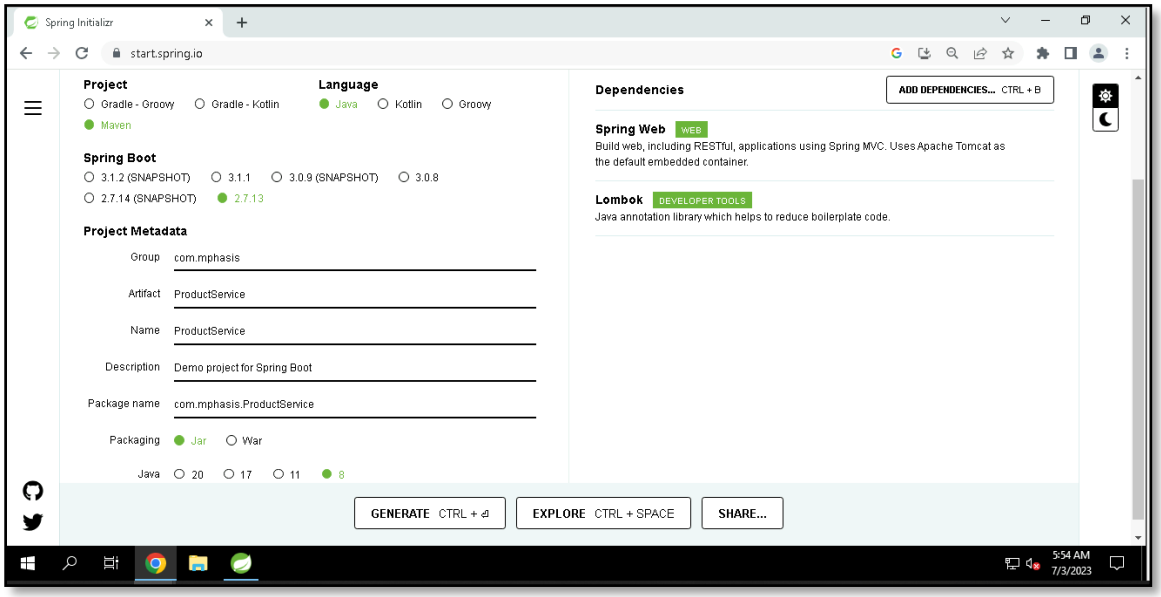
- Spring Web
- Lombok

Building a RESTful web application where CRUD operations to be carried out on entities.

Initially we will have only controller layers into the application:

1. Create a new Project for **ProductService** using the **Spring Initalizr** (start.spring.io).

   Refer the below screenshots:



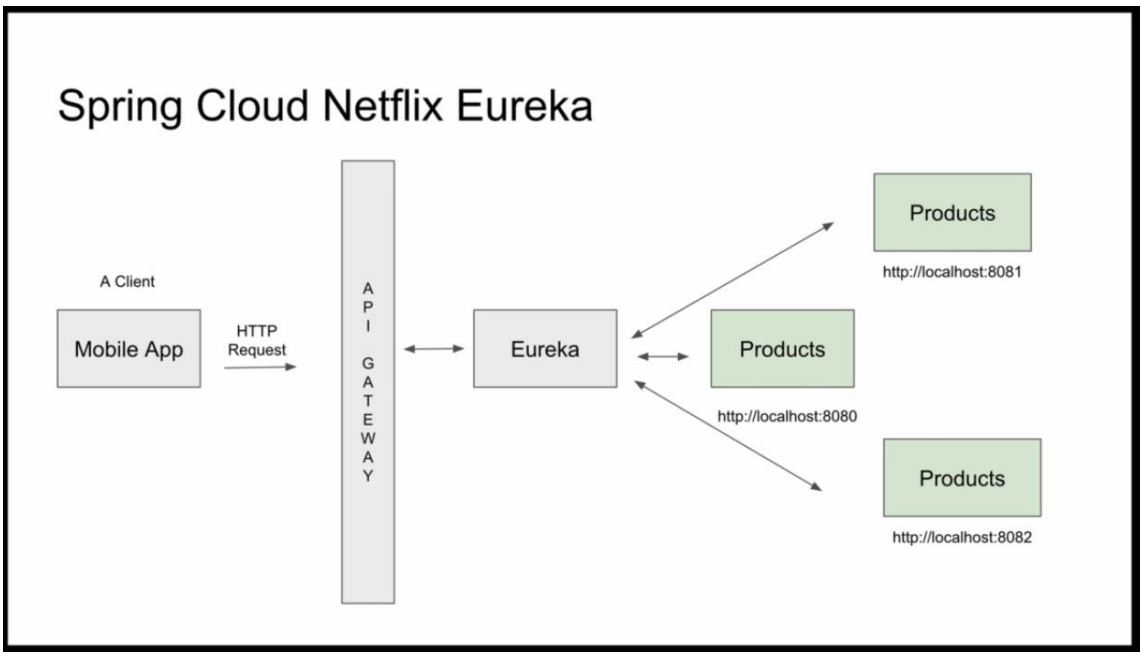

2. Select the **Spring Web**, **Lombok, and Eureka Discovery Client** dependencies.
3. Click on GENERATE button or CTRL + Enter to create the project structure.
4. Let's import the ProductService maven project in STS.
5. Will take some time for the project to be imported and the maven dependencies to be downloaded once you click **Finish**.
6. Let's start building our application.
7. Finally, create a ProductController will have the following Uri's:

| URI | METHODS | Description |
|---|---|---|
| /products | POST | Return a String - "HTTP POST Method Handled" |
| /products | PUT | Return a String - "HTTP PUT Method Handled" |
| /products | GET | Return a String - "HTTP GET Method Handled" |
| /products | DELETE | Return a String - "HTTP DELETE Method Handled" |

8. Running on a web server tier (using tomcat).

## Problem Statement 3: Configure Microservices with Eureka Service Registry Server

The "eBookStore" application instance will exposes a remote API such as HTTP/REST at a particular location (host and port). To overcome the challenge of dynamically changing service instances and their locations. The code deployers intended to create a service registry, which is a database containing information about services, their instances, and their locations.



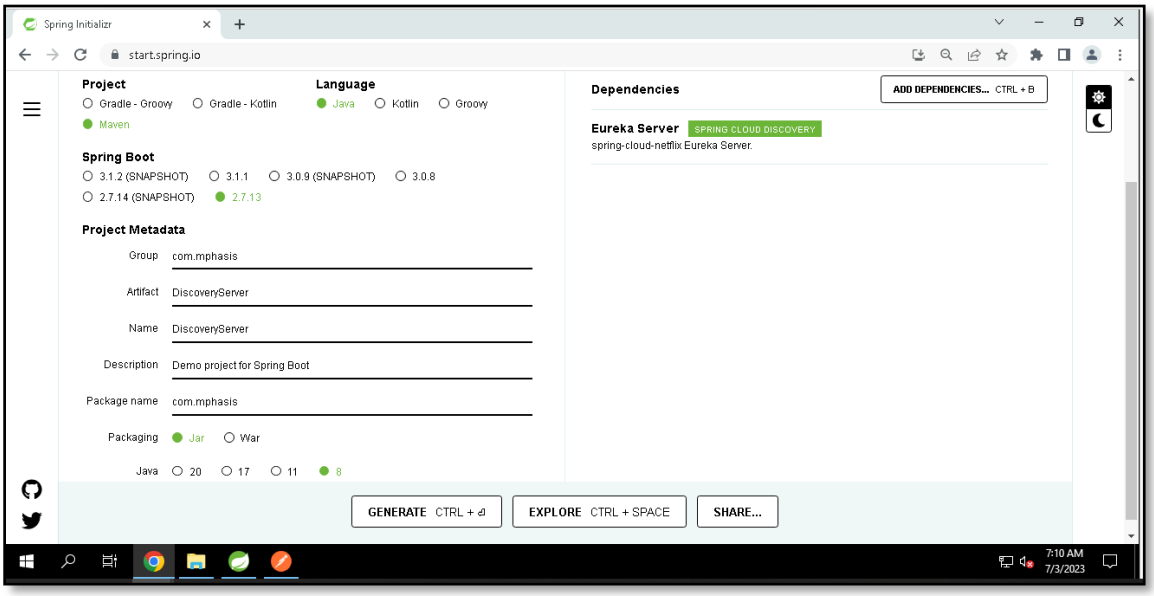

**Technology stack:**

- Spring Cloud Eureka Server

**Steps for Spring Cloud Config Server:**

1. Create a new Project for **DiscoveryServer** using the **Spring Initalizr** (start.spring.io).

   Refer the below screenshots:



2. Select the **Eureka Server** dependency.

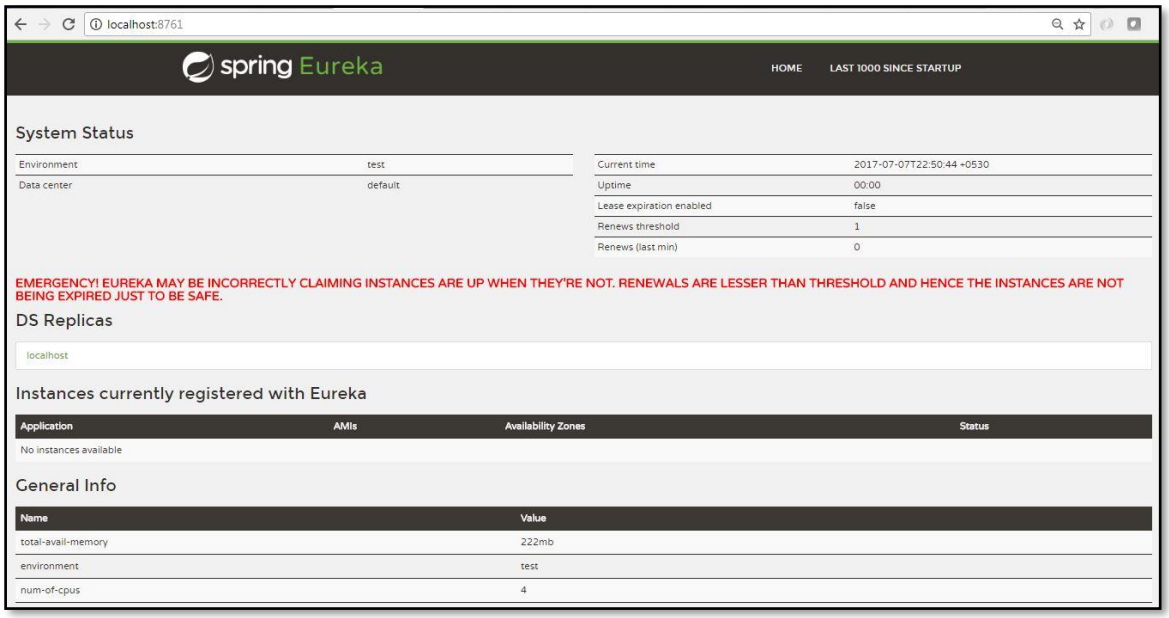
```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

3. Click on GENERATE button or CTRL + Enter to create the project structure.
4. Let's import the DiscoveryServer maven project in STS.
5. Will take some time for the project to be imported and the maven dependencies to be downloaded once you click **Finish**.
6. In the Application class, Add **@EnableEurekaServer** annotation.
7. Ensure the server is running on 8761.

application.properties

```
1
2 server.port=8761
3 eureka.client.register-with-eureka=false
4 eureka.client.fetch-registry=false
5 eureka.instance.prefer-ip-address=true
6 #eureka.instance.hostname=localhost
7 eureka.client.service-url.defaultZone=http://localhost:8761/eureka
8
```

8. Start the application.
9. Verify the Eureka Server: http://localhost:8761/



## Problem Statement 4: Enable Dynamic Registration to Product Microservice

Now we will configure the ProductService to register with Spring Cloud Eureka Server.

**Technology stack:**

- Spring Cloud Eureka Client

**Steps for Spring Cloud Config Client:**

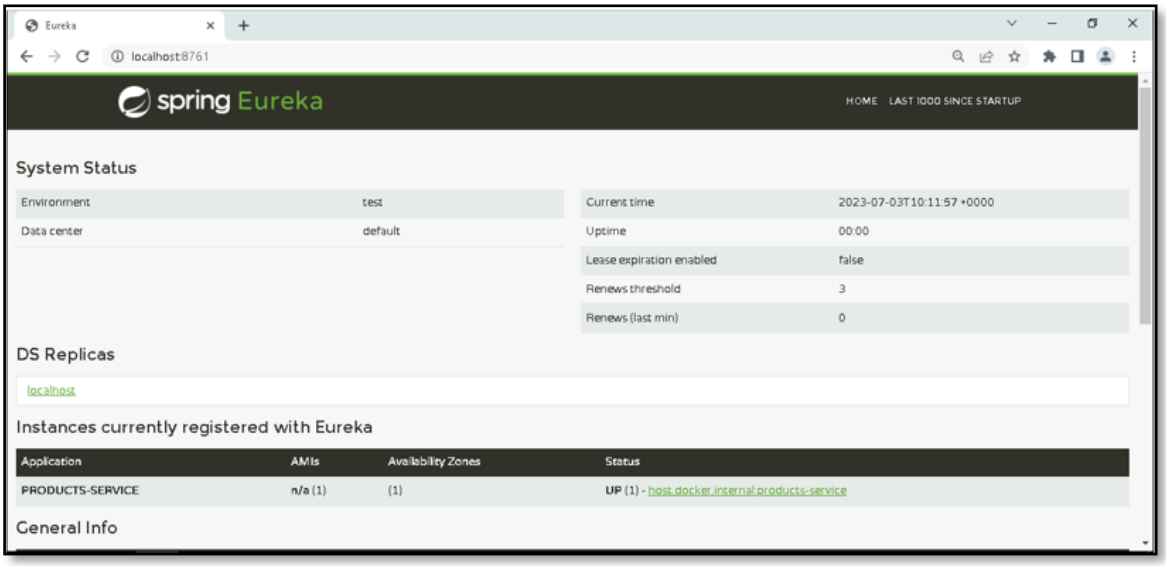
1. Refer the **ProductService** created in the problem statement – 2 and add the **Eureka Client** starter to the application.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

2. Include the **application name** and **eureka.client.serviceUrl.defaultZone** in the application.properties files. For the Product Service application to dynamically register to Discovery Server.

```
application.properties
1
2 eureka.client.service-url.defaultZone=http://localhost:8761/eureka
3 spring.application.name=products-service
4
```

3. In the Application class, Add **@EnableEurekaClient/@EnableDiscoveryClient** annotation.
4. Ensure that the Discovery Server and Product Service is running.
5. Again, verify the Eureka Server: http://localhost:8761/



## Problem Statement 5: Implementing Spring Cloud Gateway in Microservices

In this problem statement, we will use Spring Cloud Gateway to implement API Gateway. The Spring Cloud Gateway is a non-blocking API. A thread is always available to process the incoming request while using non-blocking API. These requests are then handled asynchronously in the background, and the response is returned once completed. When using Spring Cloud Gateway, no incoming request is ever blocked.

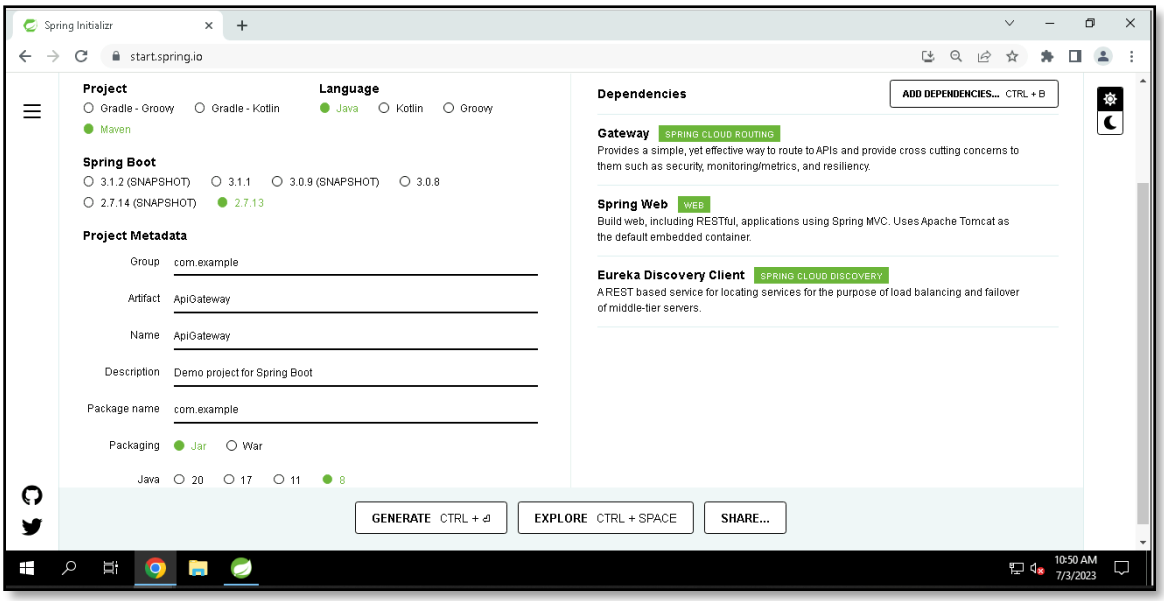
**Technology stack:**

- Spring Cloud Routing - Gateway

**Steps for implementing API Gateway:**

1. Ensure the Discovery Server and Product Service is running.
2. Now let's implement an API gateway that acts as a single-entry point for a collection of microservices.
3. Create a new Project for **ApiGateway** using the **Spring Initalizr** (start.spring.io).

   Refer the below screenshots:

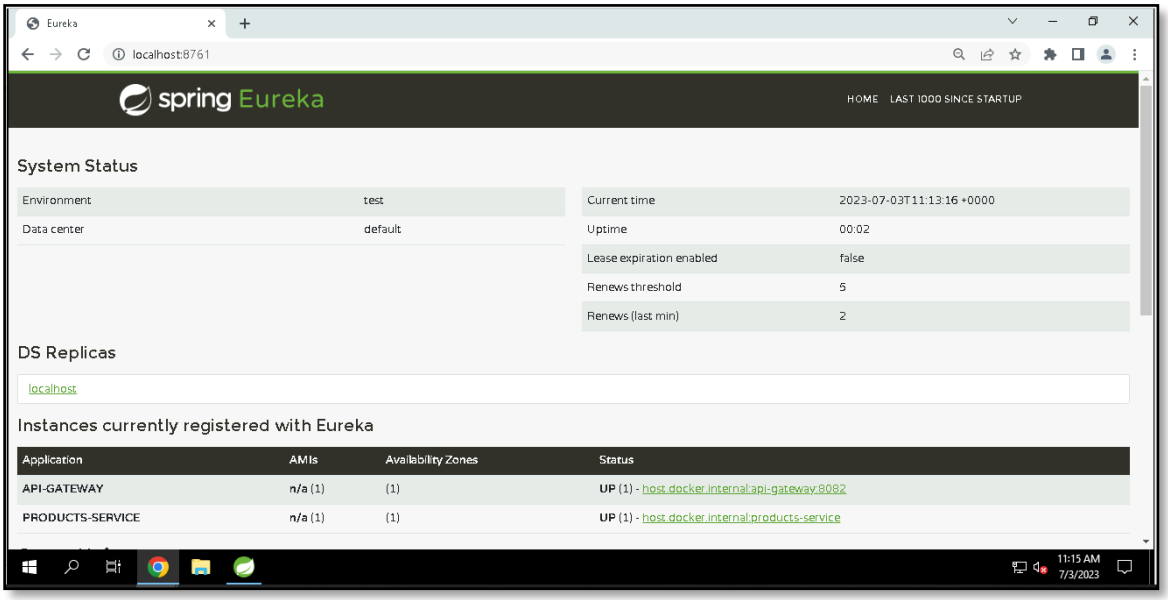



4. Select the **Gateway**, **Spring Web**, and **EurekaDiscoveryClient** dependencies.
5. Click on GENERATE button or CTRL + Enter to create the project structure.
6. Let's import the ApiGateway maven project in STS.
7. Will take some time for the project to be imported and the maven dependencies to be downloaded once you click **Finish**.
8. Add **@EnableEurekaClient/@EnableDiscoveryClient** in the Application class.

9. In application.properties file, enable the automatic mapping of gateway routes and add the application name and eureka client serviceUrl.

```
application.properties
1
2 spring.application.name=api-gateway
3 server.port=8082
4 eureka.client.service-url.defaultZone=http://localhost:8761/eureka
5
6 spring.cloud.gateway.discovery.locator.enabled=true
7 spring.cloud.gateway.discovery.locator.lower-case-service-id=true
8
```

10. Start the ApiGateway.
11. Check the proxy running instances is also registered with the Eureka Server.

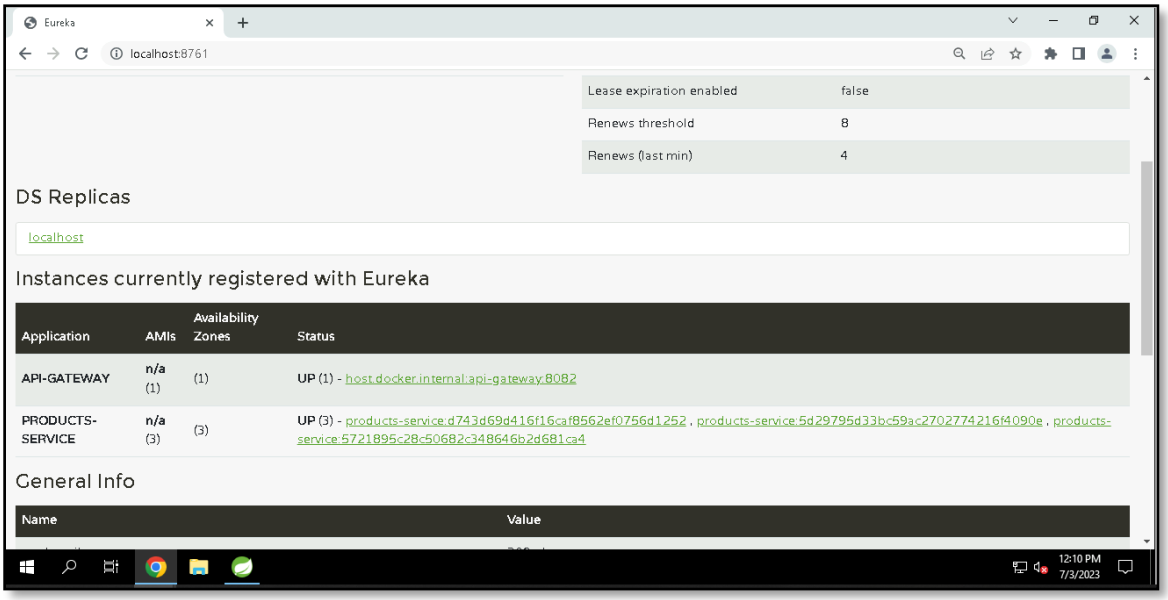



12. Test the Proxy: [http://localhost:8082/products-service/products](http://localhost:8082/products-service/products)


13. Let's add the random port and instance-id property to **ProductService/application.properties** file:

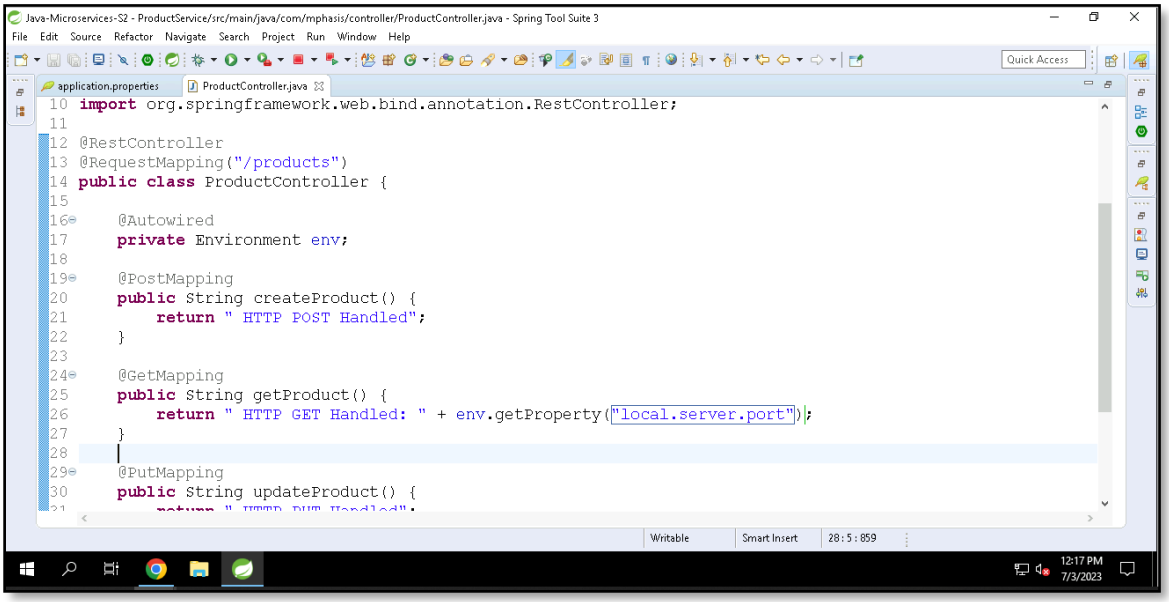
```
application.properties
1
2 server.port=0
3 eureka.client.service-url.defaultZone=http://localhost:8761/eureka
4 spring.application.name=products-service
5
6 eureka.instance.instance-id=${spring.application.name}:${instanceId:${random.value}}
7
```

14. Restart the Discovery Server and execute Product Service three times, you will find 3 instances are running.

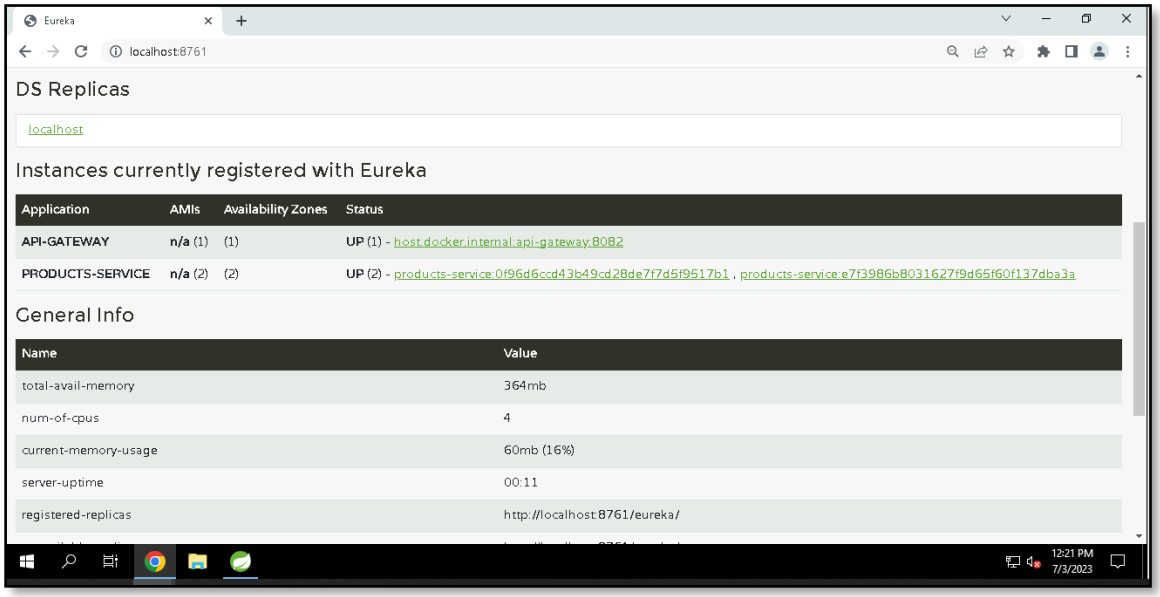



15. Test how the Load Balancing works.

16. Modify the code of GET handler method in ProductController class.



17. Restart the Discovery Server and execute Product Service two times, you will find 2 instances are running.
18. Restart the ApiGateway also.




19. Test the Proxy: http://localhost:8082/products-service/products


20. Refresh again:

## Problem Statement 6: Running Axon Server in Docker Container

Axon Server is a zero-configuration message router and event store. The message router has a clear separation of different message types: **events**, **queries**, and **commands**. The event store is optimized to handle huge volumes of events without performance degradation.

Axon Server is initially built to support distributed Axon Framework microservices. Starting from Axon Framework version 4 Axon Server is the default implementation for the **CommandBus**, **EventBus/EventStore**, and **QueryBus** interfaces.

**Steps for Running Axon Server in Docker Container:**

1. Create a folder docker-data folder on C Drive and create three sub-folders: data, event, config.
2. The "/data" and "/eventdata" directories are created as volumes, and their data will be accessible on your local filesystem somewhere in Docker's temporary storage tree. Alternatively, you can tell docker to use a specific directory, which will allow you to put it at a more convenient location. A third directory, not marked as a volume in the image, is important for our case: If you put an "axonserver.properties" file in "/config", it can override the settings and add new ones.
3. Add the below properties to **axonserver.properties**:

```
server.port=8024
axoniq.axonserver.name=My Axon Server
axoniq.axonserver.hostname=localhost
axoniq.axonserver.devmode.enabled=true
```

4. Run the following command to start Axon Server in a Docker container:

   docker run -d \

   --name axonserver \

   -p 8024:8024 \

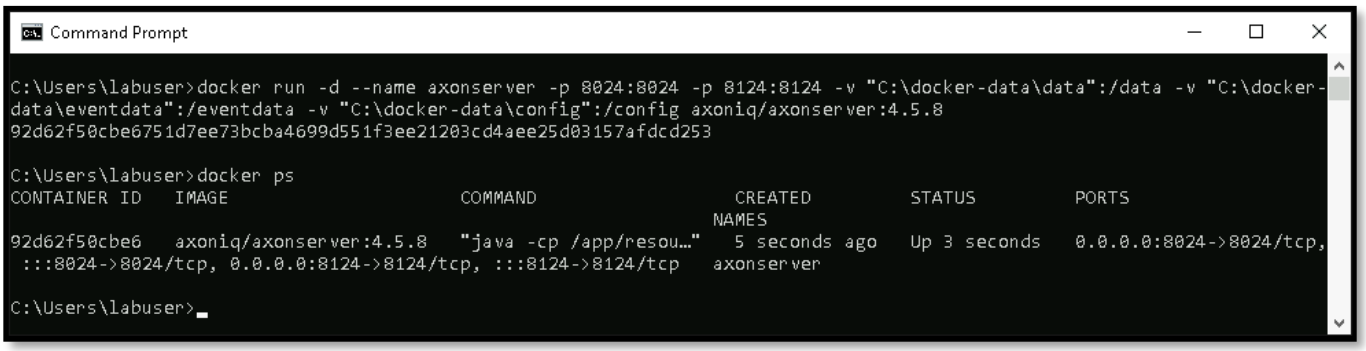
   -p 8124:8124 \

   -v "C:\docker-data\data":/data \

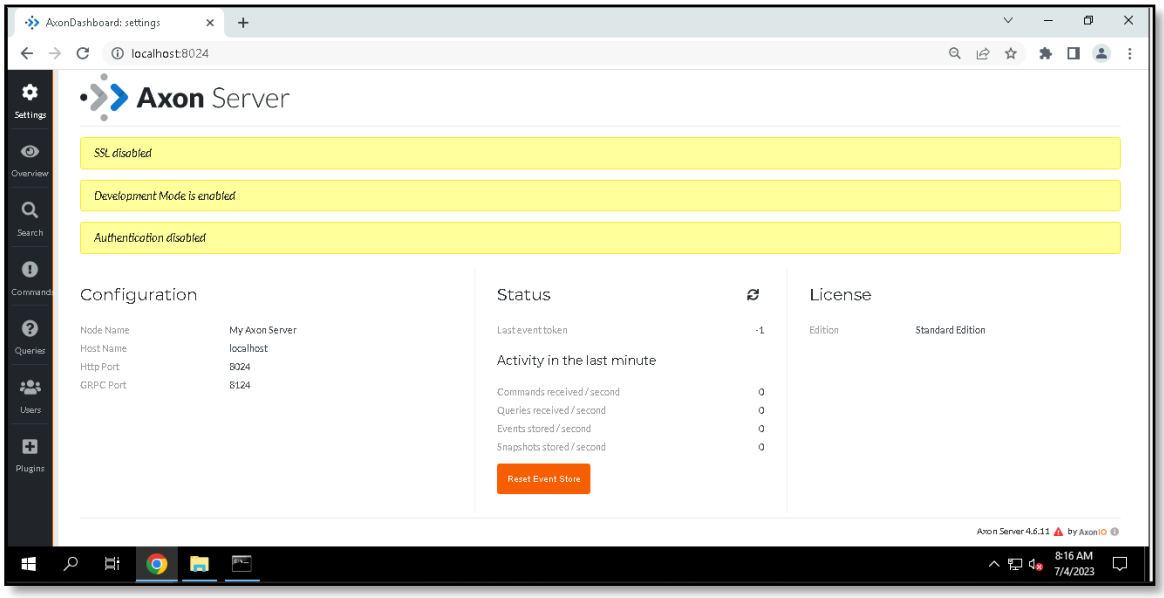
   -v "C:\docker-data\eventdata":/eventdata \

   -v "C:\docker-data\config":/config \

   axoniq/axonserver:4.5.8



Access the Axon Server, via, http://localhost:8024

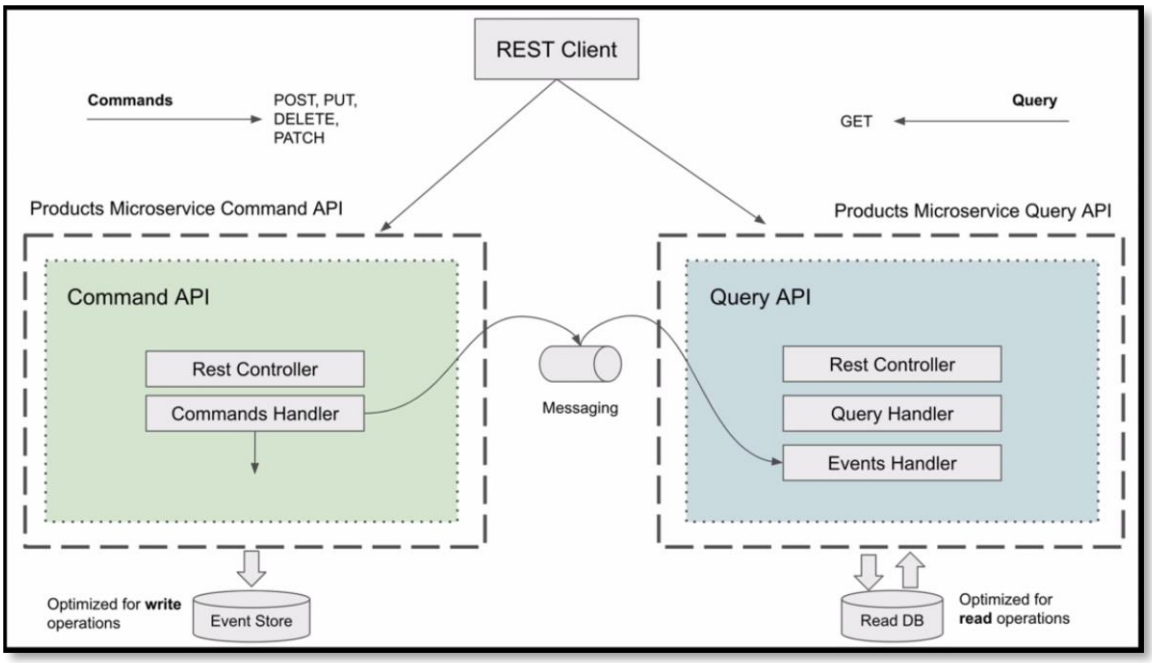


## Problem Statement 7: Implementing the CQRS & Event Sourcing Design Pattern in Product Microservice

Now in the era of distributed system, you run a large-scale ecommerce store. You have a large user base who query your system for products much more than they buy them. In other words, your system has more read requests that write requests. As a result, you'd prefer to handle the high load of read requests separately from the relative low write requests. Also, suppose you need to write complex queries to read data from the same database that you write to, and this might impact its performance. Assume you also want to add additional security while writing data to the database. How do you design your system to cater these specific use cases?

**CQRS (Command Query Responsibility Segregation)** design pattern addresses these concerns. On a high level, you separate your read and write systems and keep them in sync.

Here is a diagram explaining the same:

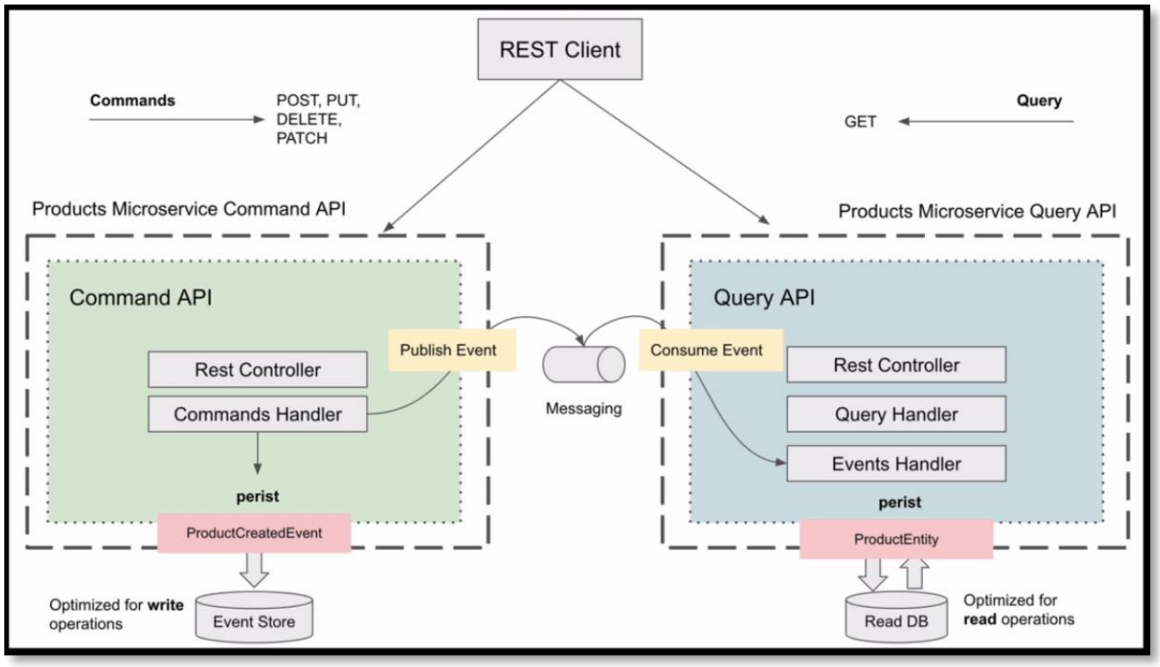



In some scenarios though you would want more than the current state, you might need all the states which the customer entry went through. For such cases, the design pattern **"Event Sourcing"** helps.

When a client application sends a POST request to create a Product, the Command Handler will handle the Command and Product Create Event will be created. And will be persisted into a database which is now going to be called **Event Store**. The ProductCreatedEvent will be published to all the other components who are interested in this event to consume it.

The **Events Handler** in the Query API on the right side, will consume the ProductCreatedEvent and read database will be updated with a new Record.

Here is a diagram explaining the same:




Now the difference between the Event Store on the left side and Read DB on the right side is that the Event Store database will contain the record of every single event that took place for the product but the read database on the right side will only contain 1 single record which is the latest state of the Product Details Entity.

We have an **historical data** that we can use for **audit purposes**, or we can use to reconstruct the state of the Object at any given time.

**Technology stack:**

- Spring Web
- Spring Data JPA
- H2 Database
- Spring Cloud Eureka Client
- Lombok
- Axon Spring Boot Starter
- Google Guava
- Spring Boot Starter Validation

**Steps for implementing CQRS and Event Sourcing using Axon Server:**

1. Refer the **ProductService** updated in the problem statement – 4.

2. Add the Spring Web, Spring Data JPA, H2 Database, Spring Cloud Eureka Client, Lombok, Axon Spring Boot Starter, Google Guava, and Spring Boot Starter Validation dependencies in pom.xml.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <scope>provided</scope>
</dependency>

<dependency>
    <groupId>org.axonframework</groupId>
    <artifactId>axon-spring-boot-starter</artifactId>
    <version>4.5.8</version>
</dependency>

<dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>30.1-jre</version>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

3. Will take some time for the project to be imported and the maven dependencies to be downloaded once you click **Finish**.
4. Will have a separate package for Command API (com.mphasis.command) and Query API (com.mphasis.query).
5. Update the ProductController class.
6. The method that accepts the HTTP Post request, should accept the CreateProductRestModel payload as a request body.

7. The CreateProductRestModel annotated with @Data and should have the following fields:

   private String title;
   private BigDecimal price;
   private Integer quantity;

8. This controller class should use the **Axon's *CommandGateway*** and publish the CreateProductCommand.
9. The CreateProductCommand annotated with @Data, @Builder and should have the following fields:

   private final String productId;
   private final String title;
   private final BigDecimal price;
   private final Integer quantity;

   Where:
   productId - is a randomly generated value. For example, UUID.randomUUID().toString() and annotated with **@TargetAggregateIdentifier**.
   Should return the productId as String. If the exception raised by published code, handle and return the Localized Message as String.

10. Create a new class called ProductAggregate and make it handle the CreateProductCommand using **@CommandHandler** and publish the ProductCreatedEvent.
11. The ProductCreatedEvent class annotated with @Data and should have the following fields:

    private String productId;
    private String title;
    private BigDecimal price;
    private Integer quantity;

12. Apply the below validation to price and title in the Command Handler.

```
@CommandHandler
public ProductAggregate(CreateProductCommand createProductCommand) {
    // Validate Create Product Command

    if (createProductCommand.getPrice().compareTo(BigDecimal.ZERO) <= 0) {
        throw new IllegalArgumentException("Price cannot be less or equal than zero");
    }

    if(createProductCommand.getTitle() == null
            || createProductCommand.getTitle().isEmpty()) {
        throw new IllegalArgumentException("Title cannot be empty");
    }
}
```

13. Use **AggregateLifecycle.apply(Object payload)** which apply a **DomainEventMessage** with given payload without metadata. Applying events means they are immediately applied (published) to the aggregate and scheduled for publication to other event handlers.

```
ProductCreatedEvent productCreatedEvent = new ProductCreatedEvent();

BeanUtils.copyProperties(createProductCommand, productCreatedEvent);

AggregateLifecycle.apply(productCreatedEvent);
```

14. The ProductAggregate class should also have an **@EventSourcingHandler** method that sets values for all fields in the ProductAggregate.

**CQRS Persisting Event in the Product database:**

15. Add the below DB properties in application.properties:

```
application.properties
server.port=0
eureka.client.service-url.defaultZone=http://localhost:8761/eureka
spring.application.name=products-service
eureka.instance.instance-id=${spring.application.name}:${instanceId:${random.value}}

spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

spring.h2.console.settings.web-allow-others=true

spring.datasource.url=jdbc:h2:mem:mphasisdb
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password

#Accessing the H2 Console
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
```

16. Create a new @Component class called ProductEventsHandler inside com.mphasis.query package.
17. Create a new JPA Repository called ProductRepository inside com.mphasis.core.data package and inject it into ProductEventsHandler using constructor-based dependency injection.
18. Add two find methods in ProductRepository interface:

```
ProductEntity findByProductId(String productId);
ProductEntity findByProductIdOrTitle(String productId, String title);
```

19. The ProductEventsHandler class should have one **@EventHandler** method that handles the ProductCreatedEvent and persists product details into the **"read"** database.
20. To persist product details into the database, create a new JPA Entity class called ProductEntity inside com.mphasis.core.data package. Annotate the ProductEntity class with:

    @Data
    @Entity
    @Table(name = "products")

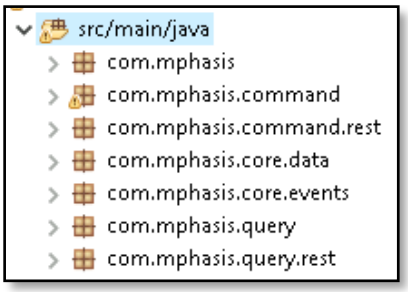
    and make the ProductEntity class have the following fields:

    @Id
    @Column(unique = true)
    private String productId;
    @Column(unique = true)
    private String title;
    private BigDecimal price;
    private Integer quantity;

**CQRS, Querying Data:**

Further, we will have one Controller class for Command API and one Controller class for Query API which will help you to split the microservices if required tomorrow.

21. Refactor Command API REST Controller:
    - Rename ProductController to ProductCommandController for better visualization.
    - Rename the Package com.mphasis.controller to com.mphasis.command.rest.
    - Rename the Package com.mphasis.core to com.mphasis.core.event.
    - So, total we will have 3 main package – command, core, and query.




22. Create the ProductsQueryController class inside com.mphasis.query.rest package.
23. The method that accepts the HTTP Get request, should have List<ProductRestModel> as a response body.
24. The ProductRestModel annotated with @Data and should have the following fields:

    private String productId;
    private String title;
    private BigDecimal price;
    private Integer quantity;

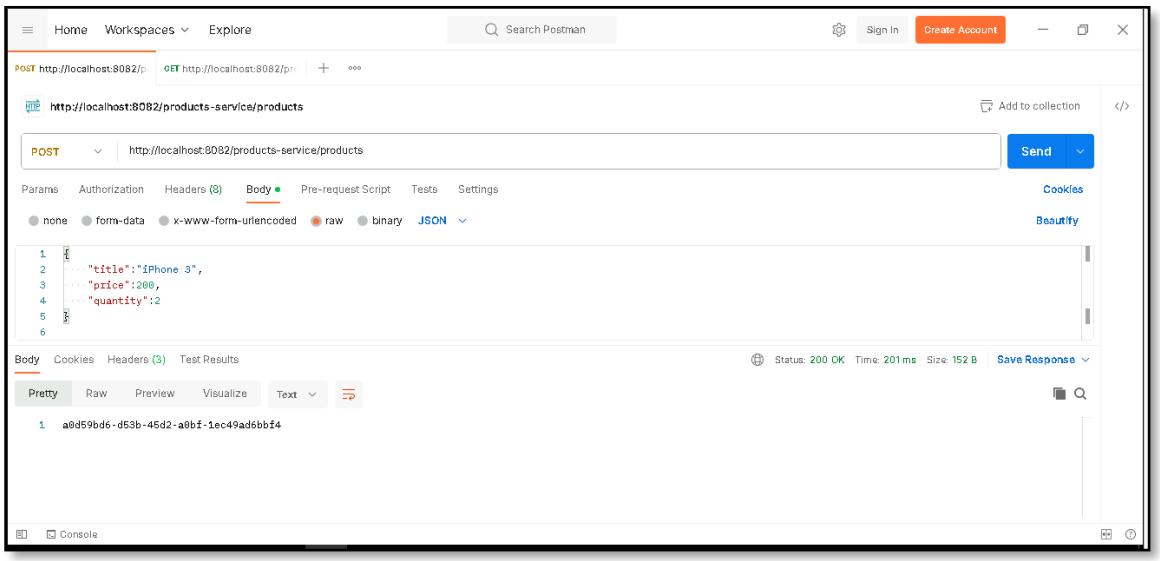
25. This controller class should use the **Axon's *QueryGateway*** to dispatch an instance of FindProductQuery. As we use the gateway's query() method to issue a point-to-point query. Because we are specifying ResponseTypes.multipleInstancesOf(ProductRestModel.class), Axon knows we only want to talk to query handlers whose return type is a collection of ProductRestModel objects.
26. Create a new @Component class called ProductsQueryHandler inside com.mphasis.query package.
27. Refer the JPA Repository called ProductRepository and inject it into ProductsQueryHandler using constructor-based dependency injection.
28. The ProductsQueryHandler class should have one **@QueryHandler** method that handles the FindProductsQuery and fetch all the product details from the **"read"** database.

**Run and make it work:**

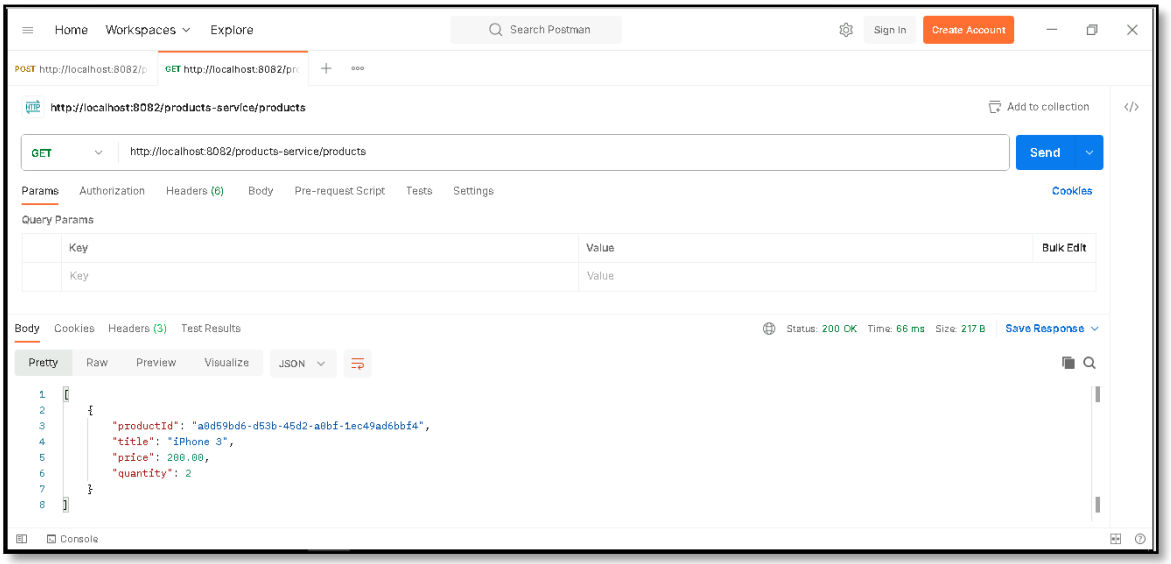
29. Let's comment below methods as we don't require them now.

```
ProductCommandController.java
44          }
45          return returnValue;
46      }
47  /*
48      @GetMapping
49      public String getProduct() {
50          return " HTTP GET Handled: " + env.getProperty("local.server.port");
51      }
52
53      @PutMapping
54      public String updateProduct() {
55          return " HTTP PUT Handled";
56      }
57
58      @DeleteMapping
59      public String deleteProduct() {
60          return " HTTP DELETE Handled";
61      }
62      */
63  }
64
```

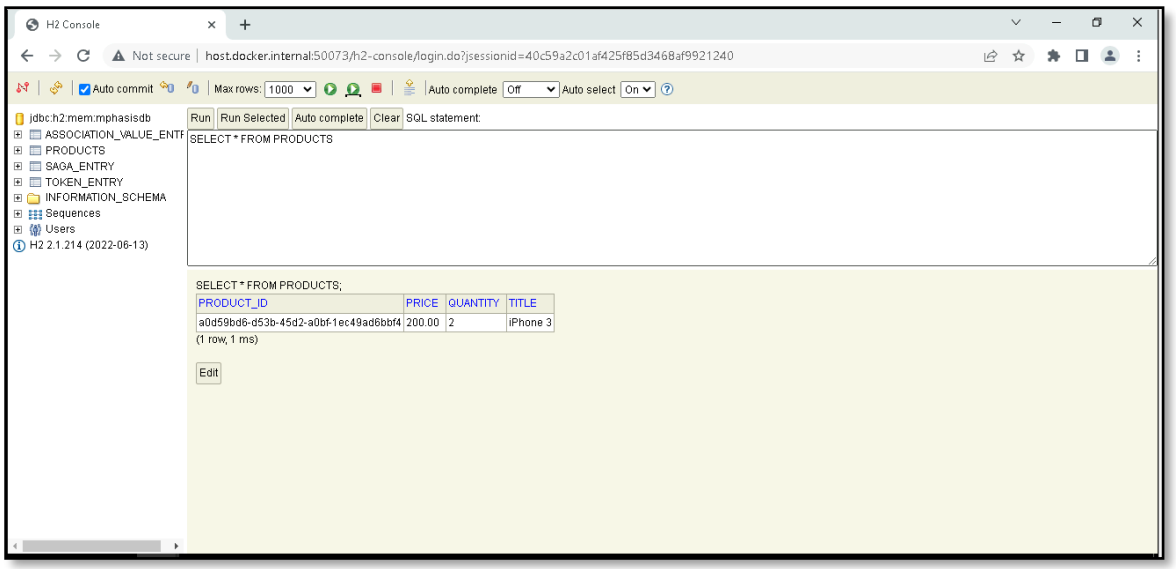
30. Run the Axon Server using Docker command.
31. Start the Discovery Server (Eureka Server), Product Service, and ApiGateway.
32. Send a POST request to Create Product.



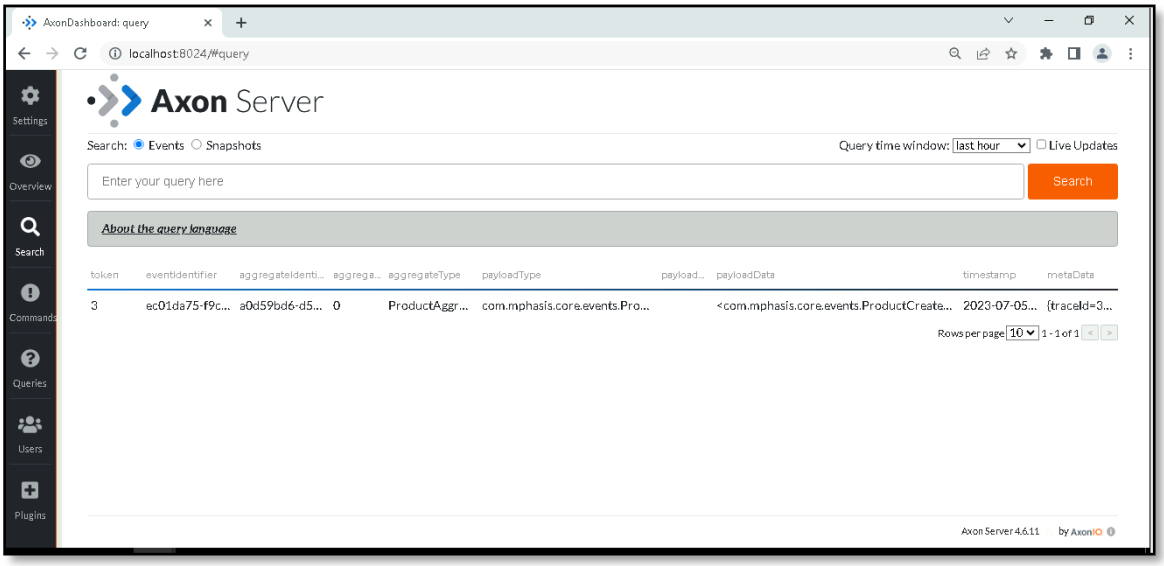
33. Send a GET request to Query the Products.



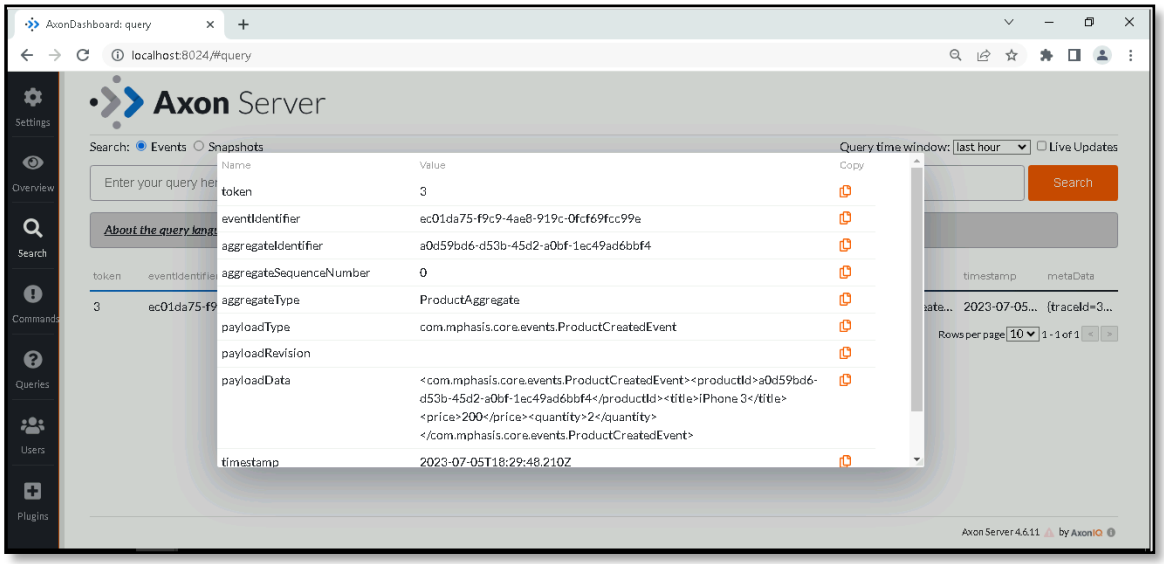

34. Using the /h2-console connect to the Products database and make sure that the product details are stored there as well.



35. Check the Event Store in the Axon server and make sure that the ProductCreatedEvent gets persisted,




36. Let's review the individual ProductCreatedEvent description in the Axon Server.



## Problem Statement 8: Validate Request Body, Bean Validation in Product Microservice

When it comes to validating user input, Spring Boot provides strong support for this **common**, yet critical, task straight out of the box.

Although Spring Boot supports seamless integration with custom validators, the **de-facto standard for performing validation is Hibernate Validator**, the Bean Validation framework's reference implementation.

In this problem statement, we'll look at how to **validate domain objects in Spring Boot**.

**Technology stack:**

- Spring Boot Starter Validation

**Steps for Spring Boot Starter Validation:**

1. Refer the **ProductService** updated in the problem statement – 7.
2. Ensure the Spring Boot Starter Validation is added to ProductService/pom.xml file.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

3. Add the below error properties to application.properties:

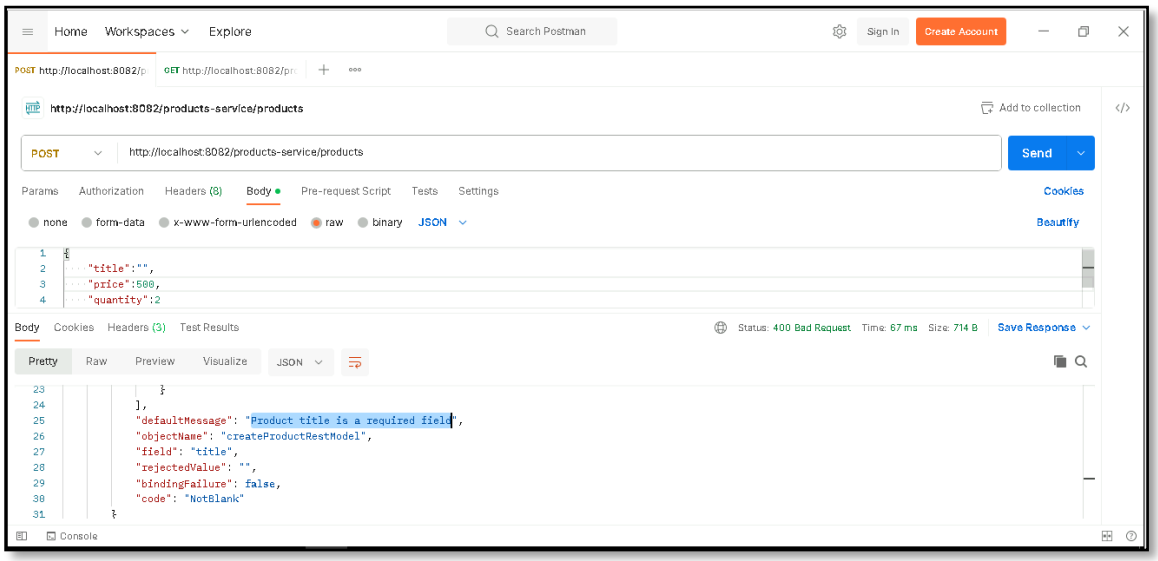
application.properties

```
server.port=0
eureka.client.service-url.defaultZone=http://localhost:8761/eureka
spring.application.name=products-service
eureka.instance.instance-id=${spring.application.name}:${instanceId:${random.value}}

spring.datasource.url=jdbc:h2:mem:mphasisdb
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

#Accessing the H2 Console
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
spring.h2.console.settings.web-allow-others=true

server.error.include-message=always
server.error.include-binding-errors=always

```

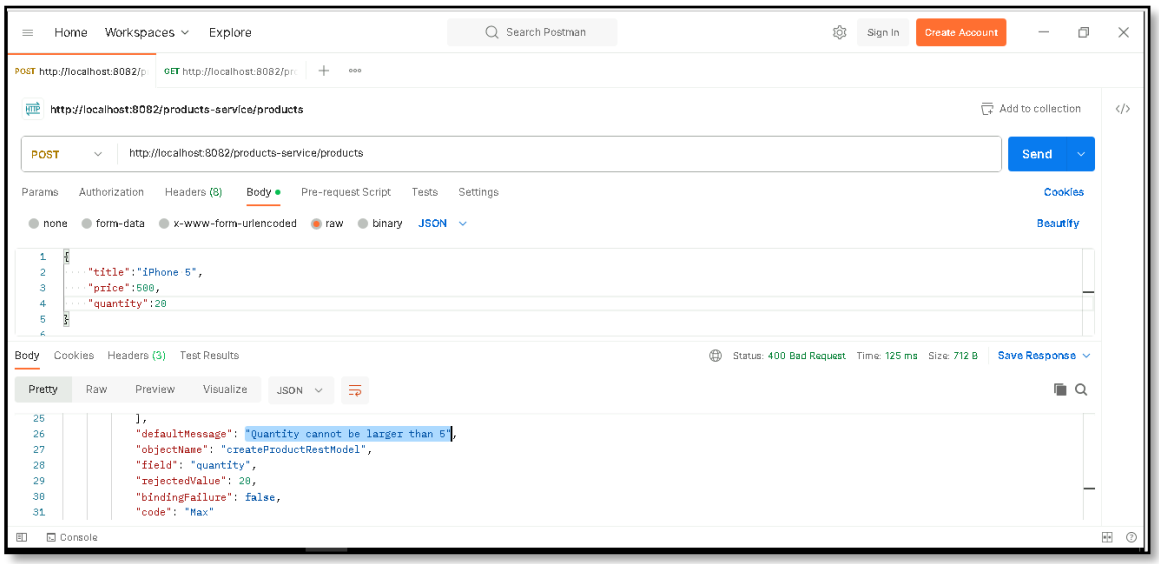
4. Apply the common validation annotation on the **CreateProductRestModel** class.
    - Use @NotBlank to say that a title field must not be the empty string.
    - Use @Min to say that the price is a numerical field is only valid when its value is above 1.
    - Use @Min and @Max to say that the quantity is a numerical field is only valid when its value is above 1 and below 5.
5. Adding the **@Valid** annotation will trigger validation of the request body on ProductCommandController handler methods.

**Run and make it work:**

6. Run the Axon Server using Docker command.
7. Start the Discovery Server (Eureka Server), Product Service, and ApiGateway.
8. Try sending a POST request with title as blank.




9. Try sending a POST request with Quantity greater than 5.



## Problem Statement 9: Apply Command Validation using Message Dispatch Interceptor

Message Dispatch Interceptor is invoked when a message is dispatch from Command Gateway to Command Bus. We can create a Message Dispatch Interceptor to intercept immediately where they are dispatched on a Command Bus. You can use Message Dispatch Interceptor to perform additional logging, command validation, change a command message by adding META-DATA, and block the command by throwing an Exception.

**Steps for implementing MessageDispatchInterceptor:**

1. Refer the **ProductService** updated in the problem statement – 8.
2. Create a new com.mphasis.command.interceptor package.
3. Create a new class CreateProductCommandInterceptor which implements **MessageDispatchInterceptor** and override the **handle** method.
4. Let apply similar validation on the CreateProductCommand object used in ProductAggregate class previously.
    - Price cannot be less than or equal to zero.
    - Title cannot be empty.
5. Register the CreateProductCommandInterceptor in the Application class.

```
@Autowired
public void registerCreateProductCommandInterceptor(
        ApplicationContext context, CommandBus commandBus) {

    commandBus.registerDispatchInterceptor(context.getBean(CreateProductCommandInterceptor.class));
}
```

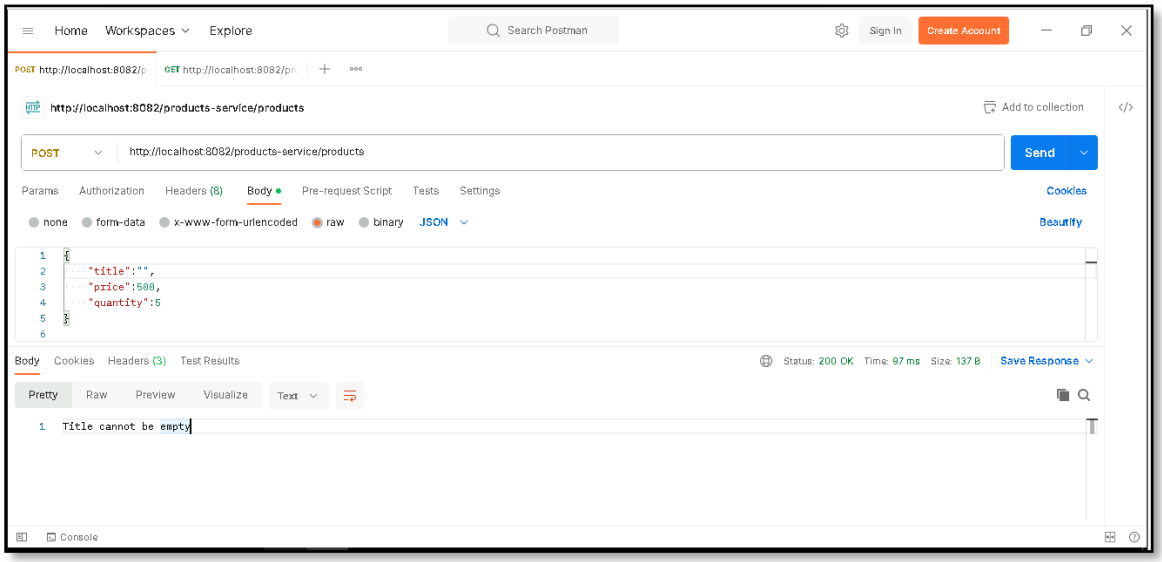
**Run and make it work:**

6. Let's comment the @NotBlank annotation for demonstration purpose.

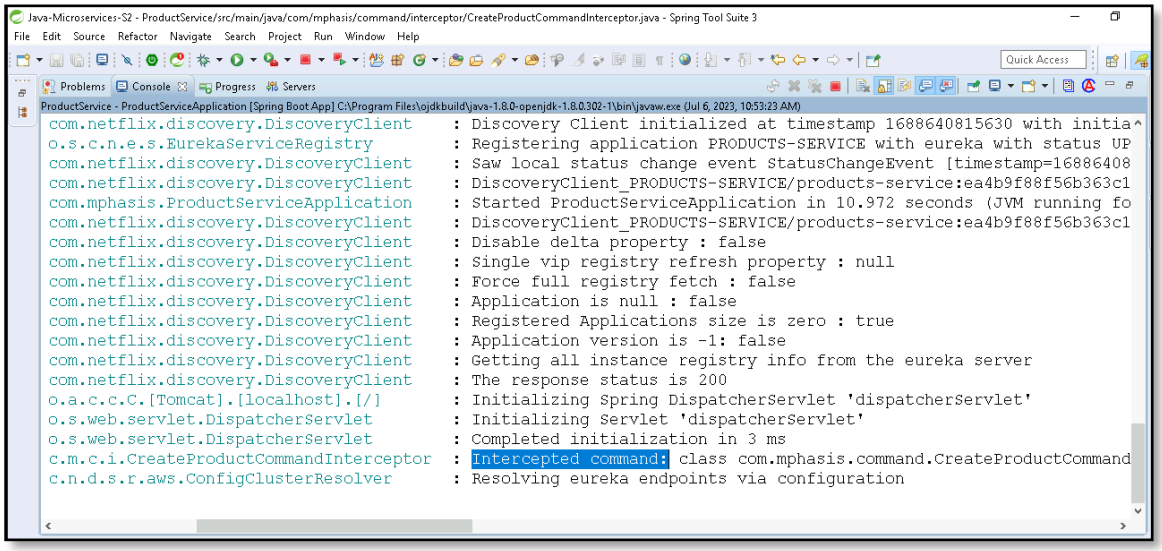
```
@Data
public class CreateProductRestModel {

    //@NotBlank(message = "Product title is a required field")
    private String title;

    @Min(value=1, message = "Price cannot be lower than 1")
    private BigDecimal price;

    @Min(value=1, message = "Quantity cannot be lower than 1")
    @Max(value=5, message = "Quantity cannot be larger than 5")
    private Integer quantity;
}
```

7. Run the Axon Server using Docker command.
8. Start the Discovery Server (Eureka Server), Product Service, and ApiGateway.

9. Send a POST request to Create Product with title as Empty.




10. Let's review the logs in the Console.




11. Let's uncomment the @NotBlank annotation.

## Problem Statement 10: Set Based Consistency Validation is CQRS and Event Sourcing Application

A very common question asked by developers:

- How to check if record already exists in a database table?
- How to check if Product already exists?
- As the communication between Command API and Query API is via Messaging Architecture.
- How can Command API quickly check the record already exists for the Persistent Event in the Event Store?
- How can I validate a uniqueness constraint throughout the entire application using **CQRS** and **Event Sourcing**, while dealing with eventual consistency?

This issue is not related to Axon Server, but rather to the CQRS design pattern.

In this problem statement, we will discuss a solution that you can use if you are developing your application with the Axon Framework.

**Steps for implementing Set Based Consistency Validation:**

1. Refer the **ProductService** updated in the problem statement – 9.
2. Create a new @Component class called ProductLookupEventsHandler inside com.mphasis.command package and should be annotated with **@ProcessingGroup("product-group")**.
3. Create a new JPA Repository called ProductLookupRepository inside com.mphasis.core.data package and inject it into ProductLookupEventsHandler using constructor-based dependency injection.
4. Add a find method in ProductLookupRepository interface:

```
ProductLookupEntity findByProductIdOrTitle(String productId, String title);
```

5. The ProductLookupEventsHandler class should have one **@EventHandler** method that handles the ProductCreatedEvent and persists product lookup details into the **"read"** database.
6. To persist lookup details into the database, create a new JPA Entity class called ProductLookupEntity in com.mphasis.core.data package. Annotate the ProductLookupEntity class with:

   @Entity
   @Table(name = "productlookup")
   @Data
   @NoArgsConstructor
   @AllArgsConstructor

   and make the ProductLookupEntity class have the following fields:

   @Id
   public String productId;
   @Column(unique = true)
   private String title;
7. How do we query this lookup table before the command handler processes the command?
8. We use Message Dispatch Interceptor, which we have already created. The command will be intercepted by the Message Dispatch Interceptor before it is handled by the command handler method. It will use the JPA repository to query the lookup table and if the record already exists, the command will be blocked.

9. Update the CreateProductCommandInterceptor class and inject the ProductLookupRepository using constructor-based dependency injection.
10. Let's remove the if conditions of Price & Title from the CreateProductCommandInterceptor class.
11. In the handle method, check whether the ProductLookupEntity exists using the productId and title.
12. If exists, throw IllegalStateException with the below message:

```
throw new IllegalStateException(
        String.format("Product with productId %s or title %s already exist",
                createProductCommand.getProductId(),
                createProductCommand.getTitle())
        );
```

13. Verify the CreateProductCommandInterceptor is registered in the Application class.
14. Add the **processing-group** property inside the application.properties file:

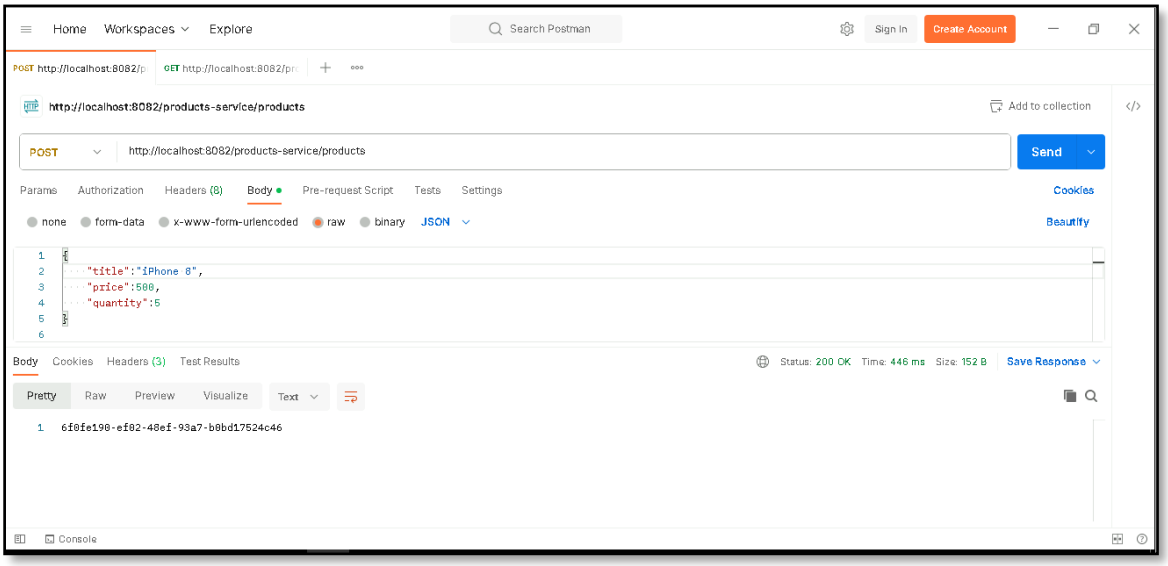
application.properties

```
server.port=0
eureka.client.service-url.defaultZone=http://localhost:8761/eureka
spring.application.name=products-service
eureka.instance.instance-id=${spring.application.name}:${instanceId:${random.value}}

spring.datasource.url=jdbc:h2:mem:mphasisdb
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

#Accessing the H2 Console
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
spring.h2.console.settings.web-allow-others=true

server.error.include-message=always
server.error.include-binding-errors=always

axon.eventhandling.processors.product-group.mode=subscribing
```

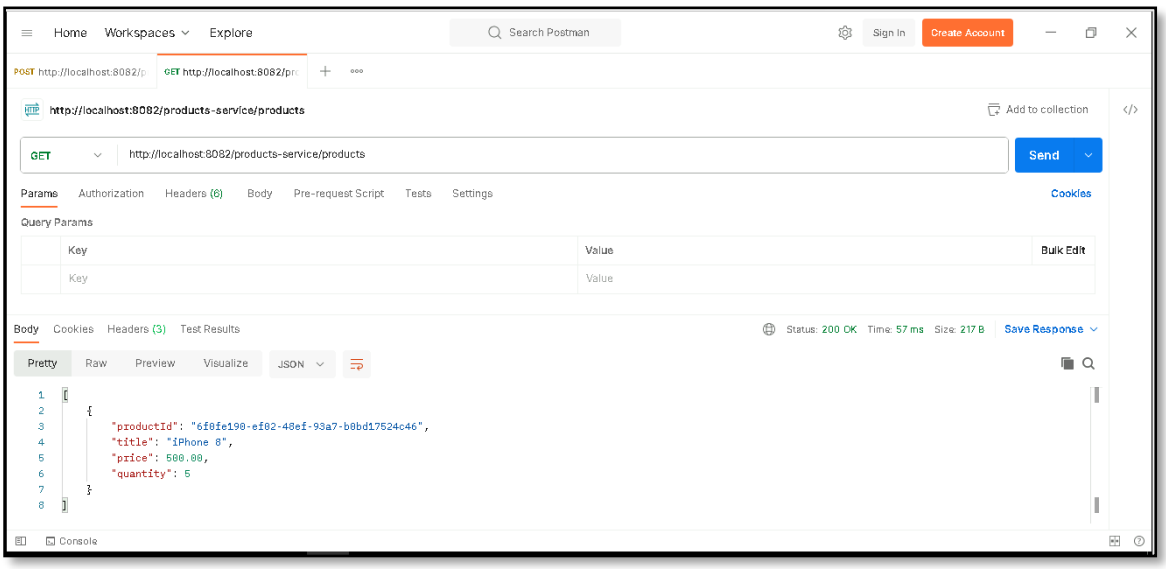
**Run and make it work:**

15. Run the Axon Server using Docker command.
16. Start the Discovery Server (Eureka Server), Product Service, and ApiGateway.

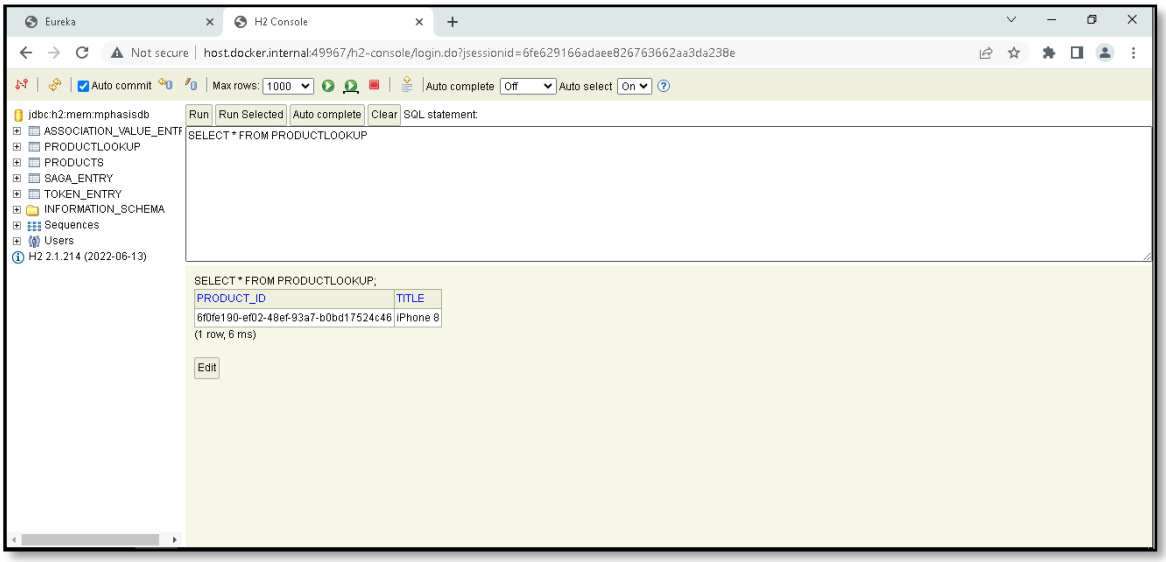
17. Send a POST request to Create Product.



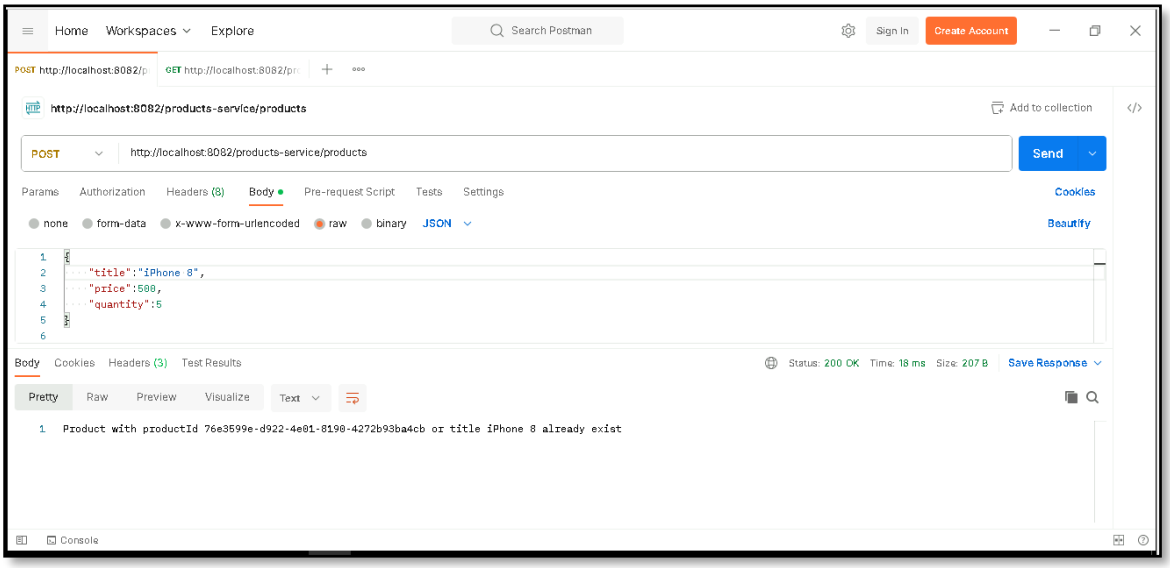

18. Send a GET request to Query the Products.



19. Let's review the ProductLookup table data.




20. Send a POST request to Create Product with Same JSON.



## Problem Statement 11: Handling Errors and Rollback Transaction with Axon

In this problem statement, we will discuss how to handle an error in the event handling method. How to rollback changes made in various event handler methods.

This is very helpful when you have multiple error handler methods, and you want to undo changes made in all error handlers that are in the **same processing group**.

**Steps for implementing Handling Errors and Rollback Transaction:**

1. Refer the **ProductService** updated in the problem statement – 10.
2. Create a centralized ProductServiceErrorHandler class in the com.mphasis.core.errorhandling package and should be annotated with **@ControllerAdvice** annotation.
3. In this case, we will use **@ExceptionHandler** to handle the IllegalStateException and other Exceptions, which will return the ResponseEntity set with a ErrorMessage object (current date and message) and the status code INTERNAL_SERVER_ERROR.
4. Create a ErrorMessage class in the com.mphasis.core.errorhandling package and annotated with:

   @Data
   @AllArgsConstructor

   And have the following fields:

   private final java.util.Date timestamp;
   private final String message;
5. Comment the try catch block in the ProductCommandController class.

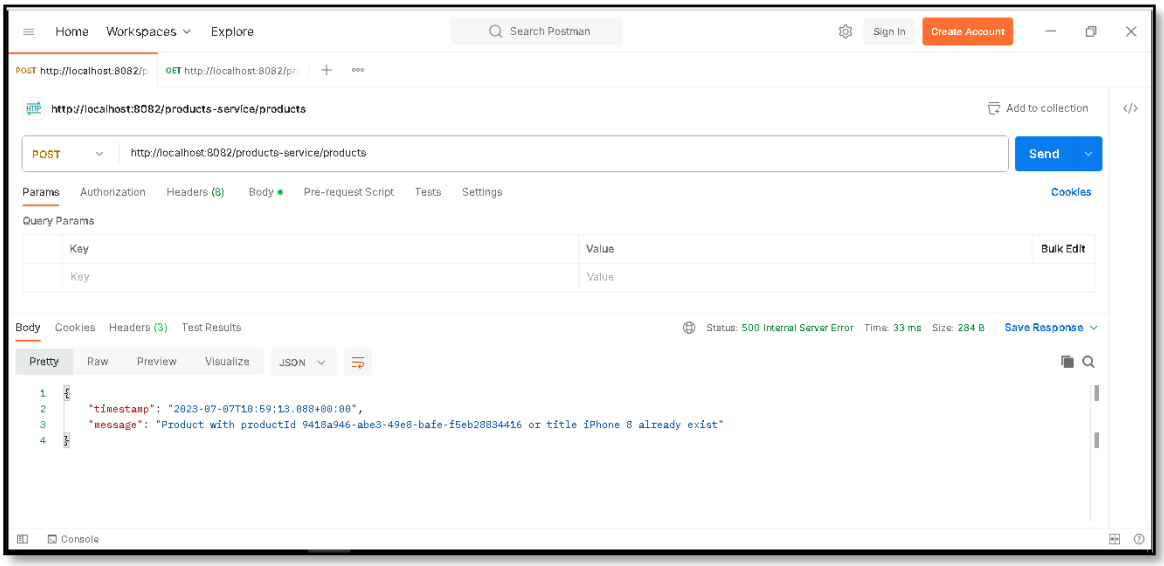
ProductCommandController.java

```
25
26      @PostMapping
27      public String createProduct(@Valid @RequestBody CreateProductRestModel createProductRestModel) {
28
29          CreateProductCommand createProductCommand = CreateProductCommand.builder()
30          .price(createProductRestModel.getPrice())
31          .quantity(createProductRestModel.getQuantity())
32          .title(createProductRestModel.getTitle())
33          .productId(UUID.randomUUID().toString())
34          .build();
35
36          String returnValue;
37
38          returnValue = commandGateway.sendAndWait(createProductCommand);
39
40  //      try {
41  //          returnValue = commandGateway.sendAndWait(createProductCommand);
42  //      }catch (Exception ex) {
43  //          returnValue = ex.getLocalizedMessage();
44  //      }
45          return returnValue;
46      }
47
```

**Run and make it work:**

6. Run the Axon Server using Docker command.
7. Start the Discovery Server (Eureka Server), Product Service, and ApiGateway.

8. Send a POST request to Create Product – twice.

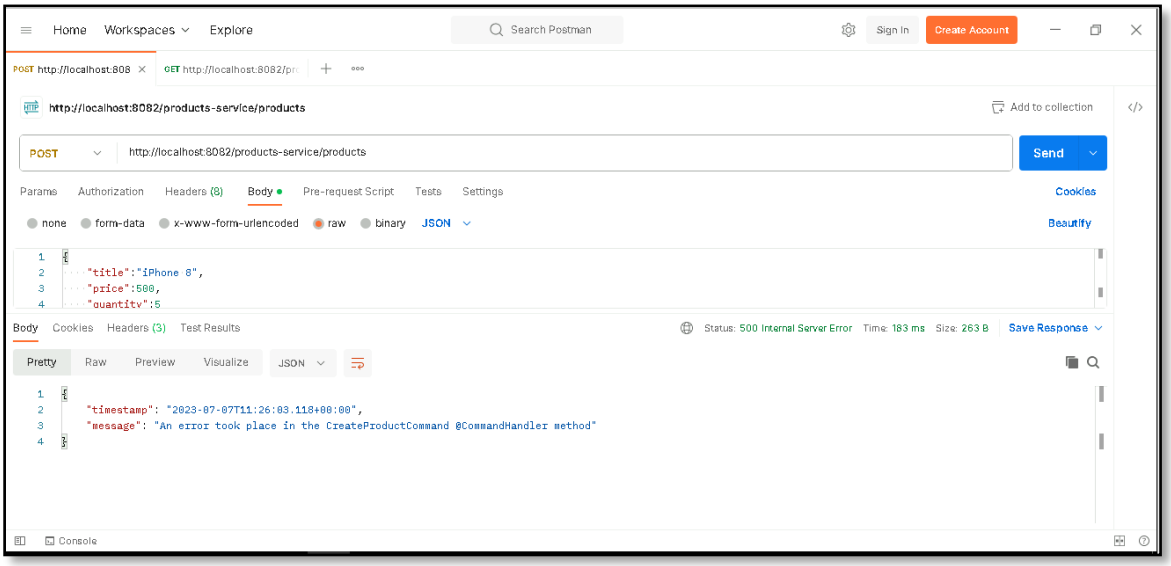



9. Further going we will handle an Exception that is thrown by Command Handler method in the Aggregate class.

```
@CommandHandler
public ProductAggregate(CreateProductCommand createProductCommand) throws Exception {
    // Validate Create Product Command

    if (createProductCommand.getPrice().compareTo(BigDecimal.ZERO) <= 0) {
        throw new IllegalArgumentException("Price cannot be less or equal than zero");
    }

    if (createProductCommand.getTitle() == null || createProductCommand.getTitle().isEmpty()) {
        throw new IllegalArgumentException("Title cannot be empty");
    }

    ProductCreatedEvent productCreatedEvent = new ProductCreatedEvent();
    BeanUtils.copyProperties(createProductCommand, productCreatedEvent);

    AggregateLifecycle.apply(productCreatedEvent);

    if(true)
        throw new Exception("An error took place in the CreateProductCommand @CommandHandler method");
}
```

10. When an Error is thrown from the command handler/query handler method then the Axon Framework wrap this Error into **CommandExecutionException/QueryExecutionException**.
11. In the ProductServiceErrorHandler, let's add an **@ExceptionHandler** to handle the CommandExecutionException, which will return the ResponseEntity set with a ErrorMessage object (current date and message) and the status code INTERNAL_SERVER_ERROR.
12. Restart the Discovery Server (Eureka Server), Product Service, and ApiGateway.

13. Send a POST request to Create Product.



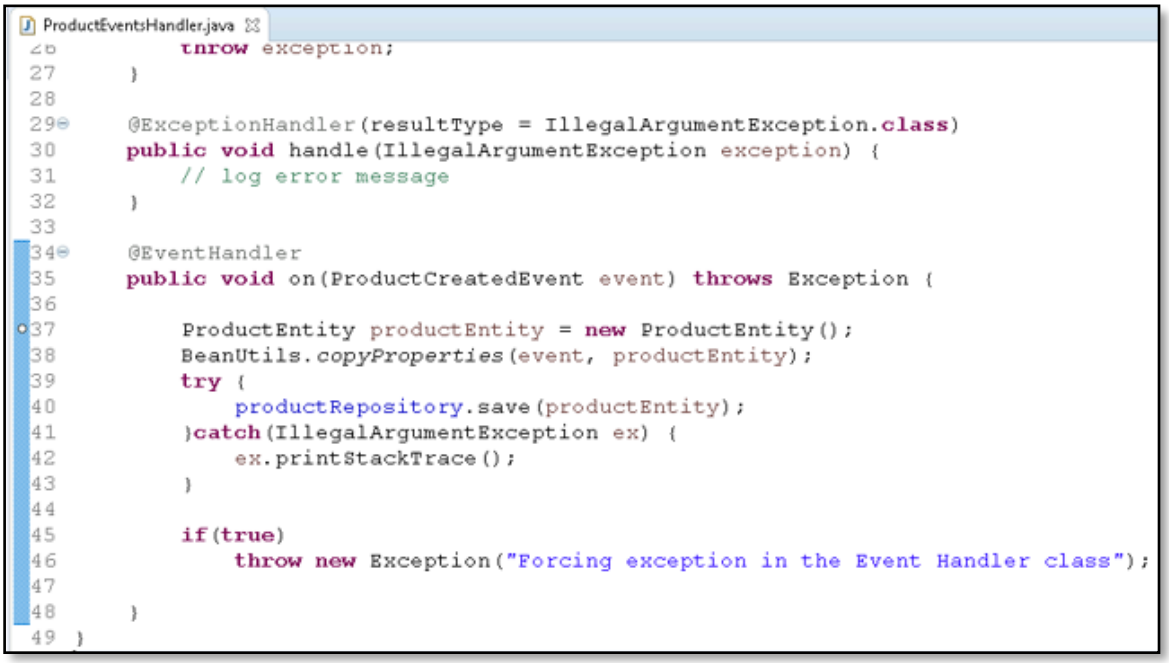

14. Further going the exception can occur in the Event Handler method that handle the ProductCreatedEvent in the ProductEventsHandler class.
15. There are different ways to handle the exception either by using try-catch block or use @ExceptionHandler annotation.
16. In this case, we will use **@ExceptionHandler** to handle the IllegalStateException and other Exceptions in ProductEventsHandler class, rethrow the exception.
17. If you look to roll back the transaction and do the database changes made by EventHandler method, you need to either handle the Exception in your EventHandler method or handle it and rethrow it again. So that it can propagate up the flow.
18. This Exception can be handled by another ExceptionHandler and be propagated further up the flow which will eventually rollback the entire transaction and none of the changes to the database or Event Store will be made.
19. This is possible if your **processing group** is configured to use **subscribing event processors**.
20. Let create a class ProductsServiceEventsErrorHandler which implements **ListenerInvocationErrorHandler** interface and overrides the **onError** method. Here we will rethrow the exception.
21. Register the ProductsServiceEventsErrorHandler in the Application class.

```
@Autowired
public void configure(EventProcessingConfigurer configurer) {

    configurer.registerListenerInvocationErrorHandler("product-group",
            config -> new ProductsServiceEventsErrorHandler());
}
```

22. Cut the below two lines from the ProductAggregate:

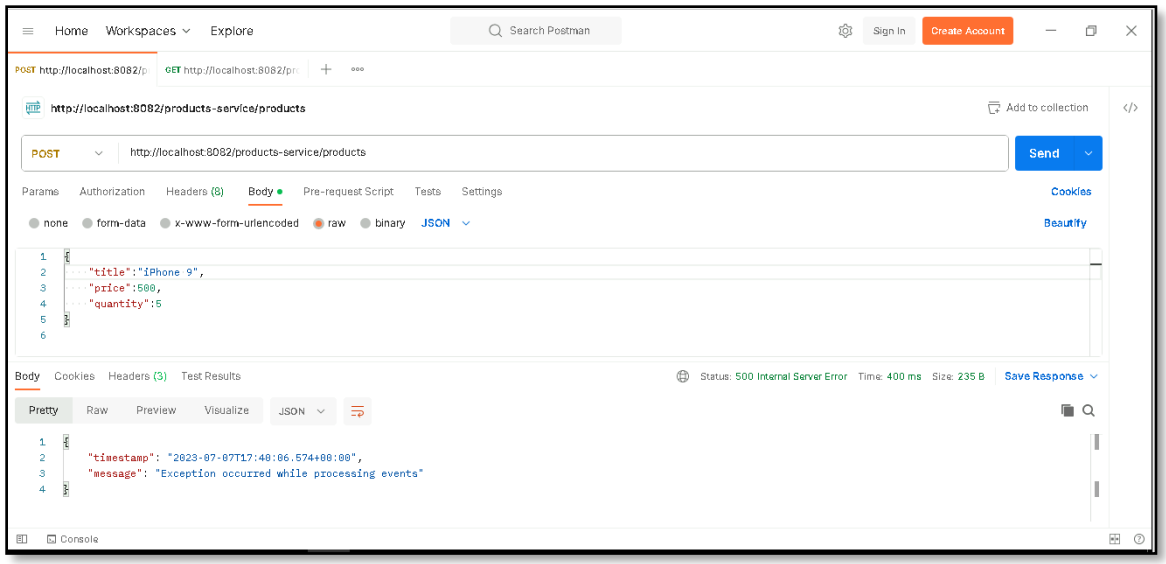
```
        if(true)
            throw new Exception("An error took place in the CreateProductCommand @CommandHandler method");
```

23. Paste it in ProductEventsHandler and modify the message:

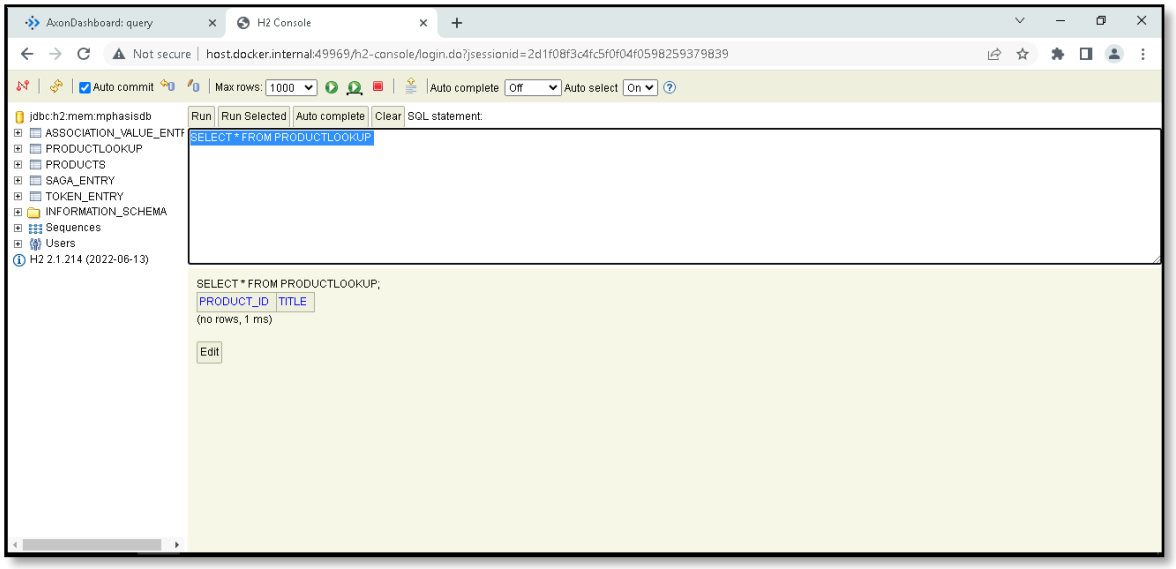



24. Restart the Discovery Server (Eureka Server), Product Service, and ApiGateway.

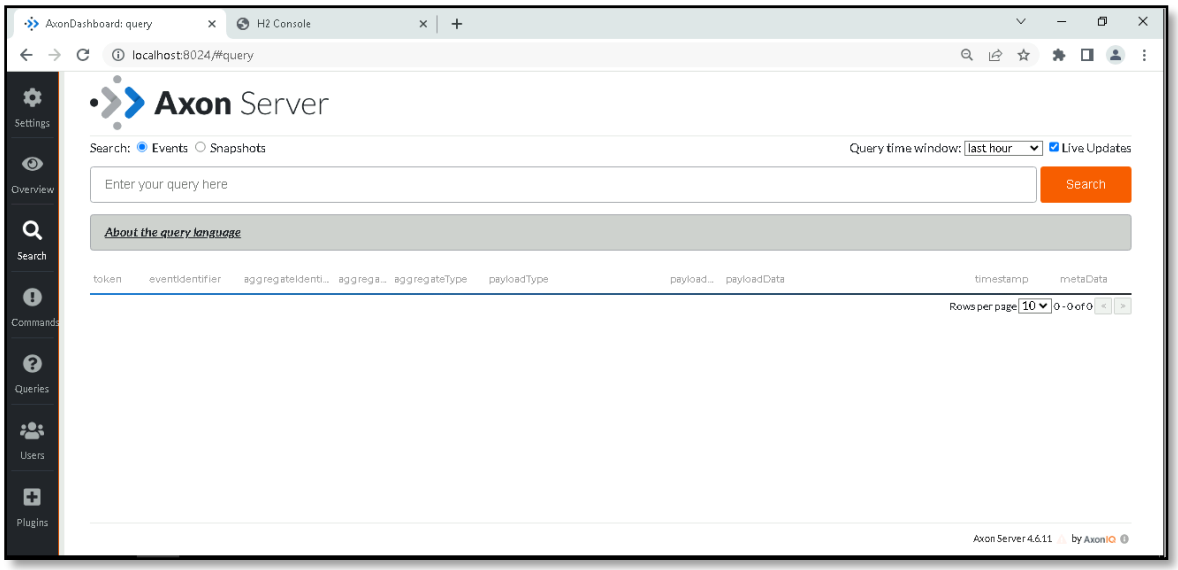
25. Send a POST request to Create Product.



26. Let's go to browser and query the ProductLookup table. Still, we don't have any record. That's good the transaction was rollback, and the Entity did not persist.




27. Now, let's go and lookup in Event Store and see if we have records here.



## Problem Statement 12: Implementing the CQRS & Event Sourcing Design Pattern in Orders Microservice

Similarly, to the Product Microservice, we will create the Orders Microservices with CQRS and Event Sourcing through the Axon Framework.

**Technology stack:**

- Spring Web
- Spring Data JPA
- H2 Database
- Spring Cloud Eureka Client
- Lombok
- Axon Spring Boot Starter
- Google Guava
- Spring Boot Starter Validation

**Steps for implementing CQRS and Event Sourcing using Axon Server:**

1. Create a new Spring Boot Project:
    - Create a new Spring Boot project using either Spring Initializer Tool(https://start.spring.io) or using your development environment.
    - Call this new project "OrdersService".
2. Add the Spring Web, Spring Data JPA, H2 Database, Spring Cloud Eureka Client, Lombok, Axon Spring Boot Starter, Google Guava, and Spring Boot Starter Validation dependencies in pom.xml.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <scope>provided</scope>
</dependency>

<dependency>
    <groupId>org.axonframework</groupId>
    <artifactId>axon-spring-boot-starter</artifactId>
    <version>4.5.8</version>
</dependency>

<dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>30.1-jre</version>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

3. Will take some time for the project to be imported and the maven dependencies to be downloaded once you click **Finish**.
4. Create a new OrdersCommandController class with a request mapping "/orders" and one method that accepts the HTTP POST request.
5. The method that accepts the HTTP Post request, should accept the OrderCreateRest JSON payload as a request body.

   {
   "productId":"f241af45-4854-43f4-95bc-ab54da338a29",
   "quantity":1,
   "addressId":"afbb5881-a872-4d13-993c-faeb8350eea5"
   }
6. Apply the common validation annotation on the OrderCreateRest class:

   Use @NotBlank to say that a productId field must not be the empty string.
   Use @Min and @Max to say that the quantity is a numerical field is only valid when its value is above 1 and below 5.
   Use @NotBlank to say that an addressId field must not be the empty string.
7. Trigger the validation of the request body on ProductCommandController handler methods.
8. This controller class should use the **Axon's *CommandGateway*** and publish the CreateOrderCommand.
9. The CreateOrderCommand annotated with @Data, @Builder and should have the following fields:

   public final String orderId;
   private final String userId;
   private final String productId;
   private final int quantity;
   private final String addressId;
   private final OrderStatus orderStatus;

Where:

- orderId - is a randomly generated value. For example, UUID.randomUUID().toString() and annotated with **@TargetAggregateIdentifier**.
- userId - is a static hard-coded value: 27b95829-4f3f-4ddf-8983-151ba010e35b. At this moment there is no user registration, authentication, and authorization implemented, so we will hard code the value of userId for now.
- orderStatus - is an Enum with the following content:
  public enum OrderStatus {
  CREATED, APPROVED, REJECTED
  }
- After sending the CreateOrderCommand, use **Axon's *QueryGateway*** instance to invoke **query** method which is used publish the FindOrderQuery with orderId and use ResponseTypes.instanceOf(OrderSummary) to get the OrderSummary and return as a response body.
- The OrderSummary annotated with @Value and should have the following fields.

  public final String orderId;
  private final OrderStatus orderStatus;
  private final String message

10. Create a new class called OrderAggregate and make it handle the CreateOrderCommand using **@CommandHandler** and publish the OrderCreatedEvent.
11. The OrderCreatedEvent class annotated with @Data and should have the following fields:

    private String orderId;
    private String productId;
    private String userId;
    private int quantity;
    private String addressId;
    private OrderStatus orderStatus;
12. The OrderAggregate class should also have an **@EventSourcingHandler** method that sets values for all fields in the OrderAggregate.
13. Create a new @Component class called OrderEventsHandler annotated with **@ProcessingGroup("order-group")** inside com.mphasis.query package.
14. Create a new JPA Repository called OrdersRepository inside com.mphasis.core.data package and inject it into OrderEventsHandler using constructor-based dependency injection.
15. Add a find method in OrdersReporitory interface:

```
OrderEntity findByOrderId(String orderId);
```

16. The OrderEventHandler class should have one **@EventHandler** method that handles the OrderCreatedEvent and persists order details into the "read" database.

17. To persist order details into the database, create a new JPA Entity class called OrderEntity inside com.mphasis.core.data package. Annotate the OrderEntity class with:

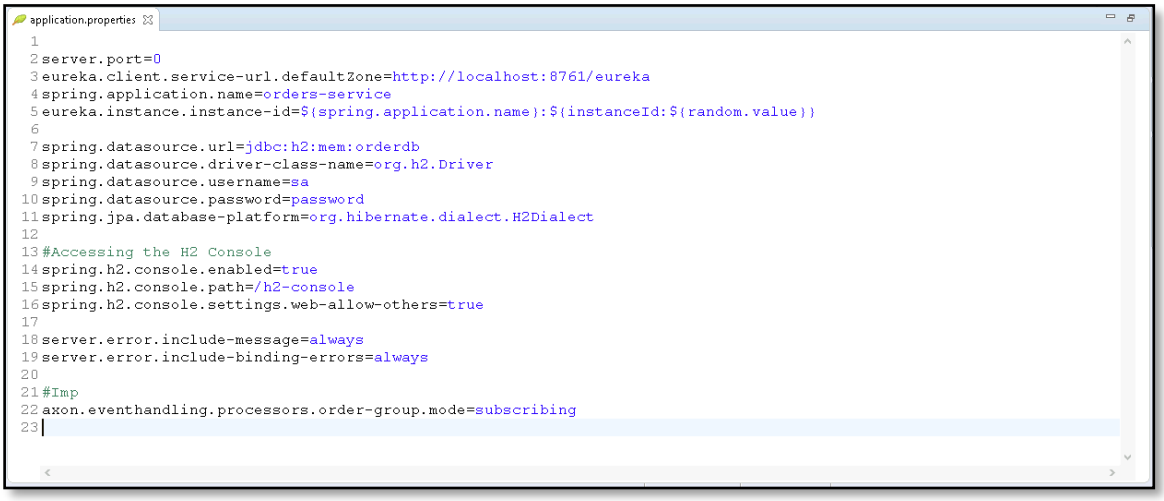
    @Data
    @Entity
    @Table(name = "orders")

    and make the OrderEntity class have the following fields:

    @Id
    @Column(unique = true)
    public String orderId;
    private String productId;
    private String userId;
    private int quantity;
    private String addressId;

    @Enumerated(EnumType.STRING)
    private OrderStatus orderStatus;

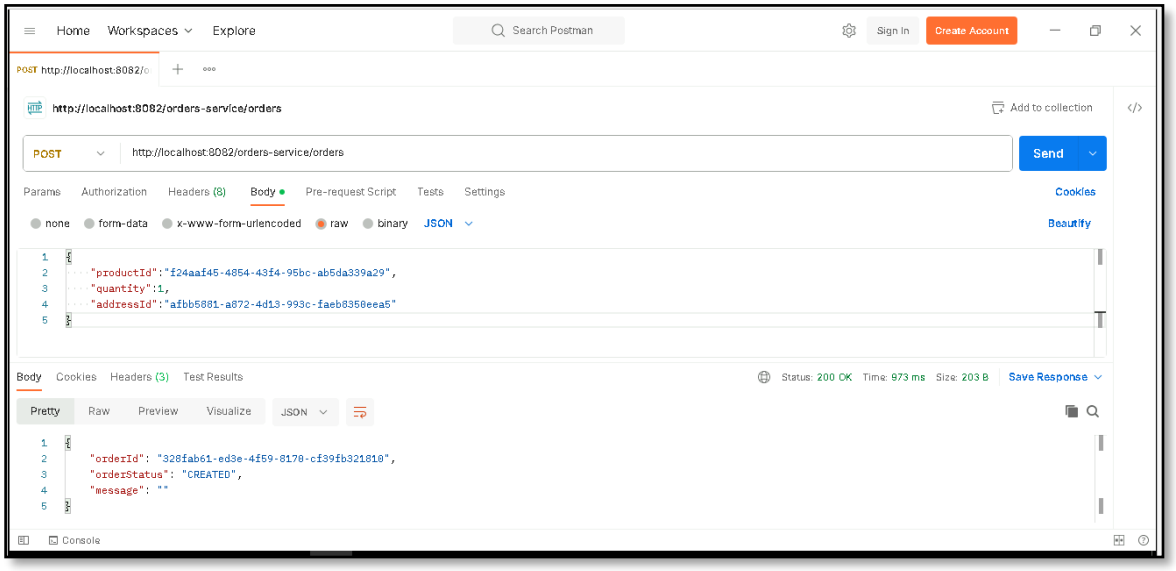
18. Create a new @Component class called OrderQueriesHandler.
19. Create a new JPA Repository called OrdersRepository and inject it into OrderQueriesHandler using constructor-based dependency injection.
20. The OrderQueriesHandler class should have one **@QueryHandler** method that handles the FindOrderQuery and fetch the order summary from the **"read"** database.
21. Register with Eureka: Make OrdersService microservice register with Eureka as a Client.
22. Since each Microservice should store data in its own database, configure this Microservice to work with a new database called "orderdb".
23. Add the below properties to application.properties:



**Run and make it work:**

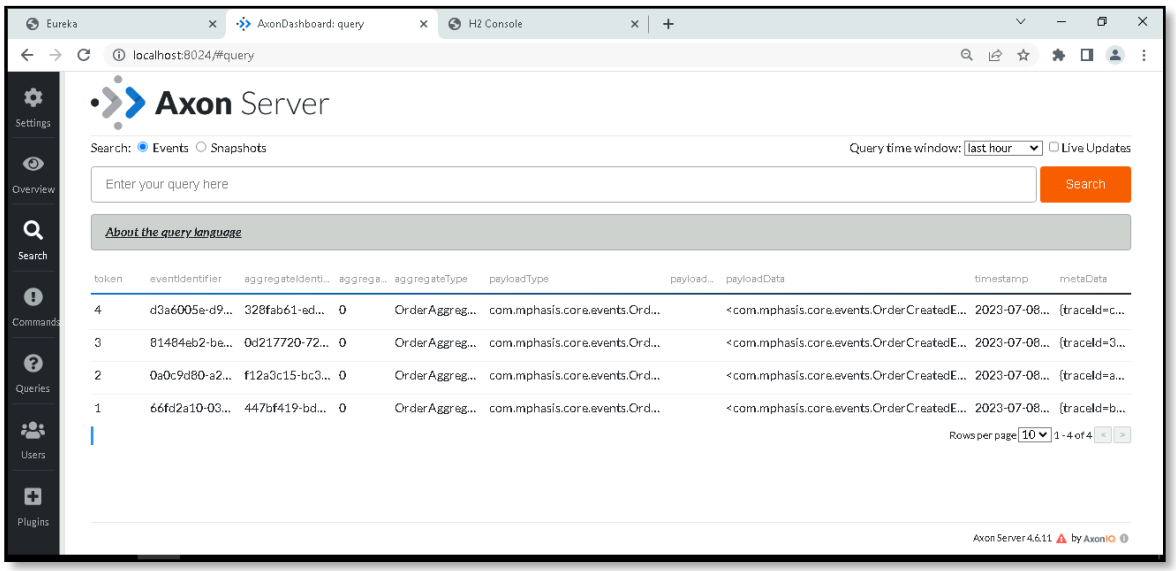
24. Run your OrdersService microservice and make it work. Send a request with the following JSON and make sure it gets successfully stored in the read database. Since you have annotated the OrderEntity class with @Table(name = "orders"), the database table name will be "orders".

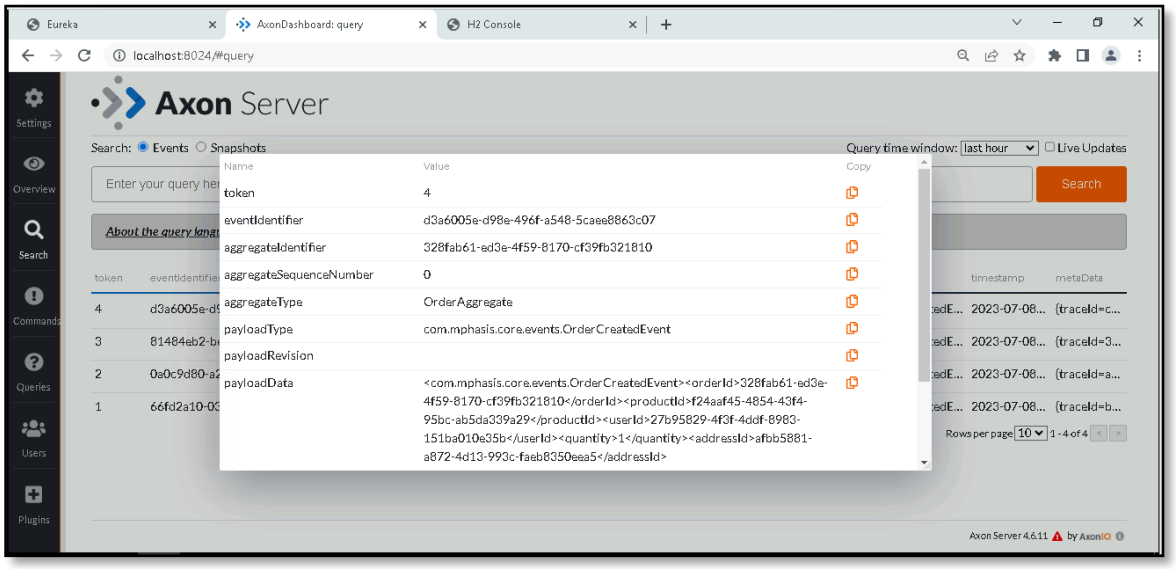
    {
    "productId":"f241af45-4854-43f4-95bc-ab54da338a29",
    "quantity":1,
    "addressId":"afbb5881-a872-4d13-993c-faeb8350eea5"
    }



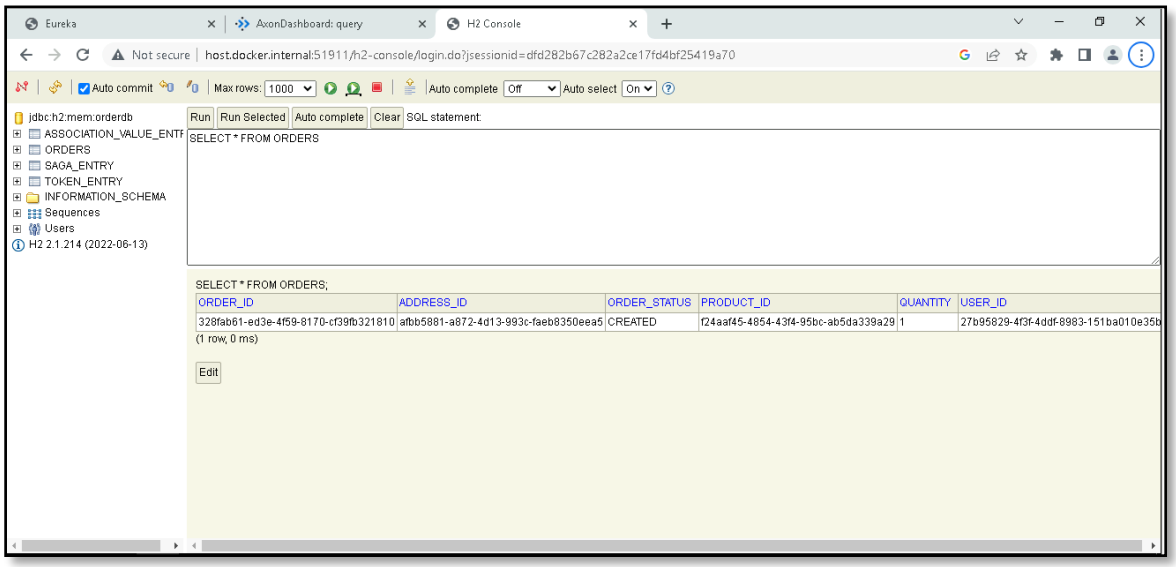

**Verify results:**

25. Check the Event Store in the Axon server and make sure that the OrderCreatedEvent gets persisted,






26. Using the /h2-console connect to the orderdb database and make sure that the order details are stored there as well.



## Problem Statement 13: Orchestration based Saga – Reserve Product in Stock

In this problem statement, we will work on the create order flow. So, I will add the order saga into my orders microservice.

Now, the flow will begin when order aggregate, received the create order command, and publishes an order created event. This is the beginning of the flow, so I will make my saga class handle the order created event and use it as the beginning of the saga flow. The order saga will then publish reserve product command, and once processed by the products microservice, the saga will handle the product reserved event. Saga will continue the flow and publish the process payment command once the product has been reserved. And once the payment is processed, it'll be the saga component that will handle the payment processed event.

So, our saga class will be an event-handling component that will manage the create order flow by handling events, and publishing commands to complete the flow. And if one of the steps on the flow fails, it will be this Saga class that manages the flow of compensating operations to rollback changes done in this flow.

The following Saga Class Structure will be implemented:

```
@Saga
public class OrderSaga {

    @Autowired
    private transient CommandGateway commandGateway;

    @StartSaga
    @SagaEventHandler(associationProperty = "orderId")
    public void handle(OrderCreatedEvent orderCreatedEvent) {
        //
    }

    @SagaEventHandler(associationProperty = "productId")
    public void handle(ProductReservedEvent productReservedEvent) {
        //
    }

    @SagaEventHandler(associationProperty = "paymentId")
    public void handle(PaymentProcessedEvent paymentProcessedEvent) {
        //
    }

    @EndSaga
    @SagaEventHandler(associationProperty = "orderId")
    public void handle(OrderApprovedEvent orderApprovedEvent) {
        //
    }
}
```

**Steps for implementing Orchestration based Saga:**

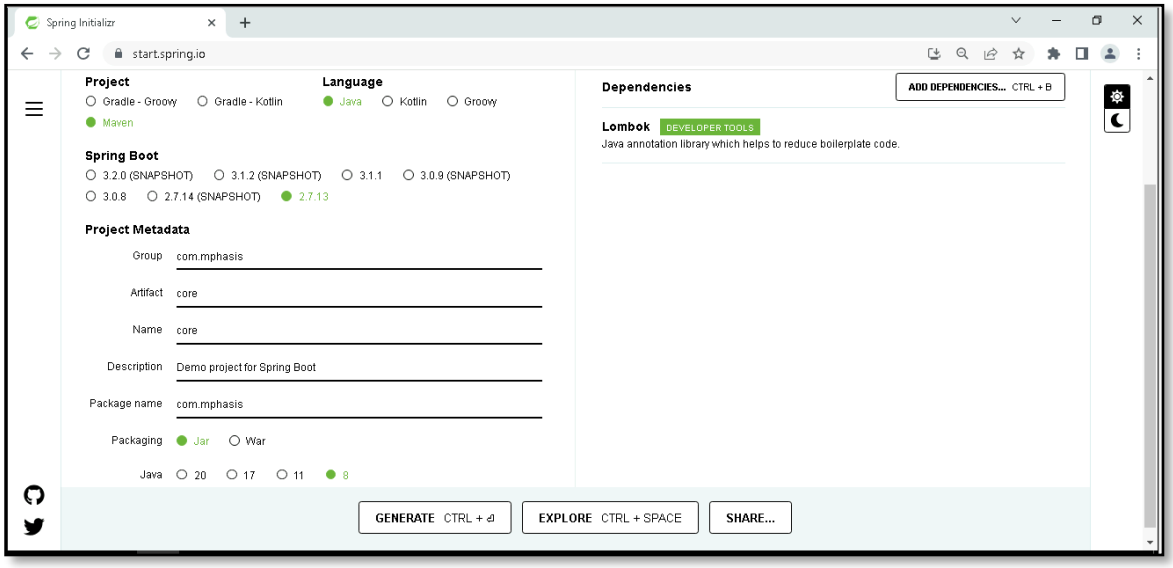
1. Refer the **ProductService** updated in the problem statement – 11.
2. Refer the **OrdersService** created in the problem statement – 12.
3. Create a new class OrderSaga annotated with **@Saga** in the com.mphasis.saga package. Also, Autowire the **Axon's *CommandGateway*** instance inside the OrderSaga class.
4. Start the Saga using **@StartSaga** annotation and create a handler for OrderCreatedEvent using **@SagaEventHandler** annotation with the **associatedProperty** "orderId".

5. The OrderCreatedEvent class annotated with @Data, @NoArgsConstructor, @AllArgsContructor and should have the following fields:

   private String orderId;
   private String productId;
   private String userId;
   private int quantity;
   private String addressId;
   private OrderStatus orderStatus;

6. Create a new Project for **Core** using the **Spring Initalizr** (start.spring.io) with the Lombok starter.

   Refer the below screenshots:




7. Select the Lombok dependencies.
8. Click on GENERATE button or CTRL + Enter to create the project structure.
9. Let's import the Core maven project in STS.
10. Will take some time for the project to be imported and the maven dependencies to be downloaded once you click **Finish**.
11. Add **axon-spring-boot-starter** dependency in pom.xml:

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
</dependency>

<dependency>
    <groupId>org.axonframework</groupId>
    <artifactId>axon-spring-boot-starter</artifactId>
    <version>4.5.8</version>
</dependency>
```

12. Delete the build tag from the pom.xml file.
13. Delete the CoreApplication class and CoreApplicationTests class.
14. So now we can use this project as a dependency to another project.
15. Adding Core project as a dependency to OrdersService/pom.xml and ProductService/pom.xml files:

```
<dependency>
    <groupId>com.mphasis</groupId>
    <artifactId>core</artifactId>
    <version>0.0.1-SNAPSHOT</version>
</dependency>
```

16. Now you can fetch the classes available in core project.
17. Create a new ReserveProductCommand class in com.mphasis.core.commands package inside the Core project.
18. The ReserveProductCommand annotated with @Data, @Builder and should have the following fields:

    private final String productId;
    private final int quantity;
    private final String orderId;
    private final String userId;

    Where:
    productId – Annotated with **@TargetAggregateIdentifier** annotation.
19. ReserveProductCommand from the core module is now available to OrdersService.
20. Inside the OrderSaga - handler method of OrderCreatedEvent, we will create the instance of ReserveProductCommand and Publish it.
21. Now let's handle the ReserveProductCommand inside the ProductService – ProductAggregate class using the **@CommandHandler** annotation.
22. Now to check this command will be successful executed, we need to check if the current quantity of the Product is not less than the requested quantity.
23. Using LOGGER.info, let's log the orderId and productId on the console.
24. If there is enough quantity of this product in stock, then we will create and publish the ProductReservedEvent which will be handled by the OrderSaga class.
25. The ProductReservedEvent class annotated with @Data, @Builder and should have the following fields:

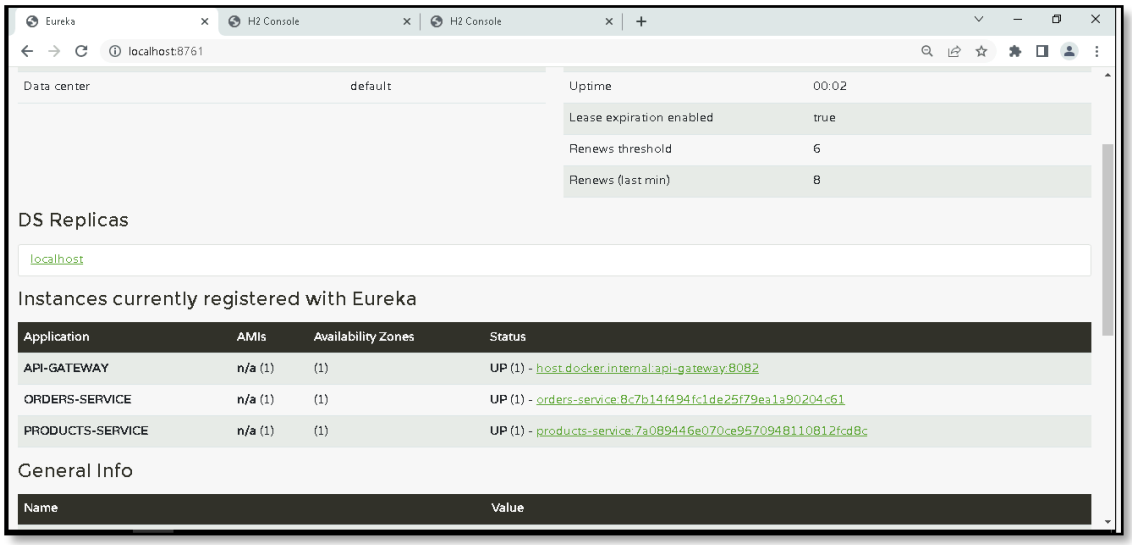
    private final String productId;
    private final int quantity;
    private final String orderId;
    private final String userId;
26. Let's go to our ProductService – ProductAggregate class and will publish the ProductReservedEvent inside the CommandHandler method.
27. Also create an **@EventSourcingHandler** method which is handling the ProductReservedEvent and updating the quantity (subtraction action).
28. To make our read database up to date with the changes that we have just made to the product quantity, we will need to handle the ProductReservedEvent and update the read database as well.

29. In ProductService – ProductEventsHandler class, we need to add one more **@EventHandler** method for the ProductReservedEvent and update the Products projection.
30. After updating, let's log the orderId and productId on the console.
31. Finally, we will Handle the ProductReservedEvent in OrdersService – OrderSaga, by creating a new handler method with the same **associationProperty** i.e., orderId.

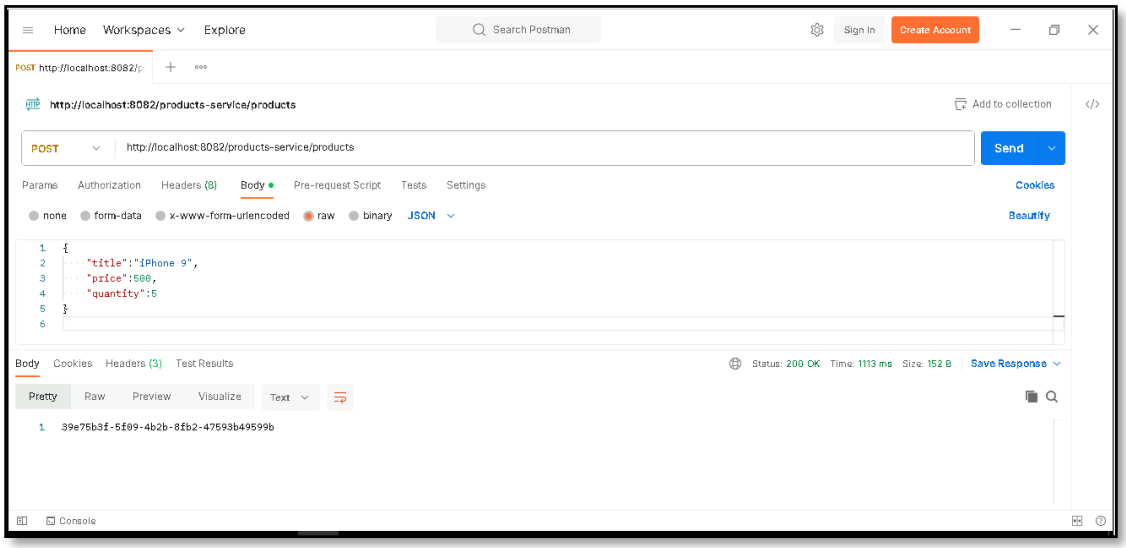
```
@SagaEventHandler(associationProperty = "orderId")
public void handle(ProductReservedEvent productReservedEvent) {
    // Process user payment

    LOGGER.info("ProductReservedEvent is called for productId: " + productReservedEvent.getProductId() +
            " and orderId: " + productReservedEvent.getOrderId());
}
```

**Run and make it work:**

32. Run the Axon Server using Docker command.
33. Start the Discovery Server (Eureka Server), Product Service, Orders Service, and ApiGateway.
34. Verify the Eureka Server: http://localhost:8761/

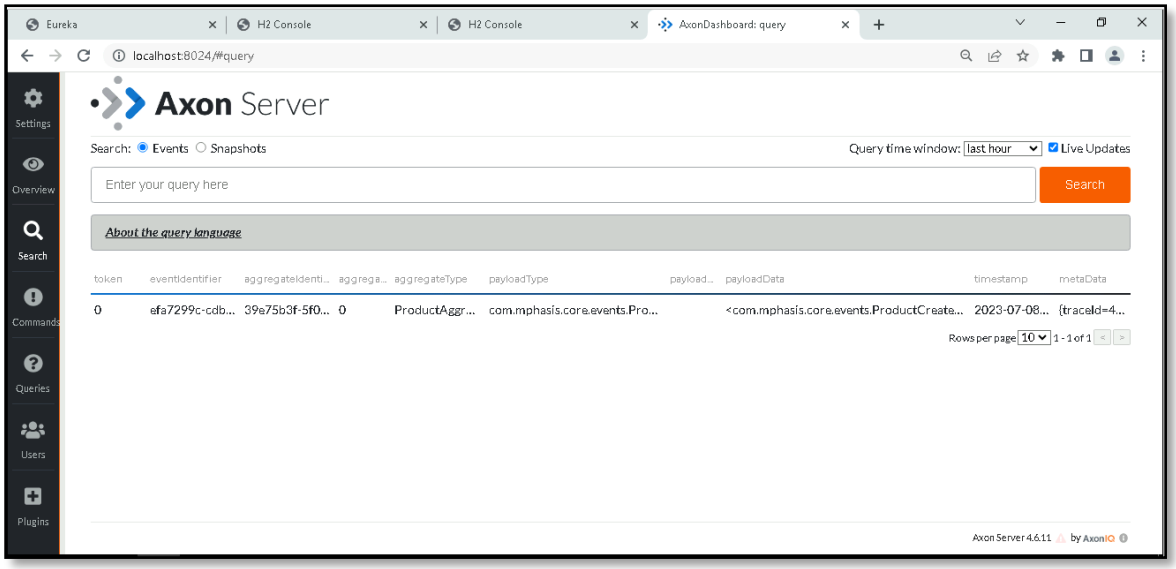


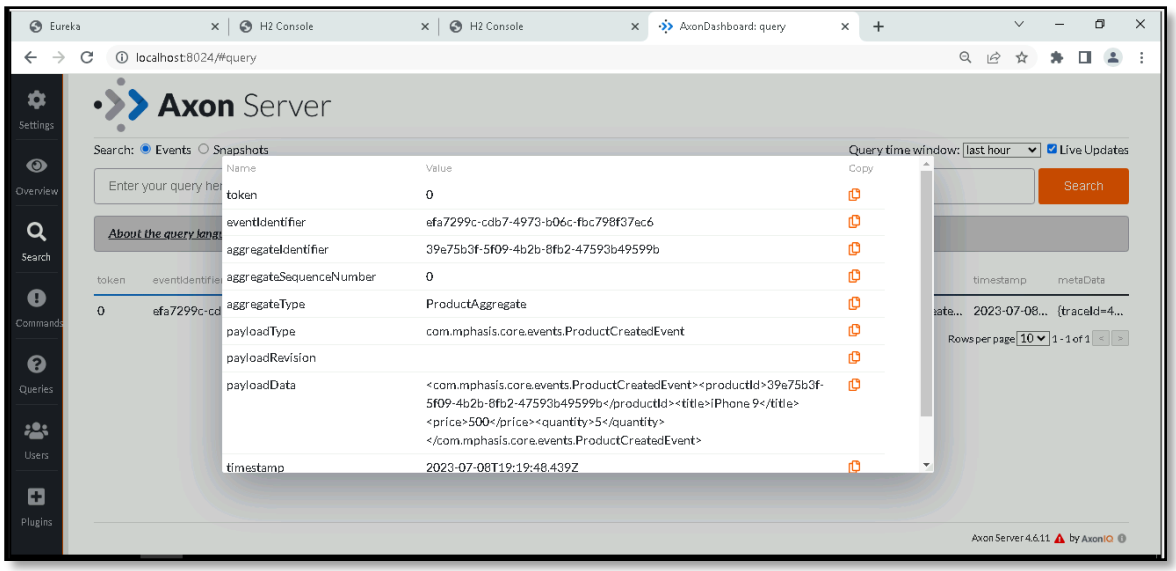

35. Send a POST request to Create Product.



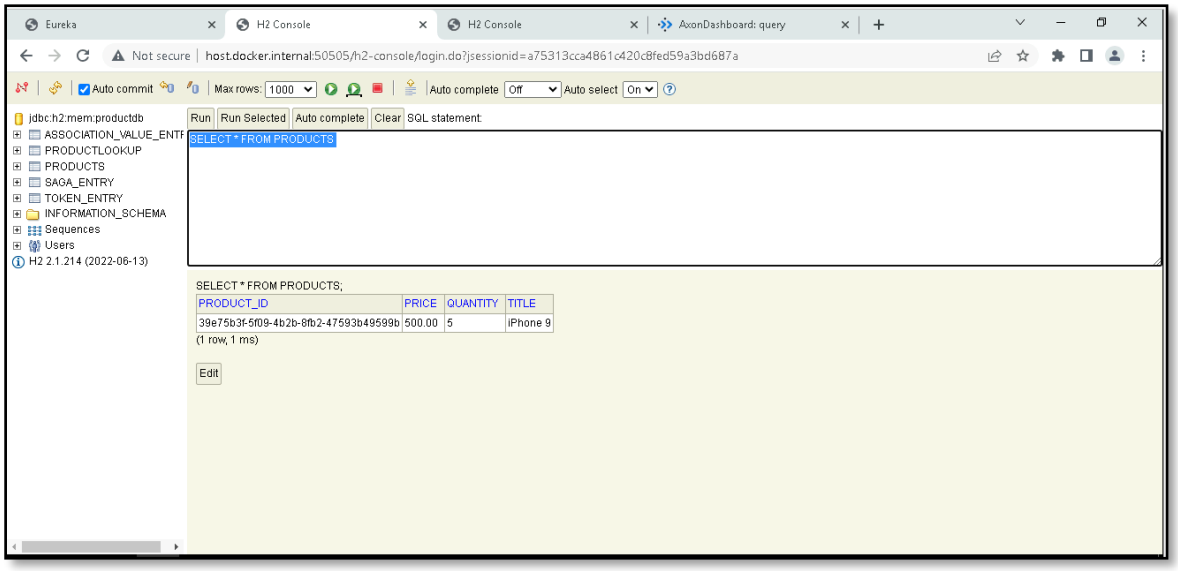
**Verify results:**

36. Check the Event Store in the Axon server and make sure that the ProductCreatedEvent gets persisted,

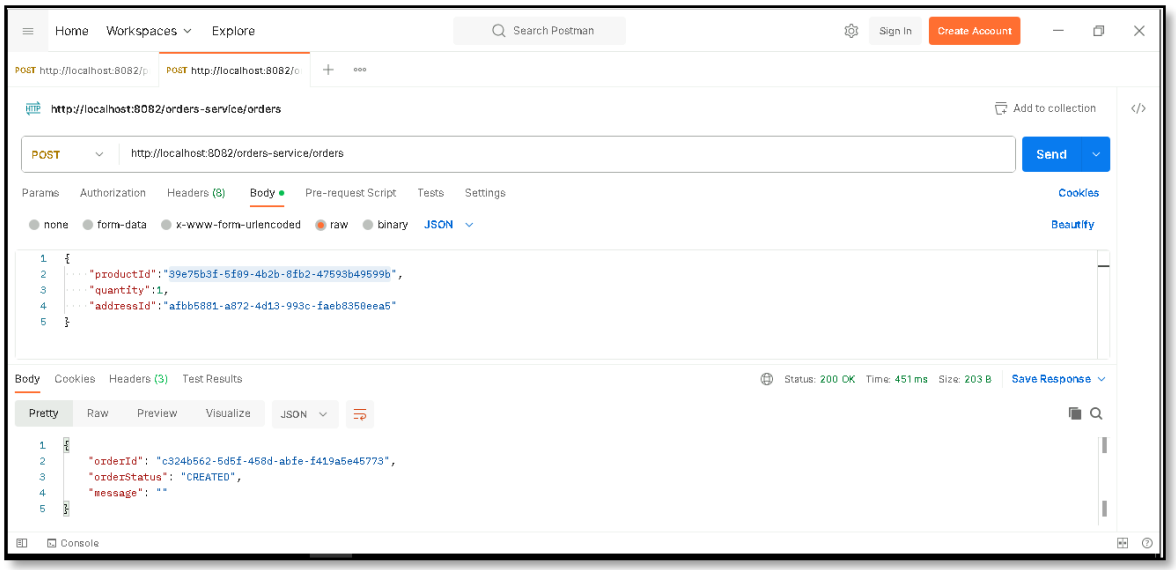


37. Using the /h2-console connect to the productdb database and make sure that the product details are stored there as well.

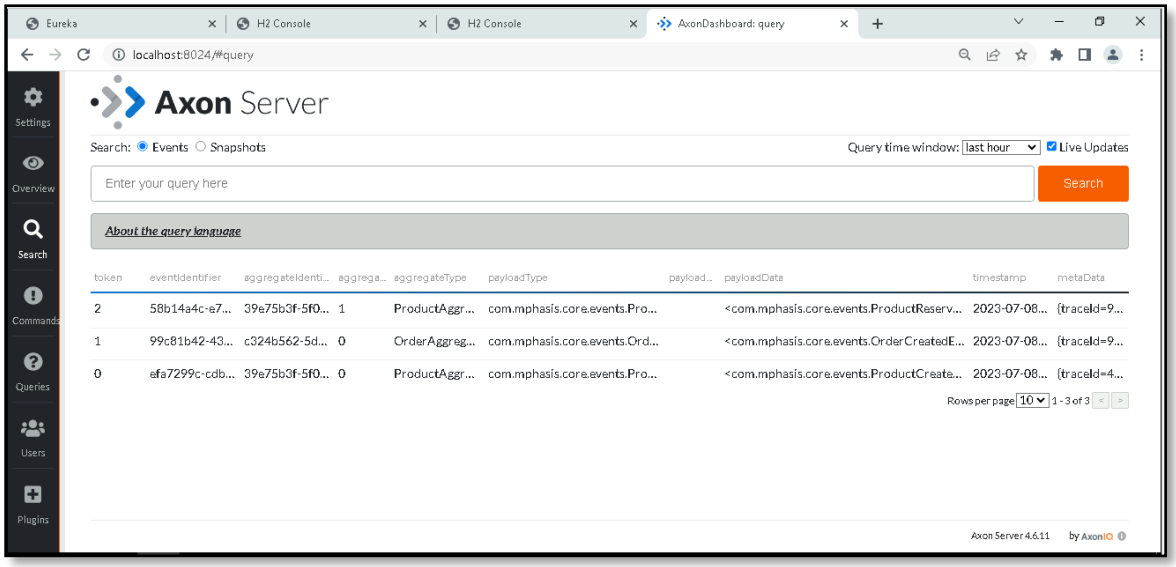


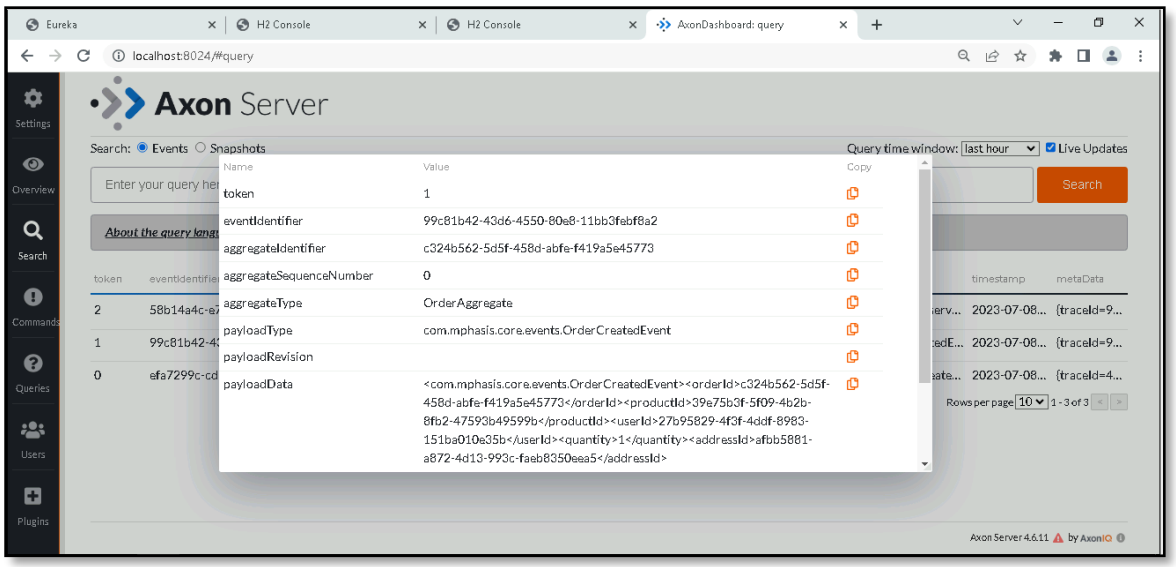

38. Copy the productId, use it in /orders-service/orders. Send a POST request to Create Order.



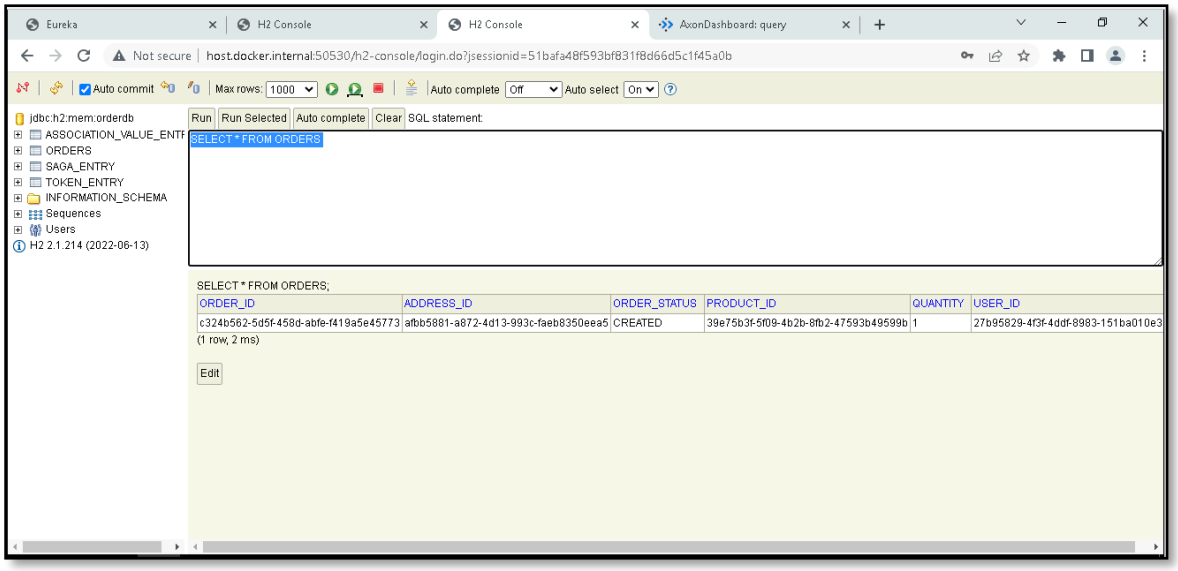
**Verify results:**

39. Check the Event Store in the Axon server and make sure that the OrderCreatedEvent gets persisted,

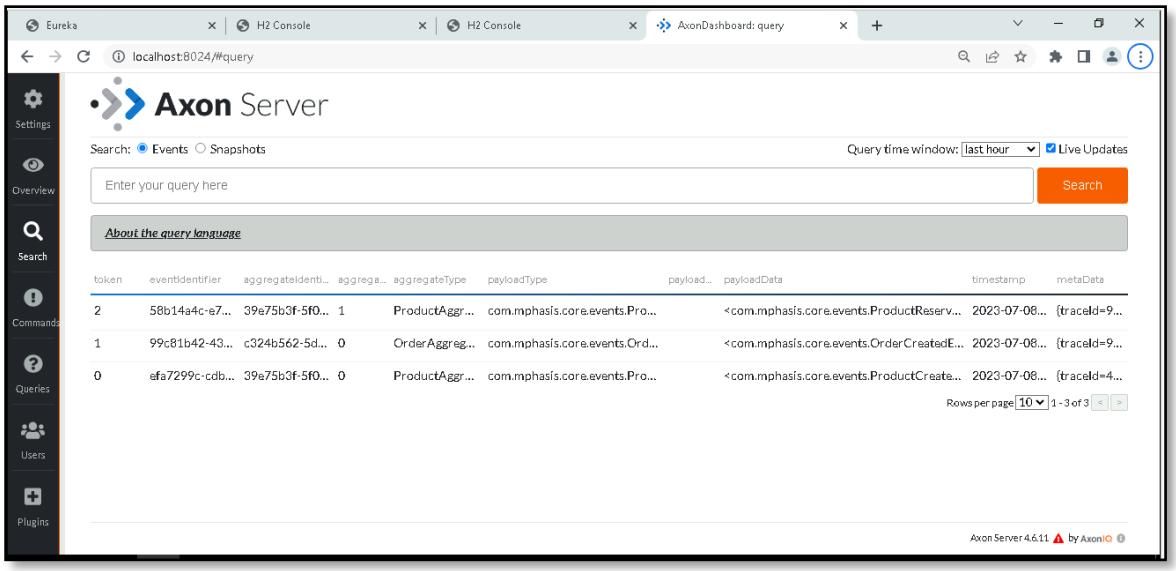


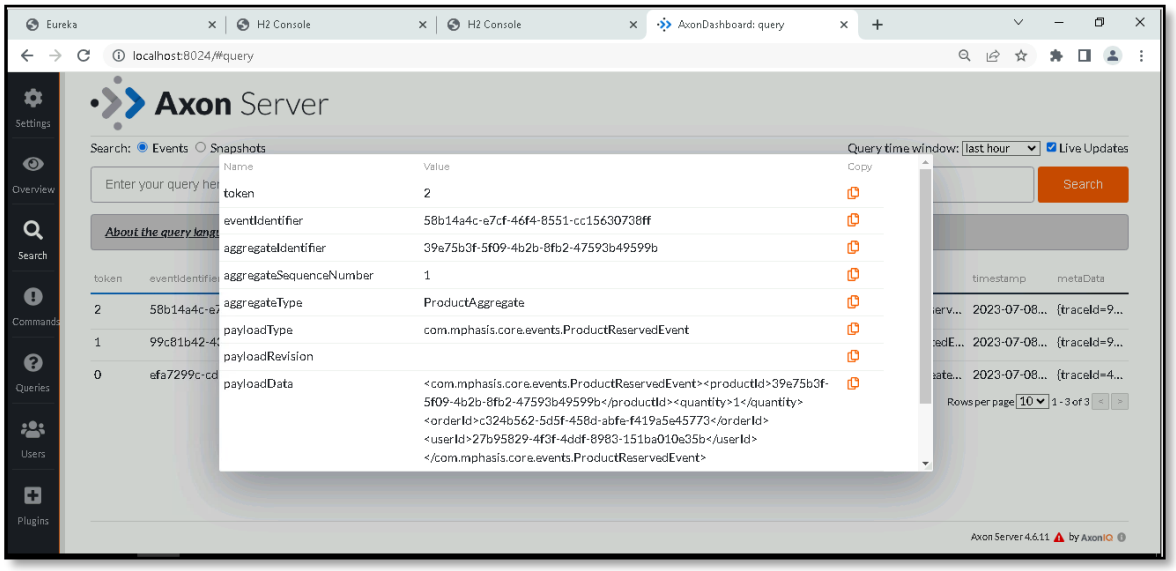



40. Using the /h2-console connect to the orderdb database and make sure that the order details are stored there as well.

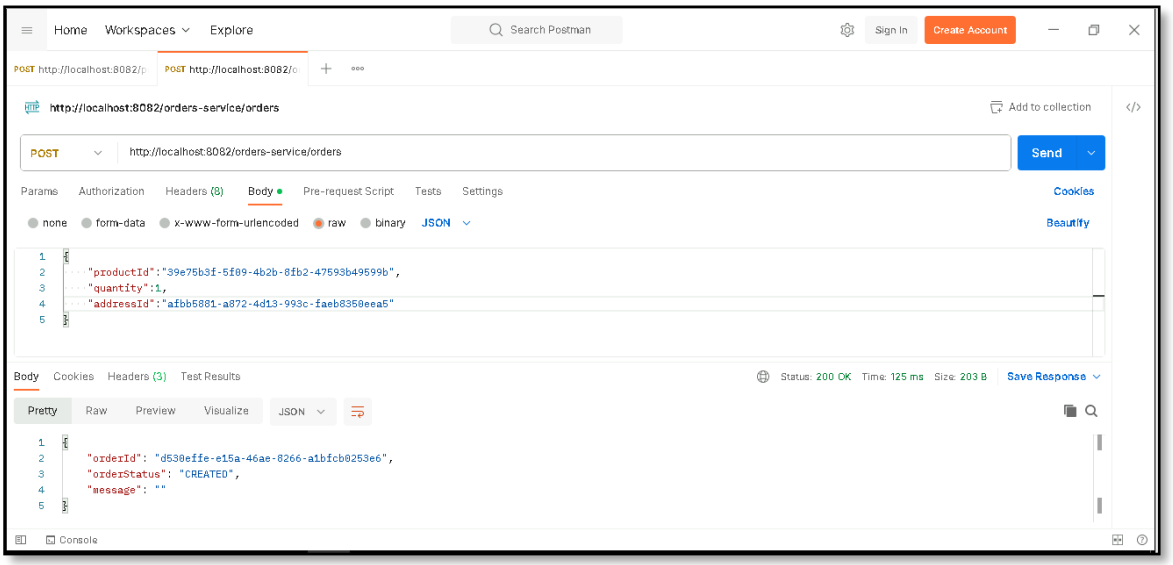



41. You can also find ProductReservedEvent Payload in Axon Server:






42. You can review the logs on the console of each Microservice.
43. Try placing one more order with the same productId. You will find new order with orderId is CREATED.



## Problem Statement 14: Orchestration based Saga – Fetch Payment Details

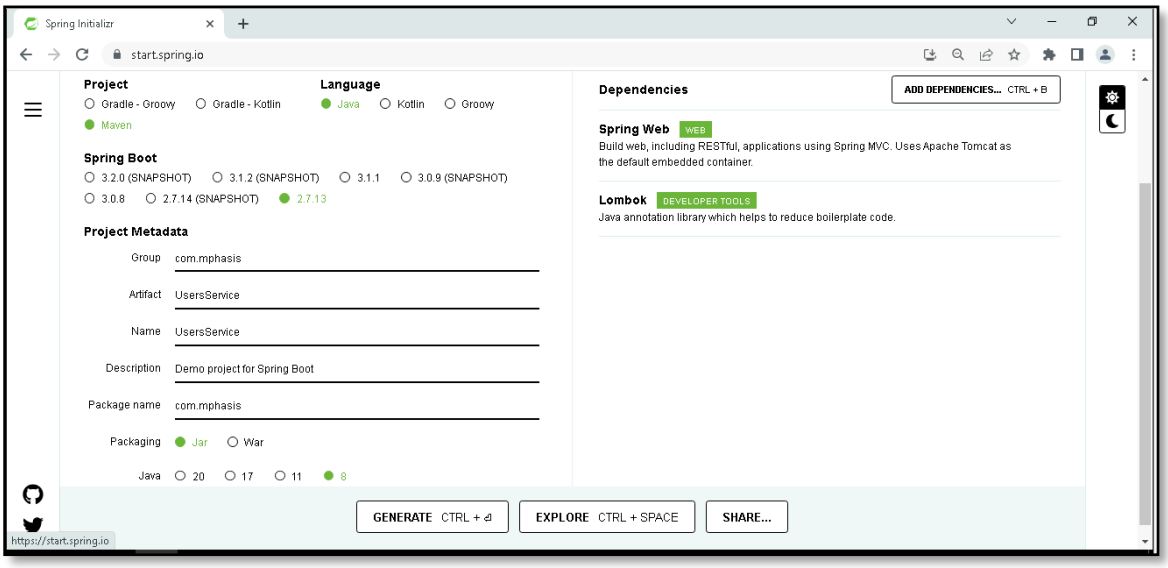
In this problem statement, we will develop a UsersService that OrderSaga will utilize to fetch the user's payment details from another Microservice.

**Technology stack:**

- Spring Web
- Axon Spring Boot Starter
- Google Guava
- Core (A project with shared classes)

**Implementation approach for Orchestration based Saga:**

1. Refer the **OrdersService** updated in the problem statement – 13.
2. Refer the **UsersService** created in the problem statement – 13.
3. Create a new Spring Boot Project:
    - Create a new Spring Boot project using either Spring Initializer Tool(https://start.spring.io) or using your development environment.
    - Call this new project " UsersService ".



4. Add the Spring Web, Lombok, Axon Spring Boot Starter, Google Guava, and Core dependencies in pom.xml.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<dependency>
    <groupId>org.axonframework</groupId>
    <artifactId>axon-spring-boot-starter</artifactId>
    <version>4.5.8</version>
</dependency>
```

```
<dependency>
    <groupId>com.mphasis</groupId>
    <artifactId>core</artifactId>
    <version>0.0.1-SNAPSHOT</version>
</dependency>

<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <scope>provided</scope>
</dependency>
```

```
<dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>30.1-jre</version>
</dependency>
```

5. Will take some time for the project to be imported and the maven dependencies to be downloaded once you click **Finish**.

6. In the Core project, create a new PaymentDetails class with the following fields.

   private final String name;
   private final String cardNumber;
   private final int validUntilMonth;
   private final int validUntilYear;
   private final String cvv;

   Annotate this class with @Data and @Builder Lombok annotations and place this class into com.mphasis.core.model package.

7. In the Core project, create a new User class with the following fields.

   private final String firstName;
   private final String lastName;
   private final String userId;
   private final PaymentDetails paymentDetails;

   Annotate this class with @Data and @Builder Lombok annotations and place this class into com.mphasis.core.model package.

8. In the Core project, create a FetchUserPaymentDetailsQuery class with a single instance property for userId and place this class into a com.mphasis.core.query package.

   private String userId;

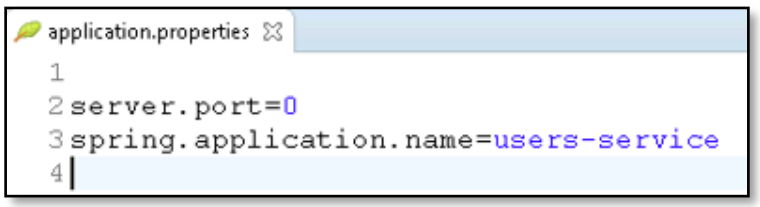
   Annotate this class with @Data and @AllArgsConstructor Lombok annotations.

9. In the UsersService project in com.mphasis.query package, create a new UserEventsHandler class annotated with @Component annotation. In this class create a single method annotated with **@QueryHandler**. Make this method accept FetchUserPaymentDetailsQuery as a method argument and return an instance of a User object with hard-coded details. For example,

   ```
   PaymentDetails paymentDetails = PaymentDetails.builder()
   .cardNumber("123Card")
   .cvv("123")
   .name("Manpreet Singh Bindra")
   .validUntilMonth(12)
   .validUntilYear(2030)
   .build();

   User userRest = User.builder()
   .firstName("Manpreet Singh")
   .lastName("Bindra")
   .userId(query.getUserId())
   .paymentDetails(paymentDetails)
   .build();
   ```

10. Create the UsersQueryController class in com.mphasis.query.rest package.
    - The method that accepts the HTTP Get request, should have User as a response body and userId as a parameter with the URI "/users/{userId}/payment-details".
    - This controller class should use the **Axon's *QueryGateway*** to dispatch an instance of FetchUserPaymentDetailsQuery. As we use the gateway's query() method to issue a point-to-point query. Because we are specifying ResponseTypes.instancesOf(User.class), Axon knows we only want to talk to query handlers whose return type is a User object.

11. Add the property to application.properties file:



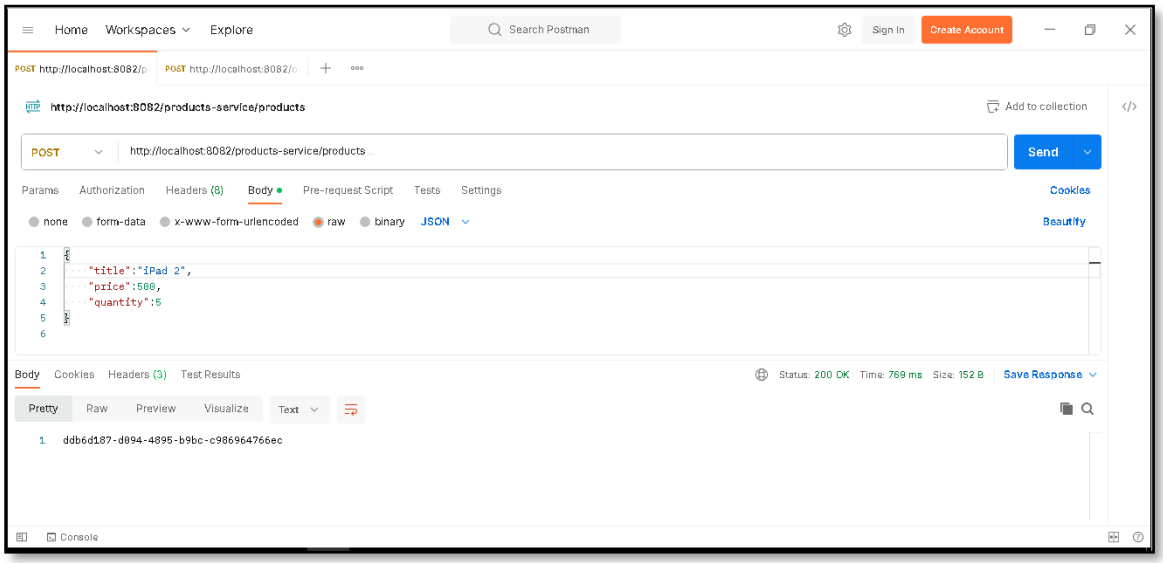

12. Finally, let's go to the OrdersService – OrderSaga class, should have one **@SagaEventHandler** method that handles the ProductReservedEvent and fetch the user payment details from the **"read"** database.

13. The OrderSaga class should use the **Axon's *QueryGateway*** to dispatch an instance of FetchUserPaymentDetailsQuery.

```java
OrderSaga.java
60    @SagaEventHandler(associationProperty = "orderId")
61    public void handle(ProductReservedEvent productReservedEvent) {
62        // Process user payment
63        LOGGER.info("ProductReservedEvent is called for productId: " + productReservedEvent.getProductId() +
64                " and orderId: " + productReservedEvent.getOrderId());
65
66        FetchUserPaymentDetailsQuery fetchUserPaymentDetailsQuery =
67                new FetchUserPaymentDetailsQuery(productReservedEvent.getUserId());
68
69        User userPaymentDetails = null;
70        try {
71            userPaymentDetails = queryGateway.query(fetchUserPaymentDetailsQuery, ResponseTypes.instanceOf(User.cla
72        } catch (Exception ex) {
73            LOGGER.error(ex.getMessage());
74            //Start compensating transaction
75            return;
76        }
77
78        if (userPaymentDetails == null) {
79            //Start compensating transaction
80            return;
81        }
82
83        LOGGER.info("Successfully fetched user payment details for user " + userPaymentDetails.getFirstName());
84    }
```

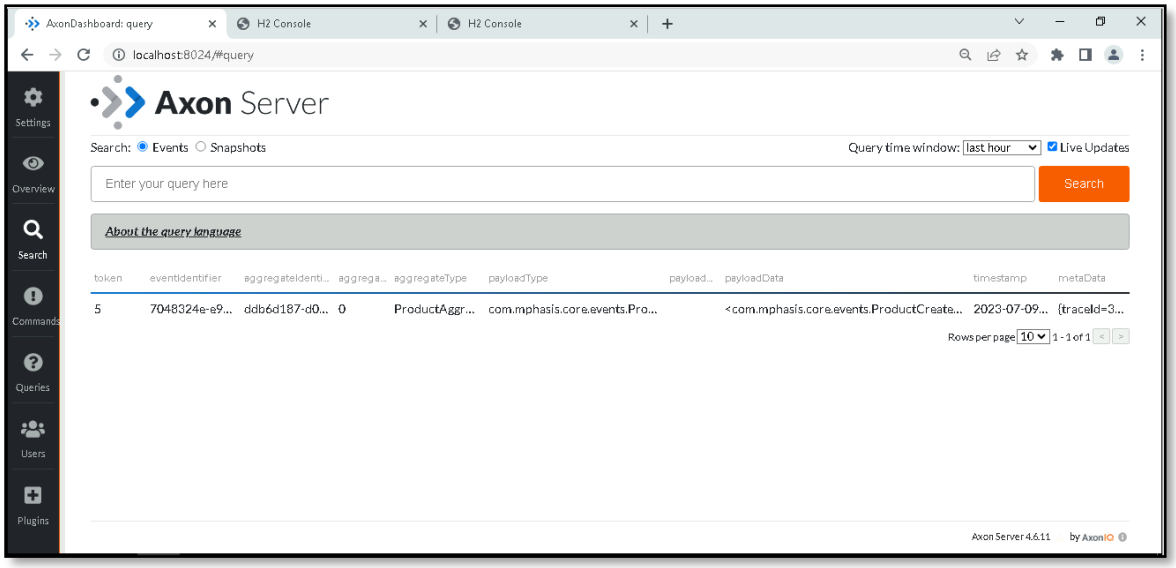
**Run and make it work:**

14. Run the Axon Server using Docker command.
15. Start the Discovery Server (Eureka Server), Product Service, OrderService, UsersService, and ApiGateway.
16. Send a POST request to Create Product.

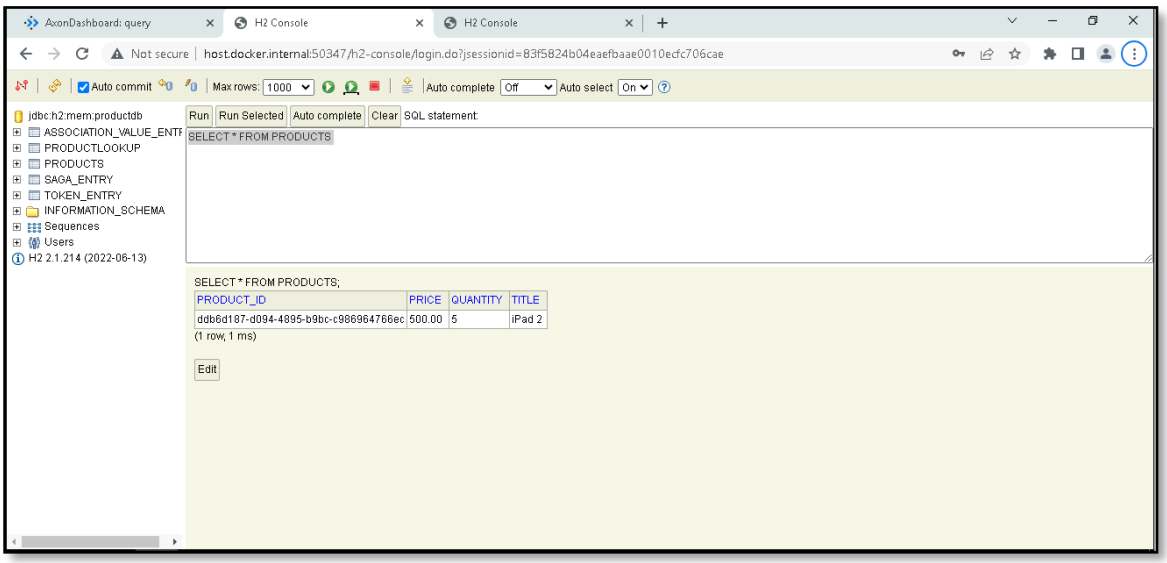


**Verify results:**

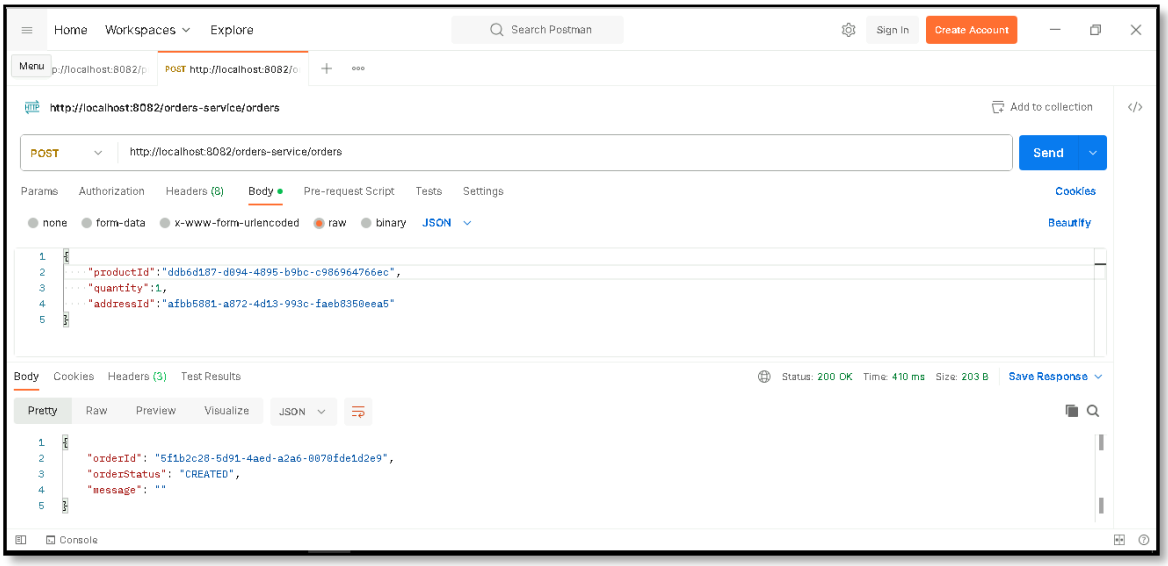
17. Check the Event Store in the Axon server and make sure that the ProductCreatedEvent gets persisted,




18. Using the /h2-console connect to the productdb database and make sure that the product details are stored there as well.

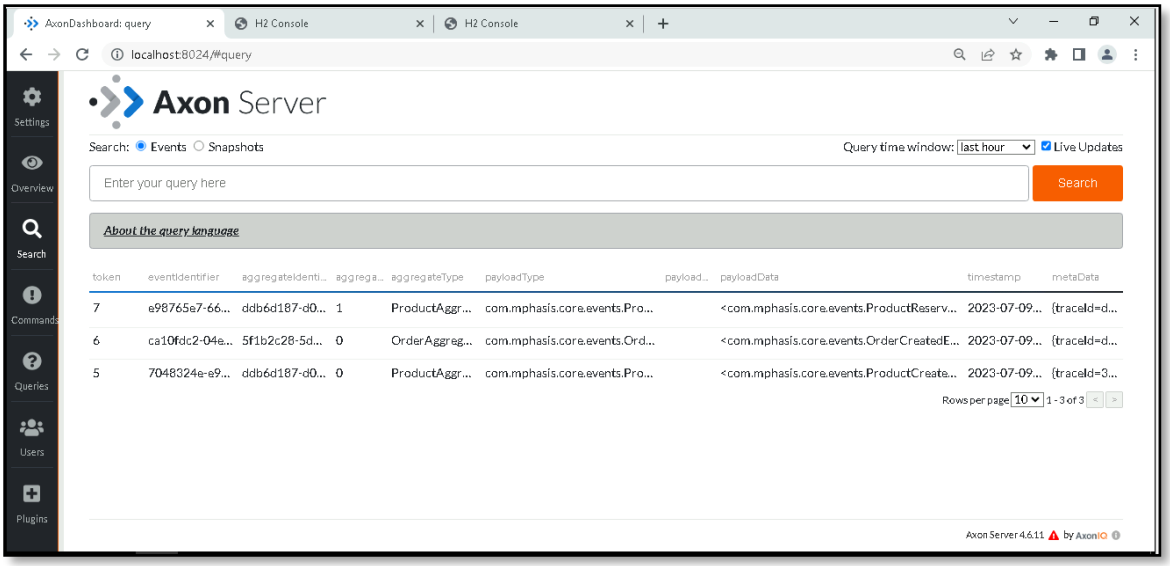


19. Copy the productId, use it in /orders-service/orders. Send a POST request to Create Order.

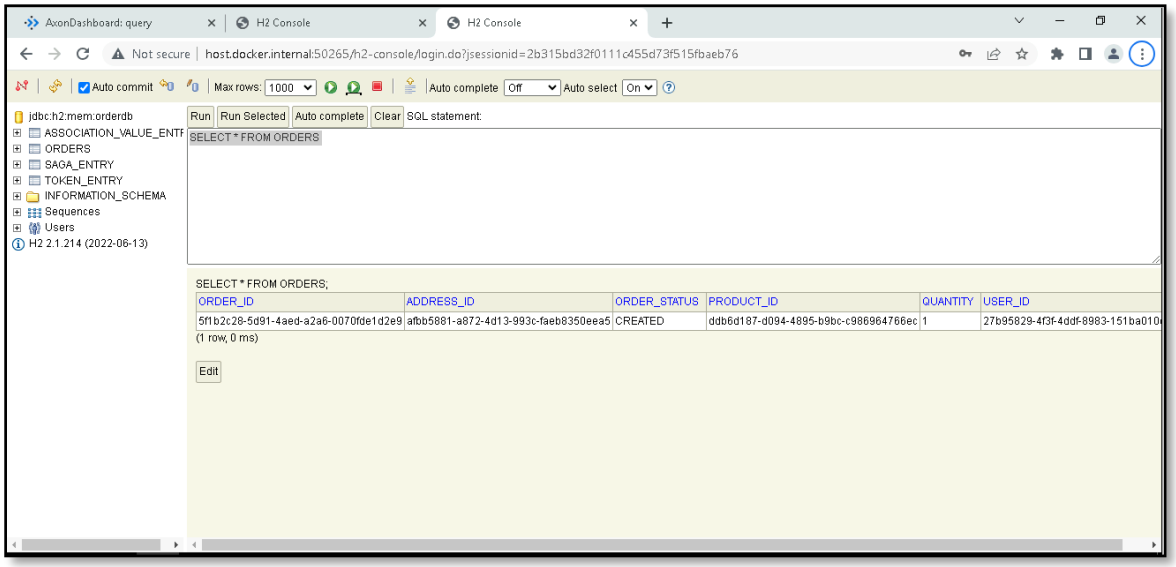



**Verify results:**

20. Check the Event Store in the Axon server and make sure that the OrderCreatedEvent gets persisted,



21. Using the /h2-console connect to the orderdb database and make sure that the order details are stored there as well.




22. Check the successful log message on the OrdersService console.

```
com.mphasis.saga.OrderSaga               : Successfully fetched user payment details for user Manpreet Singh
```

## Problem Statement 15: Orchestration based Saga – Process User Payment

As the next step in our Saga flow, we will need to process user payment. If the payment is successful, the order will be approved. If payment is not successful, we will need to initiate a compensating transaction to reject the order.

Because the goal of this problem statement is to learn how SAGA works rather than how to interact with a third-party payment system, the payment Microservice that we will construct will not really transfer user payment credentials to a real payment system. If the payment details are valid, the Payments Microservice will persist the event details to a database table. Otherwise, the payment microservice will return an error.

**Technology stack:**

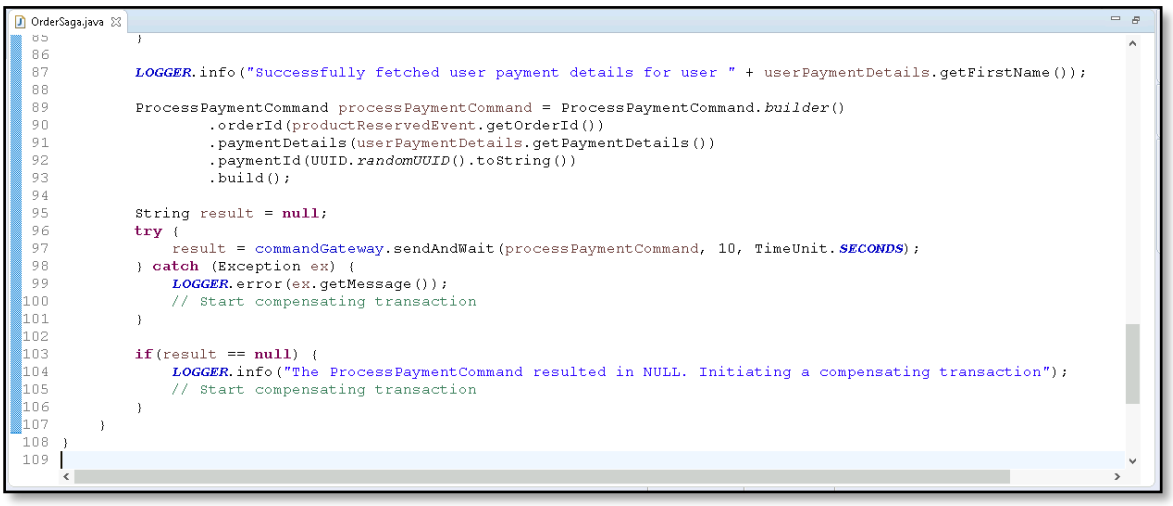
- Spring Web
- Spring Data JPA
- H2
- Axon Spring Boot Starter
- Google Guava
- Core (A project with shared classes)

**Implementation approach for Orchestration based Saga:**

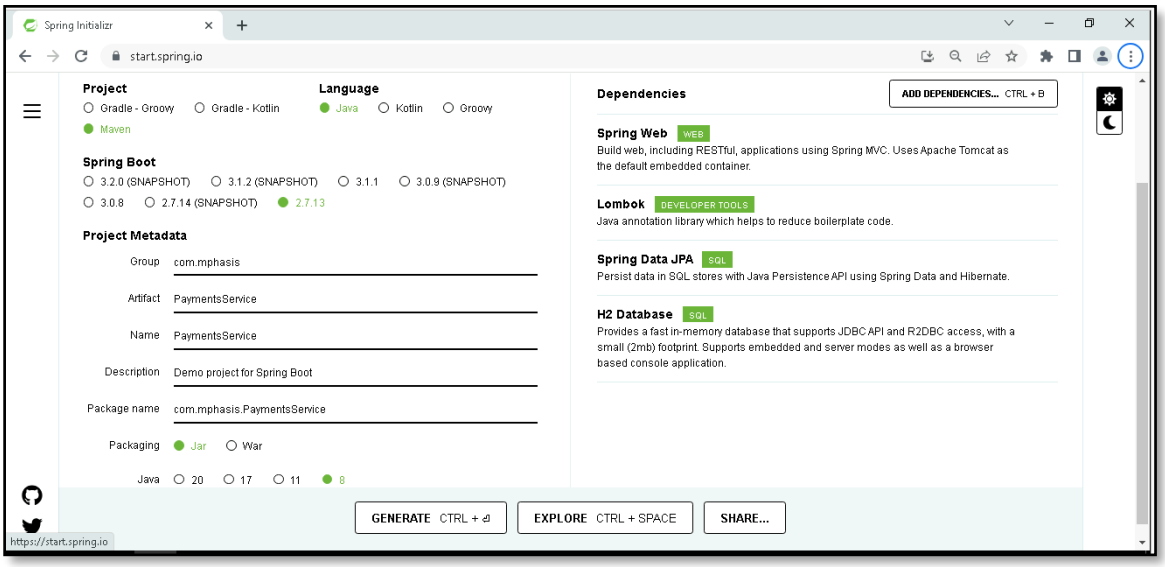
1. Refer the **OrdersService** updated in the problem statement – 14.
2. Refer the **UsersService** updated in the problem statement – 14.
3. In the Core project, create a new ProcessPaymentCommand class annotated with @Data, @Builder in the com.mphasis.core.commands package and should have the following fields:

   private final String paymentId;
   private final String orderId;
   private final PaymentDetails paymentDetails;

   Where:
   paymentId – Annotated with **@TargetAggregateIdentifier** annotation.
4. Let's go to the OrdersService – OrderSaga class, should have one **@SagaEventHandler** method that handles the ProductReservedEvent and fetch the user payment details from the **"read"** database.
5. After that the OrderSaga class should use the **Axon's *CommandGateway*** and publish the ProcessPaymentCommand.



6. Create a new Spring Boot Project:
    - Create a new Spring Boot project using either Spring Initializer Tool(https://start.spring.io) or using your development environment.
    - Call this new project " PaymentsService".




7. Add the Spring Web, Lombok, Spring Data JPA, H2, Axon Spring Boot Starter, Google Guava, and Core dependencies in pom.xml.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <scope>provided</scope>
</dependency>

<dependency>
    <groupId>org.axonframework</groupId>
    <artifactId>axon-spring-boot-starter</artifactId>
    <version>4.5.8</version>
</dependency>

<dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>30.1-jre</version>
</dependency>
```

```
<dependency>
    <groupId>com.mphasis</groupId>
    <artifactId>core</artifactId>
    <version>0.0.1-SNAPSHOT</version>
</dependency>
```

8. Will take some time for the project to be imported and the maven dependencies to be downloaded once you click **Finish**.
9. In the PaymentsService, create a new class called PaymentAggregate, make it handle the ProcessPaymentCommand, and publish the PaymentProcessedEvent.
10. In the Core project, create the PaymentProcessedEvent class will have the following fields:

    private final String orderId;
    private final String paymentId;

    Annotate this class with @Value annotations and in the com.mphasis.core.events package.
11. The PaymentAggregate class should also have an **@EventSourcingHandler** method that sets values for all fields in the PaymentAggregate.
12. In the PaymentAggregate class, create a **@CommandHandler** method that validate the ProcessPaymentCommand. If one of the required fields contains an invalid value, then throw an IllegalArumentException.

```
21    @CommandHandler
22    public PaymentAggregate(ProcessPaymentCommand processPaymentCommand) {
23
24        if(processPaymentCommand.getPaymentDetails() == null) {
25            throw new IllegalArgumentException("Missing payment details");
26        }
27
28        if(processPaymentCommand.getOrderId() == null) {
29            throw new IllegalArgumentException("Missing orderId");
30        }
31
32        if(processPaymentCommand.getPaymentId() == null) {
33            throw new IllegalArgumentException("Missing paymentId");
34        }
35
36        AggregateLifecycle.apply(new PaymentProcessedEvent(processPaymentCommand.getOrderId(),
37                processPaymentCommand.getPaymentId()));
38    }
39
```

13. Create a new @Component class called PaymentEventsHandler in com.mphasis.events package.
14. Create a new JPA Repository called PaymentsRepository in com.mphasis.data package and inject it into PaymentEventsHandler using constructor-based dependency injection.
15. The PaymentEventsHandler class should have one **@EventHandler** method that handles the PaymentProcessedEvent and persists payment details into the **"read"** database.
16. To persist payment details into the database, create a new JPA Entity class called PaymentEntity in com.mphasis.data package. Annotate the PaymentEntity class with:

    @Data
    @Entity
    @Table(name = "payments")
    and make the PaymentEntity class have the following fields:
    @Id
    private String paymentId;
    @Column
    public String orderId;
17. Since each Microservice should store data in its own database, configure Payments Microservice to work with a new database called "paymentdb".
18. Add the DB properties to application.properties file:

application.properties

```
1
2 server.port=0
3 spring.application.name=payments-service
4
5 spring.datasource.url=jdbc:h2:mem:paymentdb
6 spring.datasource.driver-class-name=org.h2.Driver
7 spring.datasource.username=sa
8 spring.datasource.password=password
9 spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
10
11 #Accessing the H2 Console
12 spring.h2.console.enabled=true
13 spring.h2.console.path=/h2-console
14 spring.h2.console.settings.web-allow-others=true
15
16
```

19. Finally, let's go to the OrdersService – OrderSaga class, should have one **@SagaEventHandler** method with the **associationProperty "orderId"** that handles the PaymentProcessedEvent and publish the ApproveOrderCommand.
20. In the OrdersService project, create a new ApproveOrderCommand class annotated with @Data and @AllArgsConstructor in the com.mphasis.command.commands package and should have the following field:

    private final String orderId;

    Where:
    orderId – Annotated with **@TargetAggregateIdentifier** annotation.
21. In the OrderAggregate class, create a **@CommandHandler** method that will handle the ApproveOrderCommand and publish the OrderApprovedEvent.
22. In the OrdersService project, create a new OrderApprovedEvent class annoted with @Value in the com.mphasis.core.events package and should have the following fields:

    private final String orderId;
    private final OrderStatus orderStatus = OrderStatus.Approved;
23. The OrderAggregate class should have **@EventSourcingHandler** method that will handle the OrderApprovedEvent and set the orderStatus field.
24. Create a new JPA Repository called OrdersRepository inside com.mphasis.core.data package and inject it into OrderEventsHandler using constructor-based dependency injection.
25. Add the find method in OrderRepository interface:

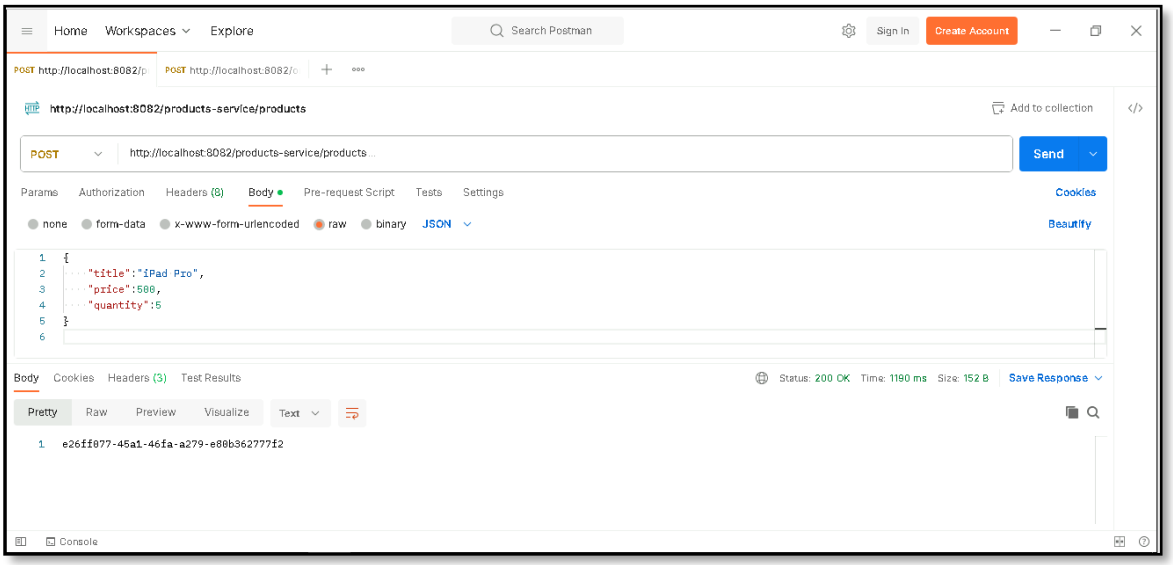
```
OrderEntity findByOrderId(String orderId);
```

26. The OrderEventsHandler class should have **@EventHandler** method that will handle the OrderApprovedEvent and persist the order details in **"read"** database.
27. Finally, let's go to the OrdersService – OrderSaga class, should have one **@SagaEventHandler** method with the **associationProperty "orderId"** that handles the OrderApprovedEvent and log the orderId.
28. This method should be annotated with **@EndSaga** which indicates the end of a Saga instance's lifecycle. When event handling completes, the Saga is destroyed and may no longer receive events.

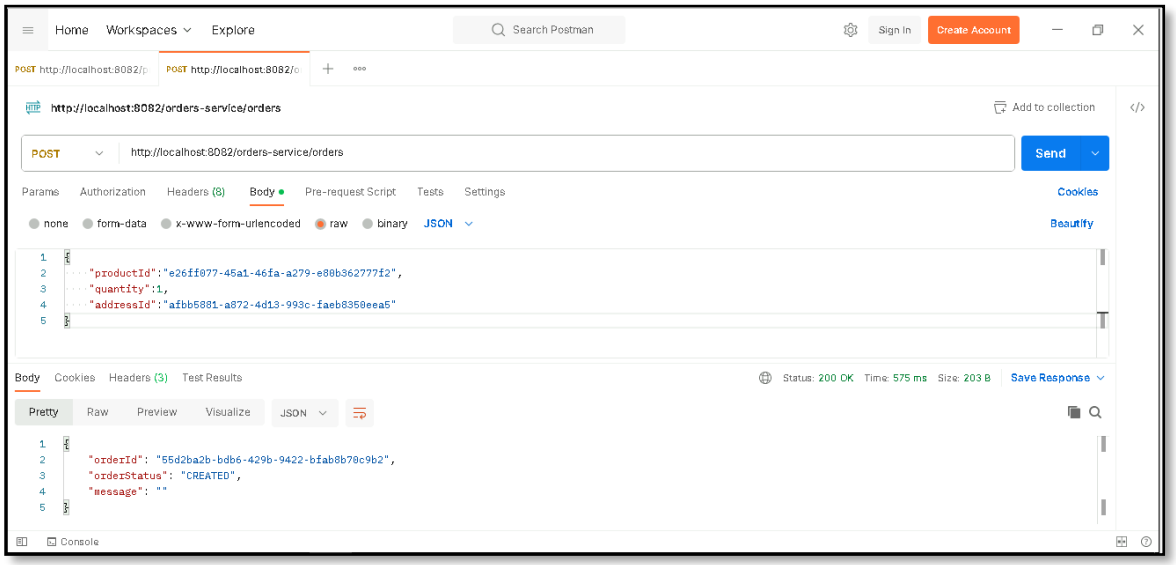
```
@EndSaga
@SagaEventHandler(associationProperty = "orderId")
public void handle(OrderApprovedEvent orderApprovedEvent) {

    LOGGER.info("Order is approved. Order Saga is complete for orderId: "
            + orderApprovedEvent.getOrderId());
    // SagaLifecycle.end();
}
```

**Run and make it work:**

29. Run the Axon Server using Docker command.
30. Start the Discovery Server (Eureka Server), Product Service, OrdersService, UsersService, PaymentsService, and ApiGateway.
31. Send a POST request to Create Product.




32. Copy the productId, use it in /orders-service/orders. Send a POST request to Create Order.



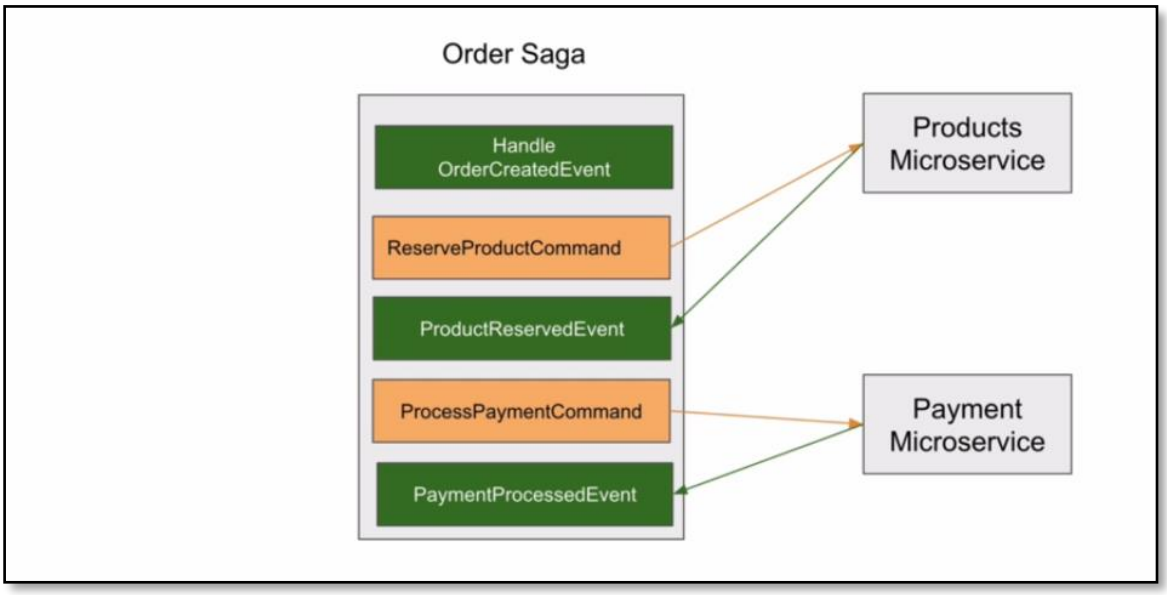

33. Check the Order approved log message on the OrdersService console.

```
com.mphasis.saga.OrderSaga               : Order is approved. Order Saga is complete for orderId: 55d2ba2b-bdb6-429b-9422-
```

## Problem Statement 16: Saga Compensating Transaction in Microservices

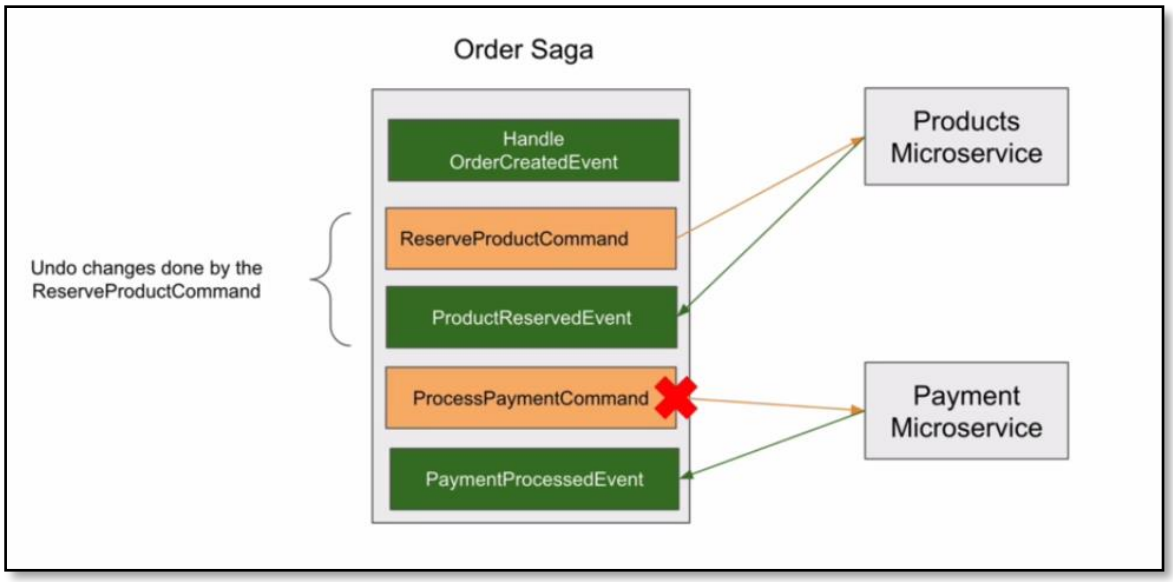
As you already know, Saga is event handling component that maintains a sequence of local transactions. In previous problem statements, we have constructed a Saga that manages a very simple order form. It will handle an event and it will publish a new command to trigger the next Saga local transactions.

Here is a diagram explaining the order flow:




If one of the local transactions fails, Saga will need to run a series of compensated transactions to undo the modifications done by the previous transactions. For example, if the process payment command fails, Saga must undo the subsequent modifying transaction initiated by the reserve product command. If there were more altering transactions, Saga will have to undo them all in reverse order.

Here is a diagram explaining the failed order flow:



**Implementation of compensating transaction in our Saga flow:**

1. Refer all the **Microservices** created in the previous problem statement.
2. In the Core project, create the CancelProductReservationCommand class annotated with @Data and @Builder in the com.mphasis.core.commands package and should have the following fields:

   private final String productId;
   private final int quantity;
   private final String orderId;
   private final String userId;
   private final String reason;

   Where:
   orderId – Annotated with **@TargetAggregateIdentifier** annotation.
3. In the OrdersService – OrderSaga class, create a new private cancelProductReservation method with the ProductReservedEvent and reason as an argument. Write a code to create an instance of CancelProductReservationCommand and use **Axon's *CommandGateway*** to publish it.
4. Now publish from 4 places in the OrderSaga class:

```
70    @SagaEventHandler(associationProperty = "orderId")
71    public void handle(ProductReservedEvent productReservedEvent) {
72        // Process user payment
73        LOGGER.info("ProductReservedEvent is called for productId: " + productReservedEvent.getProductId() +
74                " and orderId: " + productReservedEvent.getOrderId());
75
76        FetchUserPaymentDetailsQuery fetchUserPaymentDetailsQuery =
77                new FetchUserPaymentDetailsQuery(productReservedEvent.getUserId());
78
79        User userPaymentDetails = null;
80        try {
81            userPaymentDetails = queryGateway.query(fetchUserPaymentDetailsQuery, ResponseTypes.instanceOf(User.cla
82        } catch (Exception ex) {
83            LOGGER.error(ex.getMessage());
84            //Start compensating transaction
85            cancelProductReservation(productReservedEvent, ex.getMessage());
86            return;
87        }
88
89        if (userPaymentDetails == null) {
90            //Start compensating transaction
91            cancelProductReservation(productReservedEvent, "Could not fetch user payment details");
92            return;
93        }
```

```
95        LOGGER.info("Successfully fetched user payment details for user " + userPaymentDetails.getFirstName());
96
97        ProcessPaymentCommand processPaymentCommand = ProcessPaymentCommand.builder()
98                .orderId(productReservedEvent.getOrderId())
99                .paymentDetails(userPaymentDetails.getPaymentDetails())
100               .paymentId(UUID.randomUUID().toString())
101               .build();
102
103       String result = null;
104       try {
105           result = commandGateway.sendAndWait(processPaymentCommand, 10, TimeUnit.SECONDS);
106       } catch (Exception ex) {
107           LOGGER.error(ex.getMessage());
108           // Start compensating transaction
109           cancelProductReservation(productReservedEvent, ex.getMessage());
110           return;
111       }
112
113       if(result == null) {
114           LOGGER.info("The ProcessPaymentCommand resulted in NULL. Initiating a compensating transaction");
115           // Start compensating transaction
116           cancelProductReservation(productReservedEvent, "Could not process used payment with provided payment de
117       }
118   }
```

5. In the ProductService – ProductAggregate class, create a **@CommandHandler** method that will handle the CancelProductReservationCommand and publish the ProductReservationCancelledEvent.

6. In the Core project, create the ProductReservationCancelledEvent class will have the following fields:

   private final String productId;
   private final int quantity;
   private final String orderId;
   private final String userId;
   private final String reason;

   Annotate this class with @Data and @Builder annotations and in the com.mphasis.core.events package.

7. In the ProductService – ProductAggregate class, also create an **@EventSourcingHandler** method which is handling the ProductReservationCancelledEvent and updating the quantity (addition action).

8. To make our read database up to date with the changes that we have just made to the product quantity, we will need to handle the ProductReservationCancelledEvent and update the read database as well.

9. In ProductService – ProductEventsHandler class, we need to add one more **@EventHandler** method for the ProductReservationCancelledEvent and update the Products projection.

10. Finally, we will Handle the ProductReservationCancelledEvent in OrderService – OrderSaga, by creating a new handler method annotated with **@SagaEventHandler** and **associationProperty** i.e., orderId.

```
@SagaEventHandler(associationProperty = "orderId")
public void handle(ProductReservationCancelledEvent productReservationCancelledEvent) {

    // Create and send a RejectOrderCommand
}
```

11. In the OrdersService project, create the RejectOrderCommand class annotated with @Value in the com.mphasis.command.commands package and should have the following fields:

    private final String orderId;
    private final String reason;

    Where:
    orderId – Annotated with **@TargetAggregateIdentifier** annotation.

12. Let's go to OrderSaga class inside the ProductReservationCancelledEvent handler method, write a code to create an instance of RejectOrderCommand and use **Axon's *CommandGateway*** to publish it.

OrderSaga.java

```
154
155     @SagaEventHandler(associationProperty = "orderId")
156     public void handle(ProductReservationCancelledEvent productReservationCancelledEvent) {
157
158         // Create and send a RejectOrderCommand
159         RejectOrderCommand rejectOrderCommand = new RejectOrderCommand(
160                 productReservationCancelledEvent.getOrderId(), productReservationCancelledEvent.getReason());
161
162         commandGateway.send(rejectOrderCommand);
163     }
164 }
165
```

13. Also publish the RejectOrderCommand from OrderSaga class inside the OrderCreatedEvent handler method if order is not created.

```
59
60          commandGateway.send(reserveProductCommand, new CommandCallback<ReserveProductCommand, Object>() {
61
62              @Override
63              public void onResult(CommandMessage<? extends ReserveProductCommand> commandMessage,
64                      CommandResultMessage<? extends Object> commandResultMessage) {
65
66                  if (commandResultMessage.isExceptional()) {
67                      // Start a compensating transaction
68                      RejectOrderCommand rejectOrderCommand = new RejectOrderCommand(
69                              orderCreatedEvent.getOrderId(), commandResultMessage.exceptionResult().getMessage());
70
71                      commandGateway.send(rejectOrderCommand);
72                  }
73              }
74          ));
75      }
```

14. In the OrdersService – OrderAggregate class, create a **@CommandHandler** method that will handle the RejectOrderCommand and publish the OrderRejectedEvent.
15. In the OrdersService project, create the OrderRejectedEvent class will have the following fields:

    private final String orderId;
    private final String reason;
    private final OrderStatus orderStatus = OrderStatus.REJECTED;

    Annotate this class with @Value annotations and in the com.mphasis.core.events package.
16. In the OrdersService – OrderAggregate class, should also have an **@EventSourcingHandler** method that sets value of the orderStatus field.
17. In OrdersService – OrderEventsHandler class should have **@EventHandler** method for the OrderRejectedEvent and persist the order details in **"read"** database.
18. Finally, let's go to the OrdersService – OrderSaga class, should have one **@SagaEventHandler** method with the **associationProperty "orderId"** that handles the OrderRejectedEvent and log the orderId.
19. This method should be annotated with **@EndSaga** which indicates the end of a Saga instance's lifecycle. When event handling completes, the Saga is destroyed and may no longer receive events.

```
@EndSaga
@SagaEventHandler(associationProperty = "orderId")
public void handle(OrderRejectedEvent orderRejectedEvent) {

    LOGGER.info("Successfully rejected order with id " + orderRejectedEvent.getOrderId());
}
```

**Run and make it work:**

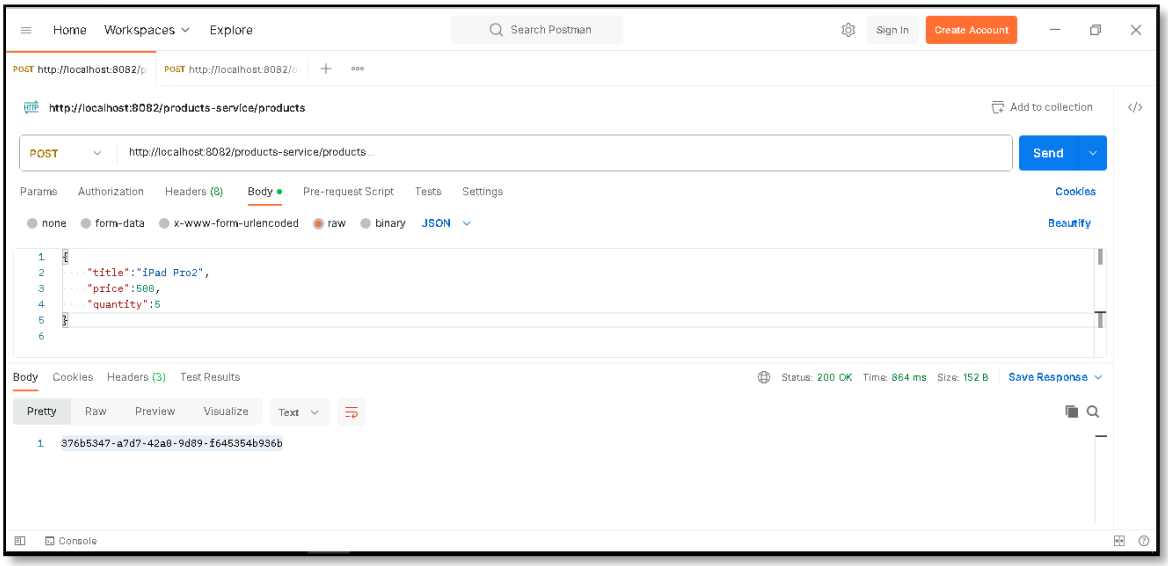
20. Run the Axon Server using Docker command.
21. Start the Discovery Server (Eureka Server), Product Service, OrdersService, UsersService, PaymentsService and ApiGateway.

22. Add logs to handler in ProductEventsHandler class:

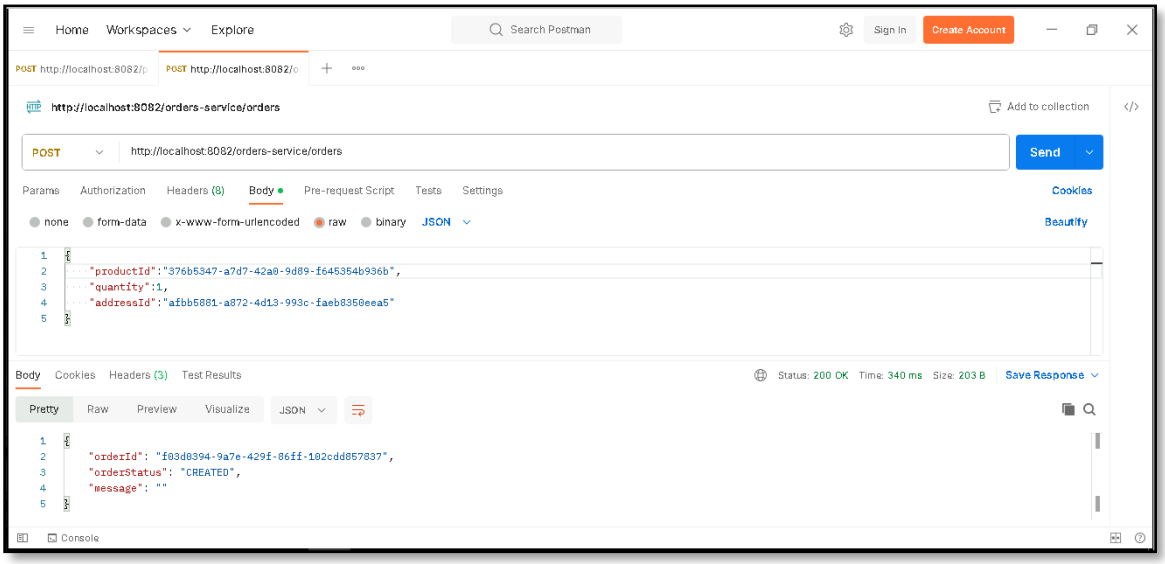
```
54    @EventHandler
55    public void on(ProductReservedEvent productReservedEvent) throws Exception {
56
57        ProductEntity productEntity = productRepository.findByProductId(productReservedEvent.getProductId());
58
59        LOGGER.info("ProductReservedEvent: Current product quantity " + productEntity.getQuantity());
60
61        productEntity.setQuantity(productEntity.getQuantity() - productReservedEvent.getQuantity());
62        productRepository.save(productEntity);
63
64        LOGGER.info("ProductReservedEvent: New product quantity " + productEntity.getQuantity());
65
66        LOGGER.info("ProductReservedEvent is called for productId:" + productReservedEvent.getProductId()+
67                " and orderId: " + productReservedEvent.getOrderId());
68    }
69
```

```
70    @EventHandler
71    public void on(ProductReservationCancelledEvent productReservationCancelledEvent) throws Exception {
72
73        ProductEntity currentlyStoredProduct =
74                productRepository.findByProductId(productReservationCancelledEvent.getProductId());
75
76        LOGGER.info("ProductReservationCancelledEvent: Current product quantity "
77                + productReservationCancelledEvent.getQuantity());
78
79        int newQuantity = currentlyStoredProduct.getQuantity() + productReservationCancelledEvent.getQuantity();
80        currentlyStoredProduct.setQuantity(newQuantity);
81
82        productRepository.save(currentlyStoredProduct);
83
84        LOGGER.info("ProductReservationCancelledEvent: New product quantity "
85                + productReservationCancelledEvent.getQuantity());
86    }
87 }
88
```

23. Send a POST request to Create Product.

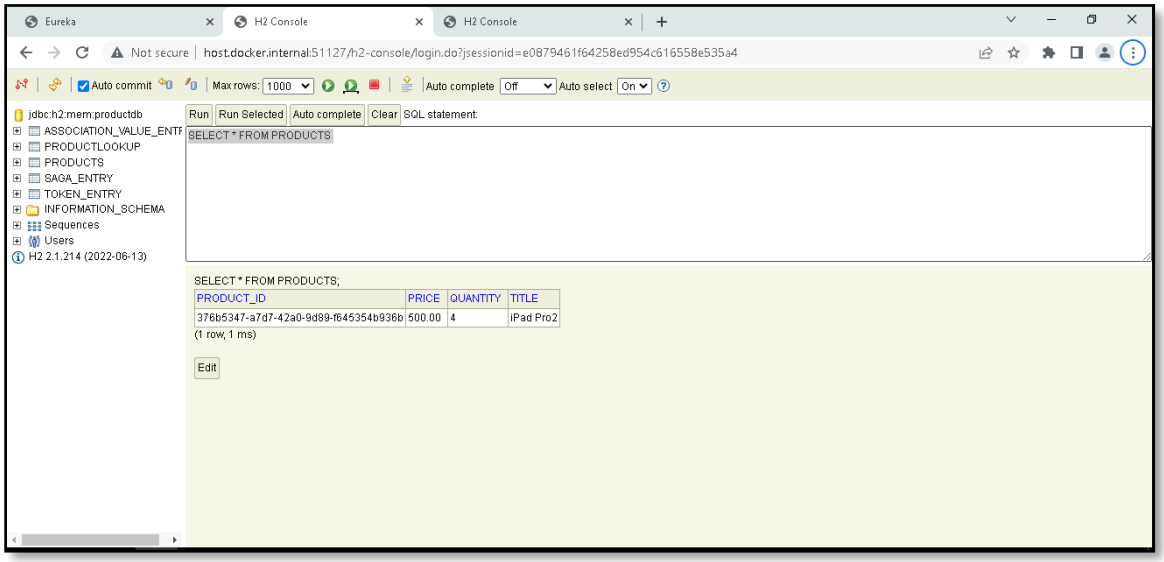


24. Copy the productId, use it in /orders-service/orders. Send a POST request to Create Order.

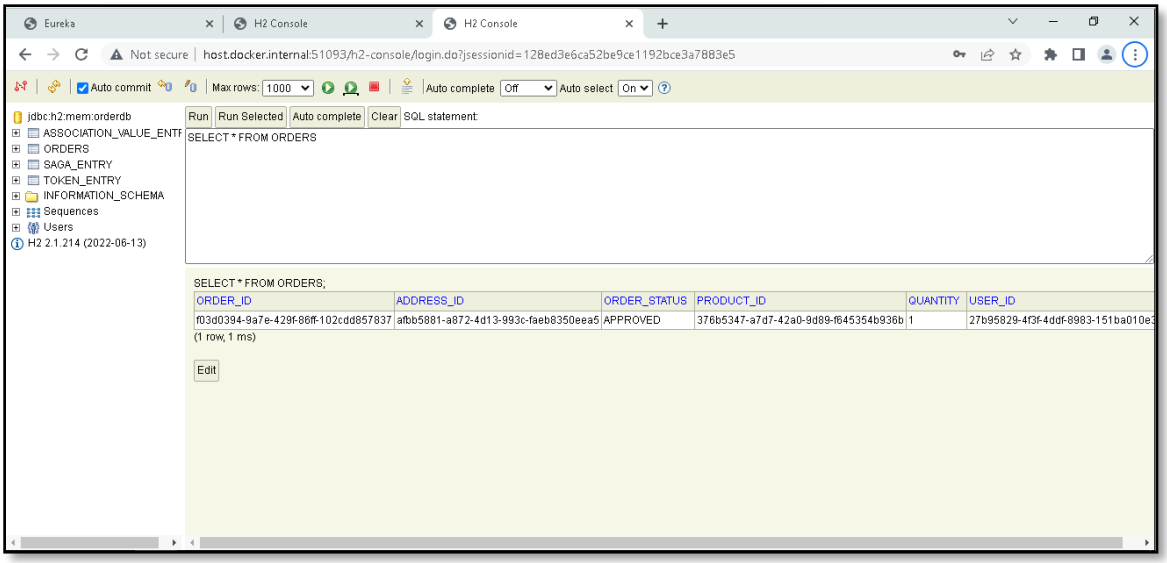



25. Check the Order approved log message on the OrdersService console.

26. Using the /h2-console connect to the Products database and make sure that the product details are stored there as well. Also, check the Quantity after the order is placed successfully.



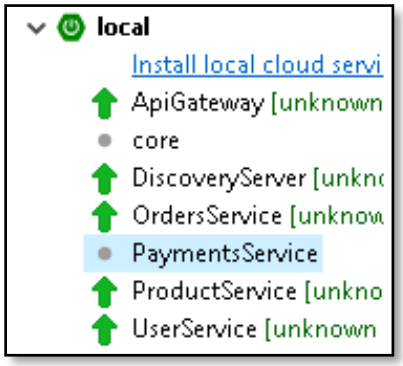
27. Using the /h2-console connect to the Orders database and make sure that the order details are stored there as well.




28. Check the Product Quantity logs messages on the ProductService console.


29. We will now Stop PaymentsService and place a new Order. A new Order reduces the number of products in stock by one more item. Because PaymentsService is unavailable and will be unable to accept user payment, it should kick in the **compensating transaction** in our **OrderSaga,** which will undo changes done by the proceeding transactions. Finally, we will notice the Product Quantity in the Product database has been increased once more.

30. Let's stop the PaymentsService:




31. Let's place the order again, to the same productId.

32. Using the /h2-console connect to the Orders database and make sure that the **rejected** order details are stored there as well.

| ADDRESS_ID | ORDER_STATUS |
|---|---|
| afbb5881-a872-4d13-993c-faeb8350eea5 | APPROVED |
| afbb5881-a872-4d13-993c-faeb8350eea5 | REJECTED |

33. Using the /h2-console connect to the Products database and check the Quantity after the order is **Rejected**.