



# Java Microservices Hands-on Mastery - S1

Manpreet Singh Bindra  
Senior Manager





## Do's and Don't

- Login to GTW session on Time
- Login with your Mphasis Email ID only
- Use the question window for asking any queries



# Welcome

1. Skill - Proficiency Introduction
2. About Me - Introduction
3. Walkthrough the Skill on TalentNext



## Overall Agenda

- Understanding CQRS and Event Sourcing Pattern
- Implementing CQRS with Event Sourcing with Axon Framework
- Distributed Transaction in Java Microservices using SAGA Pattern
- Introduce Docker and state its benefit over VM
- Understanding the Architecture of Docker and terminologies
- Describe what is Container in Docker, why to use it, and its scopes
- Deploy Microservice in Docker
- Deploy Microservice with H2 to AWS Fargate
- Implement Centralized Configuration Management with AWS Parameter Store
- Implement Auto Scaling and Load Balancing with AWS Fargate



# Day - 1

- Microservice vs Monolithic application
- Event-Driven Microservices
- Transactions in Microservices
- Choreography-Based Saga
- Orchestration-Based Saga
- Command Query Responsibility Segregation
- Types of Messaging in CQRS Pattern
- CQRS and Event Sourcing
- Building microservices with Spring Boot



Day - 1

# Microservice vs Monolithic Application



## What are Microservice?

- Microservices are a software development technique – a variant of the service-oriented architecture (SOA) architectural style that structures an application as a collection of loosely coupled services...
- In a microservices architecture services are fine-grained...
- The benefits of decomposing an application into different smaller services is that it improves modularity. This makes the application easier to understand, develop, test, and become more resilient to architecture erosion.
- It parallelized development by enabling small autonomous teams to develop, deploy and scale their respective services independently.

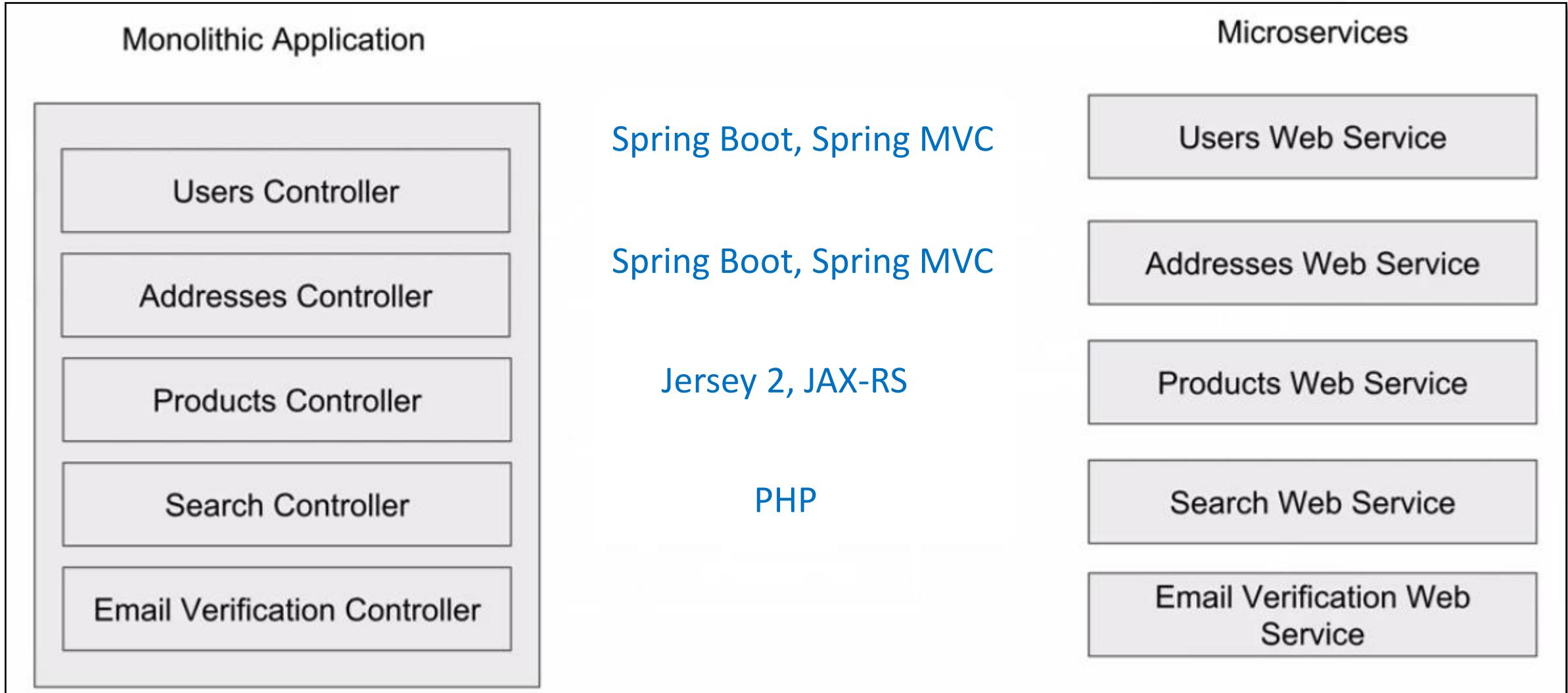


## What are Microservice?

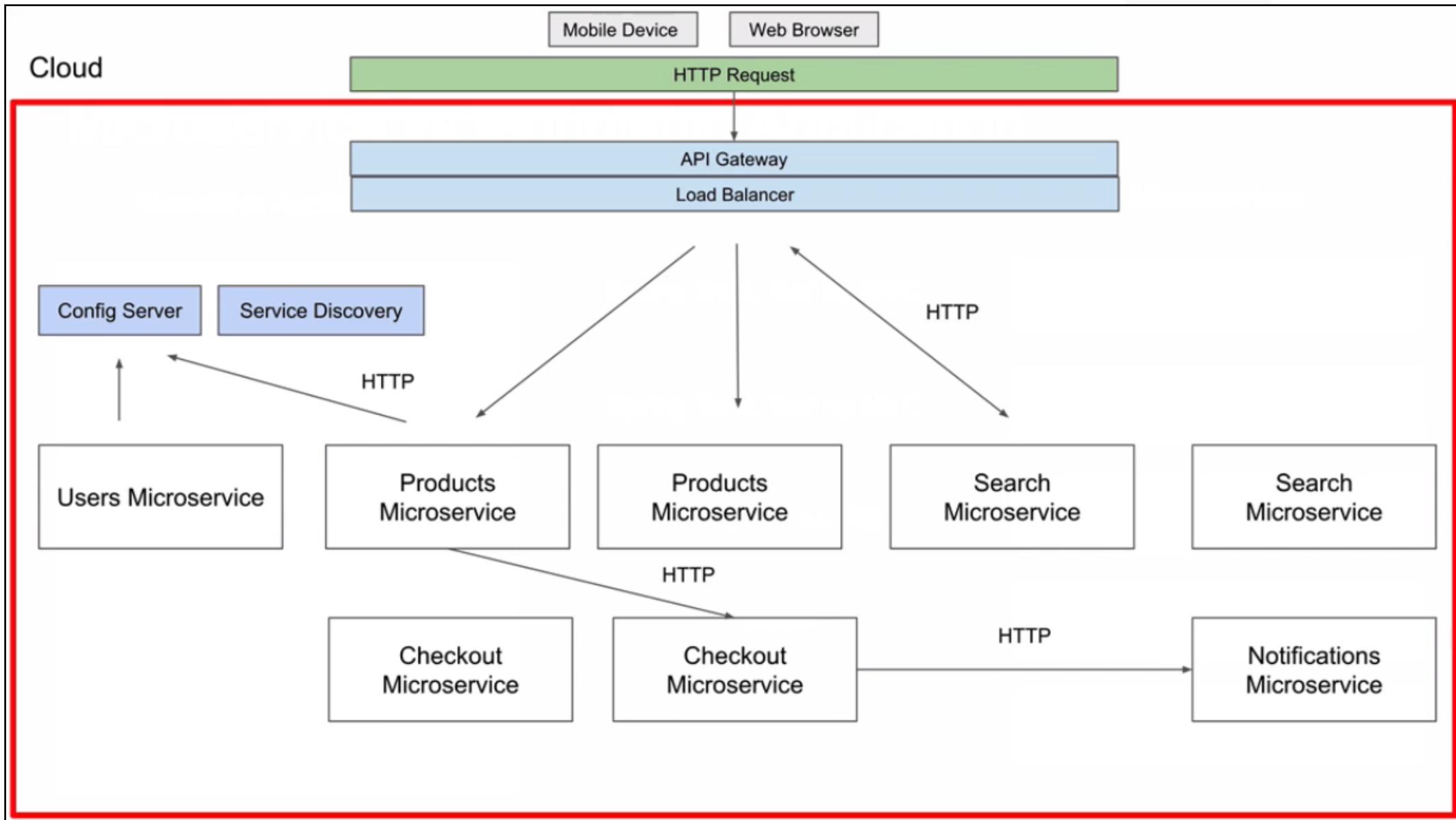
- A Web Service
- Small and Responsible for one thing (Search, Password Reset, Email Verification)
- Configured to work in the cloud and is easily scalable.



# Microservice vs Monolithic Application



# Application Diagram



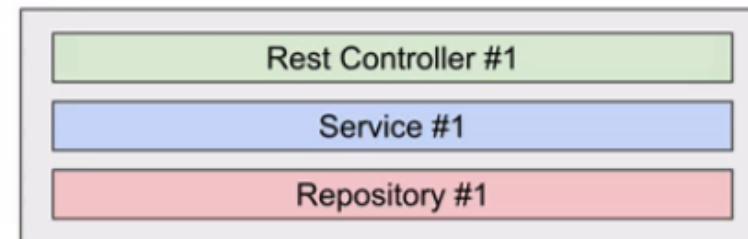


# Rest Controller - Services

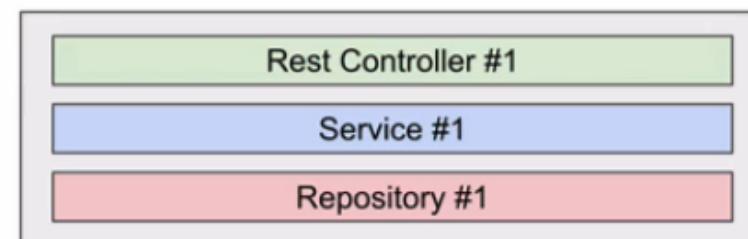
## Users



## Addresses



## Search

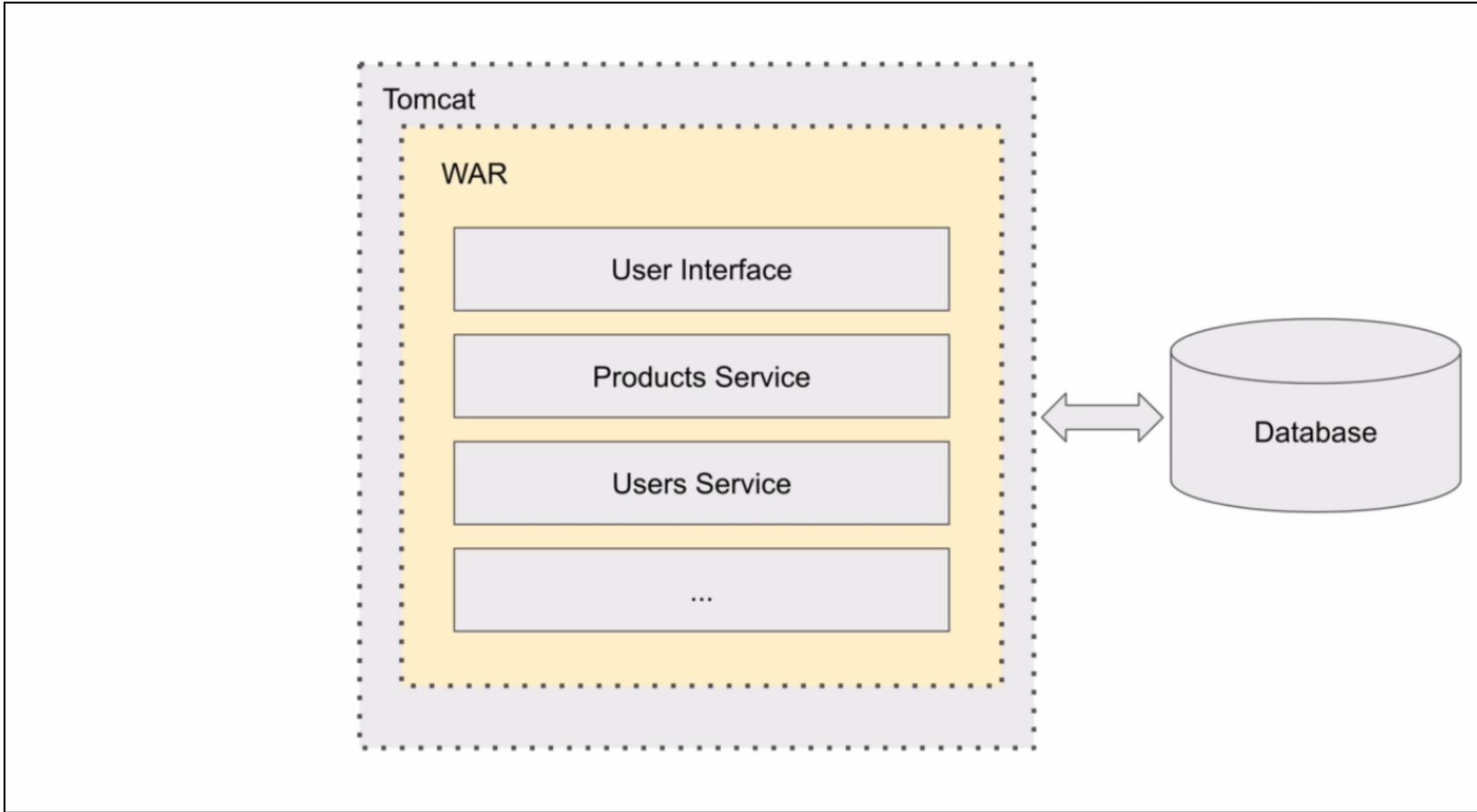




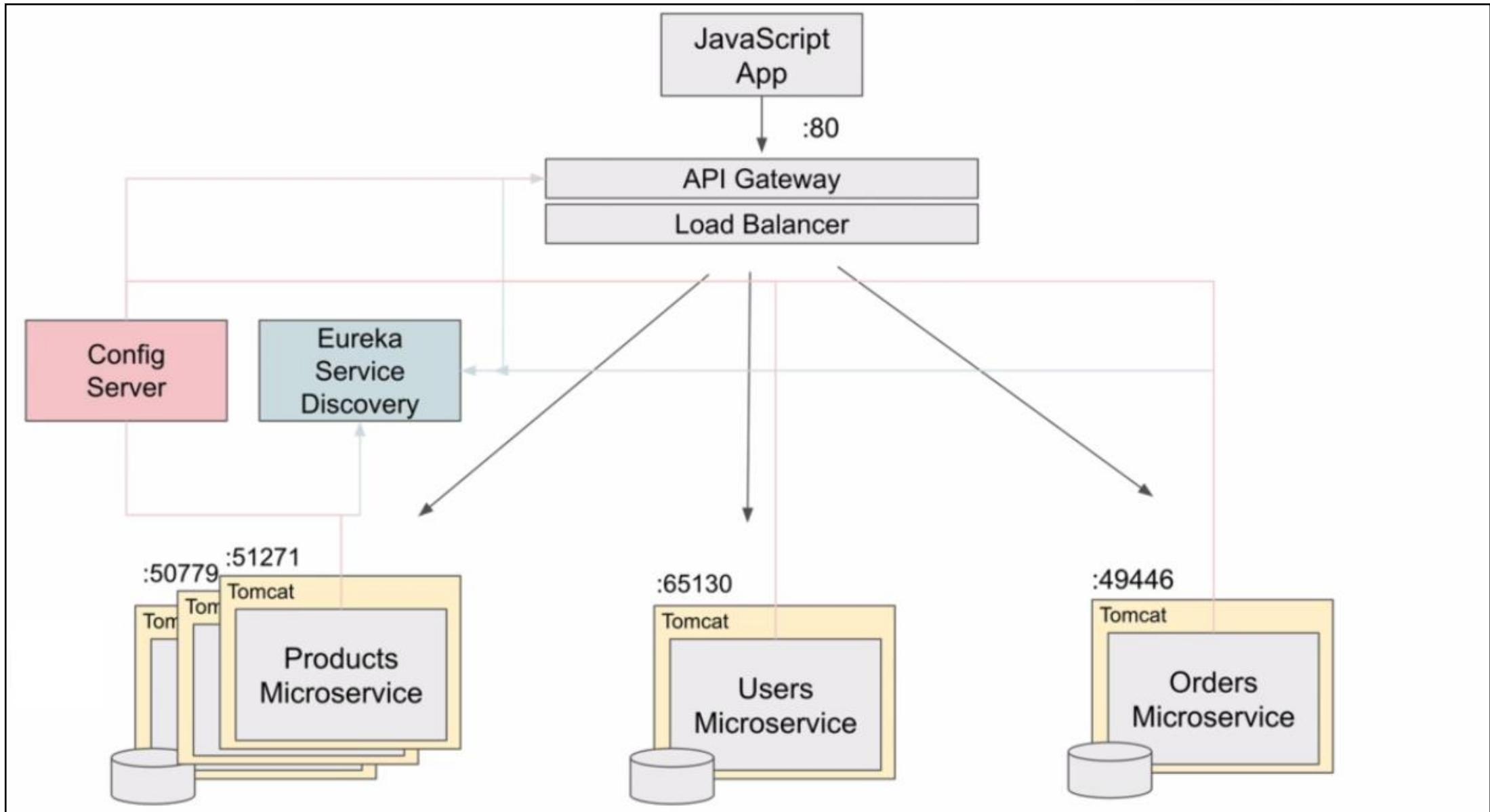
Day - 1

# Microservices Architecture Overview

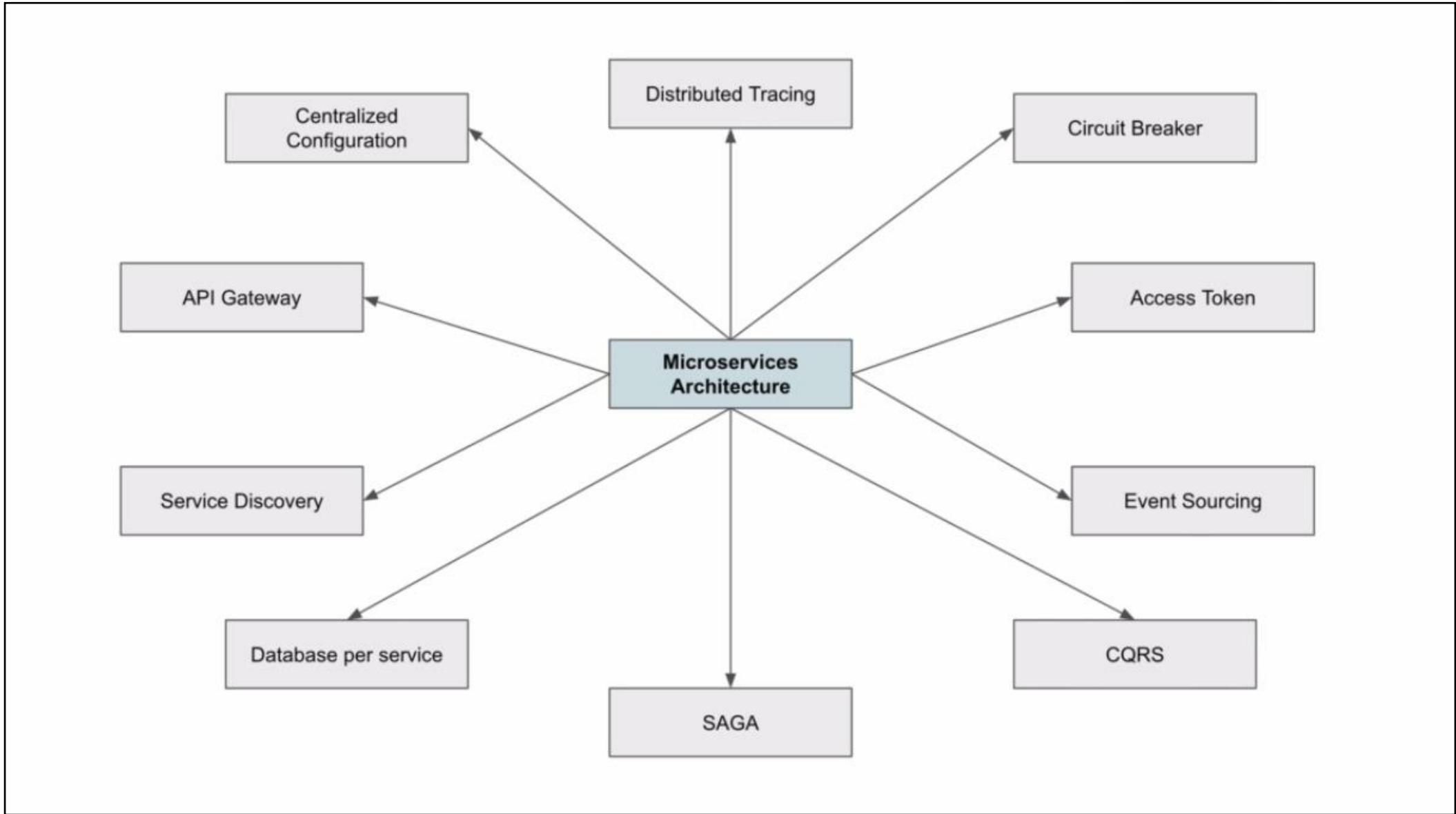
# Monolithic Architecture



# Microservices Architecture



# Pattern Diagram





Day - 1

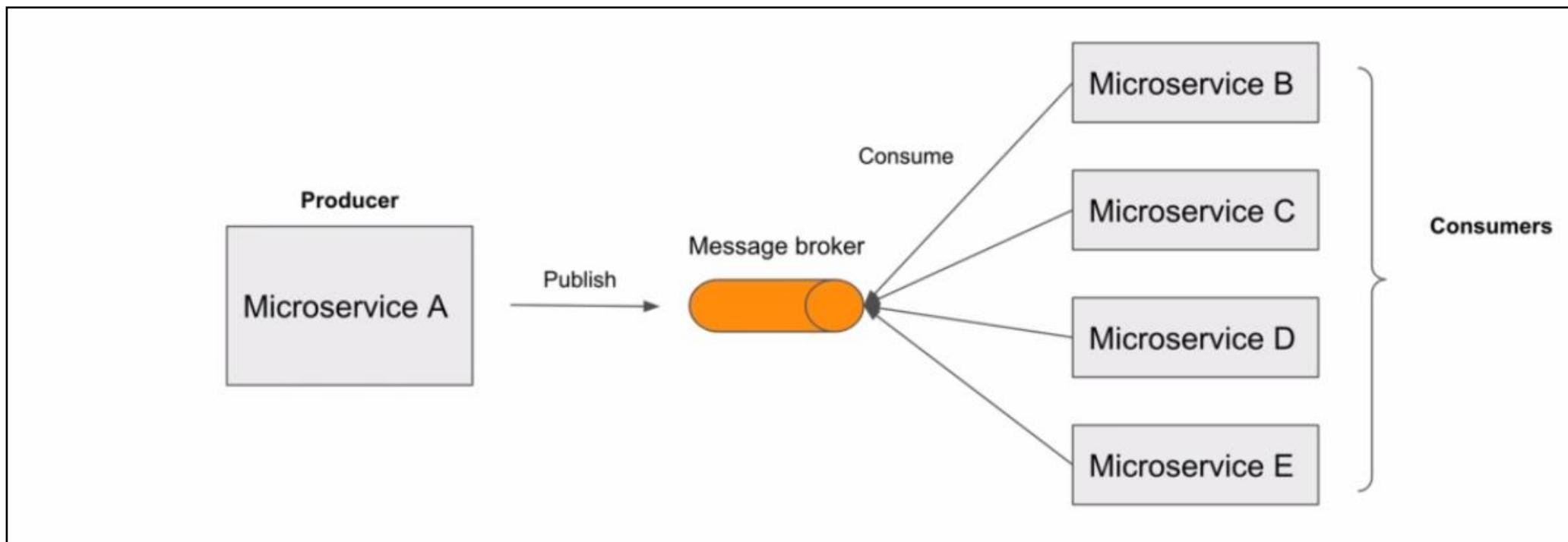
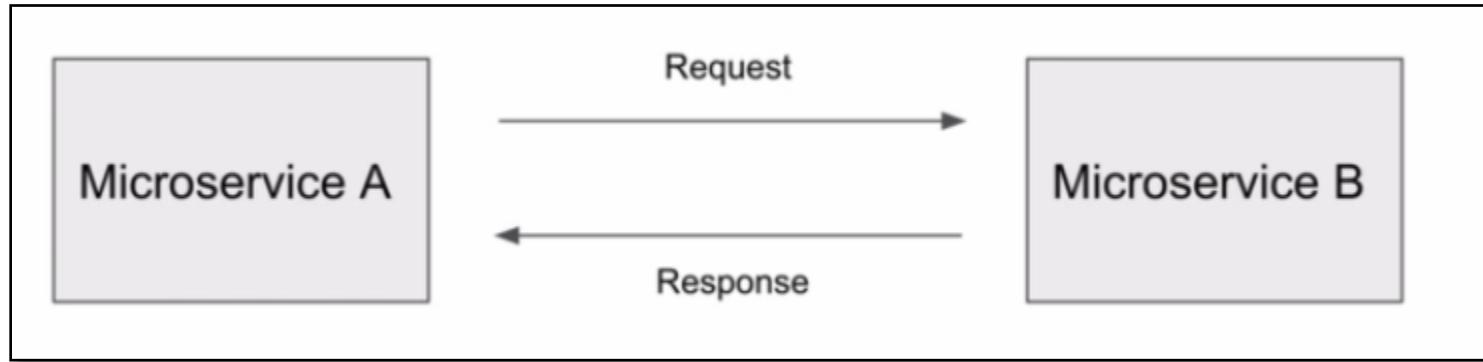
# Event-Driven Microservices



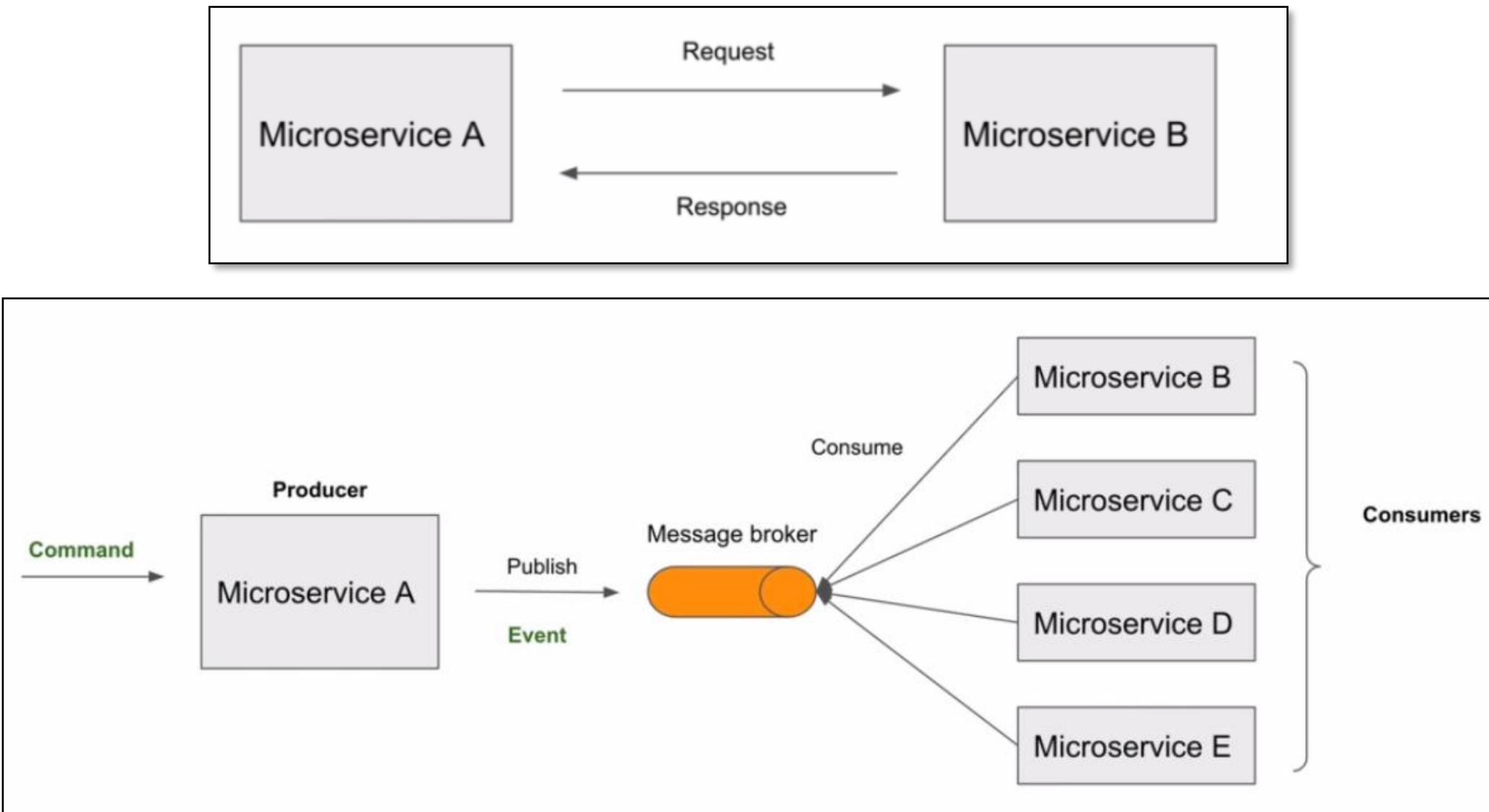
## Event-Driven Microservices

- It's asynchronous.
- If your microservices are simple, follow request-response approach.
- If you don't know how many microservices can be added/removed as consumer, we can prefer the event-driven microservices/message-driven microservices.

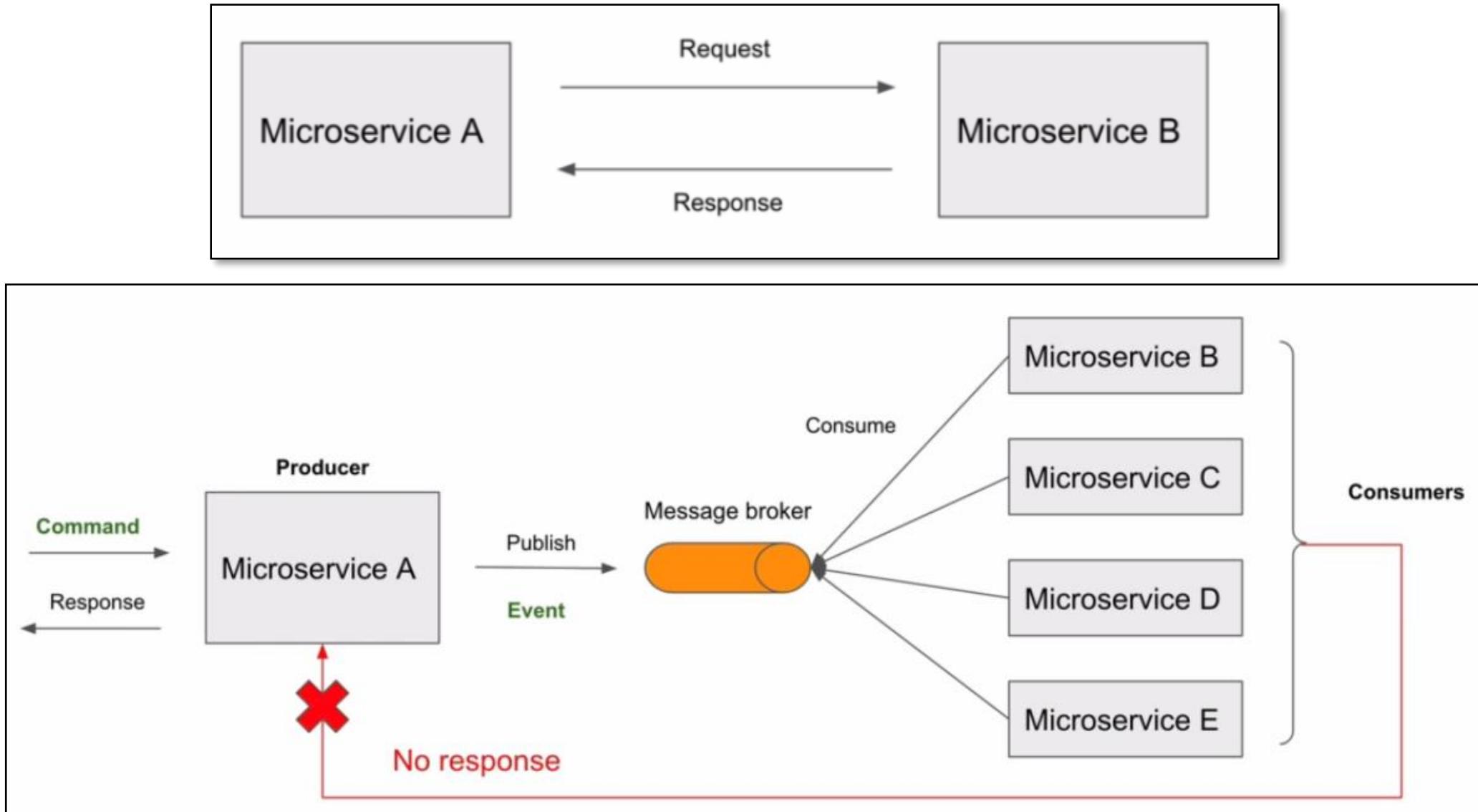
# Event-Driven Microservices



# Event-Driven Microservices



# Event-Driven Microservices

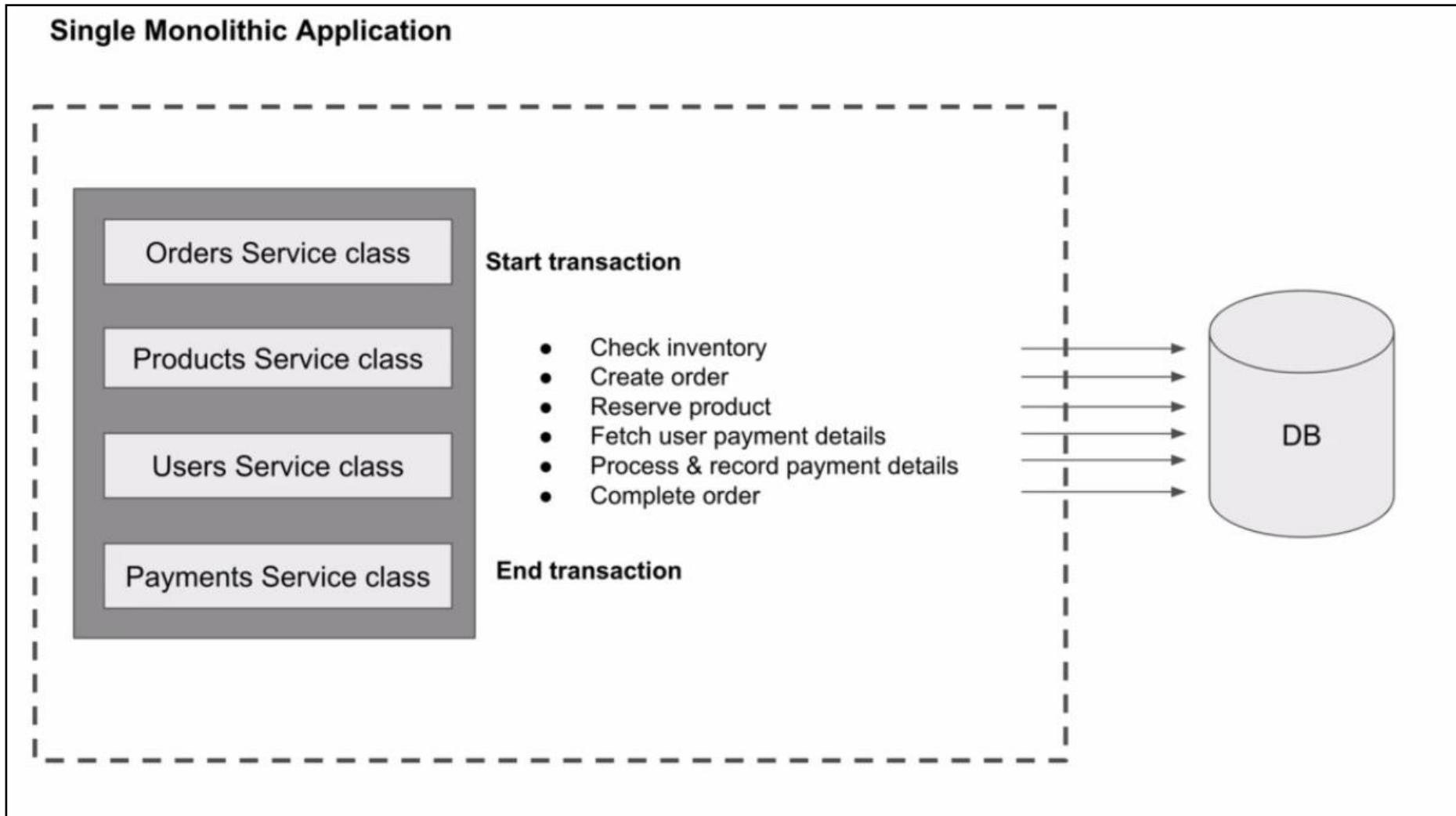




Day - 1

# Transactions in Microservices

- In monolithic applications, implementing the transaction is very easy.

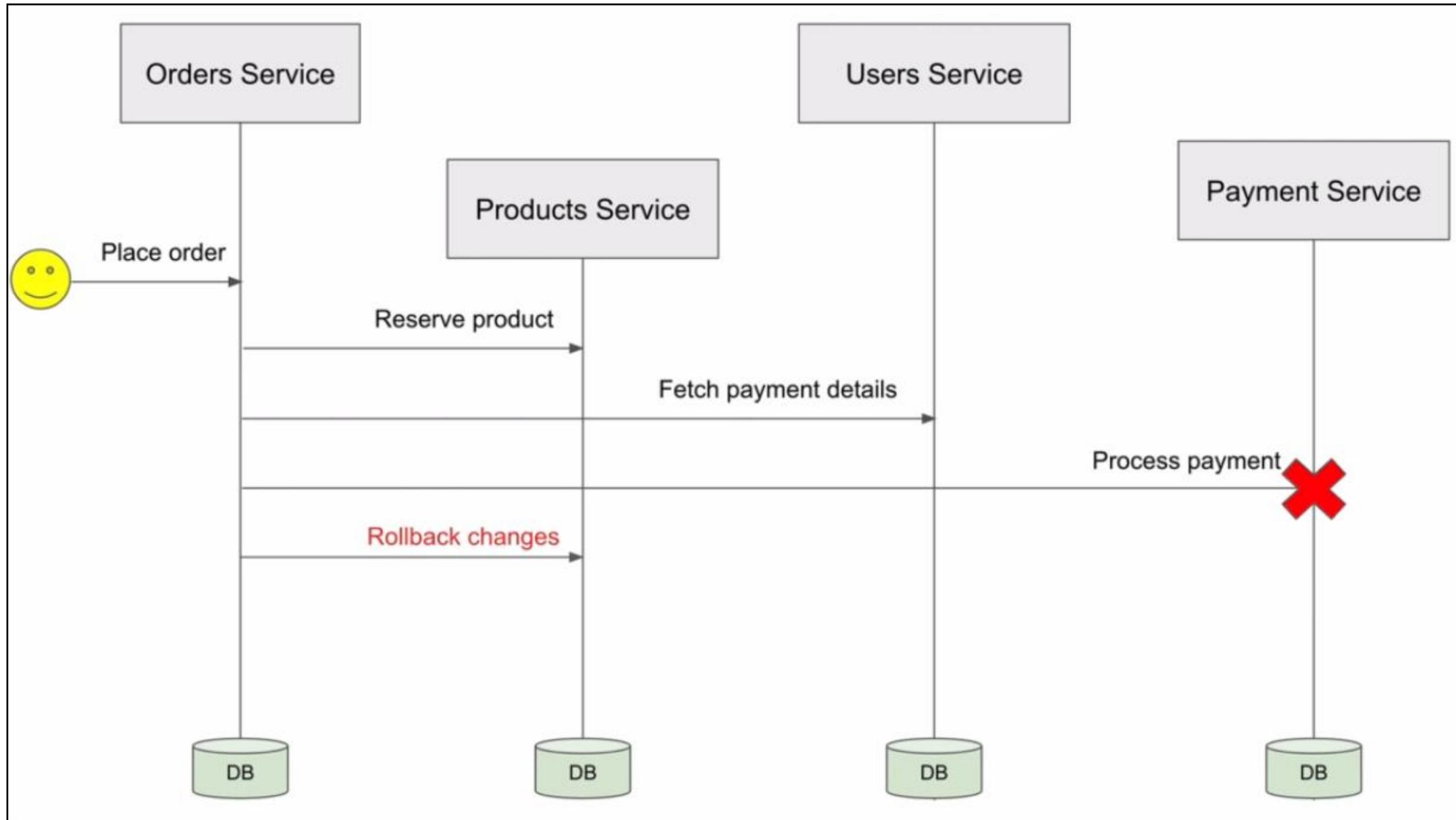




## ACID Transactions

- Atomicity
- Consistency
- Isolation
- Durability

# Transactions in Microservices





## SAGA Design Pattern

- Is a way to manage data consistency across microservices in distributed transactions scenarios.
- There are two different ways to implement SAGA patterns:
  - Choreography-Based
  - Orchestration-Based

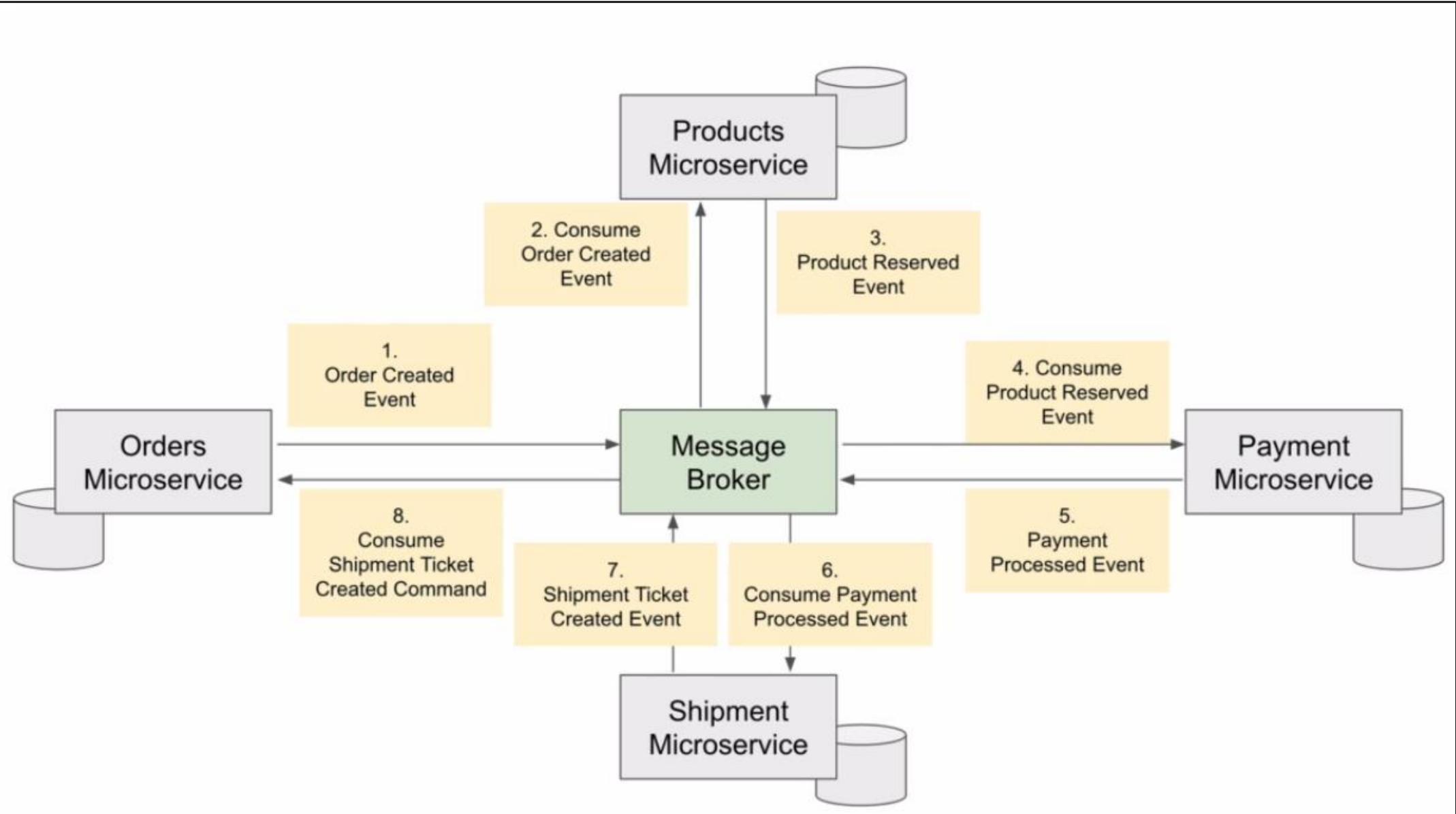


Day - 1

# Choreography-Based SAGA

- In Choreography-Based SAGA, microservices communicate with each other by exchanging events.
- When microservices perform operations, it publishes the event message to a messaging system like Message Broker.

# Choreography-Based SAGA

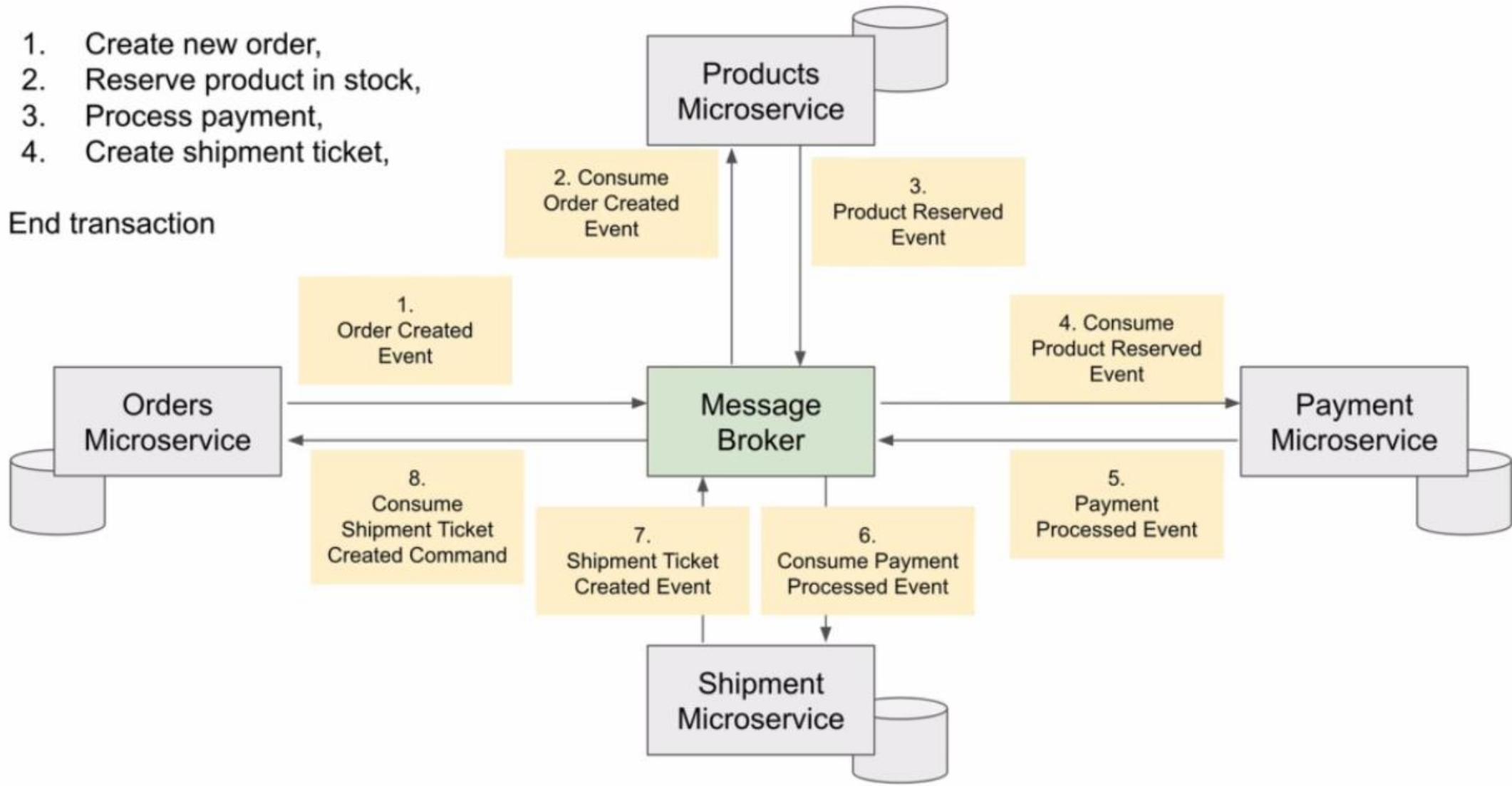


# Choreography-Based SAGA

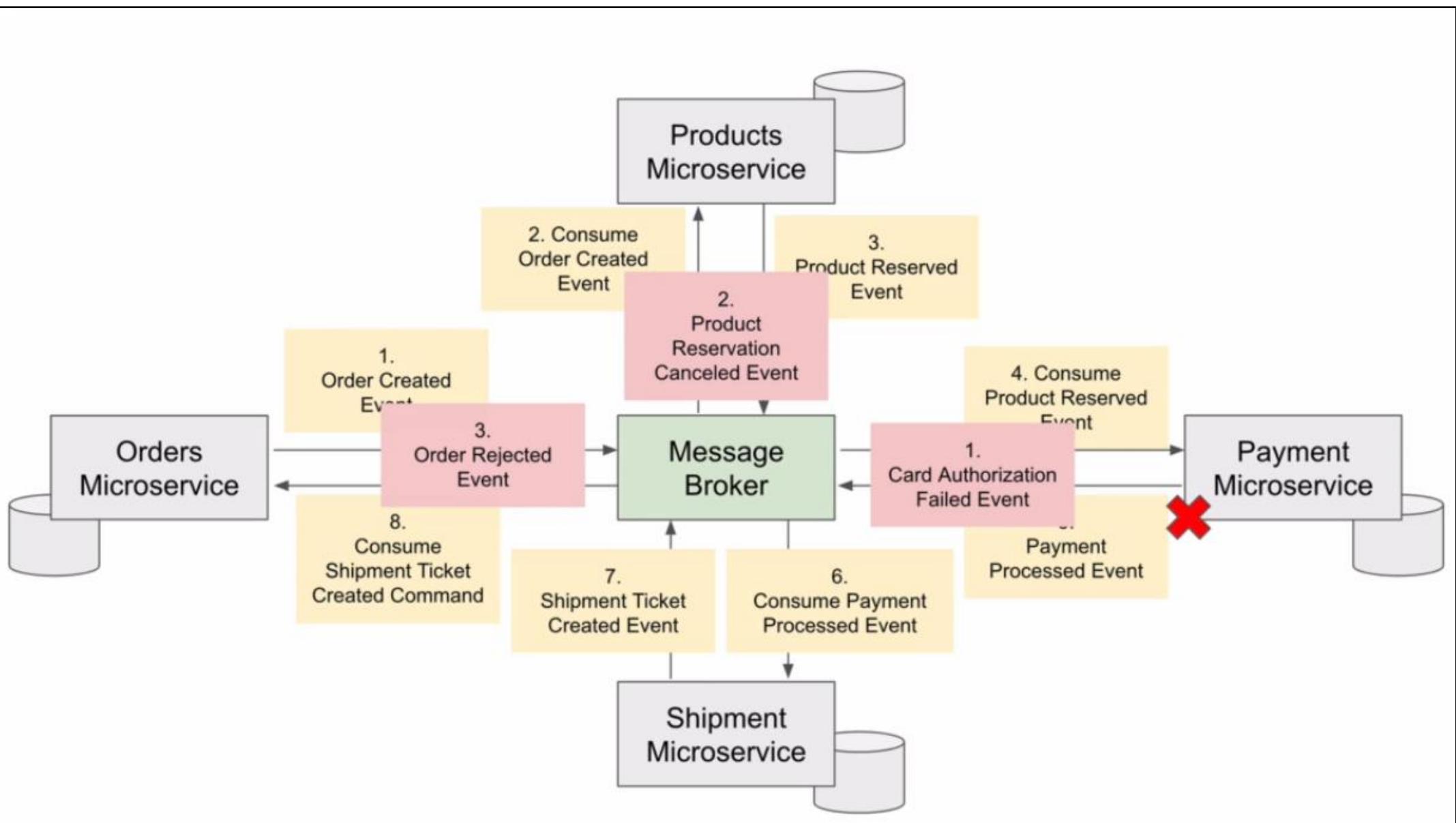
## Begin Transaction

1. Create new order,
2. Reserve product in stock,
3. Process payment,
4. Create shipment ticket,

## End transaction



# Choreography-Based SAGA



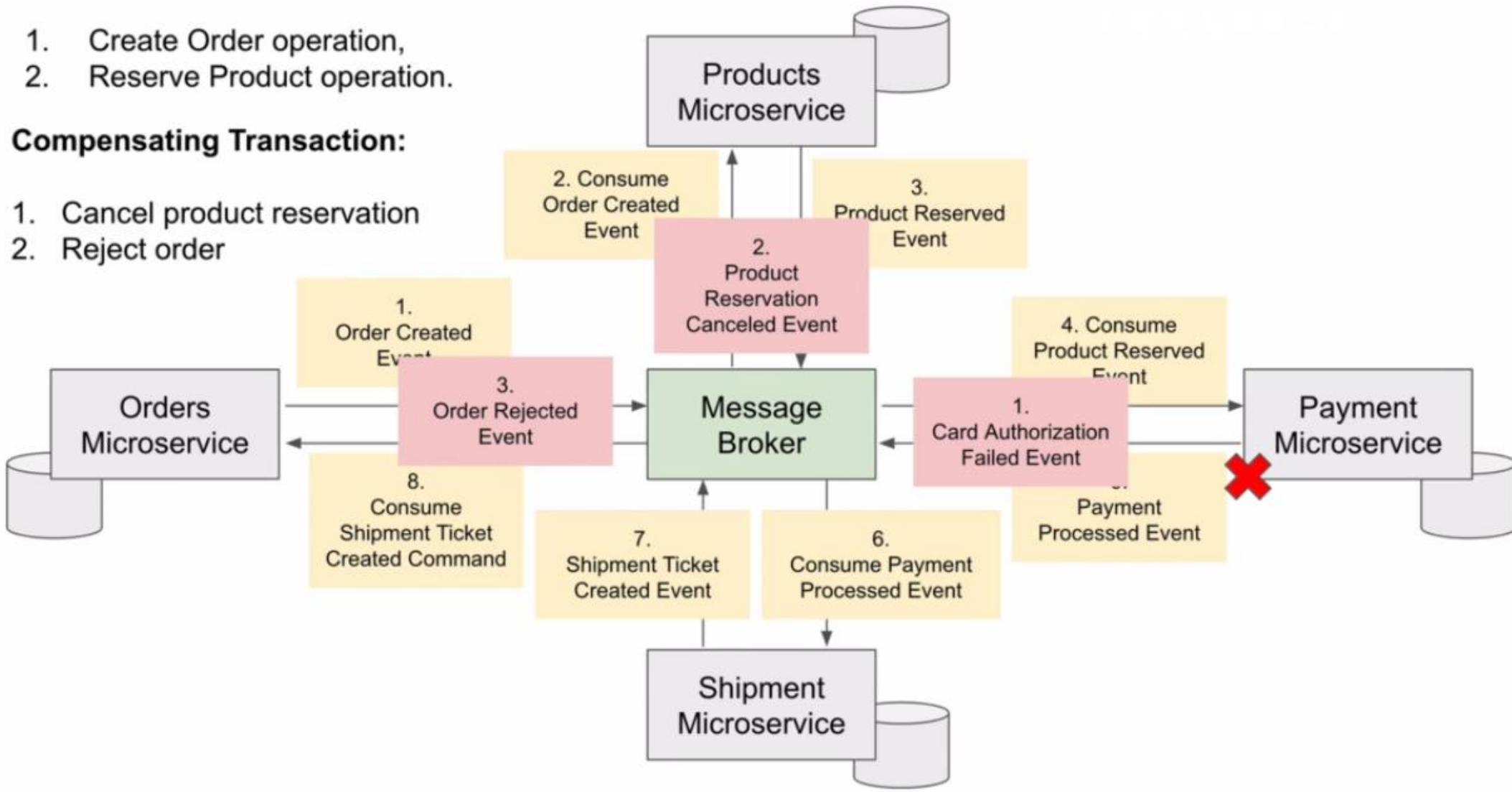
# Choreography-Based SAGA

## Initial Flow:

1. Create Order operation,
2. Reserve Product operation.

## Compensating Transaction:

1. Cancel product reservation
2. Reject order



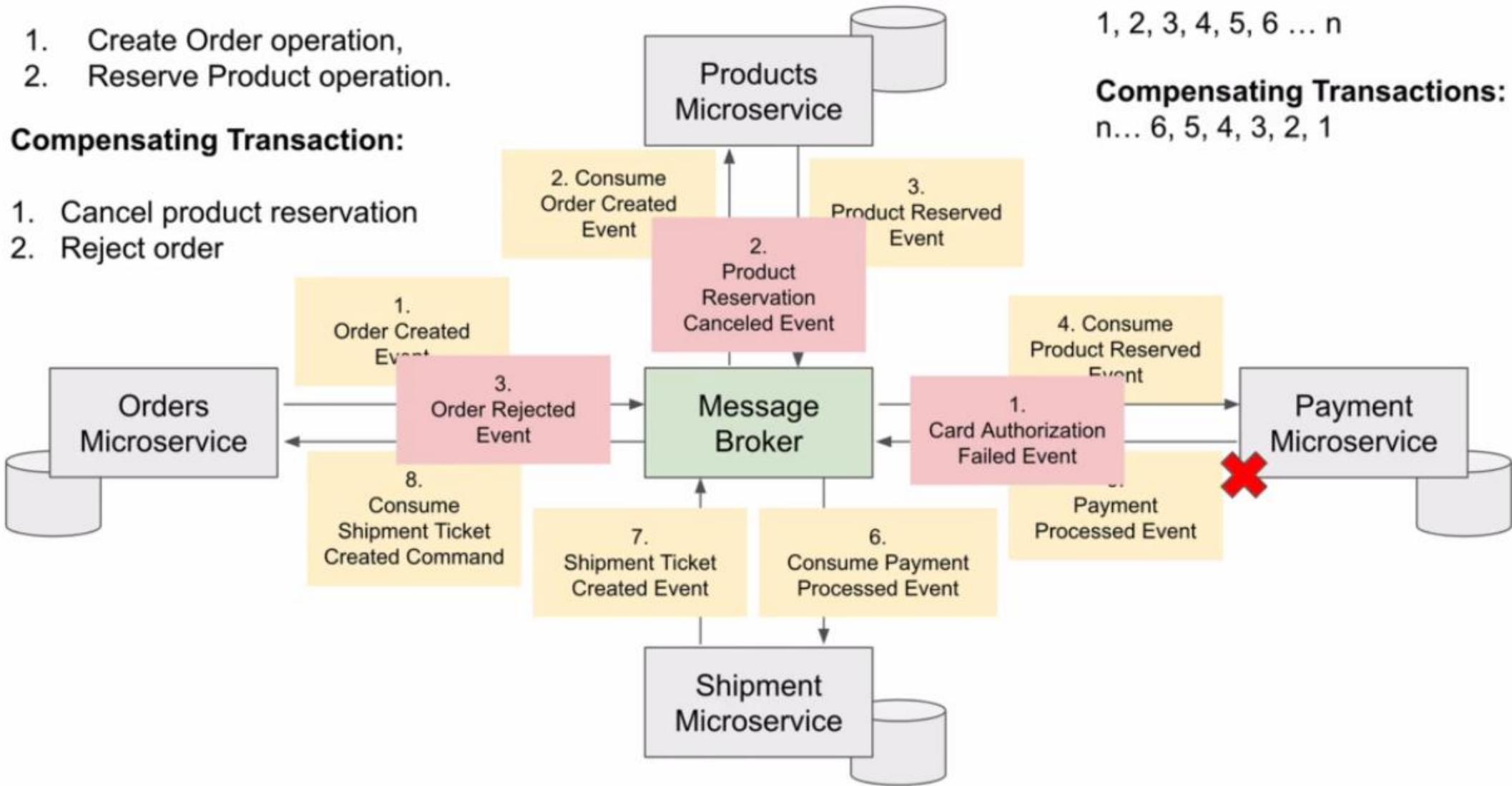
# Choreography-Based SAGA

## Initial Flow:

1. Create Order operation,
2. Reserve Product operation.

## Compensating Transaction:

1. Cancel product reservation
2. Reject order



## Initial Flow:

1, 2, 3, 4, 5, 6 ... n

## Compensating Transactions:

n... 6, 5, 4, 3, 2, 1



## Choreography-Based SAGA

- In Choreography-Based SAGA, each microservice publishes the main event that triggers event in another microservice. You can think of these events as one transaction contains multiple local transactions



Day - 1

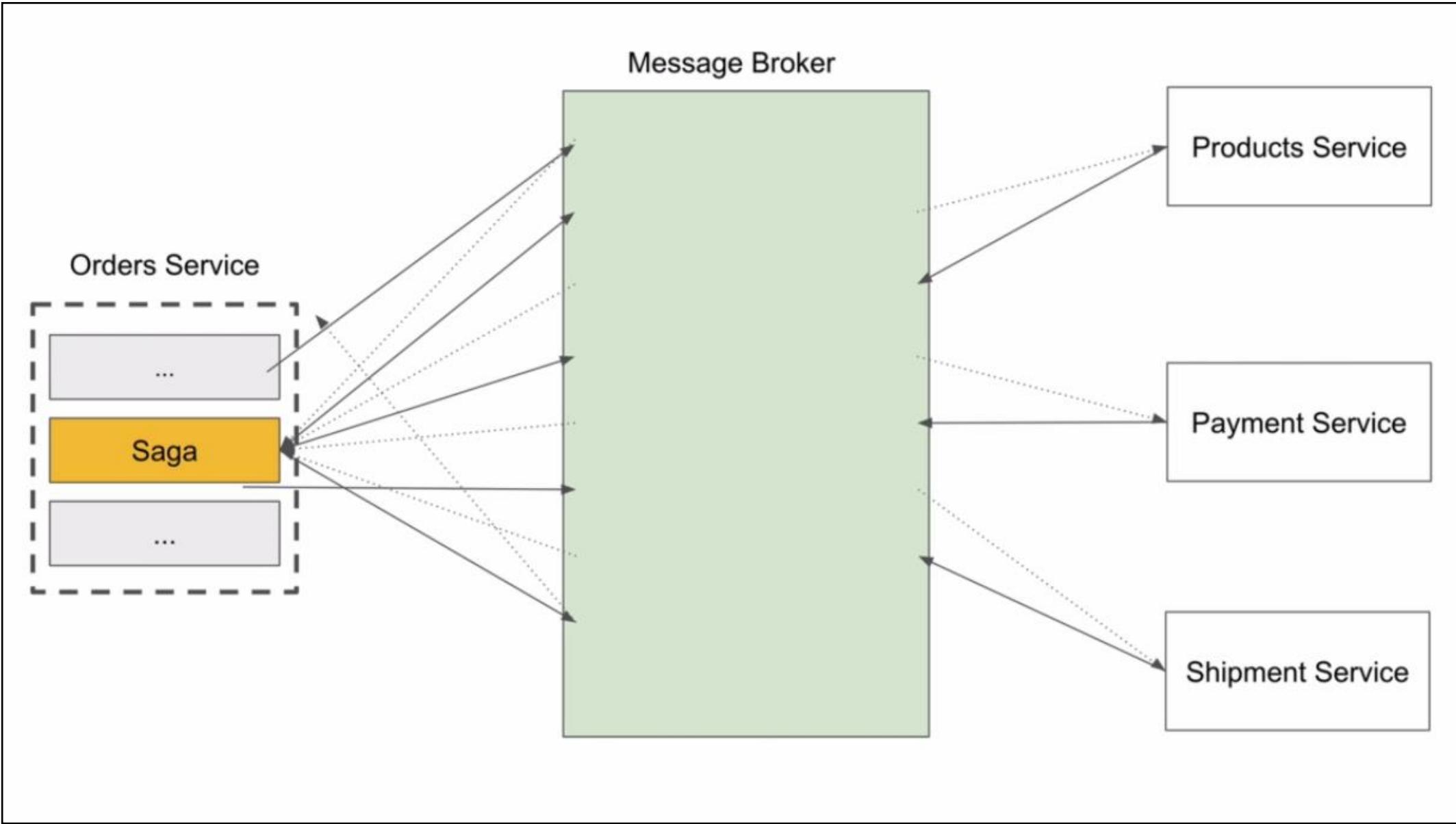
# Orchestration-Based SAGA



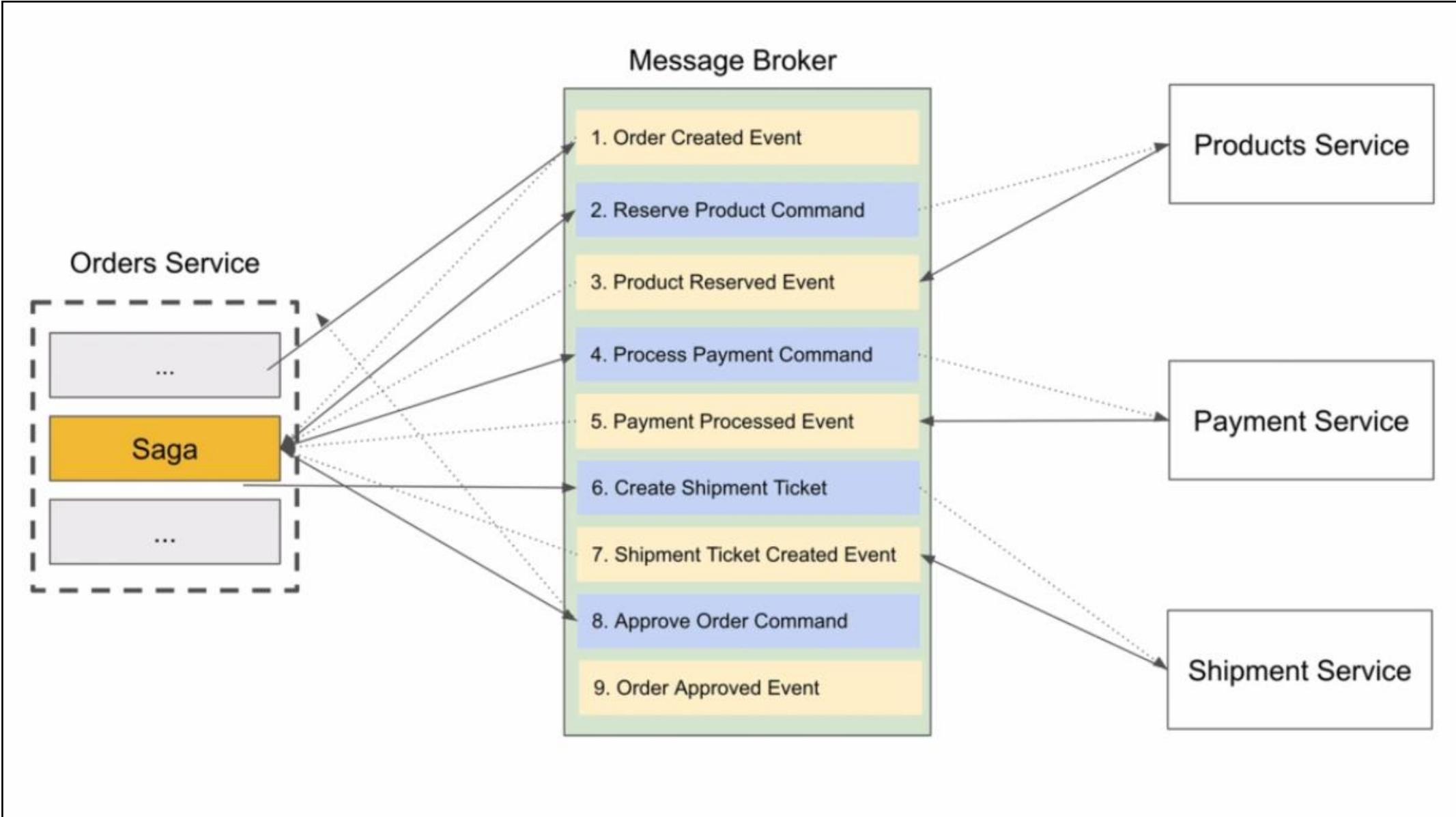
## Orchestration-Based SAGA

- In Orchestration-Based SAGA, an orchestrator (object) tells the participants what local transactions to execute.

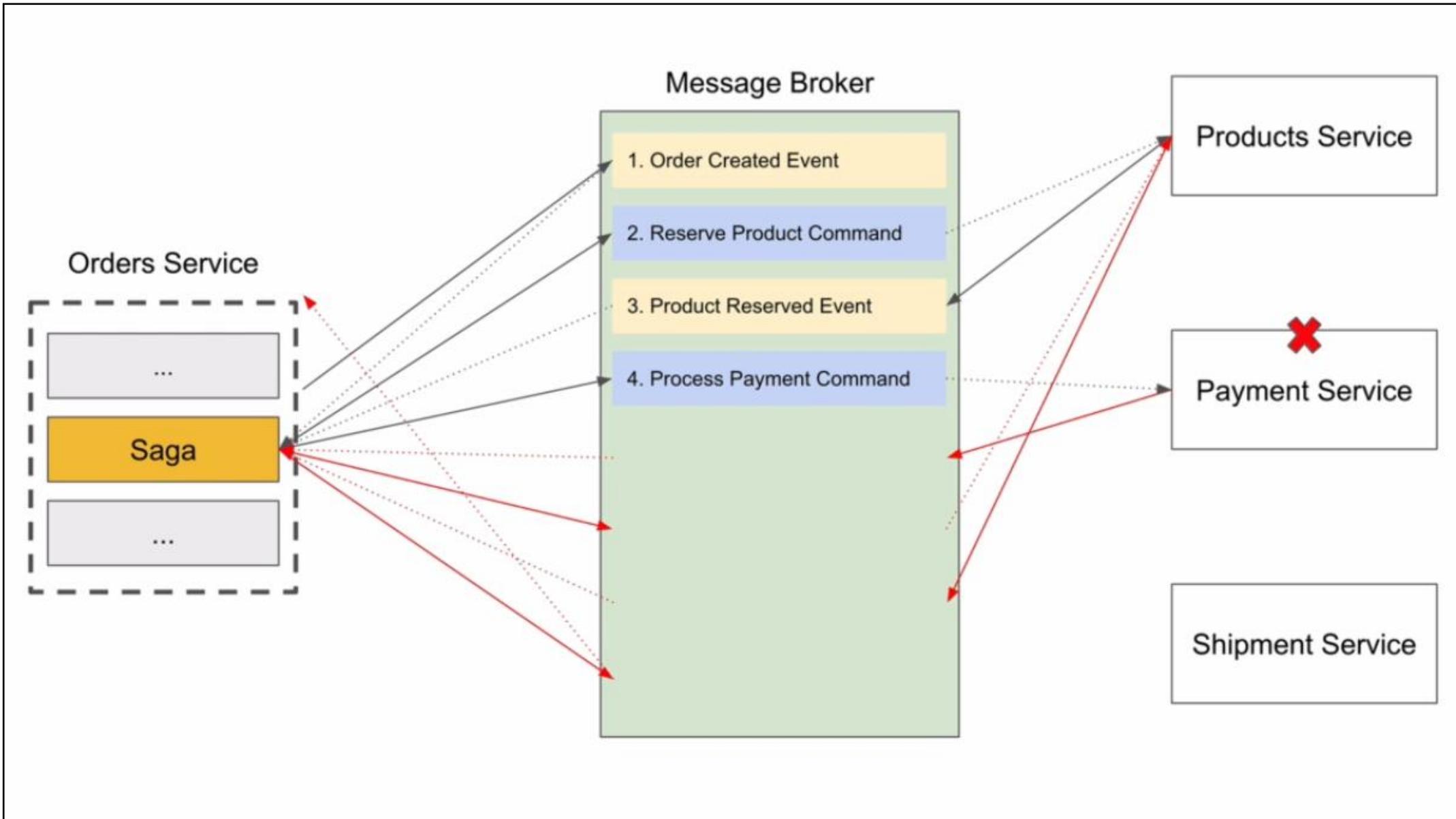
# Orchestration-Based SAGA



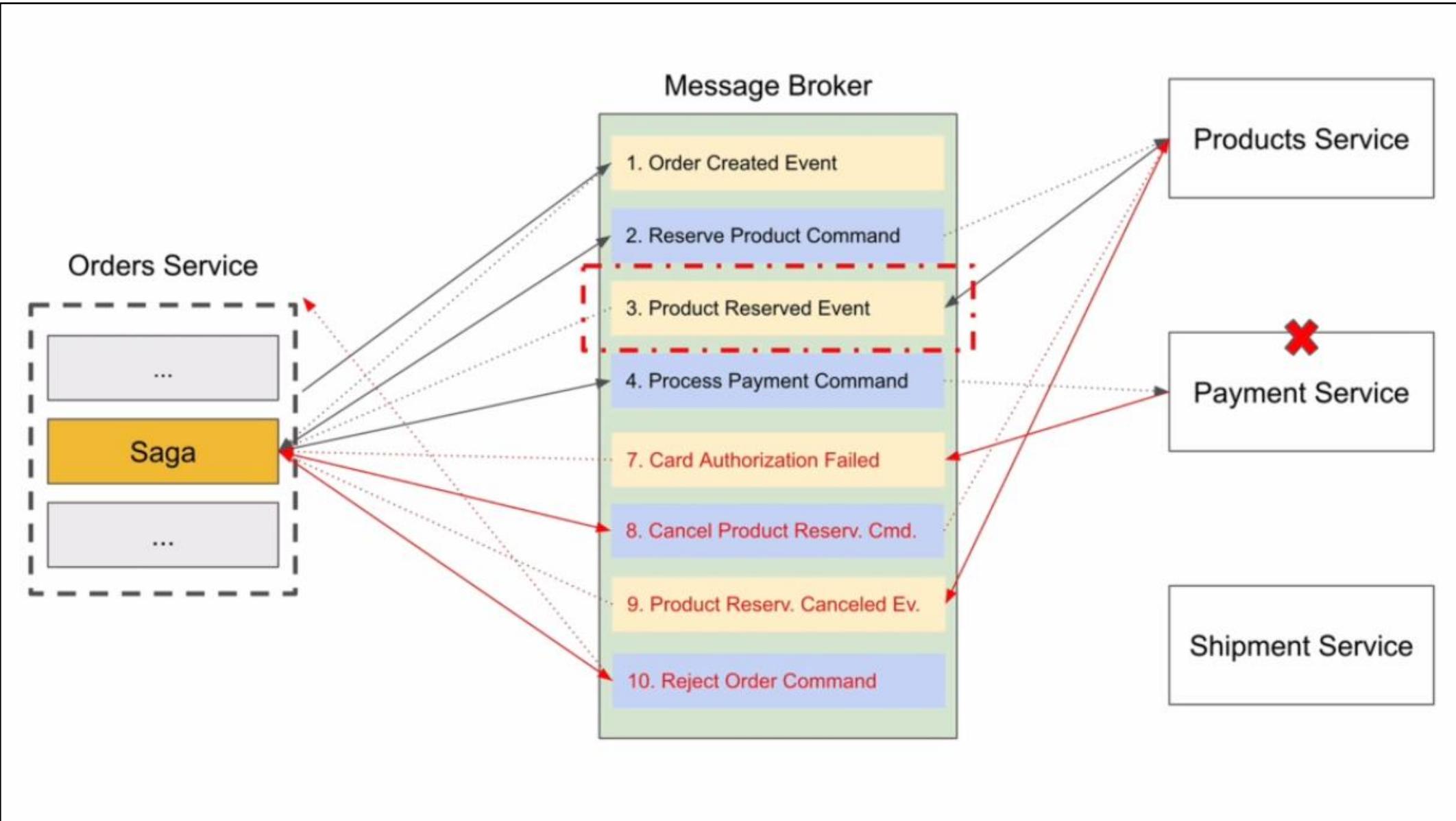
# Orchestration-Based SAGA



# Orchestration-Based SAGA



# Orchestration-Based SAGA





## Orchestration-Based SAGA

- SAGA will publish a Command, Microservices will consume this Command – process it and Publish an Event. SAGA will consume that Event and will decide what to do next. It can end SAGA or continue the process in the flow by publishing a New Command.

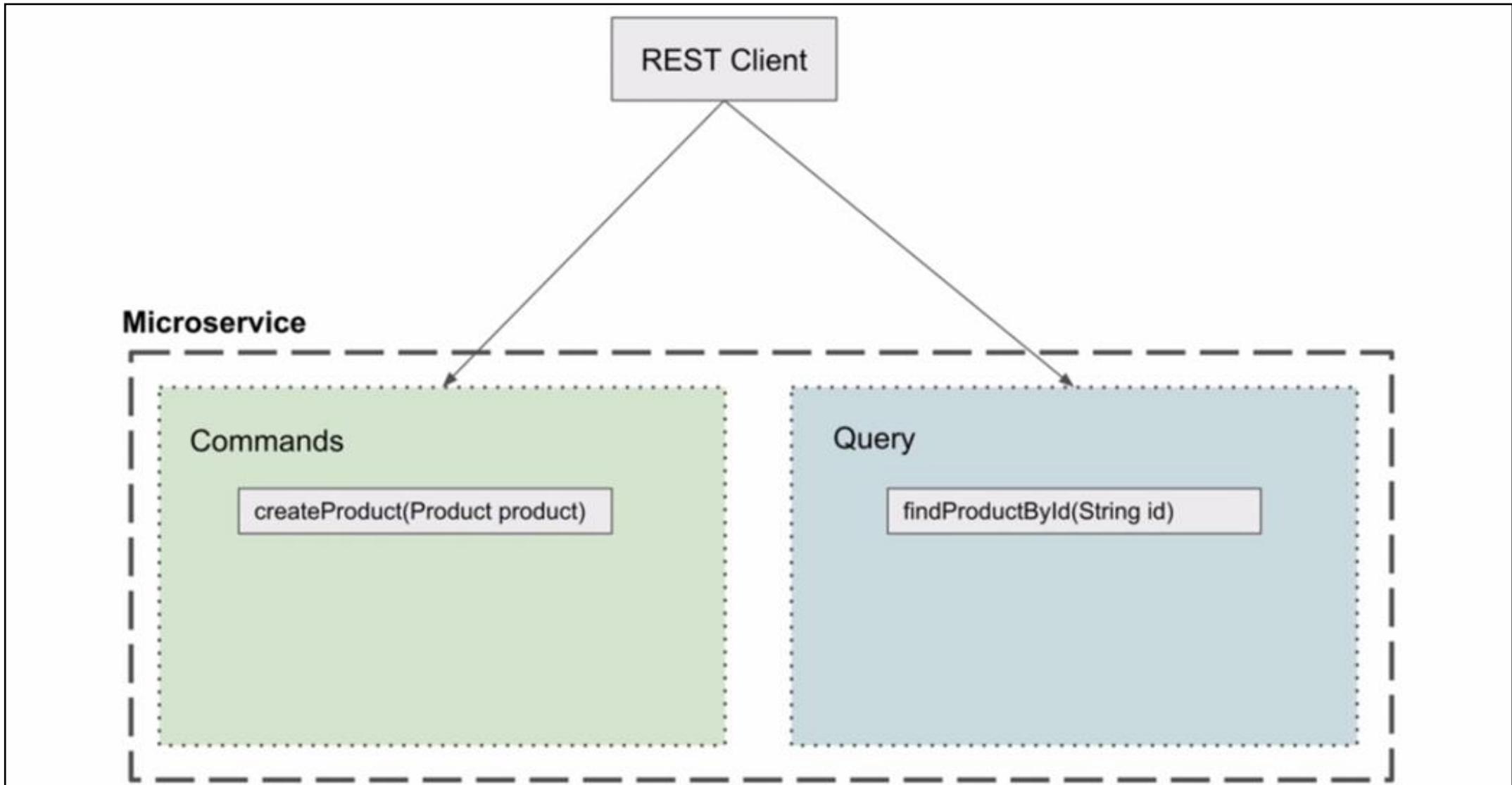


Day - 1

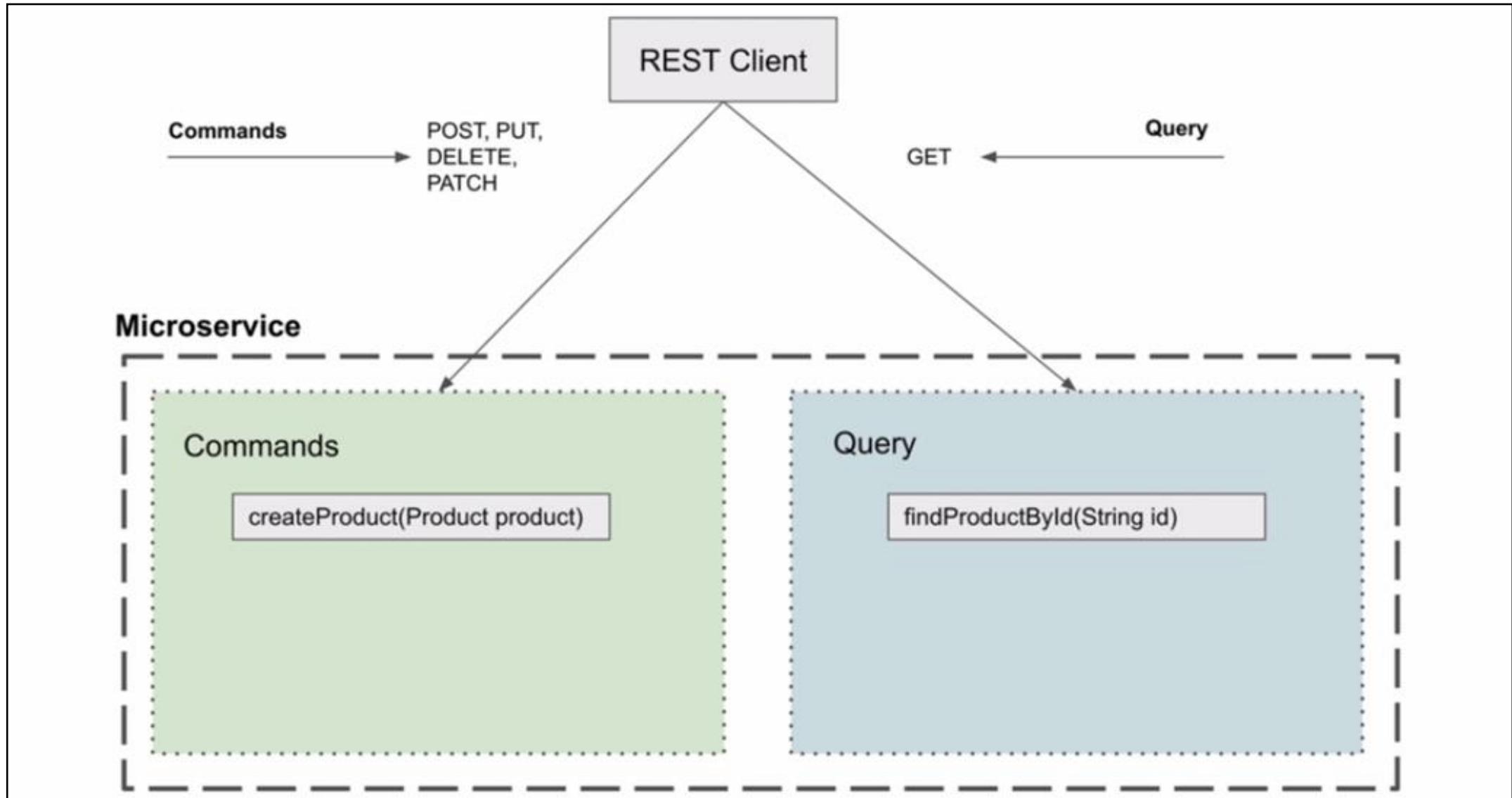
# Command Query Responsibility Segregation

- The command query responsibility segregation (CQRS) pattern separates the data mutation, or the command part of a system, from the query part.
- You can use the CQRS pattern to separate updates and queries if they have different requirements for throughput, latency, or consistency.
- The CQRS pattern splits the application into two parts: the command side and the query side.
- The command side handles create, update, and delete requests.
- The query side runs the query part by using the read replicas.

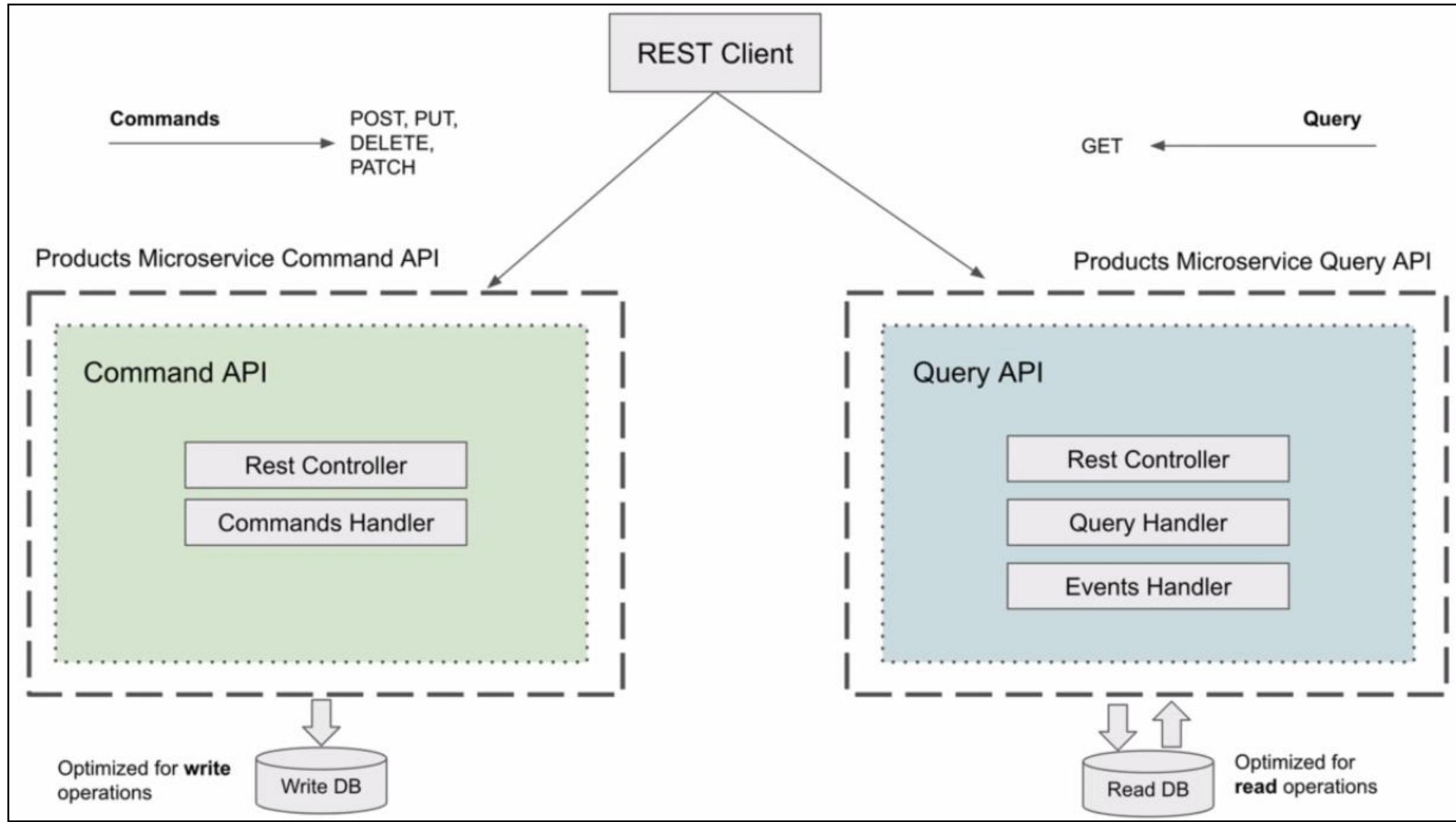
# Command Query Responsibility Segregation



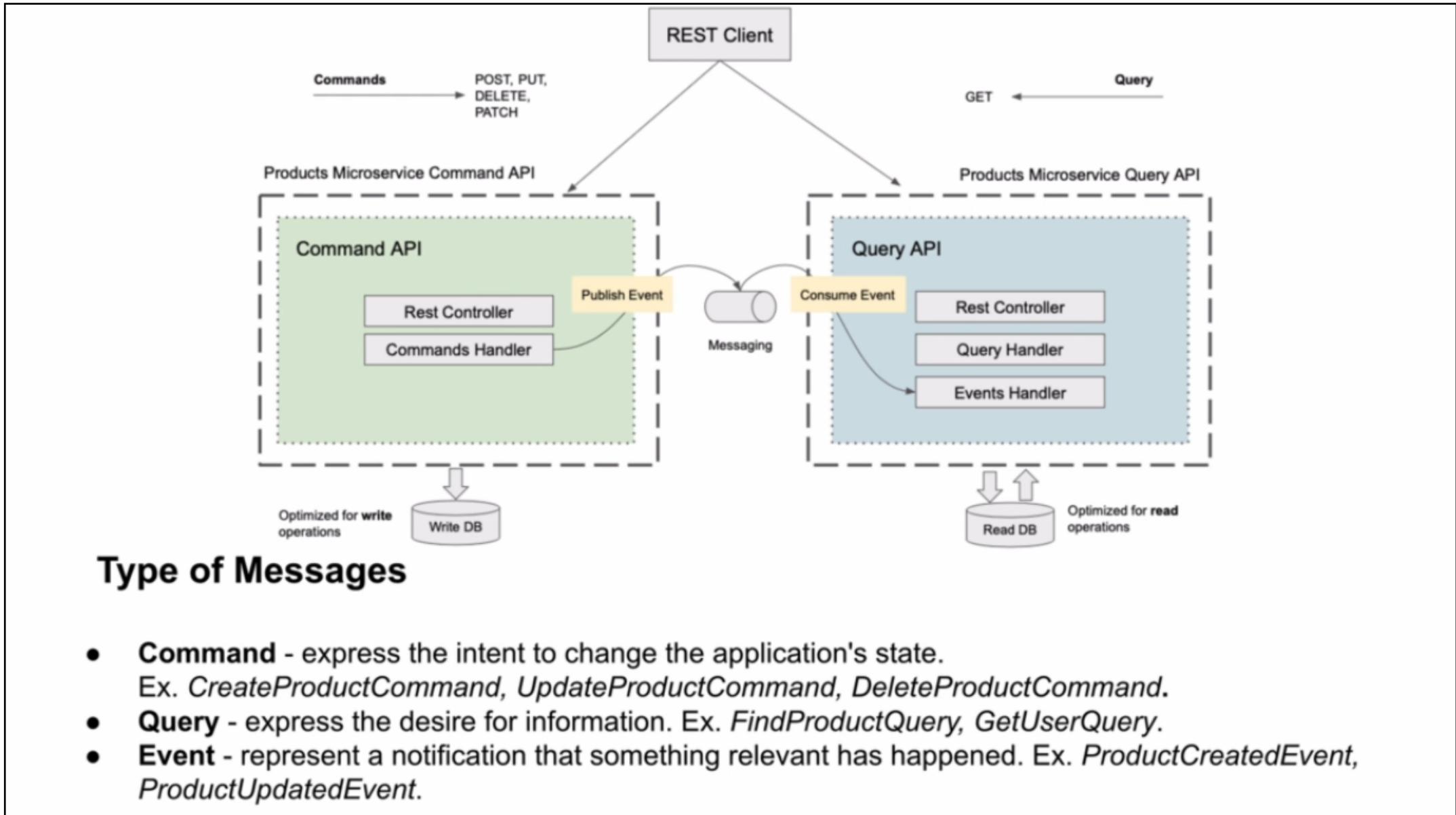
# Command vs Query – Controllers



# Read vs Write Traffic



# Type of Messages



## Type of Messages

- **Command** - express the intent to change the application's state.  
Ex. *CreateProductCommand*, *UpdateProductCommand*, *DeleteProductCommand*.
- **Query** - express the desire for information. Ex. *FindProductQuery*, *GetUserQuery*.
- **Event** - represent a notification that something relevant has happened. Ex. *ProductCreatedEvent*, *ProductUpdatedEvent*.

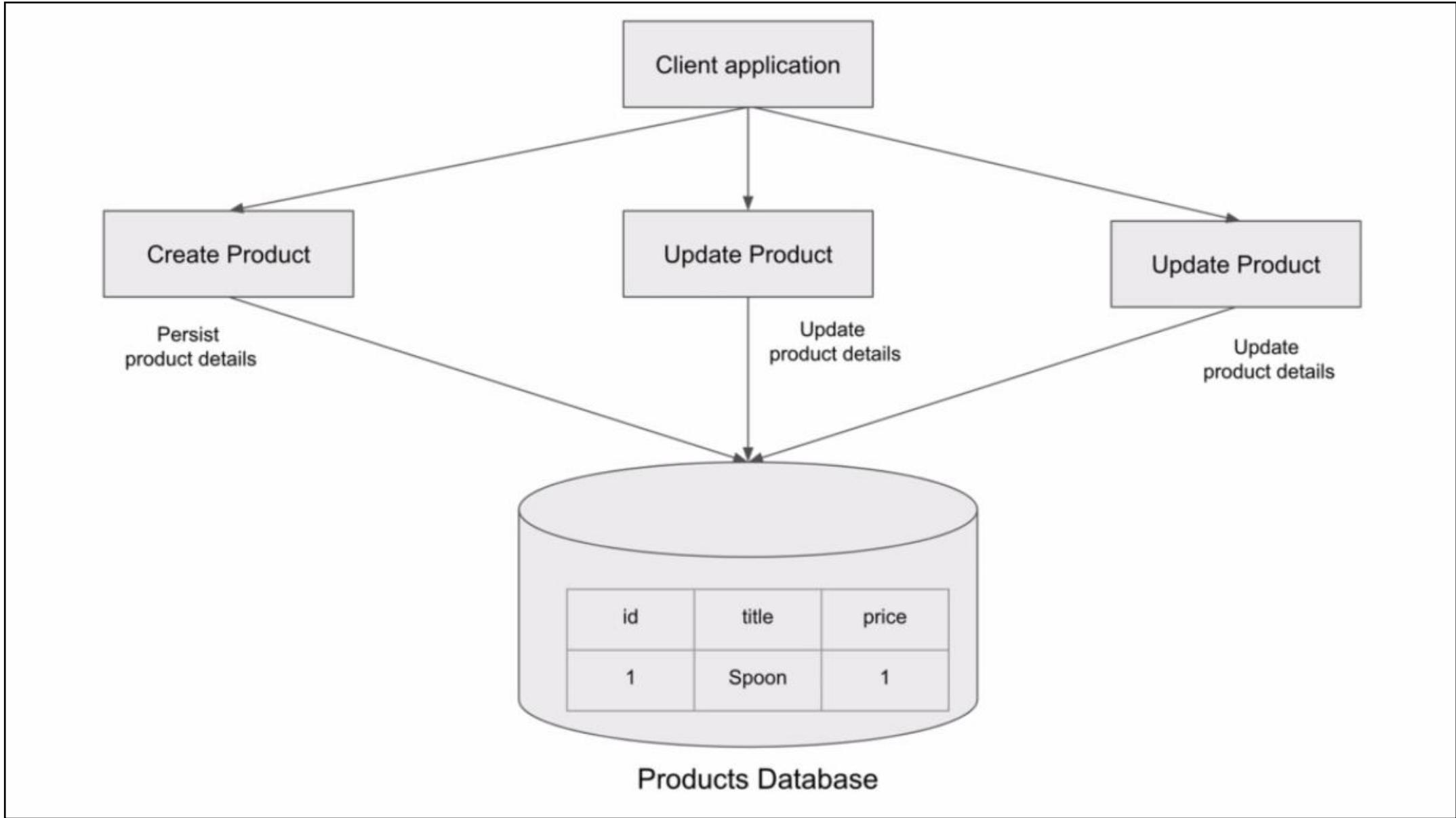


Day - 1

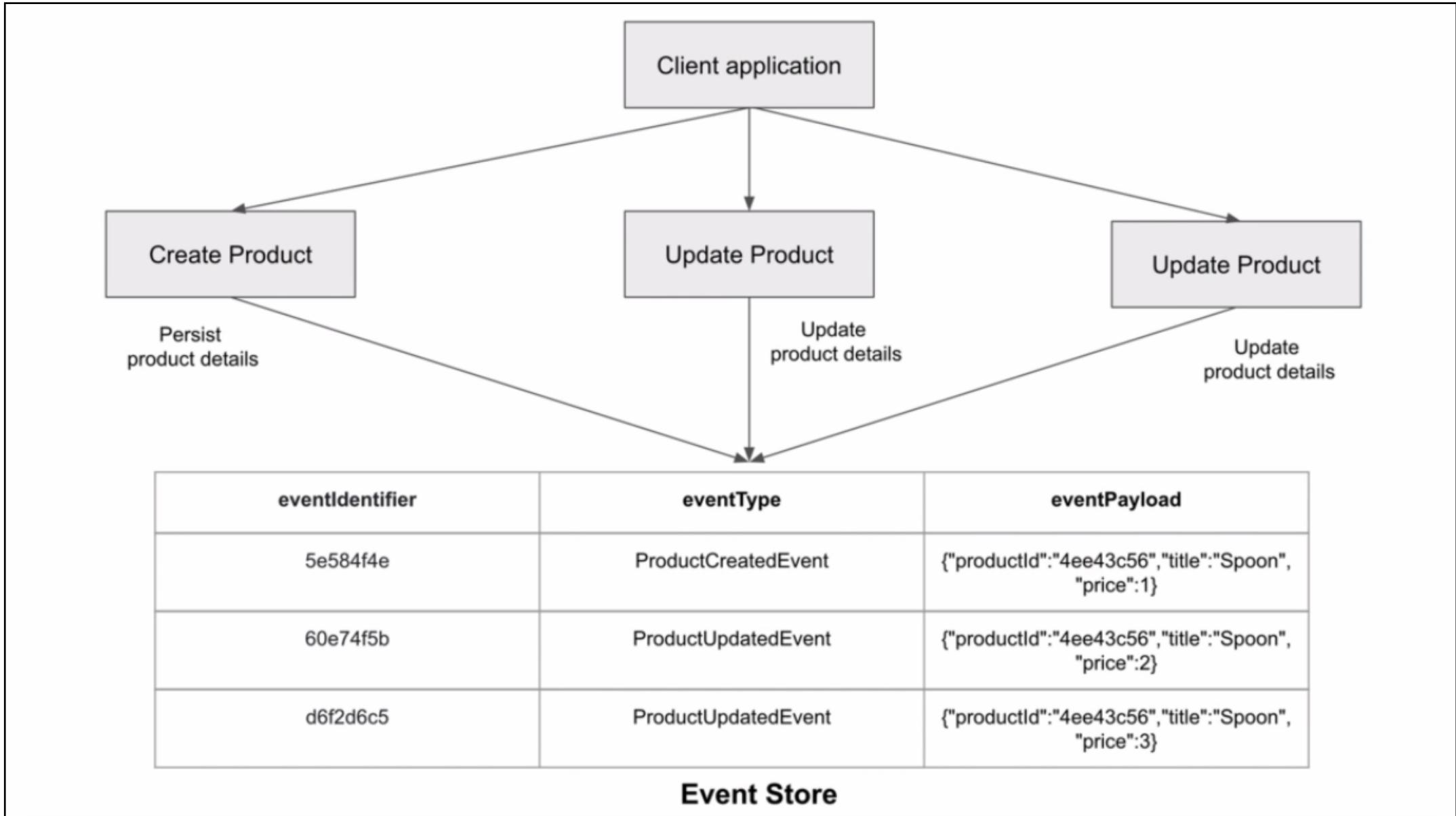
# CQRS and Event Sourcing

- 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.

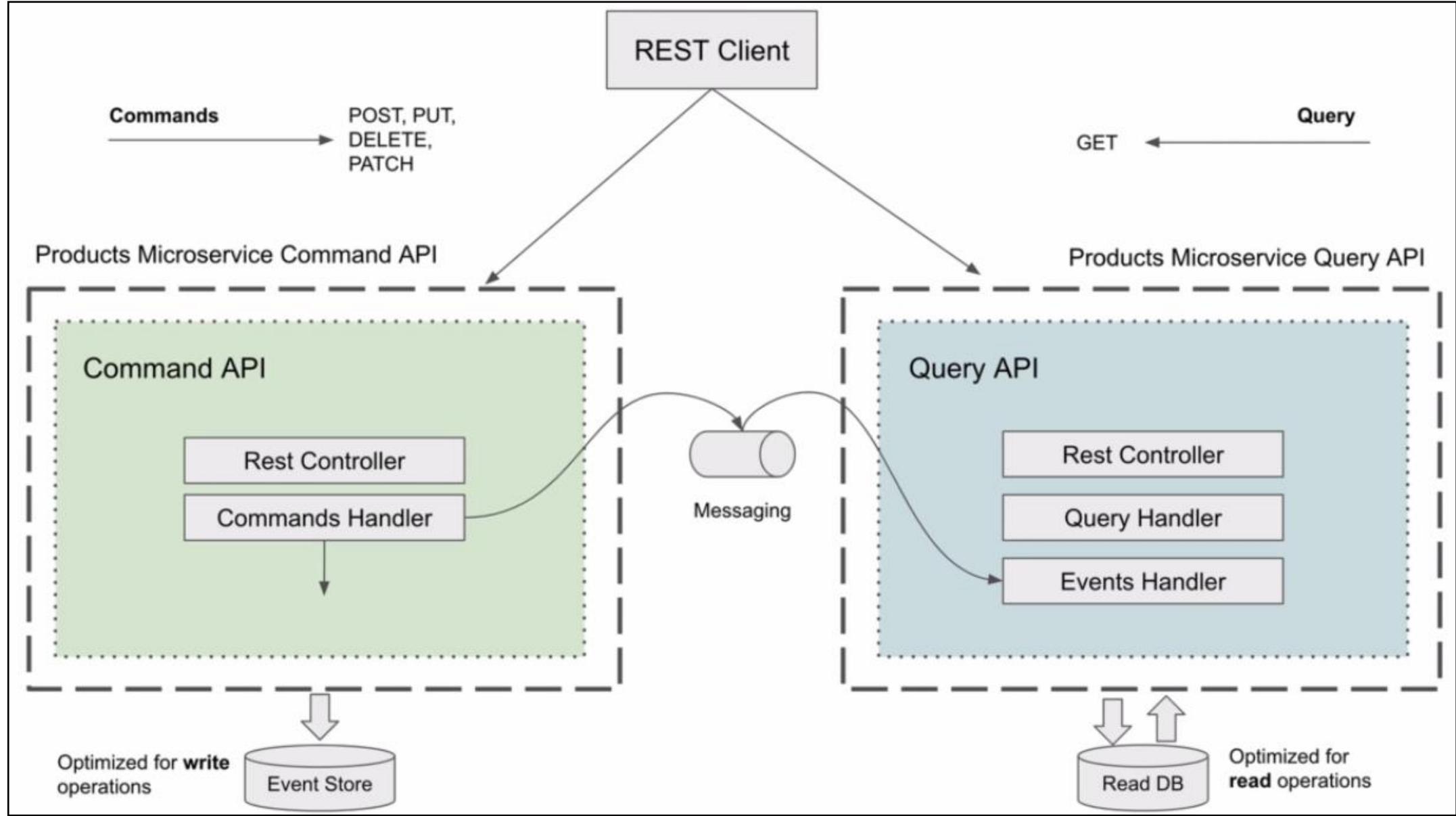
# Event Sourcing



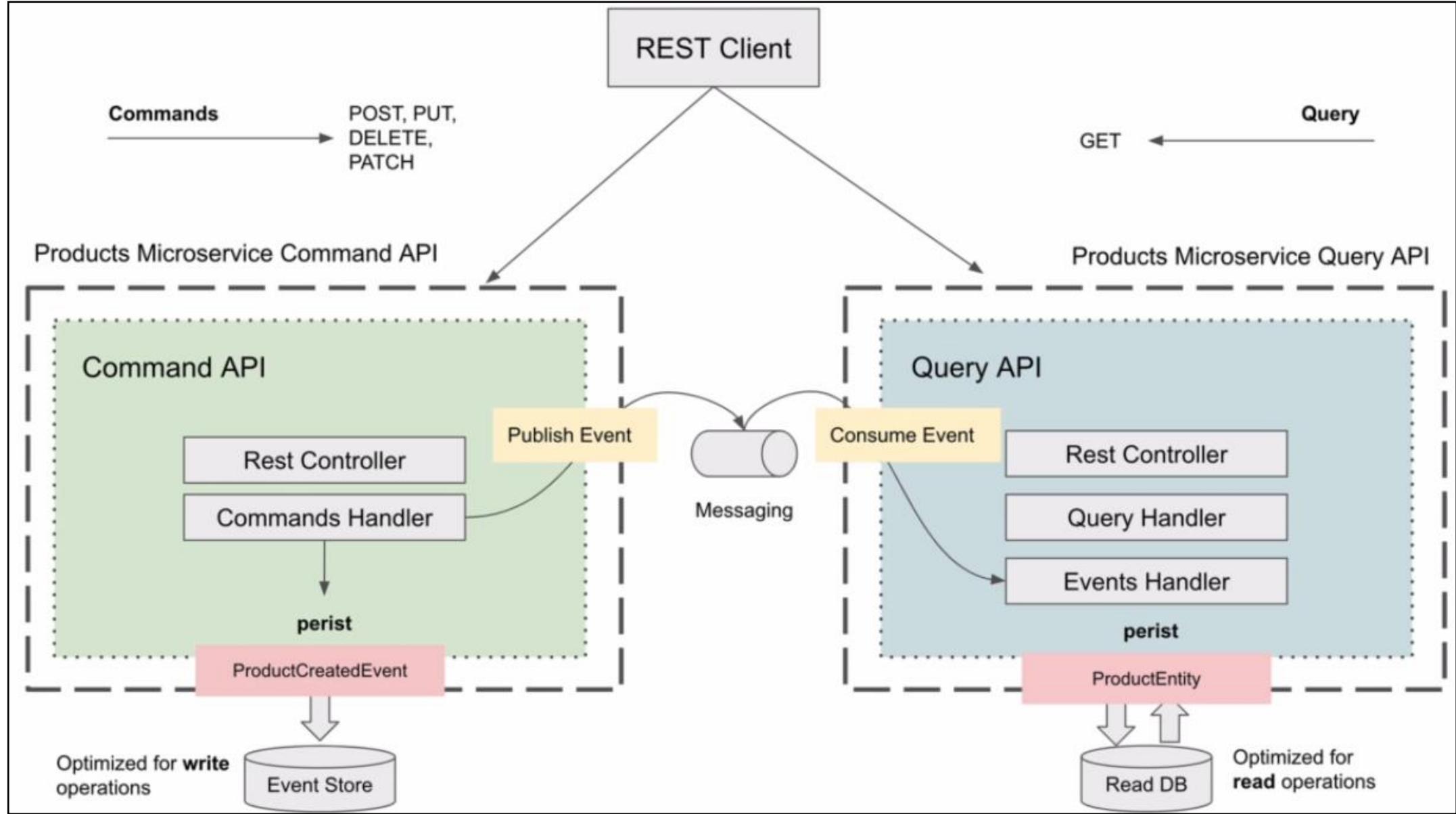
# Event Sourcing



# CQRS and Event Sourcing



# CQRS and Event Sourcing





# CQRS and Event Sourcing

**Event Store**

eventIdentifier	eventType	eventPayload
5e584f4e	ProductCreatedEvent	{"productId": "4ee43c56", "title": "Spoon", "price": 1}
60e74f5b	ProductUpdatedEvent	{"productId": "4ee43c56", "title": "Spoon", "price": 2}
d6f2d6c5	ProductUpdatedEvent	{"productId": "4ee43c56", "title": "Spoon", "price": 3}
3e73f699	ProductPriceUpdatedEvent	{"productId": "4ee43c56", "price": 3}

**Read Database**

id	title	price
1	Spoon	3

# CQRS and Event Sourcing



**Event Store**

eventIdentifier	eventType	eventPayload
5e584f4e	ProductCreatedEvent	{"productId": "4ee43c56", "title": "Spoon", "price": 1}
60e74f5b	ProductUpdatedEvent	{"productId": "4ee43c56", "title": "Spoon", "price": 2}
d6f2d6c5	ProductUpdatedEvent	{"productId": "4ee43c56", "title": "Spoon", "price": 3}
3e73f699	ProductPriceUpdatedEvent	{"productId": "4ee43c56", "price": 3}

Replay



**Read Database**

id	title	price
1	Spoon	3

**Read Database 2**

title	price
Spoon	3



Day - 1

# Building Microservices with Spring Boot



## Building a RESTful web application

1. Create a new Project for **ProductService** using the Spring Initializr ([start.spring.io](https://start.spring.io)).
2. Select the Spring Boot Version - 2.7.13.
3. Select the **Spring Web, Lombok, and Eureka Client** starter.
4. 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”



## Configure Microservices with Eureka Service Registry Server

- The “ProductService” instance will expose 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.



## Configure Microservices with Eureka Service Registry Server

1. Create a new Project for **DiscoveryServer** using the Spring Initializr ([start.spring.io](https://start.spring.io/)).
2. Select the Spring Boot Version - 2.7.13.
3. Select the **Eureka Server** starter.
4. In the Application class, Add **@EnableEurekaServer** annotation.
5. Ensure the server is running on 8761.

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

6. Start the Application.
7. Verify the Eureka Server: <http://localhost:8761/>



## Enable Dynamic Registration to Product Microservice

1. Refer the **ProductService** created previously.
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.

The screenshot shows a code editor window with the title "application.properties". The file contains the following configuration:

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



# Implementing Spring Cloud Gateway in Microservices

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 Initializr ([start.spring.io](https://start.spring.io)).
4. Select the Spring Boot Version - 2.7.13.
5. Select the **Gateway, Spring Web and Eureka Discovery Client** starter.
6. Add **@EnableEurekaClient/@EnableDiscoveryClient** in the Application class.
7. In application.properties file, enable the automatic mapping of gateway routes and add the application name and eureka client serviceUrl.

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



## Implementing Spring Cloud Gateway in Microservices

8. Start the ApiGateway.
9. Check the proxy running instances is also registered with the Eureka Server.
10. Test the Proxy: <http://localhost:8082/products-service/products>



- Microservice vs Monolithic application
- Event-Driven Microservices
- Transactions in Microservices
- Choreography-Based Saga
- Orchestration-Based Saga
- Command Query Responsibility Segregation
- Types of Messaging in CQRS Pattern
- CQRS and Event Sourcing
- Building microservices with spring boot



# Day - 2



## Day – 2 Agenda

- Introduction to Axon Server
- Download and run Axon Server as JAR application
- Axon Server configuration properties
- Run Axon Server in a Docker container
- Bringing CQRS and Event Sourcing Together with Axon Framework
- Accept HTTP Request Body
- Adding Axon Framework Spring Boot Starter
- Creating a new Command class
- Send Command to a Command Gateway



## Day – 2 Agenda

- Introduction to Aggregate
- Creating the Aggregate class
- Validate the command class
- Creating the event class
- Apply and Publish the Created Event
- @EventSourcingHandler Annotation
- Previewing Event in the EventStore



Day - 2

# Axon – Getting Started

- Axon provides the **Axon Framework** and the **Axon Server** to help build applications centered on three core concepts - **CQRS** (Command Query Responsibility Segregation) / **Event Sourcing** and **DDD** (Domain Driven Design).
- While many types of applications can be built using Axon, it has proven to be very popular for microservices architectures. Axon provides an innovative and powerful way of sensibly evolving to event-driven microservices within a microservices architecture.



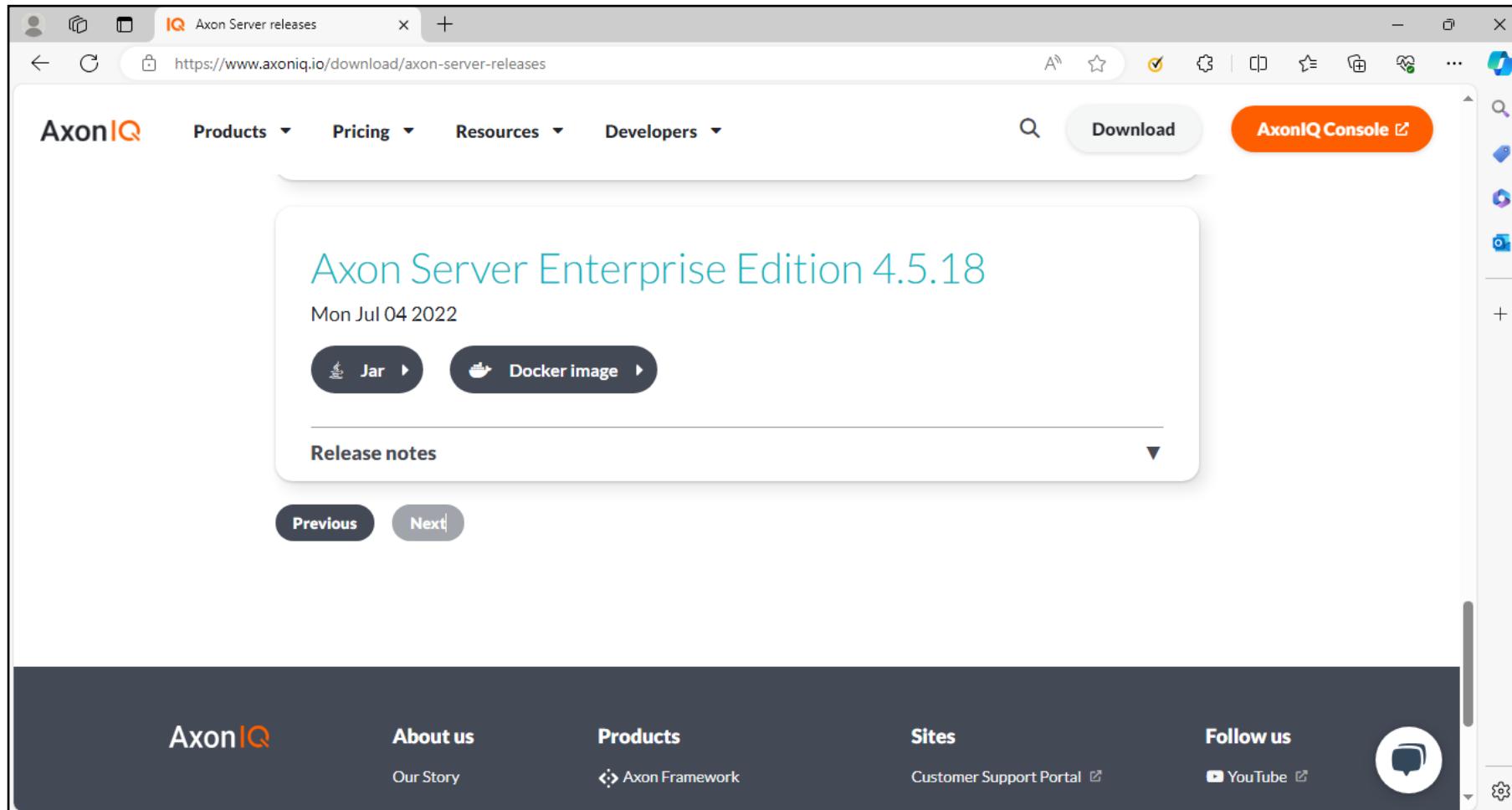
Day - 2

# Axon Server



## Download and run Axon Server as JAR application

- Go to <https://developer.axoniq.io/download>
- Select Axon Server Enterprise Edition 4.5.18, download the JAR and execute.





## Download and run Axon Server as JAR application

- Run `java -jar axonserver.jar`



## Download and run Axon Server as JAR application

- Access <http://localhost:8024>

The screenshot shows the Axon Server dashboard running at <http://localhost:8024>. The left sidebar has icons for Settings, Overview, Search, Commands, and Queries, with 'Commands' currently selected. The main area displays the Axon Server logo and two yellow status bars: 'SSL disabled' and 'Authentication disabled'. Below these are sections for Configuration, Status, and License.

Configuration	Status	License
Node Name: microservices	Last event token: -1	Edition: Standard Edition
Host Name: microservices	Activity in the last minute:	
Http Port: 8024		
GRPC Port: 8124	Commands: 0	



Day - 2

## Run Axon Server in a Docker Container



# Run Axon Server in a Docker Container

- Docker Command:

```
docker run -d --name axonserver -p 8024:8024 -p 8124:8124 axoniq/axonserver:4.5.8
```

The screenshot shows a Windows Command Prompt window titled "Command Prompt". The command entered is "docker run -d --name axonserver -p 8024:8024 -p 8124:8124 axoniq/axonserver:4.5.8". The output of the command is displayed, showing the progress of pulling the image from the repository. The command "docker ps" is then run to list the running containers, showing the newly created "axonserver" container.

```
C:\Users\labuser>docker run -d --name axonserver -p 8024:8024 -p 8124:8124 axoniq/axonserver:4.5.8
Unable to find image 'axoniq/axonserver:4.5.8' locally
4.5.8: Pulling from axoniq/axonserver
ec52731e9273: Pull complete
8907fc4ab049: Pull complete
a1f1879bb7de: Pull complete
5347aaaf66df0: Pull complete
2bdcc0330791: Pull complete
b1bb79b3cfdb: Pull complete
191655b822f9: Pull complete
11e10d67a998: Pull complete
d3d5ea2e5b82: Pull complete
07ff33aa477f: Pull complete
338c3c26206a: Pull complete
Digest: sha256:7b004facd1da1fa791e4a2752b6cdec68ae7aeb2cd216a2c4dd95b26beeb6ab1
Status: Downloaded newer image for axoniq/axonserver:4.5.8
a2006525b087bd82d16e57e92b6ed5f786f93364b20638006e992b64753c52b5

C:\Users\labuser>docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
NAMES
a2006525b087      axoniq/axonserver:4.5.8   "java -cp /app/resou..."   About a minute ago   Up About a minute   0.0.0.0:8024->8024/tcp, ::::8024->8024/tcp, 0.0.0.0:8124->8124/tcp, ::::8124->8124/tcp   axonserver

C:\Users\labuser>
```



## Start, Stop, Delete Axon Server Docker Container By ID

- docker ps -a
- docker start <container-id>
- docker stop <container-id>
- docker rm <container-id>



Day - 2

# Bringing CQRS and Event Sourcing Together with Axon Framework



## Introduction

- Apply the CQRS design pattern to our Products Microservices.



## Accept HTTP Request Body

- Add the Lombok dependency:

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



## Accept HTTP Request Body

- Create a **CreateProductRestModel** class for Accept HTTP Request Body:

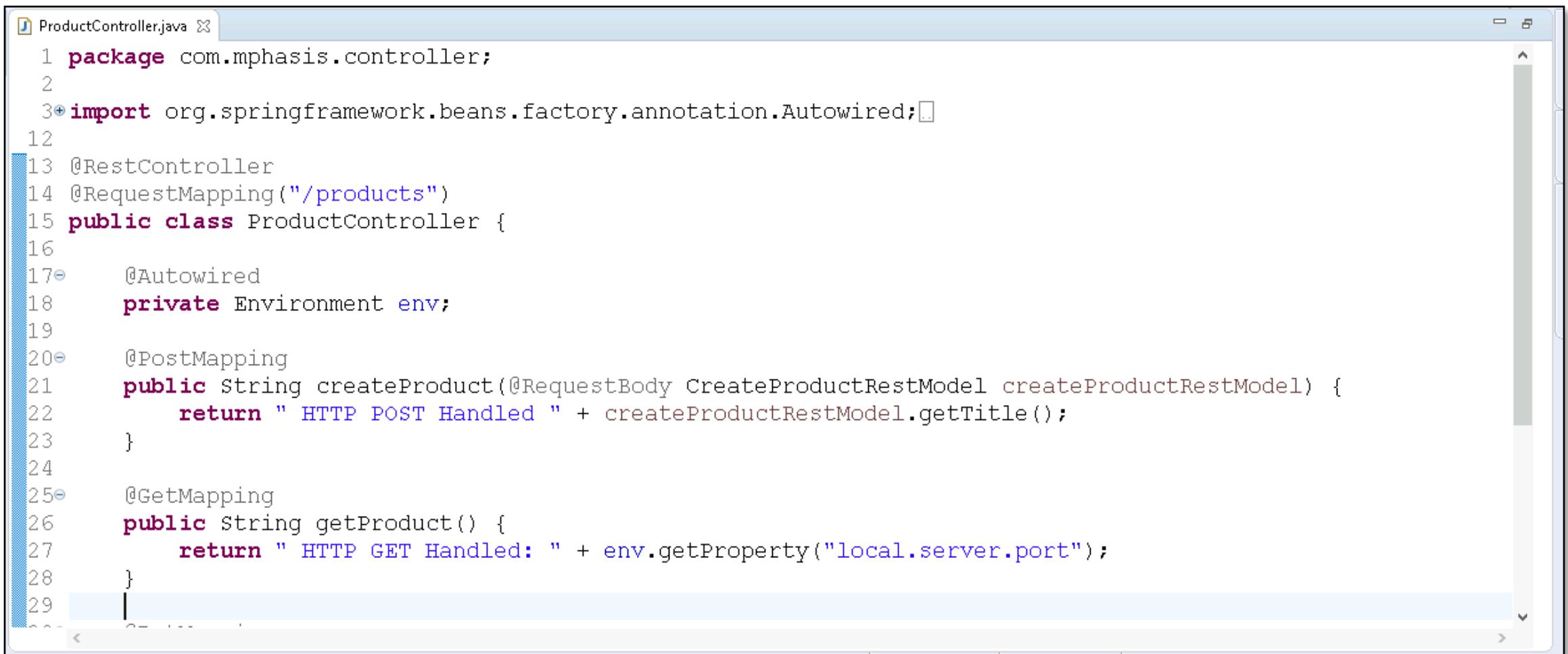
The screenshot shows a Java code editor window with the title bar "CreateProductRestModel.java". The code is as follows:

```
1 package com.mphasis.controller;
2
3 import java.math.BigDecimal;
4
5 import lombok.Data;
6
7 @Data
8 public class CreateProductRestModel {
9
10     private String title;
11     private BigDecimal price;
12     private Integer quantity;
13 }
14
```



## Accept HTTP Request Body

- Apply **CreateProductRestModel** to the Controller:



The screenshot shows a Java code editor window with the file `ProductController.java` open. The code defines a REST controller for products. It includes imports for `com.mphasis.controller`, `org.springframework.beans.factory.annotation.Autowired`, and `org.springframework.web.bind.annotation.RestController`, `@RequestMapping`, `@PostMapping`, and `@GetMapping`. The controller has two methods: `createProduct` which handles POST requests and returns a message with the product title, and `getProduct` which handles GET requests and returns a message with the local server port.

```
ProductController.java
1 package com.mphasis.controller;
2
3+import org.springframework.beans.factory.annotation.Autowired;□
12
13 @RestController
14 @RequestMapping("/products")
15 public class ProductController {
16
17@    @Autowired
18    private Environment env;
19
20@    @PostMapping
21    public String createProduct(@RequestBody CreateProductRestModel createProductRestModel) {
22        return " HTTP POST Handled " + createProductRestModel.getTitle();
23    }
24
25@    @GetMapping
26    public String getProduct() {
27        return " HTTP GET Handled: " + env.getProperty("local.server.port");
28    }
29}
```



## Adding Axon Framework Spring Boot Starter

- We will add **axon-spring-boot-starter** starter to ProductService/pom.xml:

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



Day - 2

# Product Service API - Commands



## Creating a new Command class

- **Command** - express the intent to change the application's state.
- In terms of CQRS design pattern, when there's a modifying request, this is a Command. Because the Command is intended to make a change, but in this case Command in creating a Product.
- The name of Command should be in below format:

<Verb><Noun>Command

CreateProductCommand

UpdateProductCommand



### **TargetAggregateIdentifier**

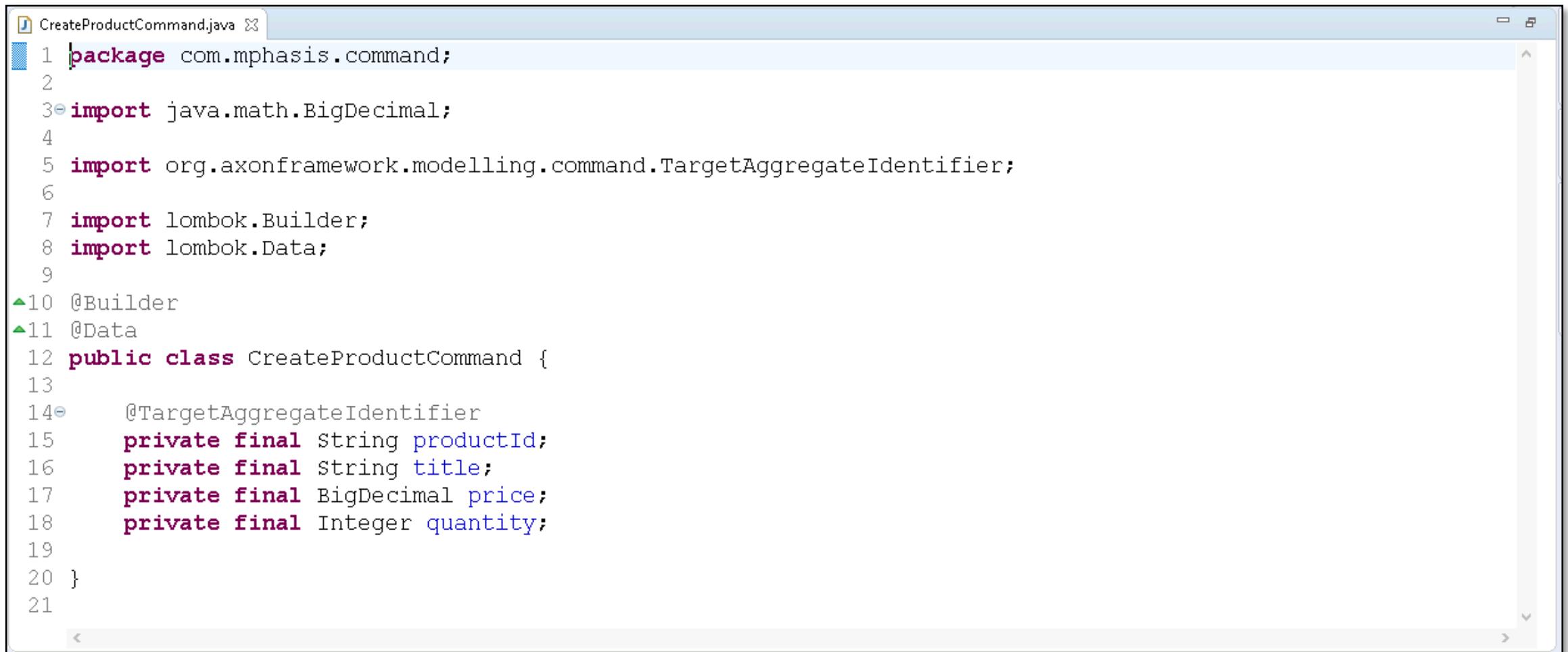


The `TargetAggregateIdentifier` annotation tells Axon that the annotated field is an id of a given aggregate to which the command should be targeted.



## Creating a new Command class

- The CreateProductCommand will be read-only:



```
1 package com.mphasis.command;
2
3 import java.math.BigDecimal;
4
5 import org.axonframework.modelling.command.TargetAggregateIdentifier;
6
7 import lombok.Builder;
8 import lombok.Data;
9
10 @Builder
11 @Data
12 public class CreateProductCommand {
13
14     @TargetAggregateIdentifier
15     private final String productId;
16     private final String title;
17     private final BigDecimal price;
18     private final Integer quantity;
19
20 }
```



## Send Command to a Command Gateway

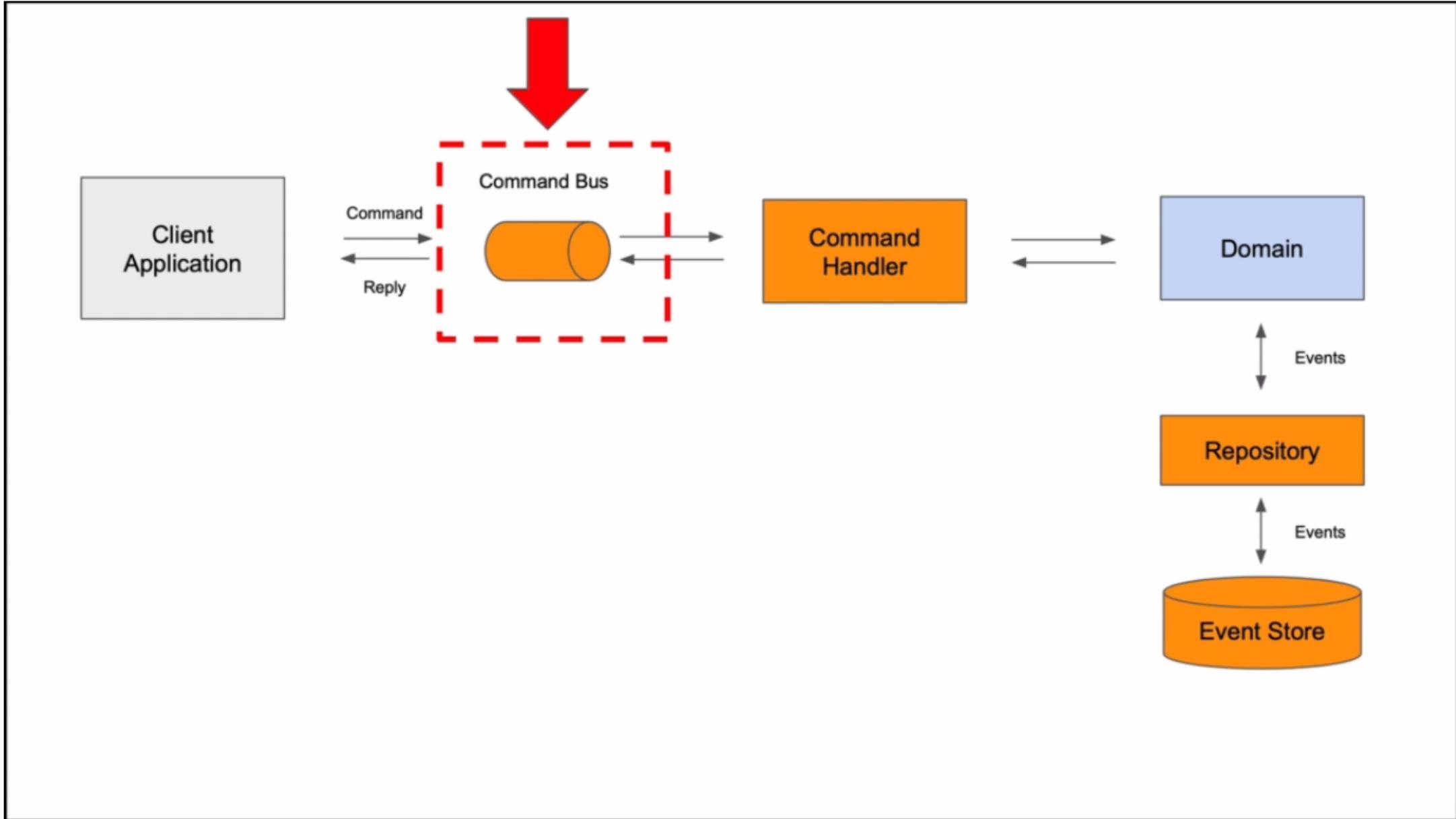


### **CommandGateway:**



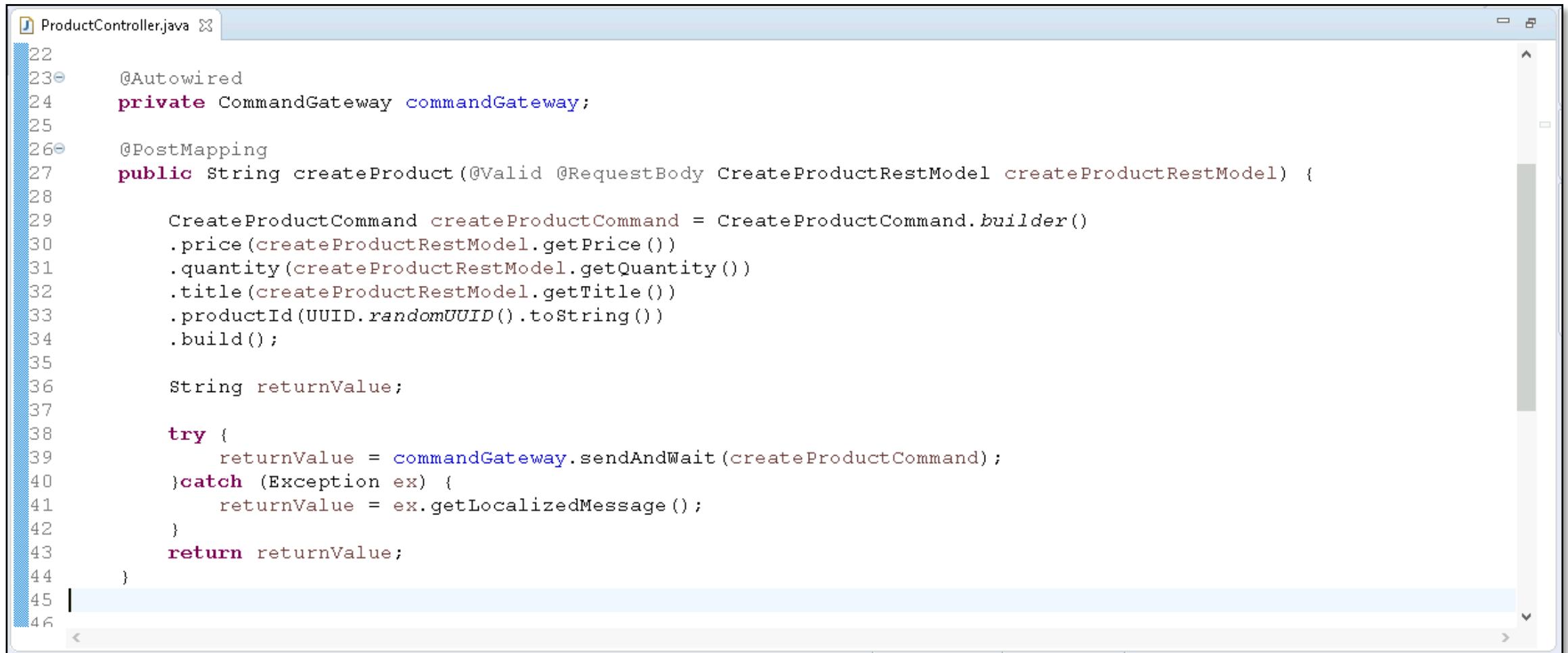
Interface towards the Command Handling components of an application. This interface provides a friendlier API toward the command bus. The CommandGateway allows for components dispatching commands to wait for the result.

# Send Command to a Command Gateway





## Send Command to a Command Gateway



The screenshot shows a Java code editor window with the file `ProductController.java` open. The code implements a `CreateProduct` command and sends it to a `CommandGateway`.

```
22
23@Autowired
24 private CommandGateway commandGateway;
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     try {
39         returnValue = commandGateway.sendAndWait(createProductCommand);
40     }catch (Exception ex) {
41         returnValue = ex.getLocalizedMessage();
42     }
43     return returnValue;
44 }
45
46
```



Day - 2

# Product Service API - Events



## Creating a new Event class

- Event - represent a notification that something relevant has happened.
- To publish an Event first we must create Event class.
- Naming conventions for Event:

<Noun><PerformedAction>Event

Product**Created**Event

Product**Shipped**Event

Product**Deleted**Event



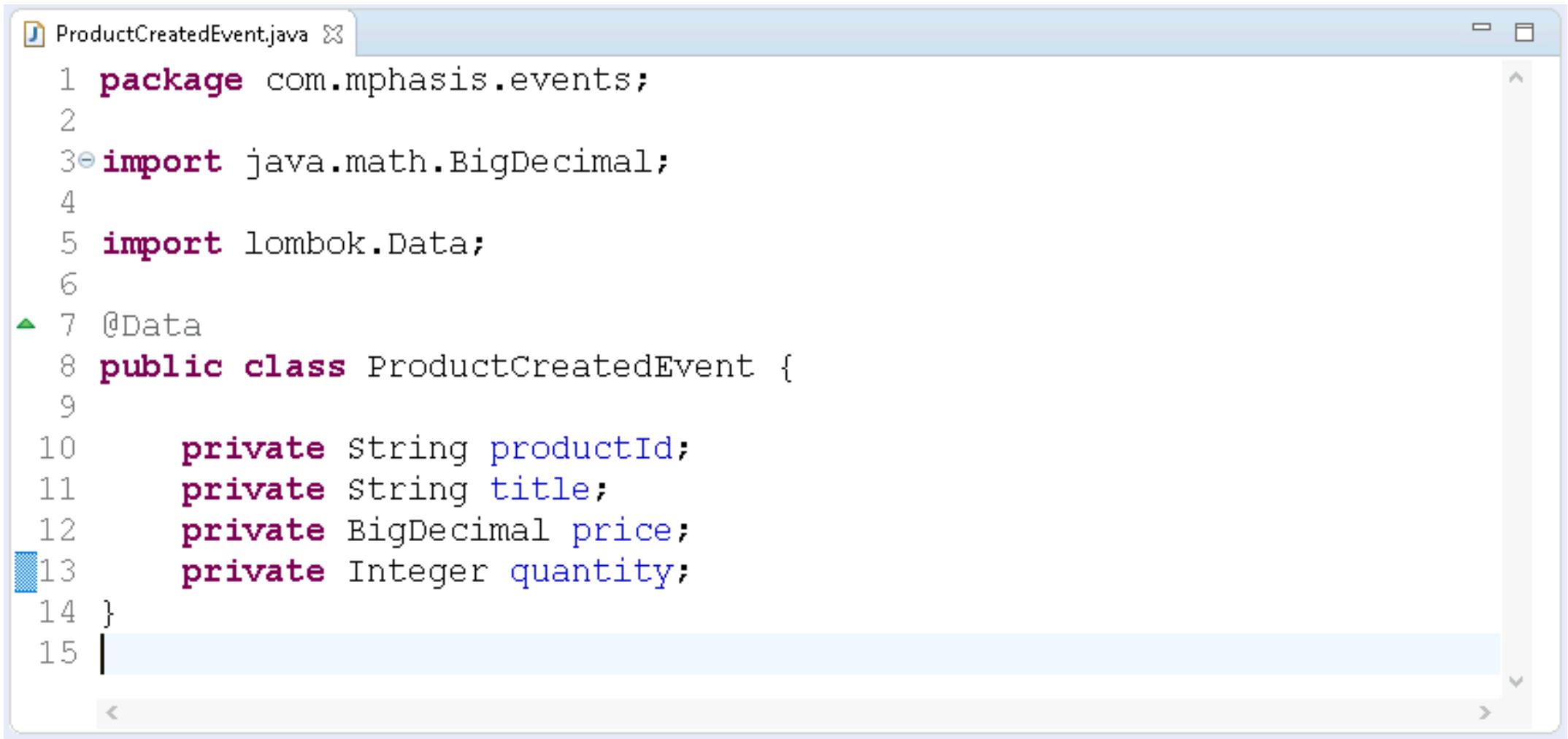
## Creating a new Event class

- Our aggregate will handle the commands, as it's in charge of deciding if a Product can be created, deleted, or shipped.
- It will notify the rest of the application of its decision by publishing an event. We'll have three types of events — ProductCreatedEvent, ProductDeletedEvent, and ProductShippedEvent.



## Creating a new Event class

- Let's create the ProductCreatedEvent class:



The screenshot shows a Java code editor window with the file `ProductCreatedEvent.java` open. The code defines a class `ProductCreatedEvent` with private fields for product ID, title, price, and quantity, annotated with `@Data` from the `lombok` library.

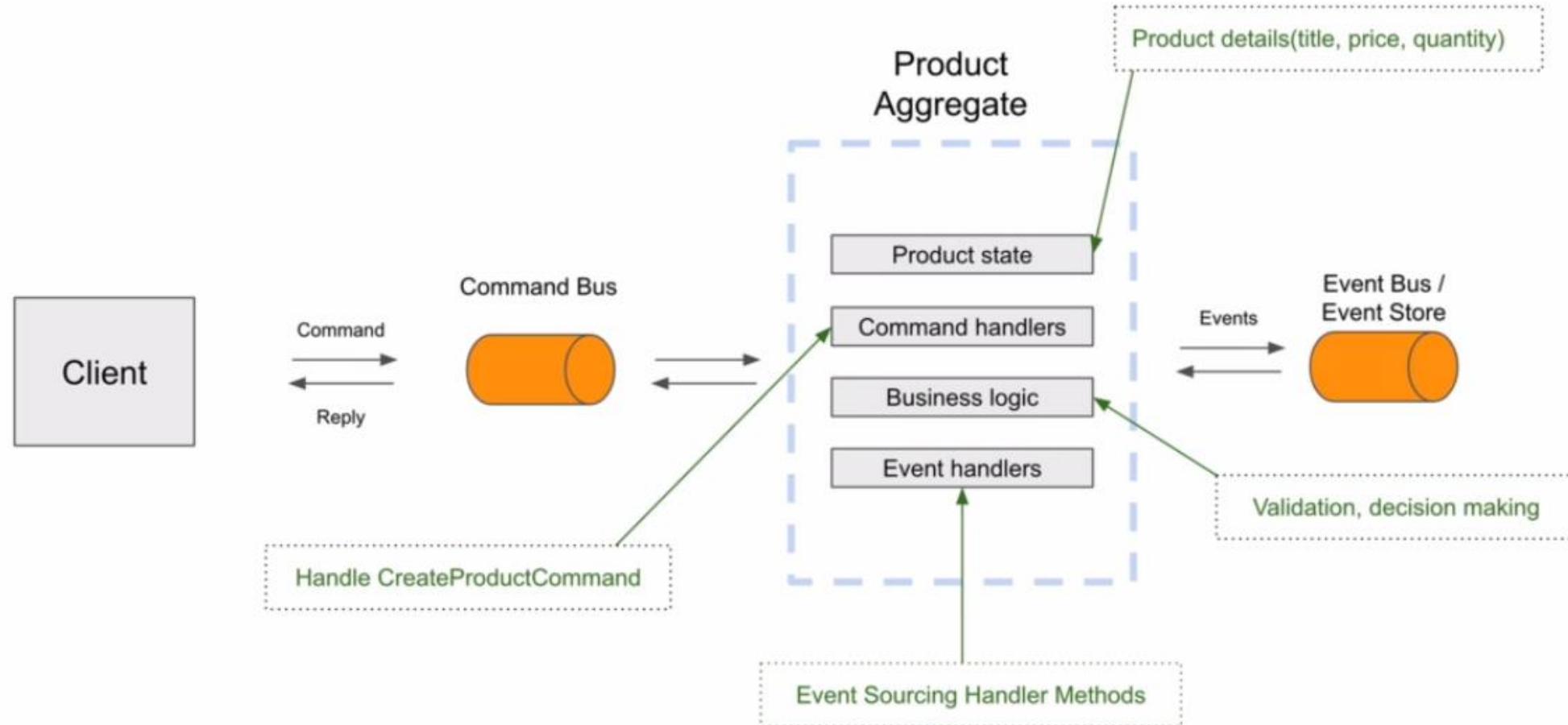
```
ProductCreatedEvent.java
1 package com.mphasis.events;
2
3 import java.math.BigDecimal;
4
5 import lombok.Data;
6
7 @Data
8 public class ProductCreatedEvent {
9
10    private String productId;
11    private String title;
12    private BigDecimal price;
13    private Integer quantity;
14 }
15 |
```



Day - 2

# The Command Model – Product Aggregate

# Product Aggregate - Introduction



- Aggregate class is at the core of your microservice.
- It holds the current state of the main object.
- In this section we are working on Product Microservice, and this means our aggregate class will be called **ProductAggregate**.
- This Product Aggregate object will hold the current state of Product object.
- It will hold the current value of the Product details(title, price, quantity).
- Additionally, the Product Aggregate will contain methods that can handle commands.
- The Product Aggregate class will also have the business logic.
- The Product Aggregate class will contain the Event Sourcing Handler Methods.
- Every time the state of the Product changes an Event Sourcing Handler Method will be invoked.



## Aggregate



The Aggregate annotation is an Axon Spring specific annotation marking this class as an aggregate. It will notify the framework that the required **CQRS** and **Event Sourcing** specific building blocks need to be instantiated for this *ProductAggregate*.



### CommandHandler



Marker annotation to mark any method on an object as being a CommandHandler. Use the AnnotationCommandHandlerAdapter to subscribe the annotated class to the command bus. This annotation can also be placed directly on Aggregate members to have it handle the commands directly.



## Create ProductAggregate class

- Product Aggregate class is annotated with **@Aggregate** annotation.
- Second constructor with CreateProductCommand argument is annotated with **@CommandHandler** annotation.



```
ProductAggregate.java X
1 package com.mphasis.command;
2
3 import org.axonframework.commandhandling.CommandHandler;
4 import org.axonframework.spring.stereotype.Aggregate;
5
6 @Aggregate
7 public class ProductAggregate {
8
9     public ProductAggregate() {
10
11
12 }
13
14 @CommandHandler
15 public ProductAggregate(CreateProductCommand createProductCommand) {
16     // Validate Create Product Command
17 }
18 }
```



## Validate the CreateProductCommand

- Product Aggregate class can be used to validate the CreateProductCommand.



The screenshot shows a Java code editor window with the file `ProductAggregate.java` open. The code defines a `ProductAggregate` class with a constructor and a command handler method. The command handler validates the `CreateProductCommand` by checking if the price is less than or equal to zero and if the title is null or empty. If either condition is true, it throws an `IllegalArgumentException`.

```
ProductAggregate.java
8 @Aggregate
9 public class ProductAggregate {
10
11     public ProductAggregate() {
12
13     }
14
15     @CommandHandler
16     public ProductAggregate(CreateProductCommand createProductCommand) {
17         // Validate Create Product Command
18
19         if (createProductCommand.getPrice().compareTo(BigDecimal.ZERO) <= 0) {
20             throw new IllegalArgumentException("Price cannot be less or equal than zero");
21         }
22
23         if (createProductCommand.getTitle() == null
24             || createProductCommand.getTitle().isEmpty()) {
25             throw new IllegalArgumentException("Title cannot be empty");
26         }
27     }
28 }
```



## AggregateLifeCycle.apply(Object payload)



### AggregateLifeCycle.apply(Object payload)



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.



## EventHandler



### EventHandler



Annotation to be placed on methods that can handle events. The parameters of the annotated method are resolved using parameter resolvers.



## Apply and Publish the Product Created Event

ProductAggregate.java

```
18
19  @CommandHandler
20  public ProductAggregate(CreateProductCommand createProductCommand) {
21      // Validate Create Product Command
22
23      if (createProductCommand.getPrice().compareTo(BigDecimal.ZERO) <= 0) {
24          throw new IllegalArgumentException("Price cannot be less or equal than zero");
25      }
26
27      if (createProductCommand.getTitle() == null
28          || createProductCommand.getTitle().isEmpty()) {
29          throw new IllegalArgumentException("Title cannot be empty");
30      }
31
32      ProductCreatedEvent productCreatedEvent = new ProductCreatedEvent();
33
34      BeanUtils.copyProperties(createProductCommand, productCreatedEvent);
35
36      AggregateLifecycle.apply(productCreatedEvent);
37  }
38 }
```

- **AggregateLifecycle.apply(Object payload)** – Apply an 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.
- **@TargetAggregateIdentifier** in CreateProductCommand and **@AggregateIdentifier** in ProductAggregate will help Axon Framework to associate the dispatch command with the right aggregate.



## AggregateIdentifier



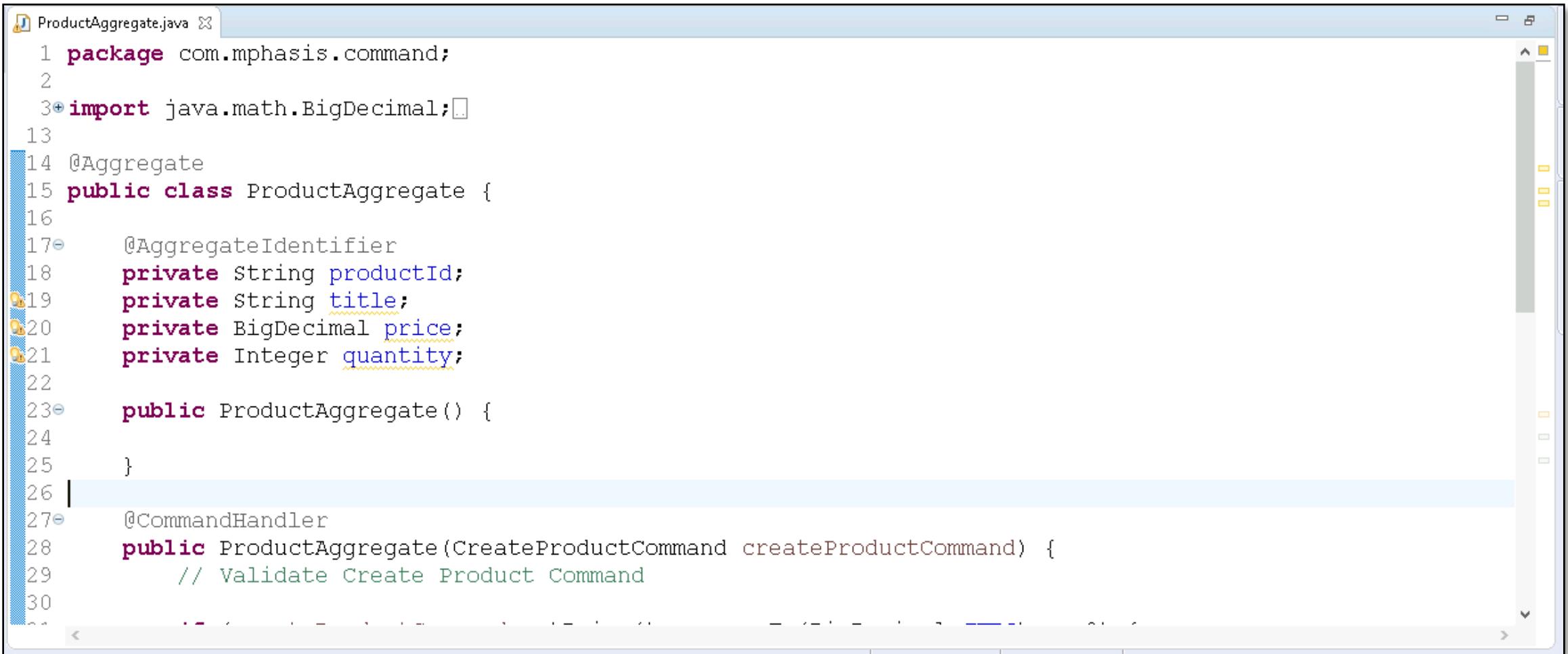
### AggregateIdentifier



Field annotation that identifies the field containing the identifier of the Aggregate.



# @EventSourcingHandler

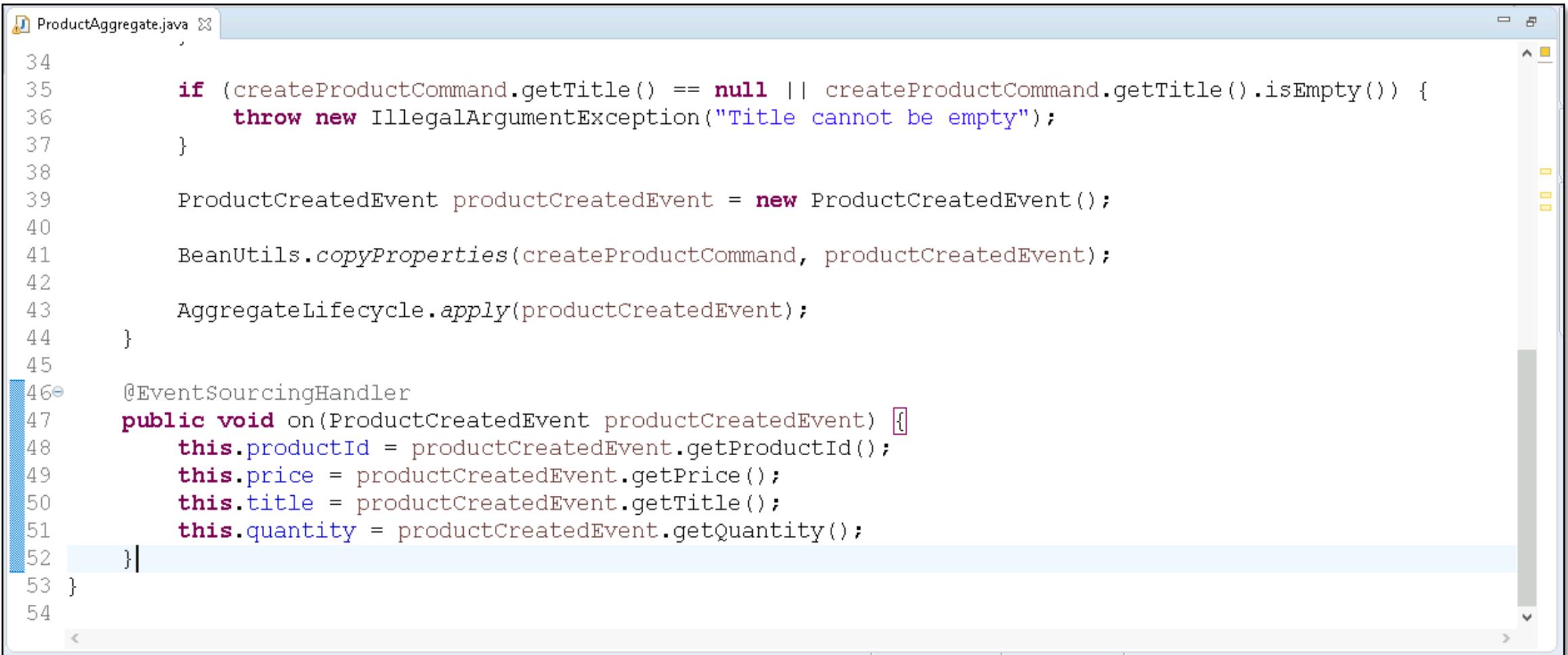


The screenshot shows a Java code editor window with the file `ProductAggregate.java` open. The code defines a class `ProductAggregate` with fields `productId`, `title`, `price`, and `quantity`. It includes a constructor and a command handler method `ProductAggregate(CreateProductCommand createProductCommand)`.

```
1 package com.mphasis.command;
2
3+import java.math.BigDecimal;□
13
14 @Aggregate
15 public class ProductAggregate {
16
17@ AggregateIdentifier
18     private String productId;
19     private String title;
20     private BigDecimal price;
21     private Integer quantity;
22
23@ Public
24     public ProductAggregate() {
25
26
27@ CommandHandler
28     public ProductAggregate(CreateProductCommand createProductCommand) {
29         // Validate Create Product Command
30
31 }
```



# @EventSourcingHandler



```
ProductAggregate.java X
34
35     if (createProductCommand.getTitle() == null || createProductCommand.getTitle().isEmpty()) {
36         throw new IllegalArgumentException("Title cannot be empty");
37     }
38
39     ProductCreatedEvent productCreatedEvent = new ProductCreatedEvent();
40
41     BeanUtils.copyProperties(createProductCommand, productCreatedEvent);
42
43     AggregateLifecycle.apply(productCreatedEvent);
44 }
45
46@EventSourcingHandler
47 public void on(ProductCreatedEvent productCreatedEvent) {
48     this.productId = productCreatedEvent.getProductId();
49     this.price = productCreatedEvent.getPrice();
50     this.title = productCreatedEvent.getTitle();
51     this.quantity = productCreatedEvent.getQuantity();
52 }
53 }
54
```



## Adding Additional Dependency

- When using Axon with Spring Cloud – Maven start demanding for the Google Guava dependency (due to transitive 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>
```



## Trying how it works

1. Run the AxonServer using Docker command.
2. Ensure the Discovery Server and Product Service is running.
3. Ensure the ApiGateway is running.

# Send a POST request

The screenshot shows the Postman application interface. At the top, there is a navigation bar with 'Home', 'Workspaces', 'Explore', a search bar 'Search Postman', and account options 'Sign In' and 'Create Account'. Below the navigation bar, a list of requests is shown, with one selected: 'POST http://localhost:8082/products-service/products'. The request details panel shows the method 'POST' and URL 'http://localhost:8082/products-service/products'. The 'Body' tab is selected, showing the JSON payload:

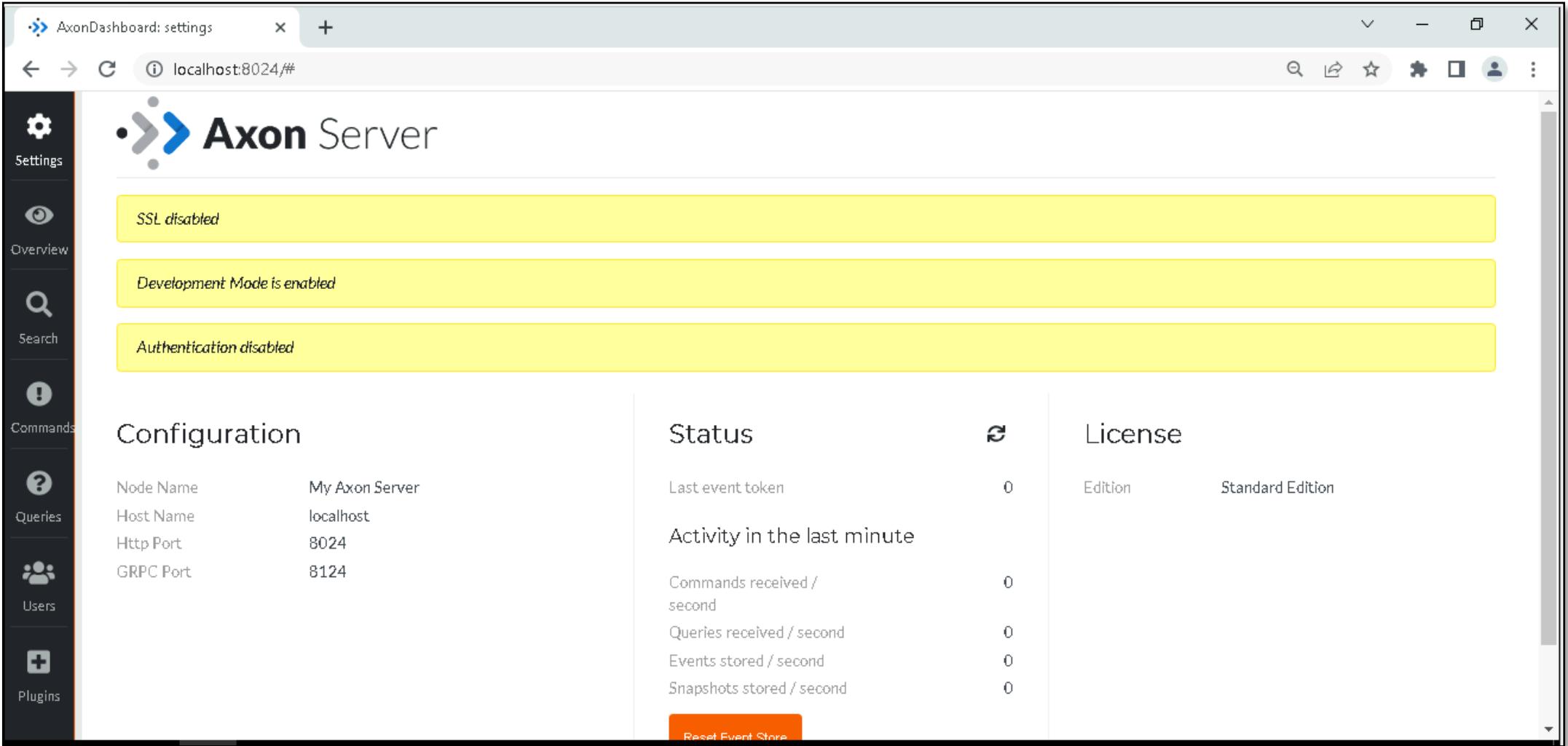
```
1 {  
2   "title": "iPhone 13",  
3   "price": 300,  
4   "quantity": 2  
5 }  
6
```

The 'Params', 'Authorization', 'Headers', 'Pre-request Script', 'Tests', and 'Settings' tabs are also visible. Below the body, the response status is shown as 'Status: 200 OK' with a timestamp of 'Time: 1m 18.78 s' and a size of 'Size: 152 B'. The 'Save Response' button is available. At the bottom, there are tabs for 'Body', 'Cookies', 'Headers', and 'Test Results', along with buttons for 'Pretty', 'Raw', 'Preview', 'Visualize', and 'Text'. The bottom right corner shows the system tray with the date '7/4/2023' and time '6:49 PM'.



# Previewing Event in the EventStore

- Go to Axon Server:



The screenshot shows the Axon Dashboard interface at `localhost:8024/#`. The left sidebar has a dark theme with icons for Settings, Overview, Search, Commands, Queries, Users, and Plugins. The main area displays the following information:

**Axon Server**

- SSL disabled
- Development Mode is enabled
- Authentication disabled

**Configuration**

Node Name	My Axon Server
Host Name	localhost
Http Port	8024
GRPC Port	8124

**Status**

Metric	Value
Last event token	0
Activity in the last minute	0
Commands received / second	0
Queries received / second	0
Events stored / second	0
Snapshots stored / second	0

**License**

Edition	Standard Edition
Edition	Standard Edition

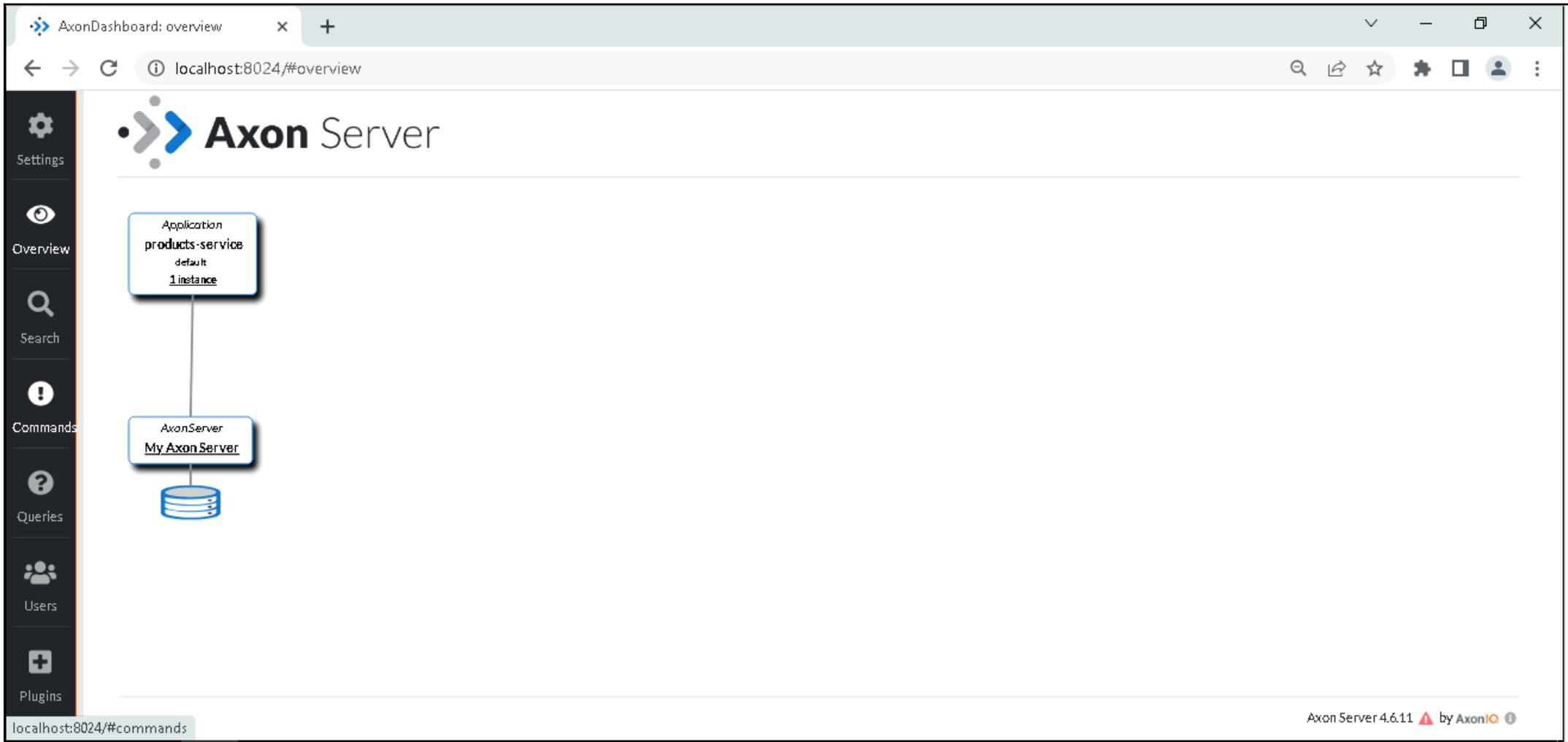
**Buttons**

- Reset Event Store



## Previewing Event in the EventStore

- Click on **Overview** tab:





## Previewing Event in the EventStore

- Select the **products-service** Application:

The screenshot shows the Axon Dashboard interface for the "products-service" application. The left sidebar contains navigation links: Settings, Overview, Search, Commands, Queries (selected), Users, and Plugins. The main content area displays the "Axon Server" logo and "Application details for products-service". It includes a "Subscription Queries" section with three metrics: #Total Subscription Queries (0), #Active Subscription Queries (0), and #Updates to Subscription Queries (0). Below this is a table showing a single instance: "3656@microsoft" under "Instance Name", "My Axon Server ← default" under "AxonServer Node", and an empty "Tags" column. Further down, a "Command" section shows "com.mphasis.command.CreateProductCommand". At the bottom, a table provides a summary of subscriptions: "Query" (empty), "Response Types" (empty), "#Subscriptions" (empty), "#Active Subscriptions" (empty), and "#Updates" (empty). The footer indicates "Axon Server 4.6.11" and "by AxonIQ".



## Previewing Event in the EventStore

- Go to **Search** tab and click on **Search** button:

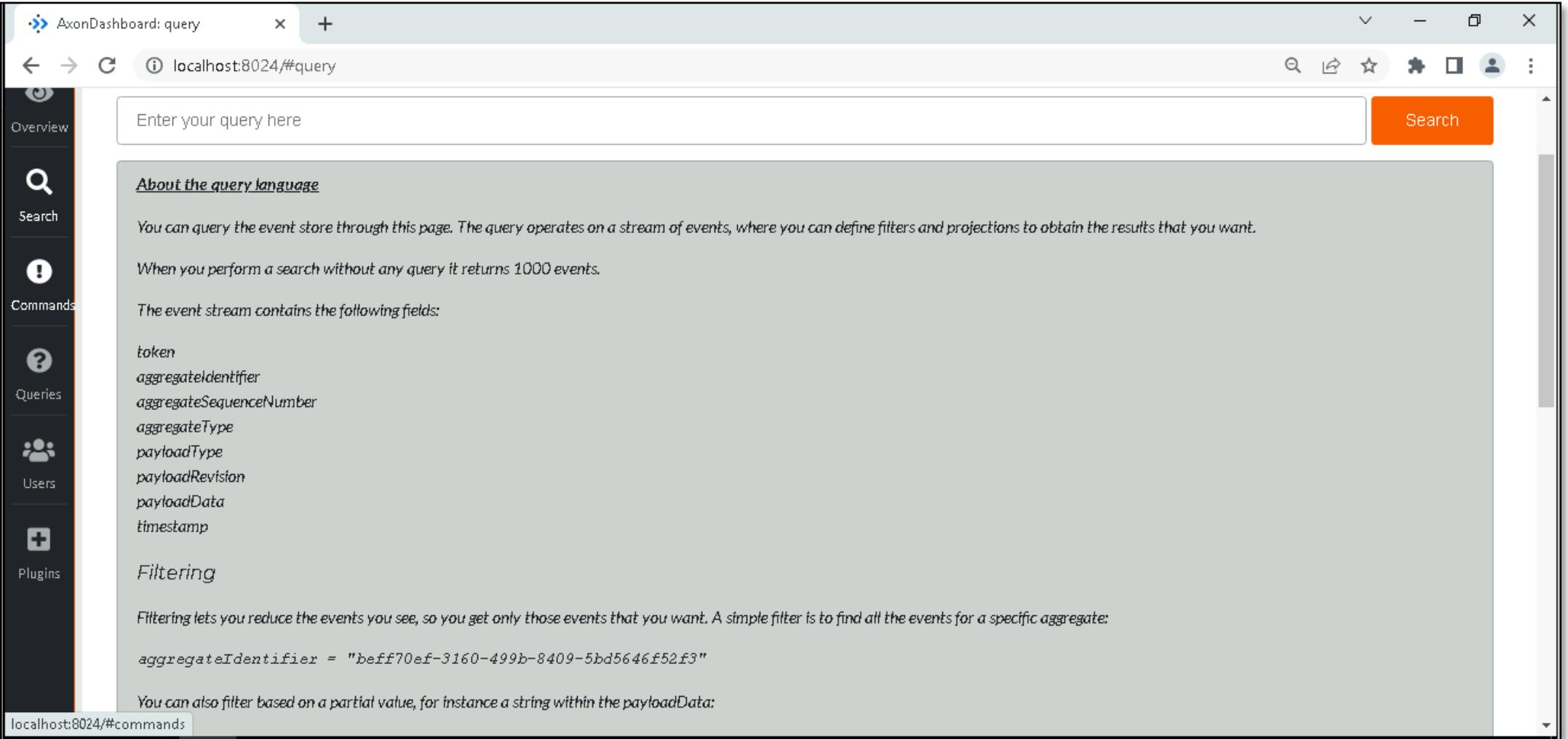
The screenshot shows the Axon Server dashboard interface. On the left, there is a vertical sidebar with icons for Settings, Overview, Search (which is selected), Commands, Queries, Users, and Plugins. The main area has a title 'Axon Server' with a logo. Below it, there are search filters: 'Search: Events' (selected) and 'Schemas'. A 'Query time window' dropdown is set to 'last hour' and a 'Live Updates' checkbox is unchecked. There is a search bar with placeholder 'Enter your query here' and an orange 'Search' button. A section titled 'About the query language' is present. The main table displays event details with columns: token, eventIdentifier, aggregateIdentifier, aggregateType, payloadType, payloadData, timestamp, and metaData. One row is shown, corresponding to the search results below. At the bottom right, there are pagination controls for 'Rows per page' (set to 10), '1 - 1 of 1', and navigation arrows. The footer indicates 'Axon Server 4.6.11' and 'by AxonIQ'.

token	eventIdentifier	aggregateIdentifier	aggregateType	payloadType	payloadData	timestamp	metaData	
0	f6ce3e9c-3fe...	51a8bedc-b3...	0	ProductAggr...	com.mphasis.events.Product...	<com.mphasis.events.ProductCreatedEve...	2023-07-04...	{traceId=c...



# Previewing Event in the EventStore

- About the **query** language:



The screenshot shows the AxonDashboard: query interface running on localhost:8024. The left sidebar has a vertical navigation menu with icons and labels: Overview (selected), Search, Commands, Queries, Users, and Plugins. The main content area displays the 'About the query language' page. It includes a search bar at the top with the placeholder 'Enter your query here' and an orange 'Search' button. Below the search bar, there's a section titled 'About the query language' with the following text:  
*You can query the event store through this page. The query operates on a stream of events, where you can define filters and projections to obtain the results that you want.*  
When you perform a search without any query it returns 1000 events.  
The event stream contains the following fields:  
`token`  
`aggregateIdentifier`  
`aggregateSequenceNumber`  
`aggregateType`  
`payloadType`  
`payloadRevision`  
`payloadData`  
`timestamp`  
A section titled 'Filtering' follows, with the text:  
*Filtering lets you reduce the events you see, so you get only those events that you want. A simple filter is to find all the events for a specific aggregate:*  
`aggregateIdentifier = "beff70ef-3160-499b-8409-5bd5646f52f3"`  
And finally, a note about partial filtering:  
*You can also filter based on a partial value, for instance a string within the payloadData:*



## Previewing Event in the EventStore

- Search using the **aggregateIdentifier**:

The screenshot shows the Axon Dashboard interface. On the left is a sidebar with icons for Settings, Overview, Search (selected), Commands, Queries, Users, and Plugins. The main area has a title 'Axon Server' with a search bar below it. The search bar contains the query 'aggregateIdentifier = "51a8bedc-b3ca-436c-b44e-64b70d0750d3"'. Below the search bar is a section titled 'About the query language'. A table is displayed with the following data:

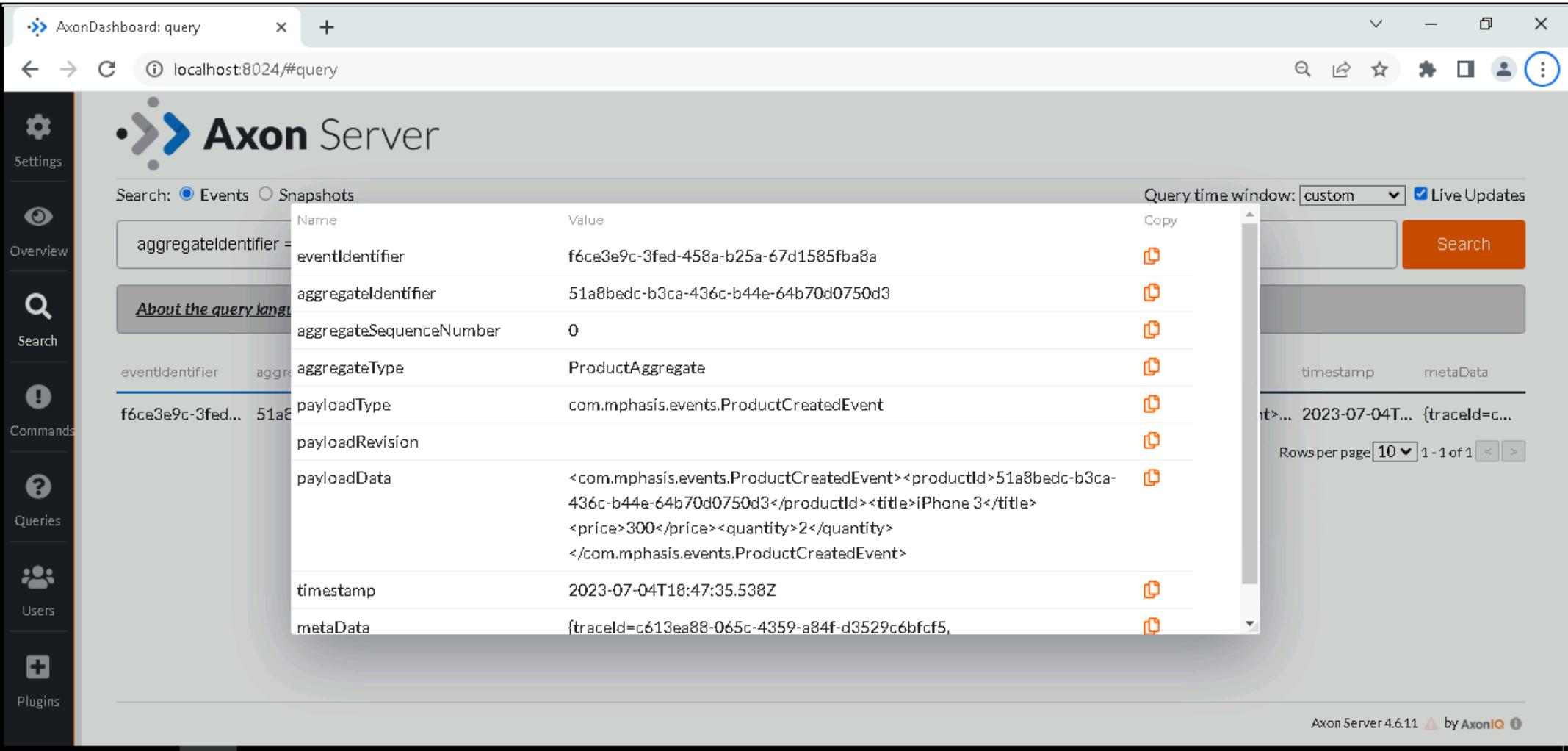
eventIdentifier	aggregateIdentifier	aggregateType	payloadType	payloadRaw	payloadData	timestamp	metaData
f6ce3e9c-3fed...	51a8bedc-b3ca...	0	ProductAggre...	com.mphasis.events.ProductCr...	<com.mphasis.events.ProductCreatedEvent>...	2023-07-04T...	{traceId=c...

At the bottom, there is a 'Rows per page' dropdown set to 10, and a footer indicating 'Axon Server 4.6.11'.



## Previewing Event in the EventStore

- View the event details available in Event Store:



The screenshot shows the Axon Server dashboard at `localhost:8024/#query`. The left sidebar has a 'Queries' tab selected. The main area displays a table of event details:

Name	Value	Actions
eventIdentifier	f6ce3e9c-3fed-458a-b25a-67d1585fba8a	<a href="#">Copy</a>
aggregateIdentifier	51a8bedc-b3ca-436c-b44e-64b70d0750d3	<a href="#">Copy</a>
aggregateSequenceNumber	0	<a href="#">Copy</a>
aggregateType	ProductAggregate	<a href="#">Copy</a>
payloadType	com.mphasis.events.ProductCreatedEvent	<a href="#">Copy</a>
payloadRevision		<a href="#">Copy</a>
payloadData	<com.mphasis.events.ProductCreatedEvent><productId>51a8bedc-b3ca-436c-b44e-64b70d0750d3</productId><title>iPhone 3</title><price>300</price><quantity>2</quantity></com.mphasis.events.ProductCreatedEvent>	<a href="#">Copy</a>
timestamp	2023-07-04T18:47:35.538Z	<a href="#">Copy</a>
metaData	{traceId=c613ea88-065c-4359-a84f-d3529c6bfcf5,	<a href="#">Copy</a>

Query time window: custom  Live Updates

Rows per page: 10 | 1 - 1 of 1

Axon Server 4.6.11 by AxonIQ



## Previewing Event in the EventStore

- Go to Commands tab:

The screenshot shows the Axon Dashboard interface with the title "AxonDashboard: commands". The browser address bar indicates the URL is "localhost:8024/#commands". On the left, a vertical sidebar menu includes "Settings", "Overview", "Search", "Commands" (which is the active tab, highlighted in blue), "Queries", "Users", and "Plugins". The main content area displays the "Axon Server" logo and the heading "Commands". Below this, a table lists a single command entry:

Command Type	Source
com.mphasis.command.CreateProductCommand	products-service@default 3656@microservices

At the bottom right of the dashboard, the text "Axon Server 4.6.11" and "by AxonIQ" is visible.

- Introduction to Axon Server
- Download and run Axon Server as JAR application
- Axon Server configuration properties
- Run Axon Server in a Docker container
- Bringing CQRS and Event Sourcing Together with Axon Framework
- Accept HTTP Request Body
- Adding Axon Framework Spring Boot Starter
- Creating a new Command class
- Send Command to a Command Gateway



## Recap of Day – 2

- Introduction to Aggregate
- Creating the Aggregate class
- Validate the command class
- Creating the event class
- Apply and Publish the Created Event
- @EventSourcingHandler Annotation
- Previewing Event in the EventStore



# Day - 3

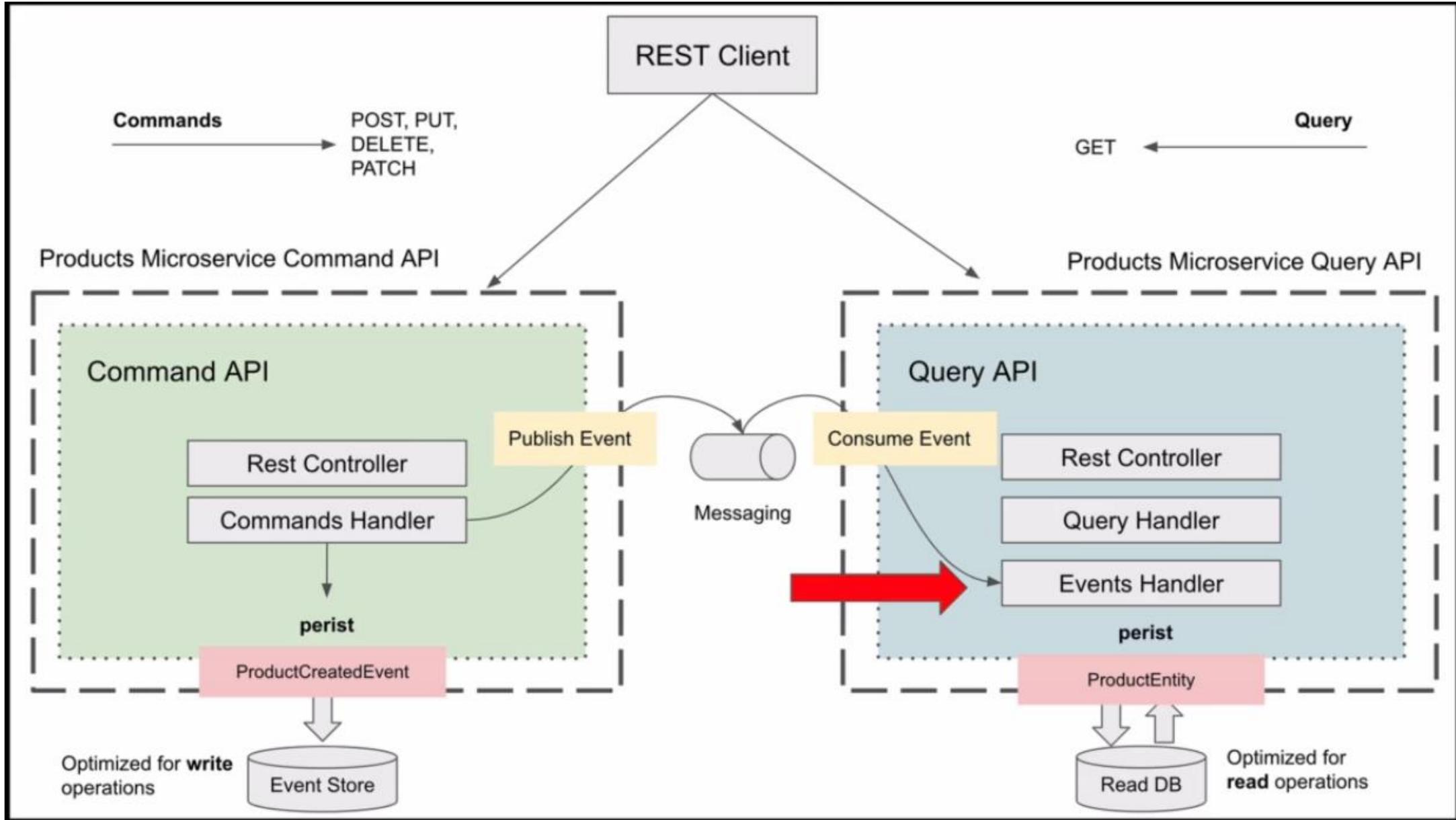
- CQRS Persisting Event in the database
- Adding Spring Data JPA & H2 dependencies
- Configure database access in the application.properties file
- Creating the Events Handler/Projection
- Implementing @EventHandler method
- CQRS, Querying Data
- Refactor Command API Rest Controller
- Create a Controller class for Query API
- Get Products Web Service Endpoint
- Implementing @QueryHandler method



Day - 3

# CQRS Persisting Event in the Product Database

# CQRS Persisting Event in the Product Database





## Adding Spring Data JPA & H2 dependencies

- Add H2 & Spring Data JPA Starters in ProductService/pom.xml.

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

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



## Configuring database access in application.properties file

- Add the DB properties in application.properties.

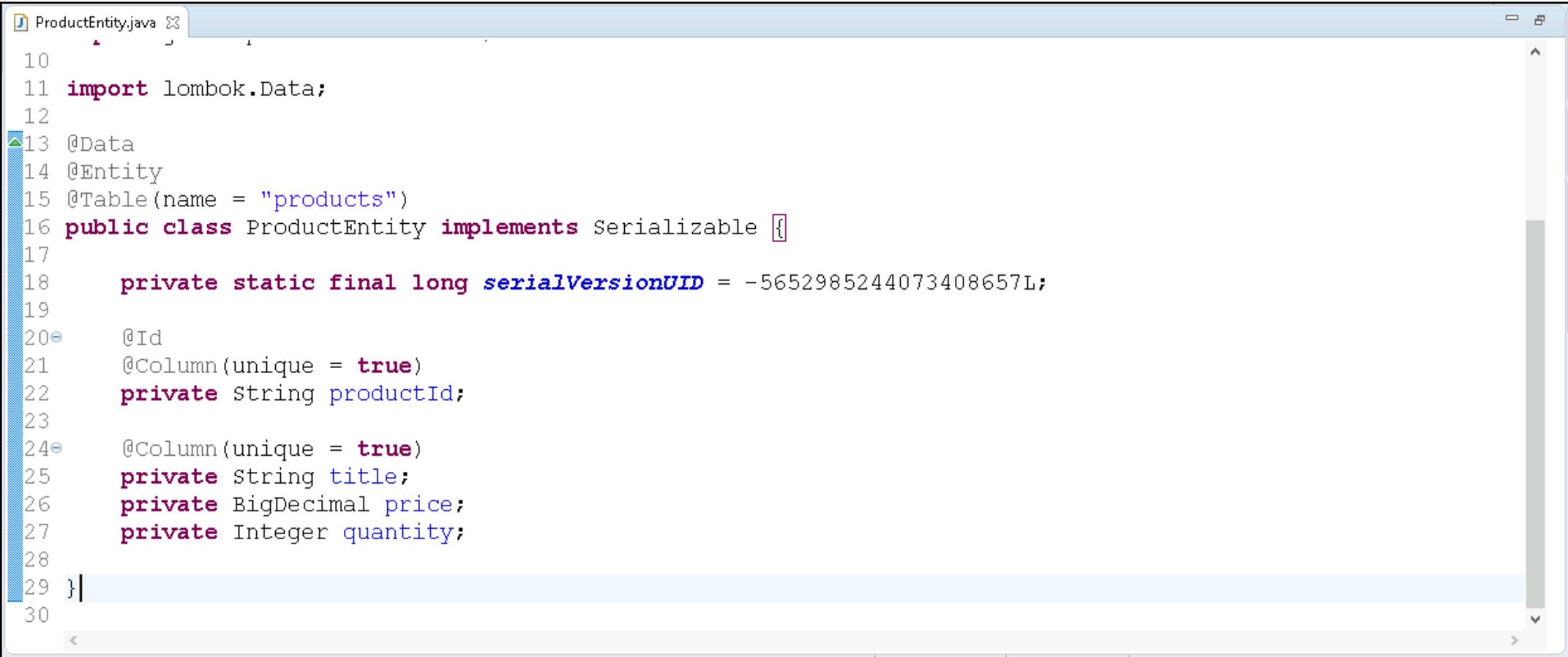
The screenshot shows a code editor window with the title bar "application.properties". The file contains configuration properties for a Spring application, specifically for a "products-service" instance. The properties include server port, Eureka client service URL, application name, instance ID, database platform (H2), H2 console settings, and JPA properties for Hibernate. The code is numbered from 1 to 22.

```
1server.port=0
2eureka.client.service-url.defaultZone=http://localhost:8761/eureka
3spring.application.name=products-service
4eureka.instance.instance-id=${spring.application.name}:${instanceId:${random.value}}
5
6
7spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
8
9spring.h2.console.settings.web-allow-others=true
10
11spring.datasource.url=jdbc:h2:mem:mphasisdb
12spring.datasource.driver-class-name=org.h2.Driver
13spring.datasource.username=sa
14spring.datasource.password=password
15
16#Accessing the H2 Console
17spring.h2.console.enabled=true
18spring.h2.console.path=/h2-console
19
20spring.jpa.properties.hibernate.show_sql=true
21spring.jpa.properties.hibernate.format_sql=true
22|
```



## Create a Product Entity class

- Create a ProductEntity Class:



```
ProductEntity.java
10
11 import lombok.Data;
12
13 @Data
14 @Entity
15 @Table(name = "products")
16 public class ProductEntity implements Serializable {
17
18     private static final long serialVersionUID = -5652985244073408657L;
19
20     @Id
21     @Column(unique = true)
22     private String productId;
23
24     @Column(unique = true)
25     private String title;
26     private BigDecimal price;
27     private Integer quantity;
28
29 }
```



## Create a Product Repository Interface

- Create a ProductRepository Interface:

The screenshot shows a Java code editor window with the file 'ProductRepository.java' open. The code defines a public interface named 'ProductRepository' that extends 'JpaRepository<ProductEntity, String>'. It includes two methods: 'findByProductId(String productId)' and 'findByProductIdOrTitle(String productId, String title)'. The code is color-coded, and the interface is highlighted with a blue selection bar.

```
1 package com.mphasis.core.data;
2
3 import org.springframework.data.jpa.repository.JpaRepository;
4
5 public interface ProductRepository extends JpaRepository<ProductEntity, String>{
6
7     ProductEntity findByProductId(String productId);
8     ProductEntity findByProductIdOrTitle(String productId, String title);
9 }
10
```



## EventHandler



### EventHandler



Annotation to be placed on methods that can handle events. The parameters of the annotated method are resolved using parameter resolvers.



## Create a Product Events Handler Class

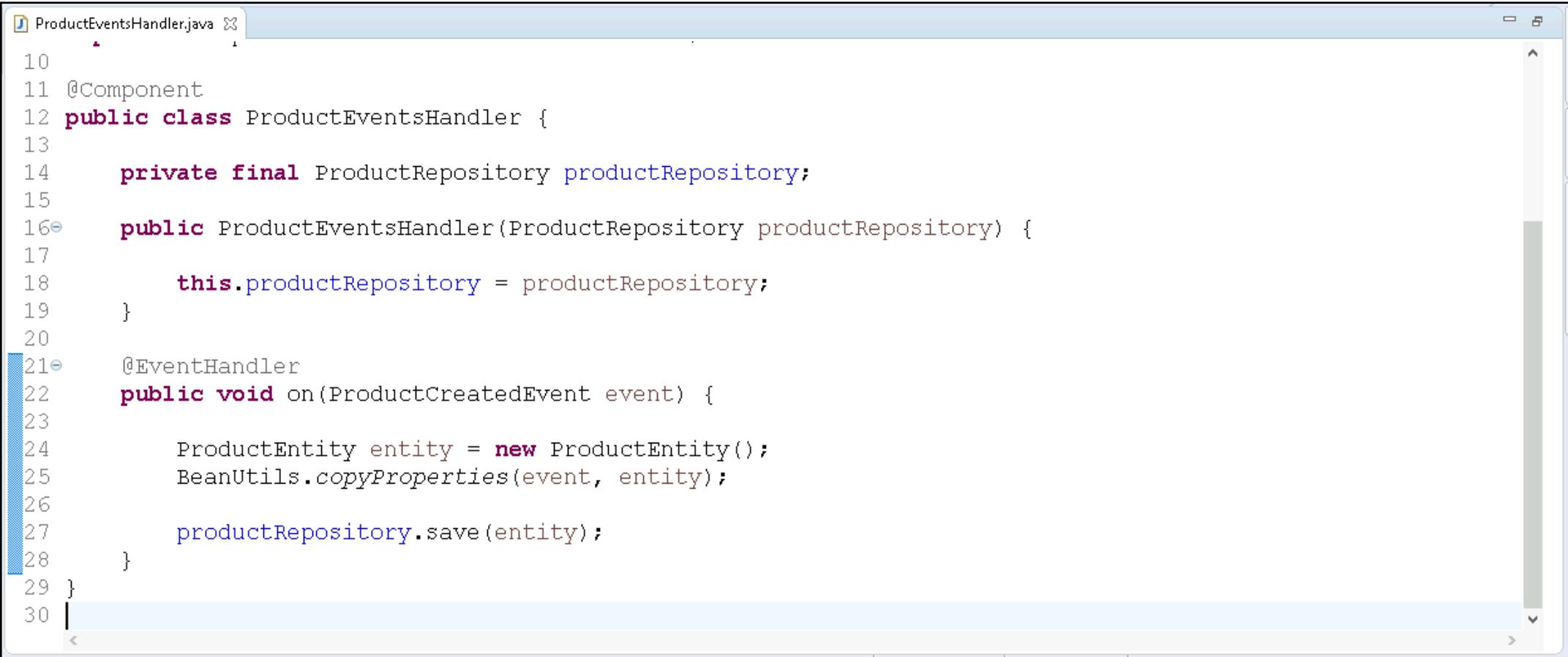
- Create a new ProductEventsHandler class:

```
ProductEventsHandler.java
1 package com.mphasis.query;
2
3 import org.axonframework.eventhandling.EventHandler;
4 import org.springframework.stereotype.Component;
5
6 import com.mphasis.events.ProductCreatedEvent;
7
8 @Component
9 public class ProductEventsHandler {
10
11     @EventHandler
12     public void on(ProductCreatedEvent event) {
13
14     }
15 }
16
```



## Implementing @EventHandler method

- Implementing @EventHandler method:



```
ProductEventsHandler.java
10
11 @Component
12 public class ProductEventsHandler {
13
14     private final ProductRepository productRepository;
15
16     public ProductEventsHandler(ProductRepository productRepository) {
17
18         this.productRepository = productRepository;
19     }
20
21     @EventHandler
22     public void on(ProductCreatedEvent event) {
23
24         ProductEntity entity = new ProductEntity();
25         BeanUtils.copyProperties(event, entity);
26
27         productRepository.save(entity);
28     }
29 }
30
```



## Trying how it works

1. Add a breakpoint in the **on** method of ProductEventsHandler class.
2. Run the AxonServer using Docker command.
3. Ensure the Discovery Server (Eureka Server) is running.
4. Execute the Product Service in Debug mode.
5. Ensure the ApiGateway is running.

# Let's create one more Product

The screenshot shows the Postman application interface. At the top, there is a navigation bar with 'Home', 'Workspaces', 'Explore', a search bar 'Search Postman', and account options 'Sign In' and 'Create Account'. Below the navigation bar, a header bar shows a 'POST' method and the URL 'http://localhost:8082/products-service/products'. To the right of the URL are buttons for 'Add to collection' and a close button. The main content area contains a request configuration panel. The 'Method' dropdown is set to 'POST' and the 'URL' field is 'http://localhost:8082/products-service/products'. Below this, tabs for 'Params', 'Authorization', 'Headers (8)', 'Body', 'Pre-request Script', 'Tests', and 'Settings' are visible, with 'Headers' currently selected. Under 'Body', the type is set to 'JSON' and the content is a JSON object:

```
1 {  
2   "title": "iPhone 1115",  
3   "price": 500,  
4   "quantity": 2  
5 }  
6
```

Below the body panel, there are tabs for 'Body', 'Cookies', 'Headers (3)', and 'Test Results', with 'Body' currently selected. On the right side of the interface, status information is displayed: 'Status: 200 OK', 'Time: 548 ms', and 'Size: 152 B'. There is also a 'Save Response' button. At the bottom of the interface, there are buttons for 'Pretty', 'Raw', 'Preview', 'Visualize', 'Text', and a copy icon. The bottom-most part of the interface shows a 'Console' tab.

1 c26818f6-8b23-4497-ba1b-ba2d172367fe

# Preview Product record in a Database

The screenshot shows the H2 Console interface running in a browser tab titled "AxonDashboard: query". The URL is "host.docker.internal:51578/h2-console/login.do?jsessionid=d0b63eeee1c64a142bb66a9af76e16e0". The left sidebar lists database objects: "ASSOCIATION\_VALUE\_ENTRY", "PRODUCTS", "SAGA\_ENTRY", "TOKEN\_ENTRY", "INFORMATION\_SCHEMA", "Sequences", and "Users". The main area contains a SQL statement "SELECT \* FROM PRODUCTS" and its execution results.

SQL statement:

```
SELECT * FROM PRODUCTS;
```

PRODUCT_ID	PRICE	QUANTITY	TITLE
51a8bedc-b3ca-436c-b44e-64b70d0750d3	300.00	2	iPhone 3
c26818f6-8b23-4497-ba1b-ba2d172367fe	500.00	2	iPhone 1115

(2 rows, 0 ms)

Edit



Day - 3

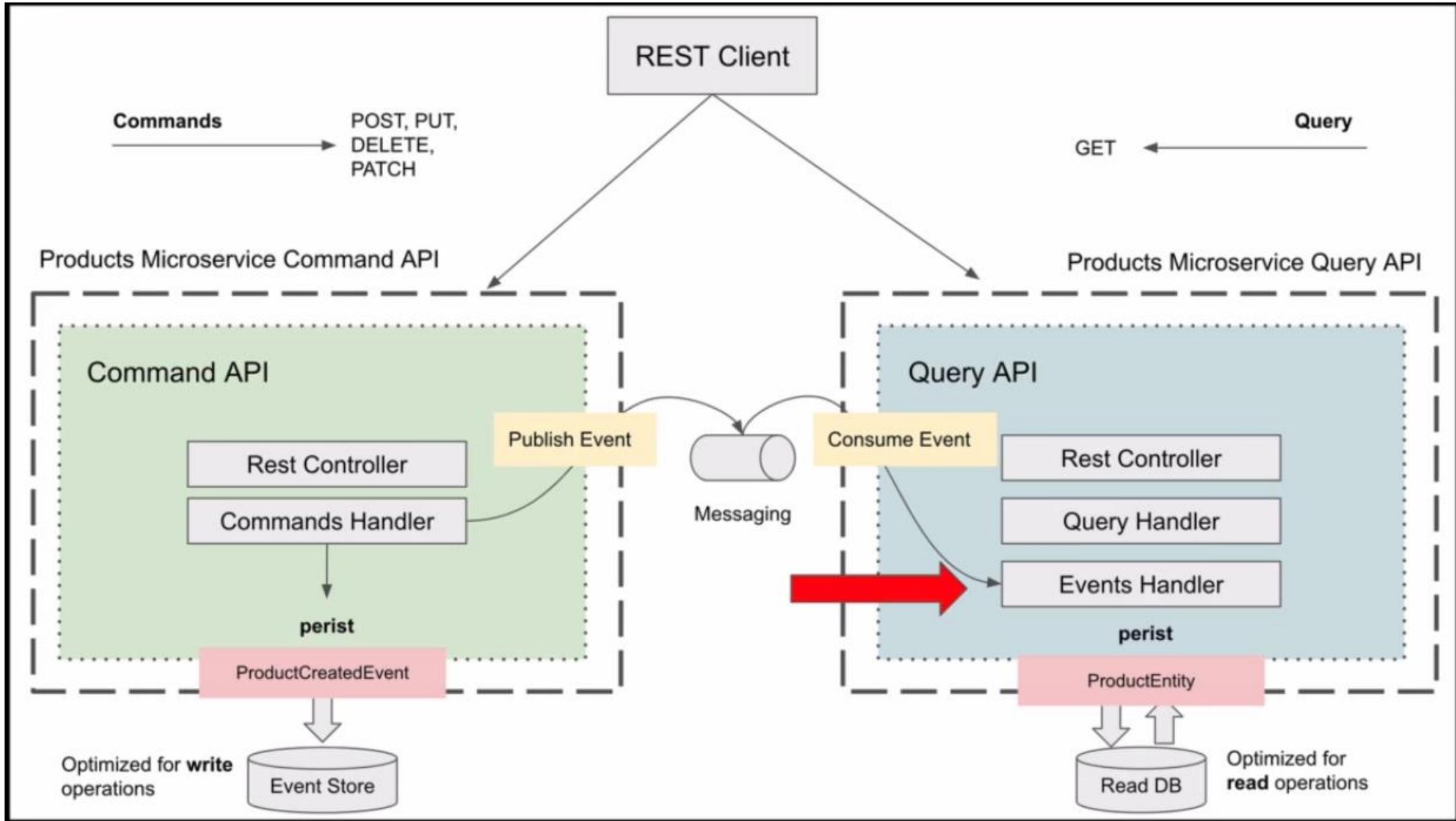
# CQRS, Querying Data



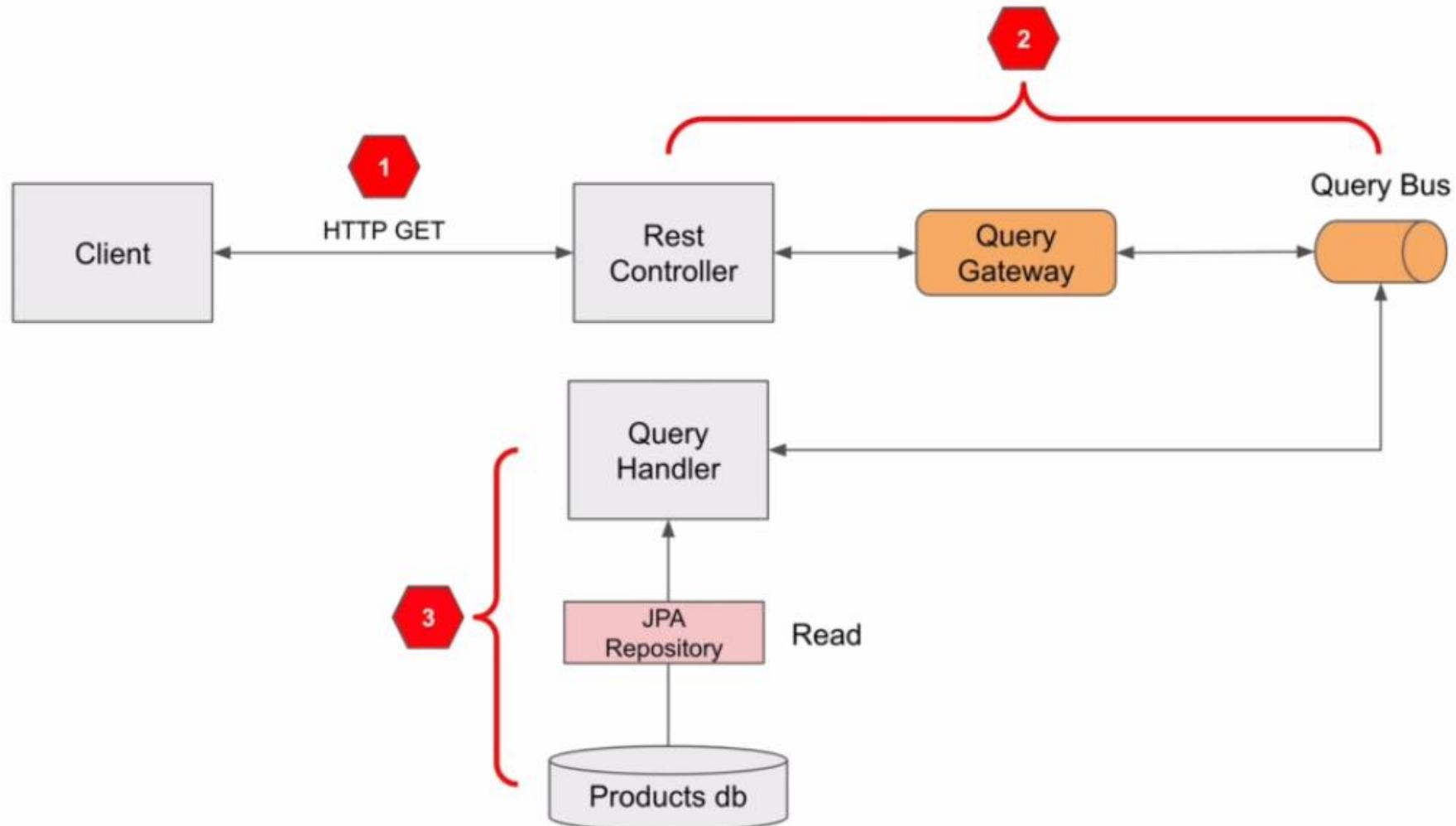
## CQRS, Querying Data

- Query - express the desire for information.
- Ex: FindProductQuery, GetUserQuery

# CQRS, Querying Data



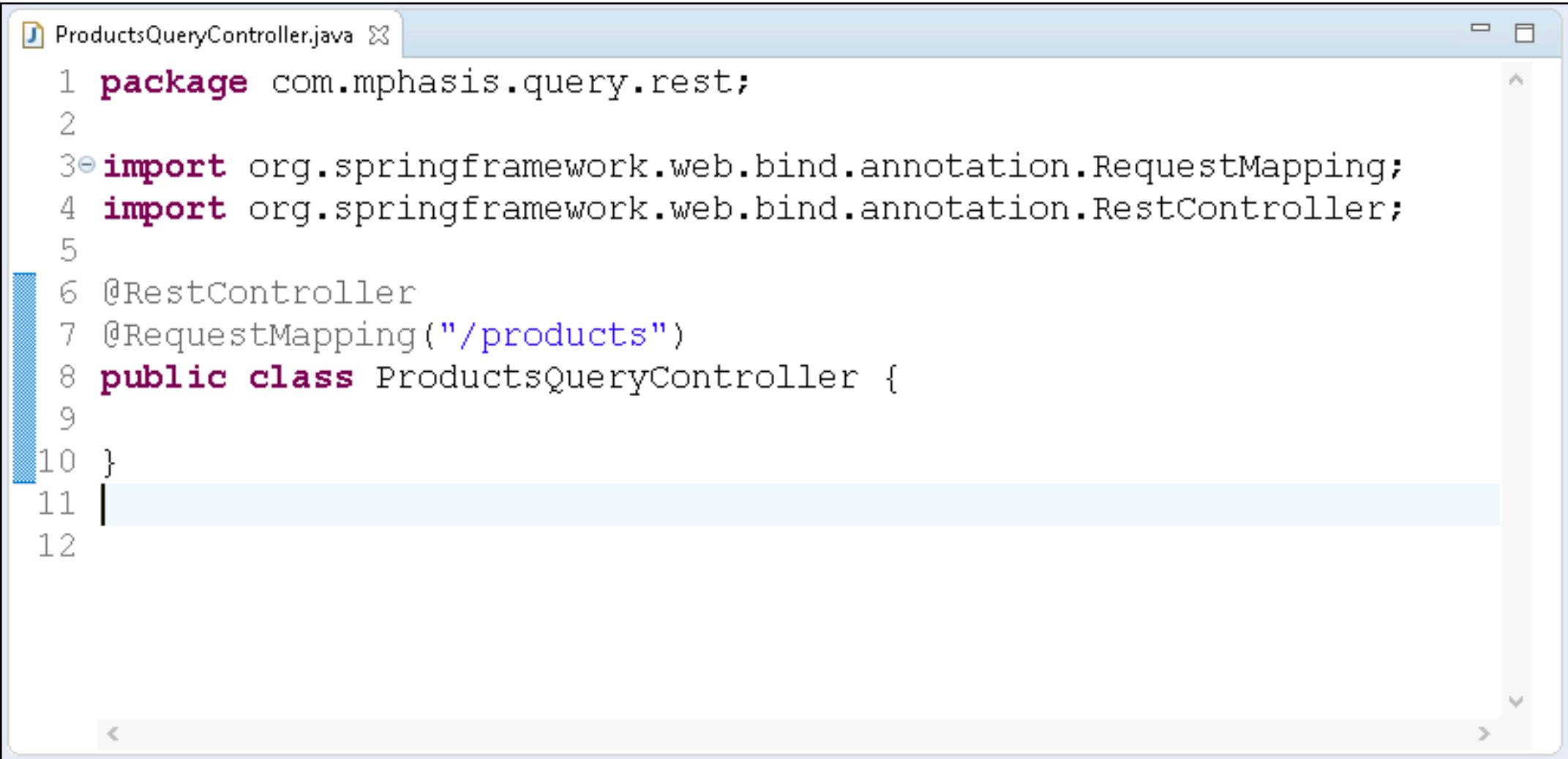
# CQRS, Querying Data





## Create a Controller class for Query API

- Create a separate Controller class for Query API:



The screenshot shows a Java code editor window with the file 'ProductsQueryController.java' open. The code defines a REST controller for products. It includes imports for RequestMapping and RestController annotations, and uses @RestController and @RequestMapping("/products") to map the class to the '/products' endpoint.

```
1 package com.mphasis.query.rest;
2
3 import org.springframework.web.bind.annotation.RequestMapping;
4 import org.springframework.web.bind.annotation.RestController;
5
6 @RestController
7 @RequestMapping("/products")
8 public class ProductsQueryController {
9
10 }
11
12
```



## 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.events to com.mphasis.core.events
- So, total we will have 3 main package – command, core, and query.





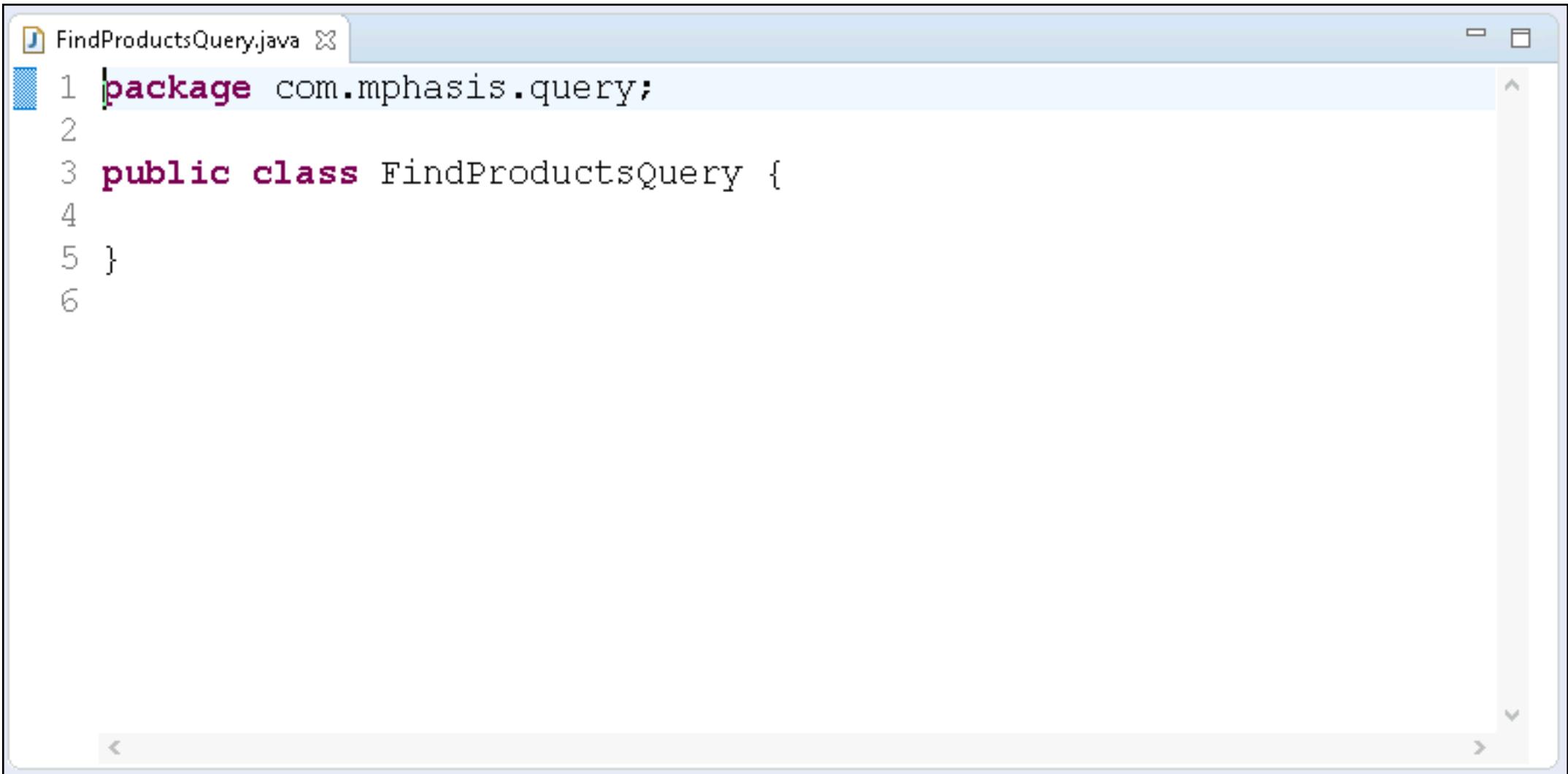
## Get Products Web Service Endpoint

- Create a ProductRestModel class:

The screenshot shows a Java code editor window with the title bar "ProductRestModel.java". The code itself is as follows:

```
1 package com.mphasis.query.rest;
2
3 import java.math.BigDecimal;
4
5 import lombok.Data;
6
7 @Data
8 public class ProductRestModel {
9
10     private String productId;
11     private String title;
12     private BigDecimal price;
13     private Integer quantity;
14 }
15 |
```

- Query the QueryGateway:



The screenshot shows a Java code editor window with a single file named "FindProductsQuery.java". The code is as follows:

```
1 package com.mphasis.query;
2
3 public class FindProductsQuery {
4
5 }
6
```



## QueryGateway



### QueryGateway



Interface towards the Query Handling components of an application.  
This interface provides a friendlier API toward the query bus.

- Query the QueryGateway:



```
ProductsQueryController.java
1 package com.mphasis.query.rest;
2
3+import java.util.List;
4
5 @RestController
6 @RequestMapping("/products")
7 public class ProductsQueryController {
8
9     @Autowired
10    private QueryGateway queryGateway;
11
12    @GetMapping
13    public List<ProductRestModel> getProducts() {
14
15        FindProductsQuery findProductsQuery = new FindProductsQuery();
16        List<ProductRestModel> products = queryGateway.query(findProductsQuery,
17            ResponseTypes.multipleInstancesOf(ProductRestModel.class)).join();
18
19        return products;
20    }
21}
```



## QueryHandler



### QueryHandler

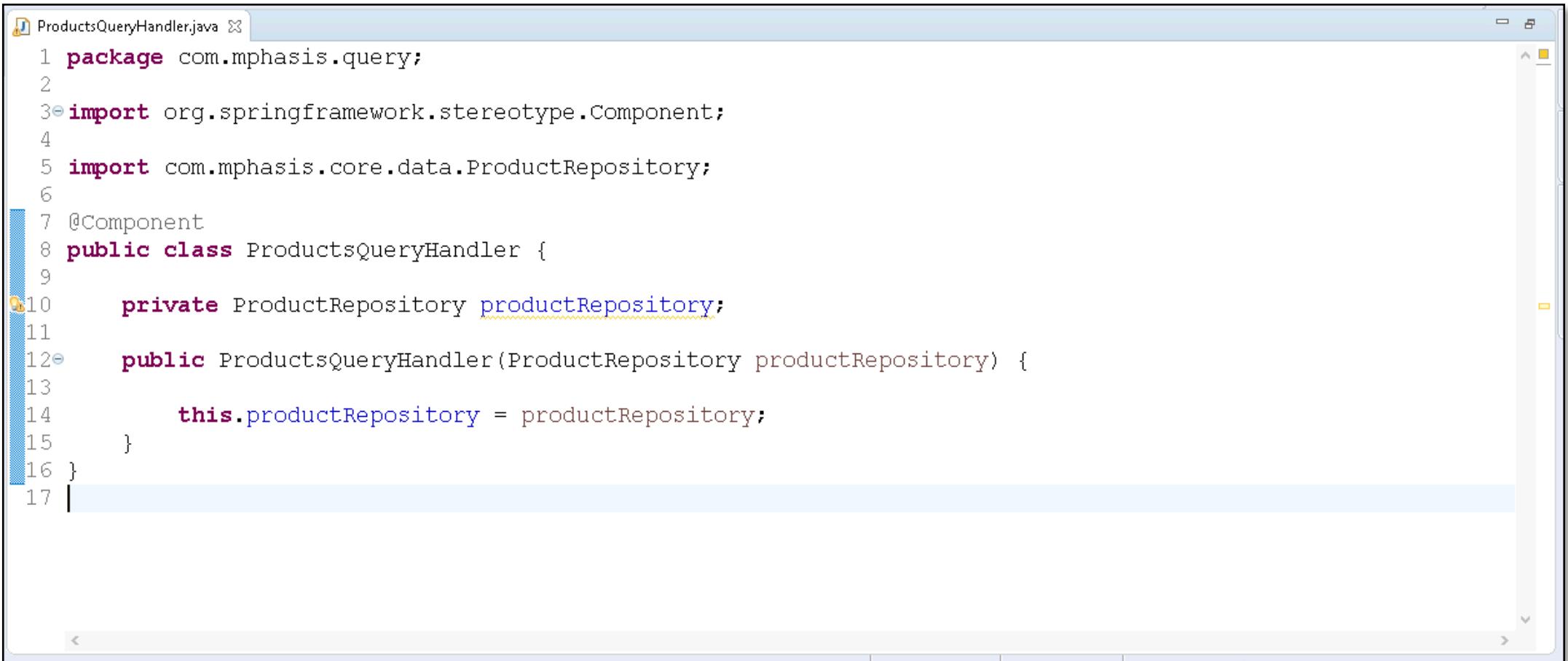


Marker annotation to mark any method on an object as being a QueryHandler.



## Create a ProductsQueryHandler class

- Create an ProductQueryHandler class:



The screenshot shows a Java code editor window with the title bar "ProductsQueryHandler.java". The code is as follows:

```
1 package com.mphasis.query;
2
3 import org.springframework.stereotype.Component;
4
5 import com.mphasis.core.data.ProductRepository;
6
7 @Component
8 public class ProductsQueryHandler {
9
10     private ProductRepository productRepository;
11
12     public ProductsQueryHandler(ProductRepository productRepository) {
13
14         this.productRepository = productRepository;
15     }
16 }
17
```



## Implementing the findProducts() method

- Implementing the findProducts() method:

```
@QueryHandler
public List<ProductRestModel> findProducts(FindProductsQuery query) {

    List<ProductRestModel> productsRest = new ArrayList<>();

    List<ProductEntity> storedProducts = productRepository.findAll();

    for (ProductEntity productEntity : storedProducts) {

        ProductRestModel productRestModel = new ProductRestModel();
        BeanUtils.copyProperties(productEntity, productRestModel);
        productsRest.add(productRestModel);
    }

    return productsRest;
}
```



## Trying how it works

1. Run the AxonServer using Docker command.
2. Ensure the Discovery Server (Eureka Server) and Product Service is running.
3. Ensure the ApiGateway is running.

# Send a POST request to Create Product

The screenshot shows the Postman application interface. At the top, there is a navigation bar with 'Home', 'Workspaces', 'Explore', a search bar 'Search Postman', and account options 'Sign In' and 'Create Account'. Below the navigation bar, a header bar displays two items: 'POST http://localhost:8082/p' and 'GET http://localhost:8082/prc'. The main workspace shows a POST request to 'http://localhost:8082/products-service/products'. The 'Body' tab is selected, showing JSON input:

```
1 {  
2   "title": "iPhone 3",  
3   "price": 200,  
4   "quantity": 2  
5 }  
6
```

The response status is shown as 'Status: 200 OK' with a timestamp of 'Time: 201 ms' and a size of 'Size: 152 B'. The response body contains the ID: 'a0d59bd6-d53b-45d2-a0bf-1ec49ad6bbf4'. The bottom of the screen features a 'Console' tab.

# Send a GET request to Query the Products

The screenshot shows the Postman application interface. At the top, there's a navigation bar with 'Home', 'Workspaces', 'Explore', a search bar 'Search Postman', and account options 'Sign In' and 'Create Account'. Below the navigation is a list of requests. The first request is a POST to 'http://localhost:8082/p'. The second request, highlighted in green, is a GET to 'http://localhost:8082/products-service/products'. This request has 'GET' selected in the method dropdown and the URL 'http://localhost:8082/products-service/products' in the body field. To the right of the body field are 'Send' and a dropdown menu. Below the request details are tabs for 'Params', 'Authorization', 'Headers (6)', 'Body', 'Pre-request Script', 'Tests', 'Settings', and 'Cookies'. The 'Params' tab is active, showing a table for 'Query Params' with two rows: one for 'Key' and one for 'Value'. Under the 'Body' tab, the response is displayed in 'Pretty' JSON format. The JSON output is:

```
1 [  
2   {  
3     "productId": "a0d59bd6-d53b-45d2-a0bf-1ec49ad6bbf4",  
4     "title": "iPhone 3",  
5     "price": 200.00,  
6     "quantity": 2  
7   }  
8 ]
```

At the bottom of the interface, there are tabs for 'Body', 'Cookies', 'Headers (3)', and 'Test Results'. On the right side, there are status indicators: 'Status: 200 OK', 'Time: 66 ms', 'Size: 217 B', and a 'Save Response' button. The bottom left corner shows a 'Console' tab.

# Preview Product record in a Database

The screenshot shows the H2 Console interface. The title bar reads "H2 Console". The address bar indicates a "Not secure" connection to "host.docker.internal:50073/h2-console/login.do?jsessionid=40c59a2c01af425f85d3468af9921240". The toolbar includes buttons for Auto commit (checked), Max rows (set to 1000), Run, Run Selected, Auto complete, Auto select (set to On), and a SQL statement input field containing "SELECT \* FROM PRODUCTS". The left sidebar lists database objects: ASSOCIATION\_VALUE\_ENTRY, PRODUCTS, SAGA\_ENTRY, TOKEN\_ENTRY, INFORMATION\_SCHEMA, Sequences, and Users. A note at the bottom left says "H2 2.1.214 (2022-06-13)". The main content area displays the query results:

```
SELECT * FROM PRODUCTS;
```

PRODUCT_ID	PRICE	QUANTITY	TITLE
a0d59bd6-d53b-45d2-a0bf-1ec49ad6bbf4	200.00	2	iPhone 3

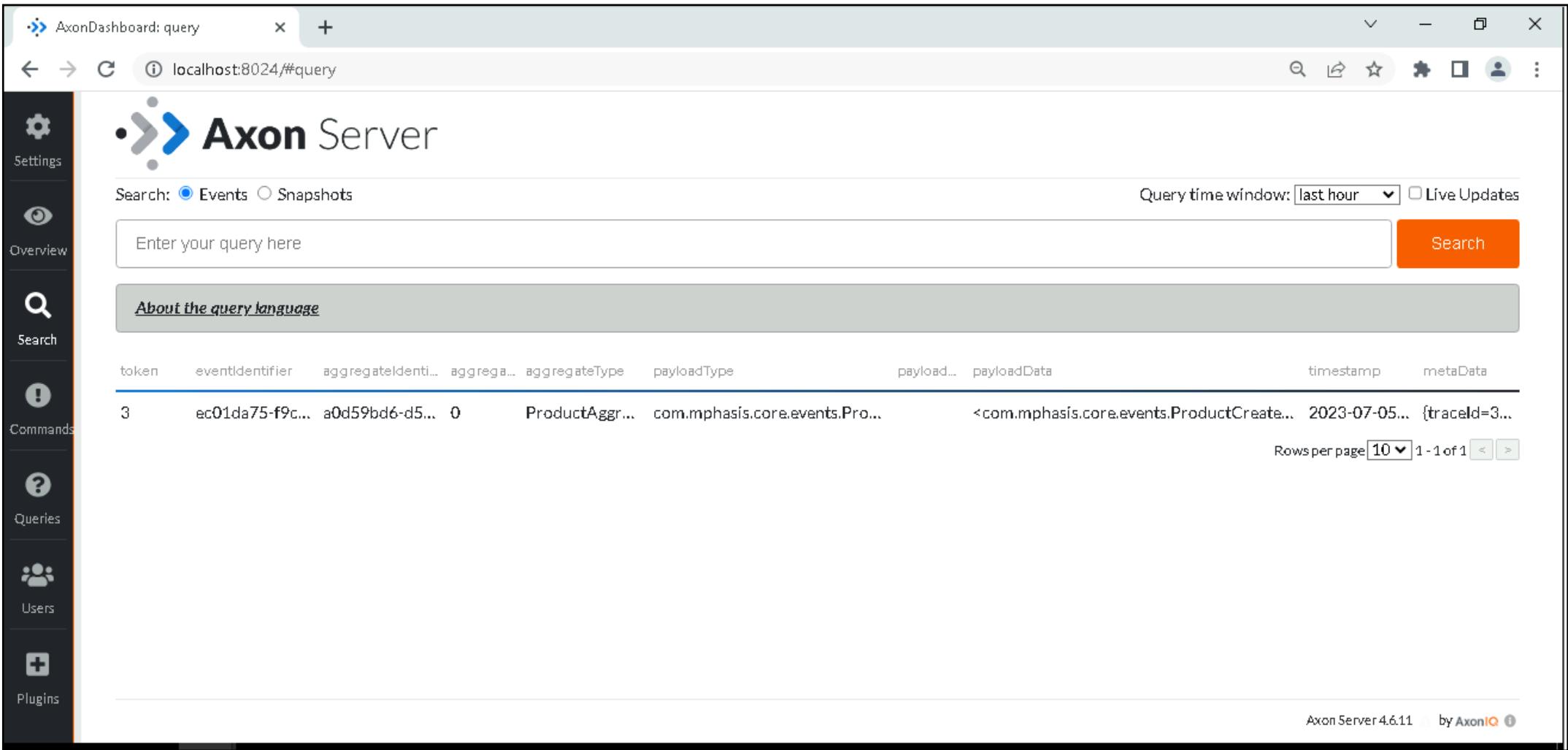
(1 row, 1 ms)

**Edit**



# Previewing Event in the EventStore

- Go to **Search** tab and click on **Search** button:



The screenshot shows the Axon Dashboard interface with the title "Axon Server". The left sidebar has a "Search" tab selected. The main area displays a table of event search results.

**Search Options:**

- Search type: Events (selected)
- Query time window: last hour
- Live Updates: Off

**Table Headers:**

token	eventIdentifier	aggregateIdent... er	aggregate... er	aggregateType	payloadType	payload... er	payloadData	timestamp	metaData
-------	-----------------	-------------------------	--------------------	---------------	-------------	------------------	-------------	-----------	----------

**Table Data:**

3	ec01da75-f9c...	a0d59bd6-d5...	0	ProductAggr...	com.mphasis.core.events.Pro...	<com.mphasis.core.events.ProductCreate...	2023-07-05...	{traceId=3...
---	-----------------	----------------	---	----------------	--------------------------------	---	---------------	---------------

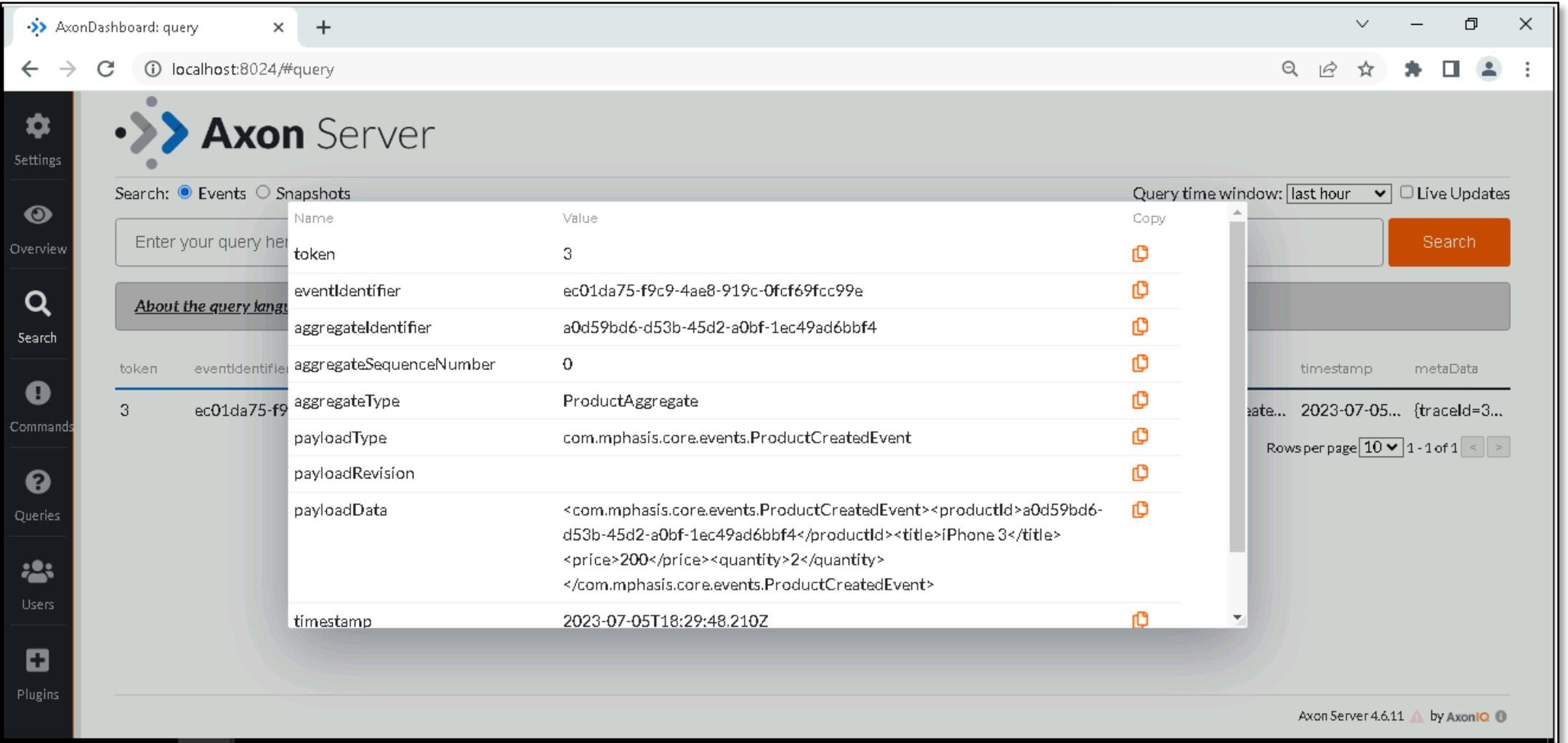
Rows per page: 10 | 1 - 1 of 1 | < >

Axon Server 4.6.11 by AxonIQ



## Previewing Event in the EventStore

- View the event details available in Event Store:



The screenshot shows the Axon Server dashboard interface. The left sidebar contains navigation links: Settings, Overview, Search, Commands, Queries, Users, and Plugins. The main area is titled "Axon Server" and displays event details. The search bar at the top says "Search: Events" and has a query input field containing "token". The event table has columns: Name, Value, and Actions (Copy, Cut, Paste). The event details are as follows:

Name	Value	Action
token	3	Copy
eventIdentifier	ec01da75-f9c9-4ae8-919c-0fcf69fcc99e	Copy
aggregateIdentifier	a0d59bd6-d53b-45d2-a0bf-1ec49ad6bbf4	Copy
aggregateSequenceNumber	0	Copy
aggregateType	ProductAggregate	Copy
payloadType	com.mphasis.core.events.ProductCreatedEvent	Copy
payloadRevision		Copy
payloadData	<com.mphasis.core.events.ProductCreatedEvent><productId>a0d59bd6-d53b-45d2-a0bf-1ec49ad6bbf4</productId><title>iPhone 3</title><price>200</price><quantity>2</quantity></com.mphasis.core.events.ProductCreatedEvent>	Copy
timestamp	2023-07-05T18:29:48.210Z	Copy

On the right side, there is a timestamped log entry: "2023-07-05T18:29:48.210Z {traceId=3...}" and a message indicating "1 - 1 of 1". The bottom right corner shows the footer: "Axon Server 4.6.11 by AxonIQ".

- CQRS Persisting Event in the database
- Adding Spring Data JPA & H2 dependencies
- Configure database access in the application.properties file
- Creating the Events Handler/Projection
- Implementing @EventHandler method
- CQRS, Querying Data
- Refactor Command API Rest Controller
- Create a Controller class for Query API
- Get Products Web Service Endpoint
- Implementing @QueryHandler method



# Day - 4

- Validating Request Body, Bean Validation
- Bean Validation – with Hibernate Validation
- Bean Validation. Enable Bean Validation
- Bean Validation. Validating Request Body
- Validation in the @CommandHandler method
- Command Validation in the Aggregate
- Introduction to Message Dispatch Interceptor
- Creating a new Command Interceptor class
- Register Message Dispatch Interceptor



## Day – 4 Agenda

- Introduction to Set Based Consistency
- Create a Product Lookup Entity
- Create a Product Lookup Repository
- Create a Product Lookup Event Handler
- Persisting information into a ProductLookup table
- Update the MessageDispatchInterceptor class

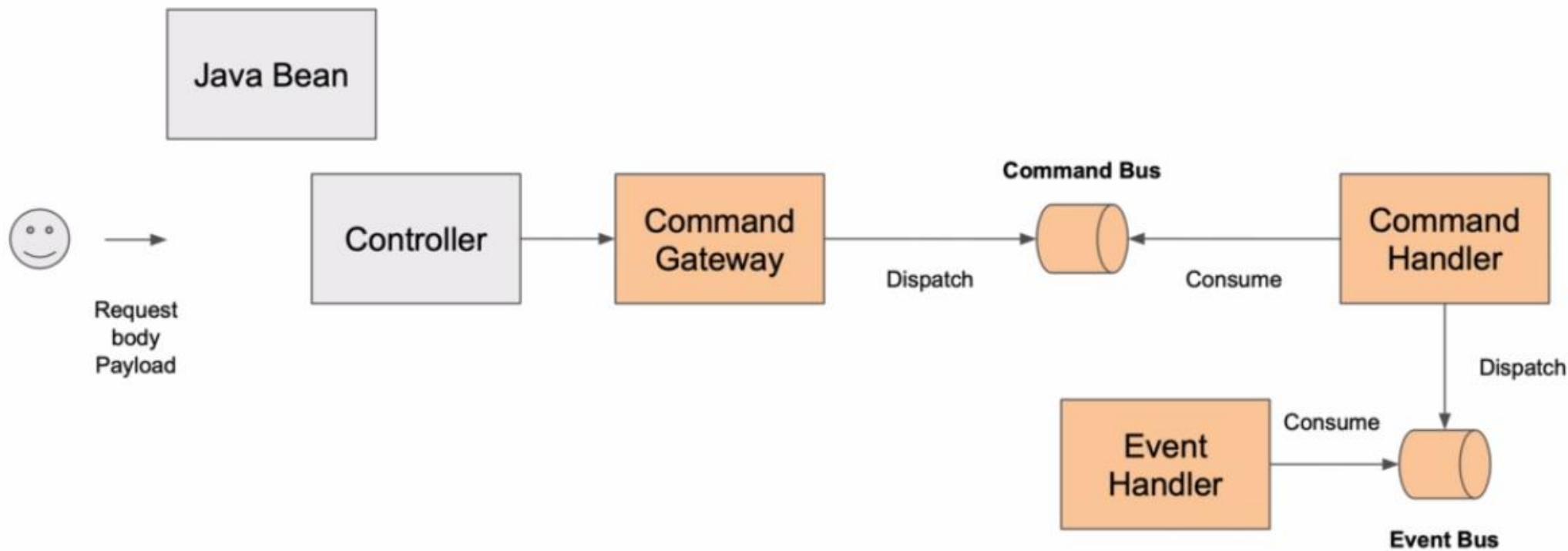


Day - 4

# Validating Request Body, Bean Validation

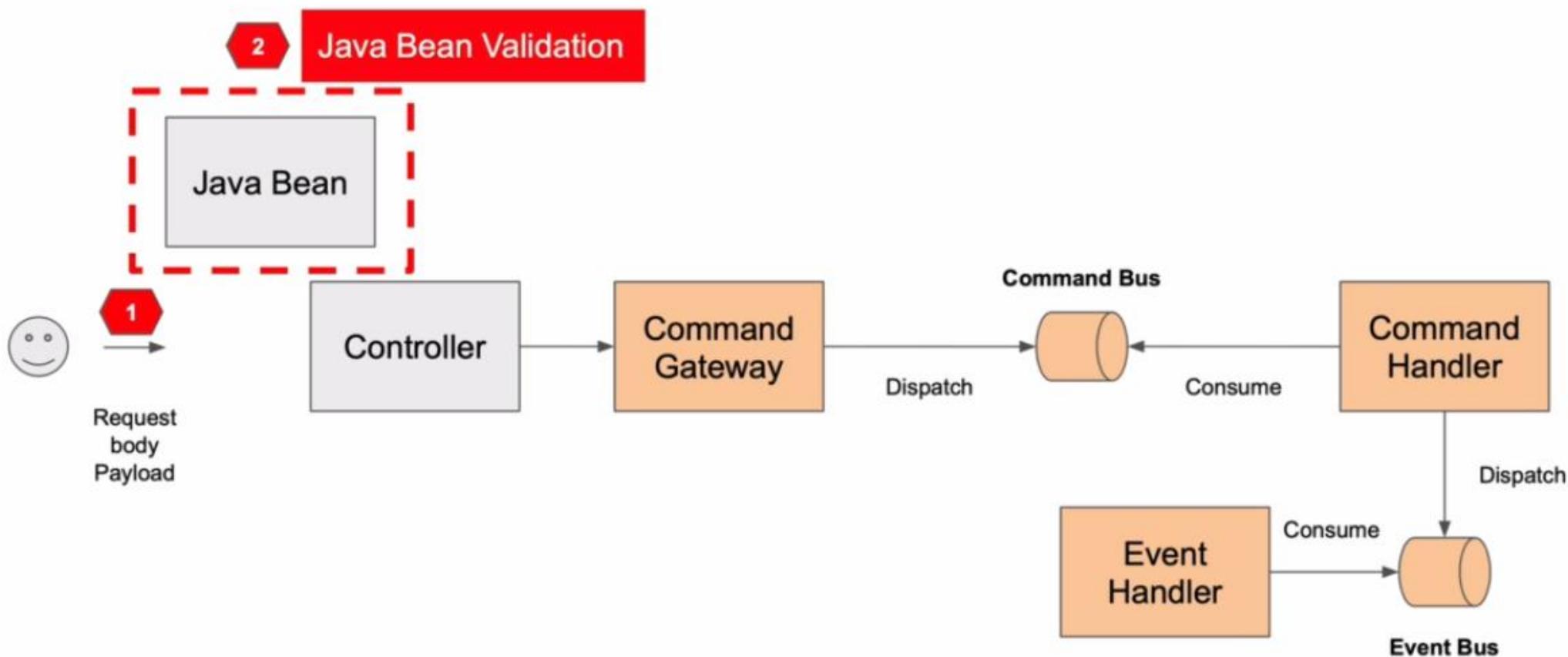
# Validating Request Body, Bean Validation

## Command API



# Validating Request Body, Bean Validation

## Command API



```
@Data  
public class CreateProductRestModel {  
  
    @NotNull(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;  
  
}
```



## Bean Validation. Enable Bean Validation

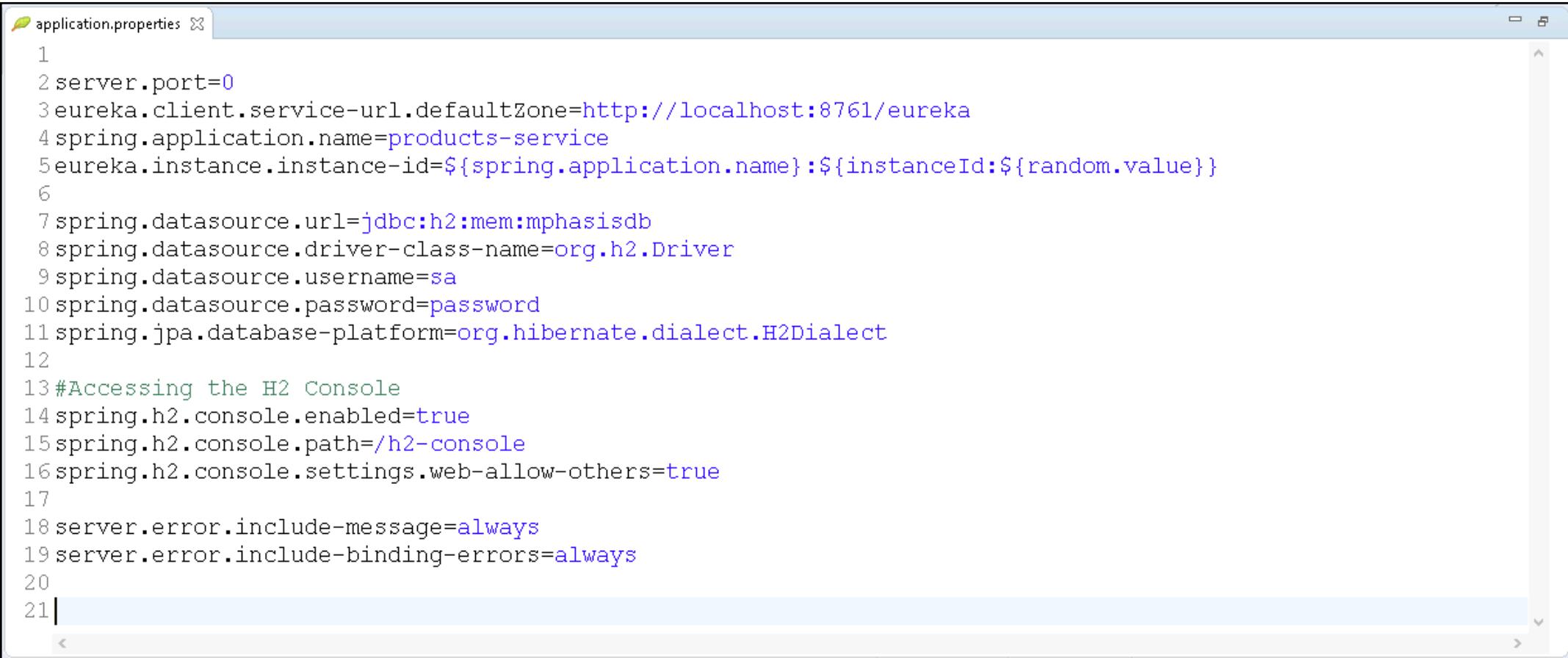
- Add the validation dependency in ProductService/pom.xml:

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



## Bean Validation. Enable Bean Validation

- Add the properties in application.properties file:



```
application.properties X
1
2server.port=0
3eureka.client.service-url.defaultZone=http://localhost:8761/eureka
4spring.application.name=products-service
5eureka.instance.instance-id=${spring.application.name}:${instanceId:${random.value}}
6
7spring.datasource.url=jdbc:h2:mem:mphasisdb
8spring.datasource.driver-class-name=org.h2.Driver
9spring.datasource.username=sa
10spring.datasource.password=password
11spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
12
13#Accessing the H2 Console
14spring.h2.console.enabled=true
15spring.h2.console.path=/h2-console
16spring.h2.console.settings.web-allow-others=true
17
18server.error.include-message=always
19server.error.include-binding-errors=always
20
21|
```



## Bean Validation. Validating Request Body

- Add the validation annotation to the fields:

```
1 package com.mphasis.command.rest;
2
3 import java.math.BigDecimal;
4
5
6 @Data
7 public class CreateProductRestModel {
8
9
10    @NotNull(message = "Product title is a required field")
11    private String title;
12
13    @Min(value=1, message = "Price cannot be lower than 1")
14    private BigDecimal price;
15
16    @Min(value=1, message = "Quantity cannot be lower than 1")
17    @Max(value=5, message = "Quantity cannot be larger than 1")
18    private Integer quantity;
19
20
21 }
```



## Bean Validation. Validating Request Body

- **@Valid annotation will trigger the validation on the request body:**

```
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      try {
39          returnValue = commandGateway.sendAndWait(createProductCommand);
40      }catch (Exception ex) {
41          returnValue = ex.getLocalizedMessage();
42      }
43      return returnValue;
44  }
45
```



## Trying how it works

1. Run the AxonServer using Docker command.
2. Ensure the Discovery Server (Eureka Server) and Product Service is running.
3. Ensure the ApiGateway is running.

# Send a POST request with title as blank

The screenshot shows the Postman application interface. At the top, there's a navigation bar with 'Home', 'Workspaces', 'Explore', a search bar 'Search Postman', and account options 'Sign In' and 'Create Account'. Below the navigation is a list of requests: 'POST http://localhost:8082/p' (selected), 'GET http://localhost:8082/prc', and others. The main workspace shows a POST request to 'http://localhost:8082/products-service/products'. The 'Body' tab is selected, showing the JSON payload:

```
1 {
2   ...
3     "title": "",
4     "price": 500,
5     "quantity": 2
}
```

The 'Headers' tab shows 8 items. The 'Tests' and 'Settings' tabs are also visible. On the right, there are buttons for 'Send' and 'Beautify'. Below the request details, the response section shows a status of '400 Bad Request'. The 'Pretty' tab is selected in the response view, displaying the error message:

```
23   },
24   ],
25   "defaultMessage": "Product title is a required field",
26   "objectName": "createProductRestModel",
27   "field": "title",
28   "rejectedValue": "",
29   "bindingFailure": false,
30   "code": "NotBlank"
31 }
```

At the bottom, there are tabs for 'Console' and other sections.

# Send a POST request with Quantity greater than 20

The screenshot shows the Postman application interface. At the top, there are navigation tabs for Home, Workspaces, and Explore, along with a search bar and account options. Below the header, a list of requests is visible, with the current one being a POST to `http://localhost:8082/products-service/products`. The request details show the method as POST, the URL, and various configuration tabs like Params, Authorization, Headers, Body, Pre-request Script, Tests, Settings, Cookies, and Beautify. The Body tab is selected, showing the JSON payload:

```
1 {
2   "title": "iPhone 5",
3   "price": 500,
4   "quantity": 20
5 }
```

Below the request details, the response section shows the status as 400 Bad Request, with time 125 ms and size 712 B. The response body is displayed in Pretty, Raw, Preview, and Visualize formats, showing an error message: "Quantity cannot be larger than 5".

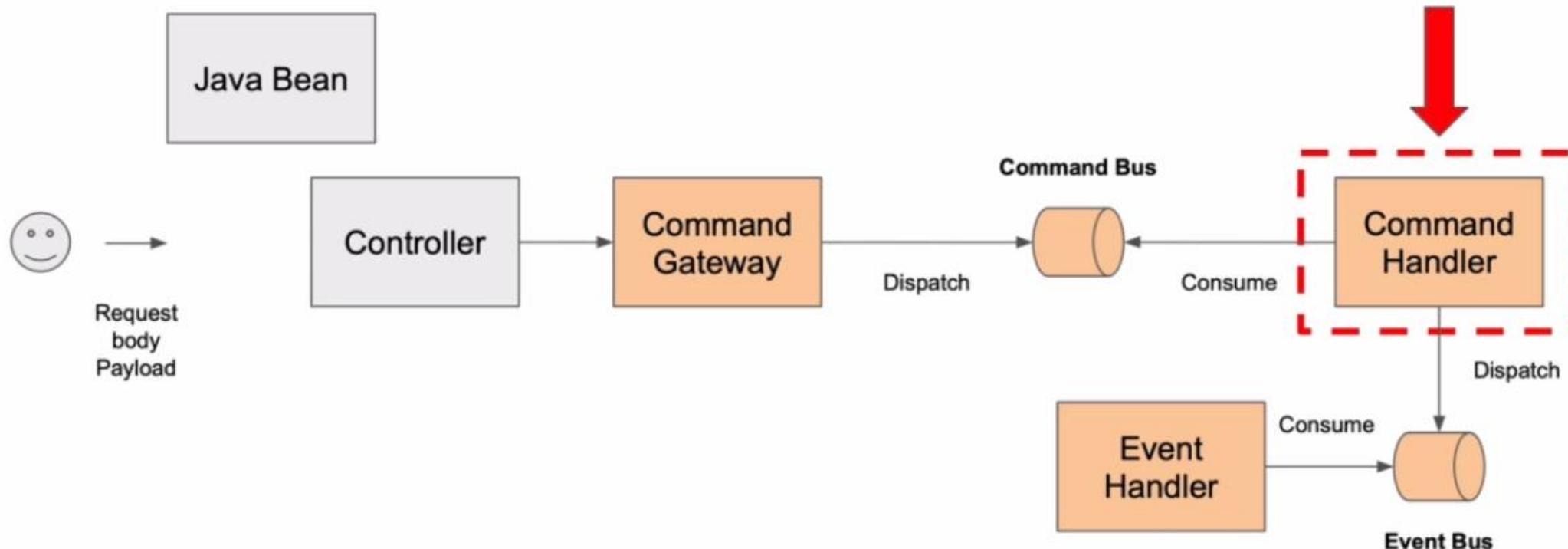


Day - 4

## Validation in the @CommandHandler method

# Validation in the @CommandHandler method

## Command API





## Command Validation in the Aggregate

```
@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().isBlank()) {
        throw new IllegalArgumentException("Title cannot be empty");
    }

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

    AggregateLifecycle.apply(productCreatedEvent);
}
```



Day - 4

# Message Dispatch Interceptor

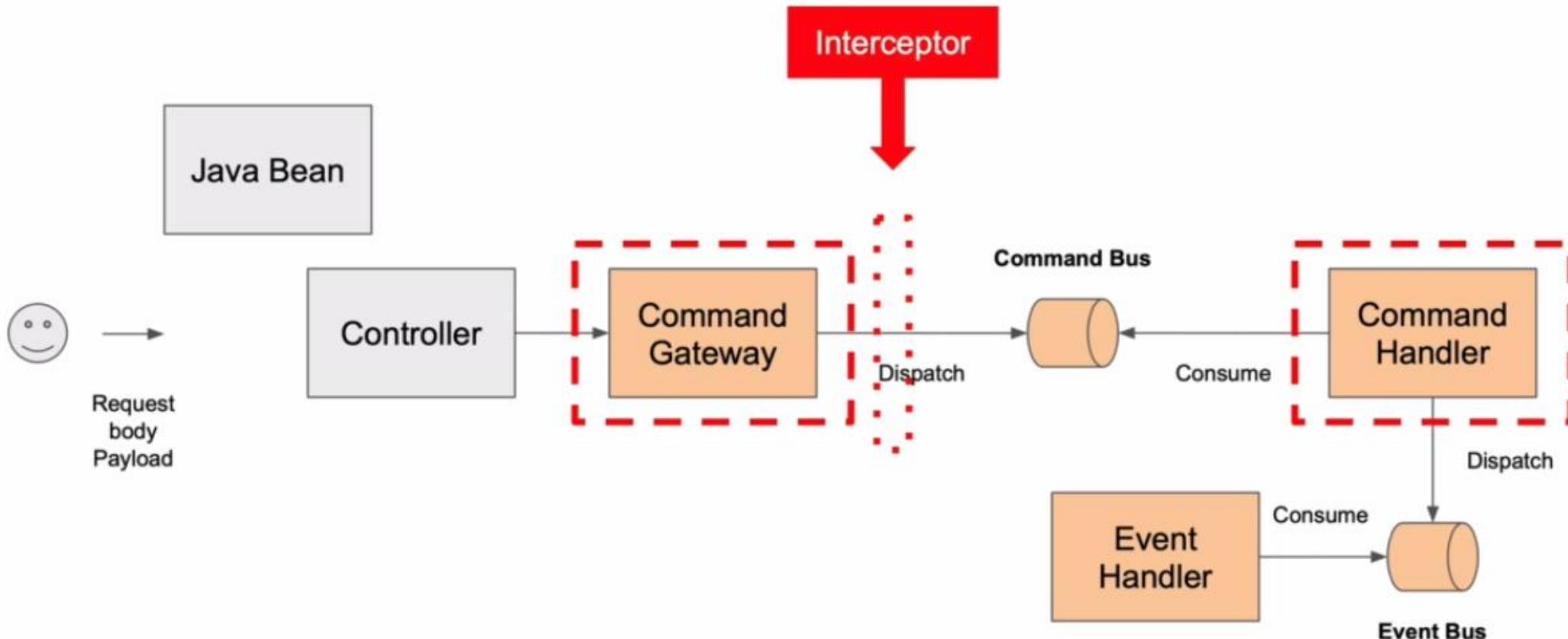


## Introduction to Message Dispatch Interceptor

- Message Dispatch Interceptor is invoked when a message is dispatch from Command Gateway to Command Bus.
- We can write a Message Dispatch Interceptor to intercept the command right where it's dispatched on a Command Bus.
- Using Message Dispatch Interceptor, you can do additional logging, you can do command validation, you can alter a command message by adding the META-DATA, you can also block the command by throwing an Exception.

# Introduction to Message Dispatch Interceptor

## Command API





## Creating a new Command Interceptor class

- Create a new CreateProductCommandInterceptor class:

```
15 @Component
16 public class CreateProductCommandInterceptor implements MessageDispatchInterceptor<CommandMessage<?>>{
17
18     private static final Logger LOGGER = LoggerFactory.getLogger(CreateProductCommandInterceptor.class);
19
20     @Override
21     public BiFunction<Integer, CommandMessage<?>, CommandMessage<?>> handle(
22         List<? extends CommandMessage<?>> messages) {
23
24         return (index, command) -> {
25
26             LOGGER.info("Intercepted command: " + command.getPayloadType());
27
28             if(CreateProductCommand.class.equals(command.getPayloadType())) {
29
30                 CreateProductCommand createProductCommand = (CreateProductCommand) command.getPayload();
31
32                 if (createProductCommand.getPrice().compareTo(BigDecimal.ZERO) <= 0) {
33                     throw new IllegalArgumentException("Price cannot be less or equal than zero");
34                 }
35
36                 if (createProductCommand.getTitle() == null || createProductCommand.getTitle().isEmpty()) {
37                     throw new IllegalArgumentException("Title cannot be empty");
38                 }
39             }
40             return command;
41         };
42     }
}
```



## Register Message Dispatch Interceptor

- Register the CreateProductCommandInterceptor class:



```
ProductServiceApplication.java
1 import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
2 import org.springframework.context.ApplicationContext;
3
4 import com.mphasis.command.interceptor.CreateProductCommandInterceptor;
5
6 @EnableDiscoveryClient
7 @SpringBootApplication
8 public class ProductServiceApplication {
9
10     public static void main(String[] args) {
11         SpringApplication.run(ProductServiceApplication.class, args);
12     }
13
14     @Autowired
15     public void registerCreateProductCommandInterceptor(
16         ApplicationContext context, CommandBus commandBus) {
17
18         commandBus.registerDispatchInterceptor(context.getBean(CreateProductCommandInterceptor.class));
19     }
20 }
```



## Trying how it works

1. Run the AxonServer using Docker command.
2. Ensure the Discovery Server (Eureka Server) and Product Service is running.
3. Ensure the ApiGateway is running.

# Send a POST request with title as blank

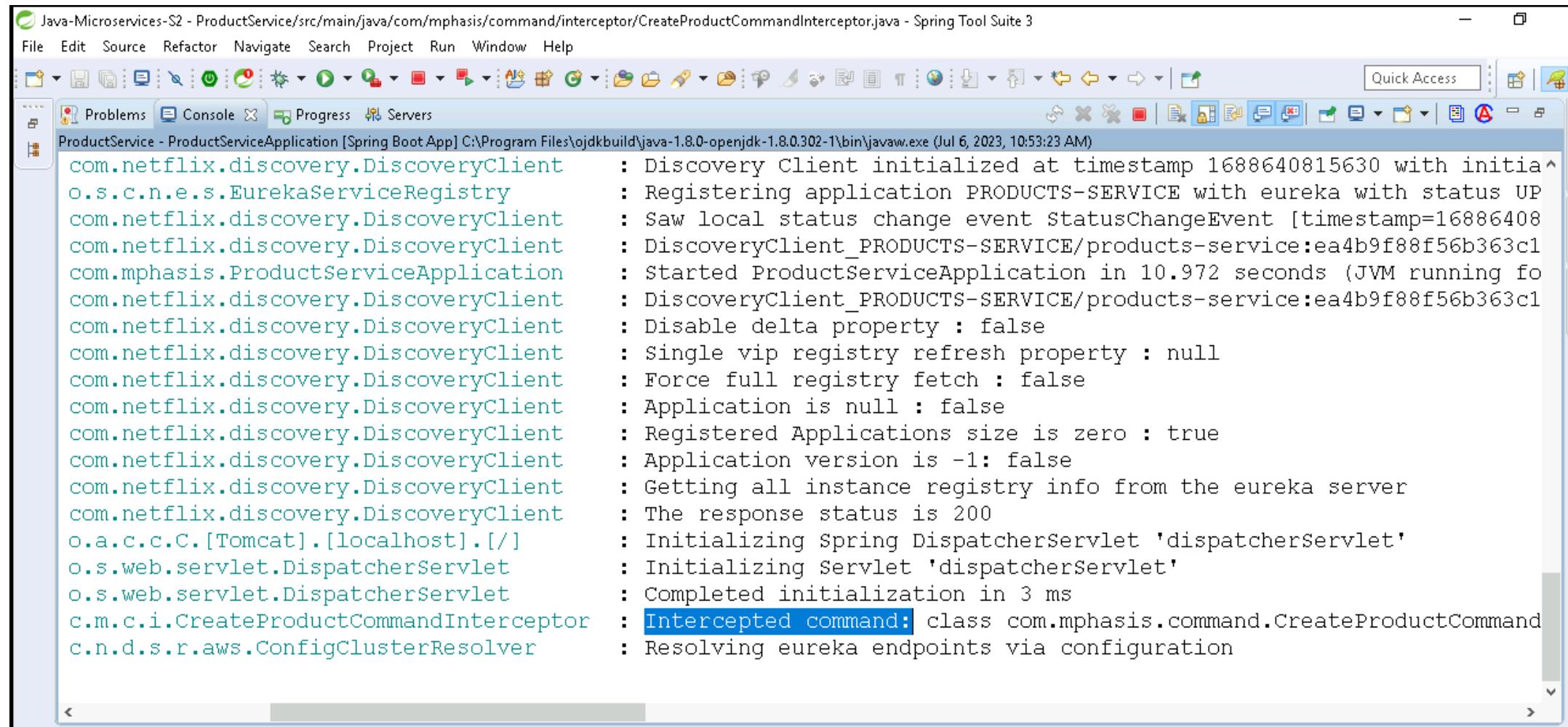
The screenshot shows the Postman application interface. At the top, there is a navigation bar with 'Home', 'Workspaces', 'Explore', a search bar 'Search Postman', and account options 'Sign In' and 'Create Account'. Below the navigation bar, a header bar displays 'POST http://localhost:8082/p' and 'GET http://localhost:8082/prc'. The main workspace shows a POST request to 'http://localhost:8082/products-service/products'. The 'Body' tab is selected, showing a JSON payload with an empty 'title' field:

```
1 {  
2   ... "title": "",  
3   ... "price": 500,  
4   ... "quantity": 5  
5 }  
6
```

The 'Pretty' tab in the preview section shows the response message: 'Title cannot be empty'.

At the bottom, the status bar indicates 'Status: 200 OK Time: 97 ms Size: 137 B' and a 'Save Response' button. The bottom left corner shows a 'Console' tab.

# Check the ProductServiceApplication Console



The screenshot shows the Spring Tool Suite 3 interface with the 'Console' tab selected. The title bar indicates the project is 'Java-Microservices-S2 - ProductService/src/main/java/com/mphasis/command/interceptor/CreateProductCommandInterceptor.java - Spring Tool Suite 3'. The console output window displays the following log entries:

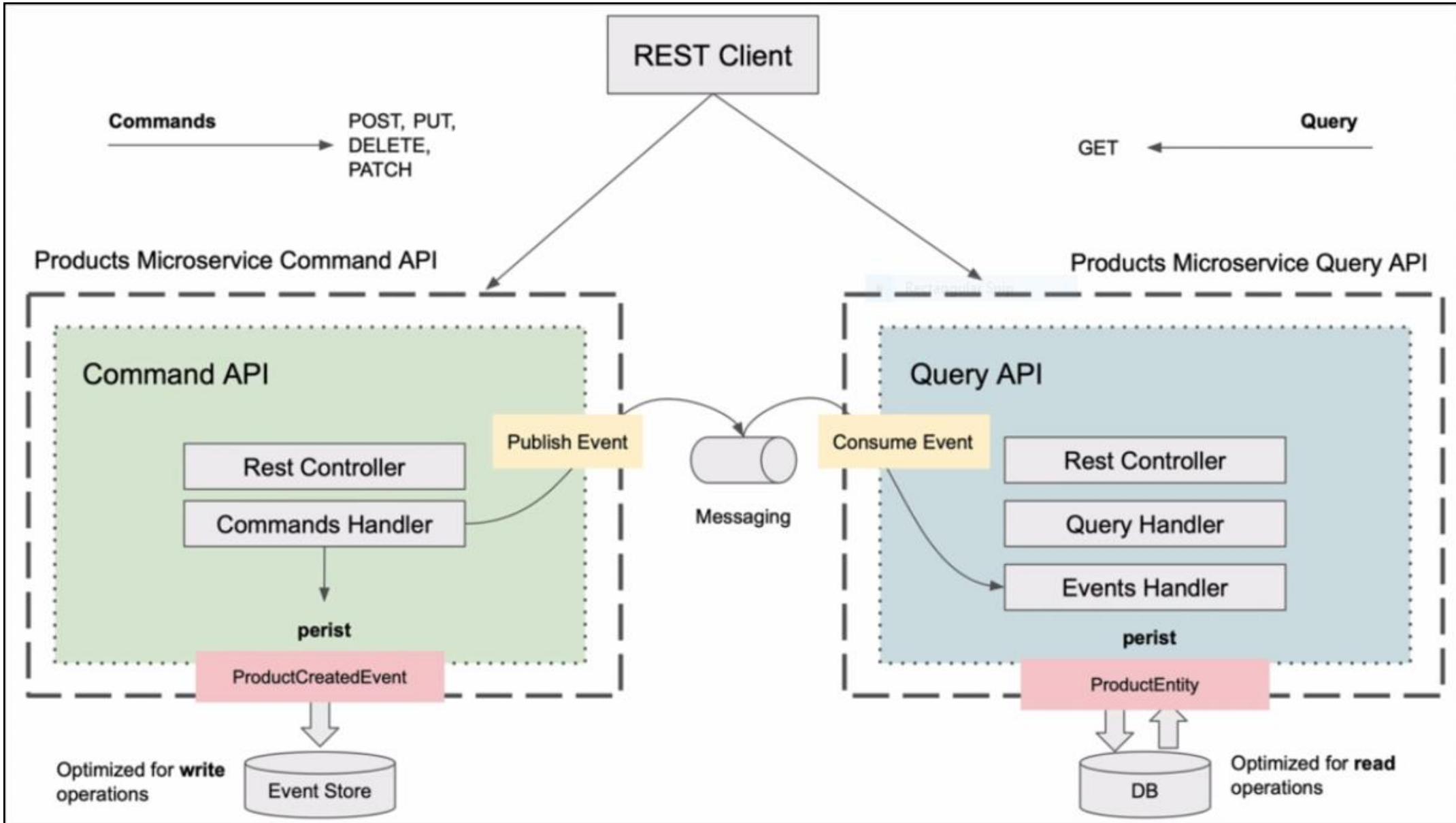
```
 ProductService - ProductServiceApplication [Spring Boot App] C:\Program Files\ojdkbuild\java-1.8.0-openjdk-1.8.0.302-1\bin\javaw.exe (Jul 6, 2023, 10:53:23 AM)
com.netflix.discovery.DiscoveryClient : Discovery Client initialized at timestamp 1688640815630 with initia^
o.s.c.n.e.s.EurekaServiceRegistry : Registering application PRODUCTS-SERVICE with eureka with status UP
com.netflix.discovery.DiscoveryClient : Saw local status change event StatusChangeEvent [timestamp=16886408
com.netflix.discovery.DiscoveryClient : DiscoveryClient_PRODUCTS-SERVICE/products-service:ea4b9f88f56b363c1
com.mphasis.ProductServiceApplication : Started ProductServiceApplication in 10.972 seconds (JVM running fo
com.netflix.discovery.DiscoveryClient : DiscoveryClient_PRODUCTS-SERVICE/products-service:ea4b9f88f56b363c1
com.netflix.discovery.DiscoveryClient : Disable delta property : false
com.netflix.discovery.DiscoveryClient : Single vip registry refresh property : null
com.netflix.discovery.DiscoveryClient : Force full registry fetch : false
com.netflix.discovery.DiscoveryClient : Application is null : false
com.netflix.discovery.DiscoveryClient : Registered Applications size is zero : true
com.netflix.discovery.DiscoveryClient : Application version is -1: false
com.netflix.discovery.DiscoveryClient : Getting all instance registry info from the eureka server
com.netflix.discovery.DiscoveryClient : The response status is 200
o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherServlet'
o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
o.s.web.servlet.DispatcherServlet : Completed initialization in 3 ms
c.m.c.i.CreateProductCommandInterceptor : Intercepted command: class com.mphasis.command.CreateProductCommand
c.n.d.s.r.aws.ConfigClusterResolver : Resolving eureka endpoints via configuration
```



Day - 4

# Set Based Consistency

# Introduction to Set Based Consistency





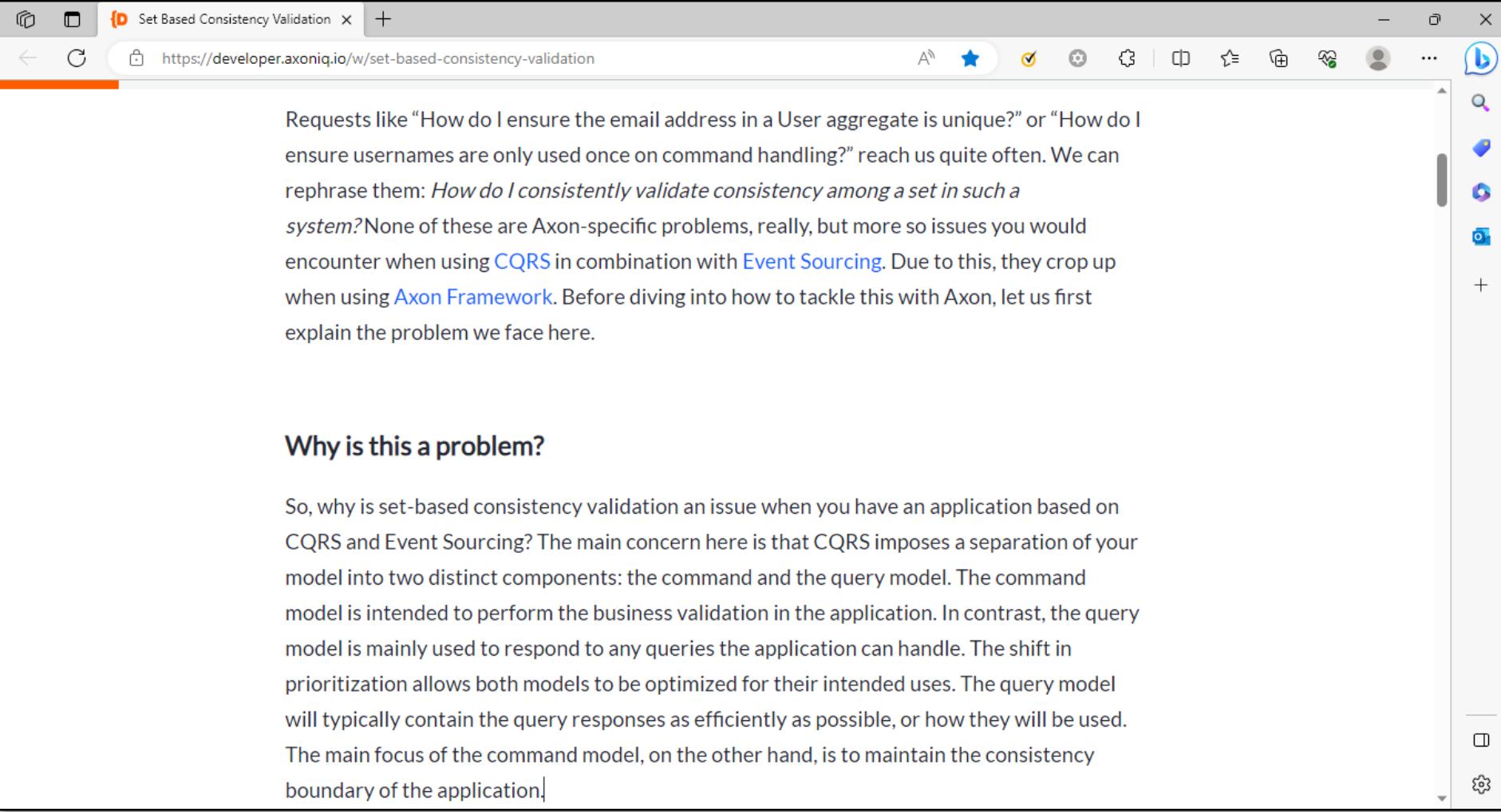
## Introduction to Set Based Consistency

- How to check if record already exists in a database table?
  - How to check if Product already exists?
  - How to check if User already exists?
- 
- 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.



# Introduction to Set Based Consistency

- Go to <https://developer.axoniq.io/w/set-based-consistency-validation>



The screenshot shows a Microsoft Edge browser window with the title "Set Based Consistency Validation". The URL in the address bar is "https://developer.axoniq.io/w/set-based-consistency-validation". The main content area contains the following text:

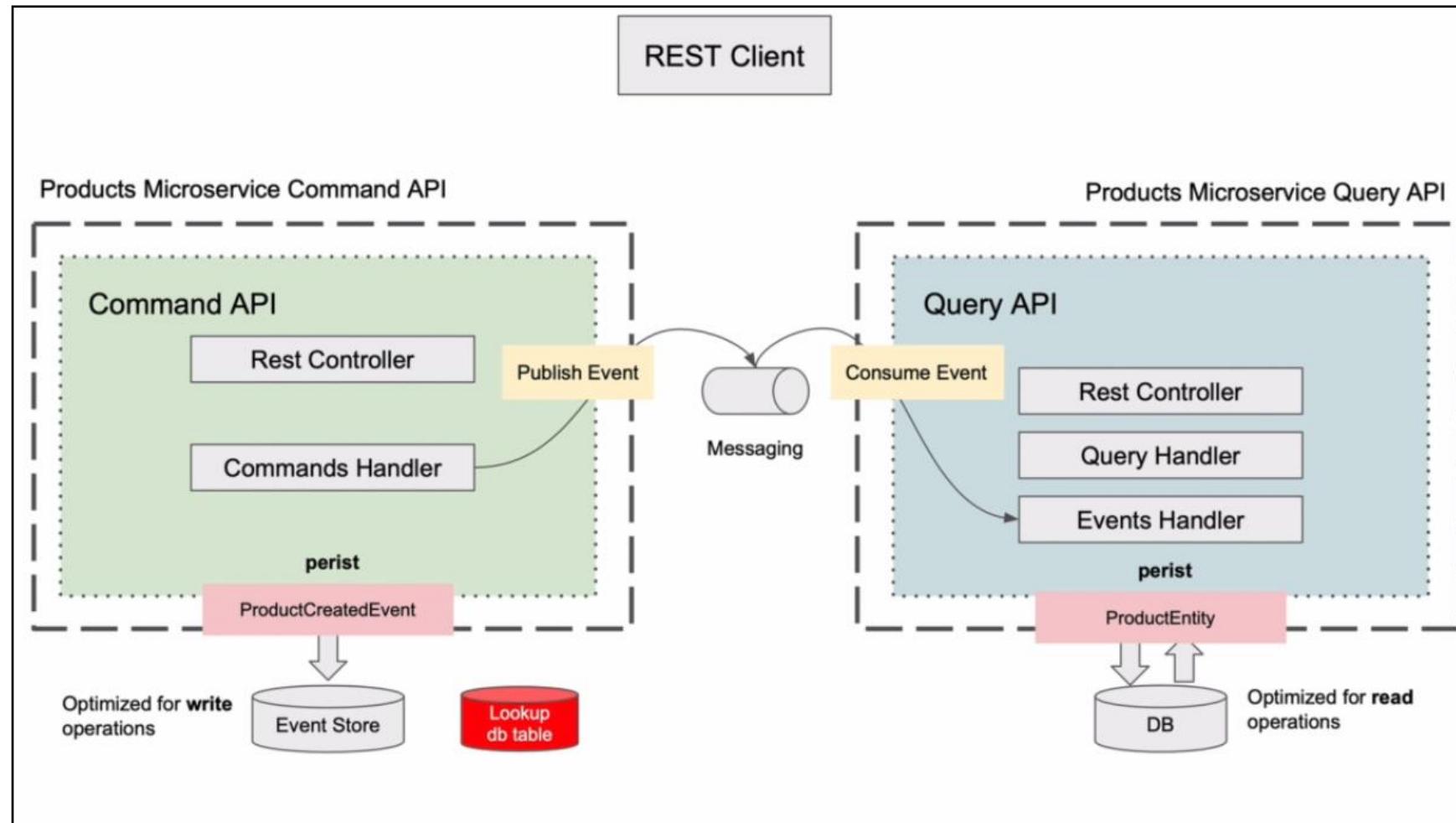
Requests like "How do I ensure the email address in a User aggregate is unique?" or "How do I ensure usernames are only used once on command handling?" reach us quite often. We can rephrase them: *How do I consistently validate consistency among a set in such a system?* None of these are Axon-specific problems, really, but more so issues you would encounter when using CQRS in combination with Event Sourcing. Due to this, they crop up when using Axon Framework. Before diving into how to tackle this with Axon, let us first explain the problem we face here.

### Why is this a problem?

So, why is set-based consistency validation an issue when you have an application based on CQRS and Event Sourcing? The main concern here is that CQRS imposes a separation of your model into two distinct components: the command and the query model. The command model is intended to perform the business validation in the application. In contrast, the query model is mainly used to respond to any queries the application can handle. The shift in prioritization allows both models to be optimized for their intended uses. The query model will typically contain the query responses as efficiently as possible, or how they will be used. The main focus of the command model, on the other hand, is to maintain the consistency boundary of the application.

# Set Based Consistency

- Because the command model can contain any form of data model. We will introduce a small lookup table which will contain product IDs and product titles.



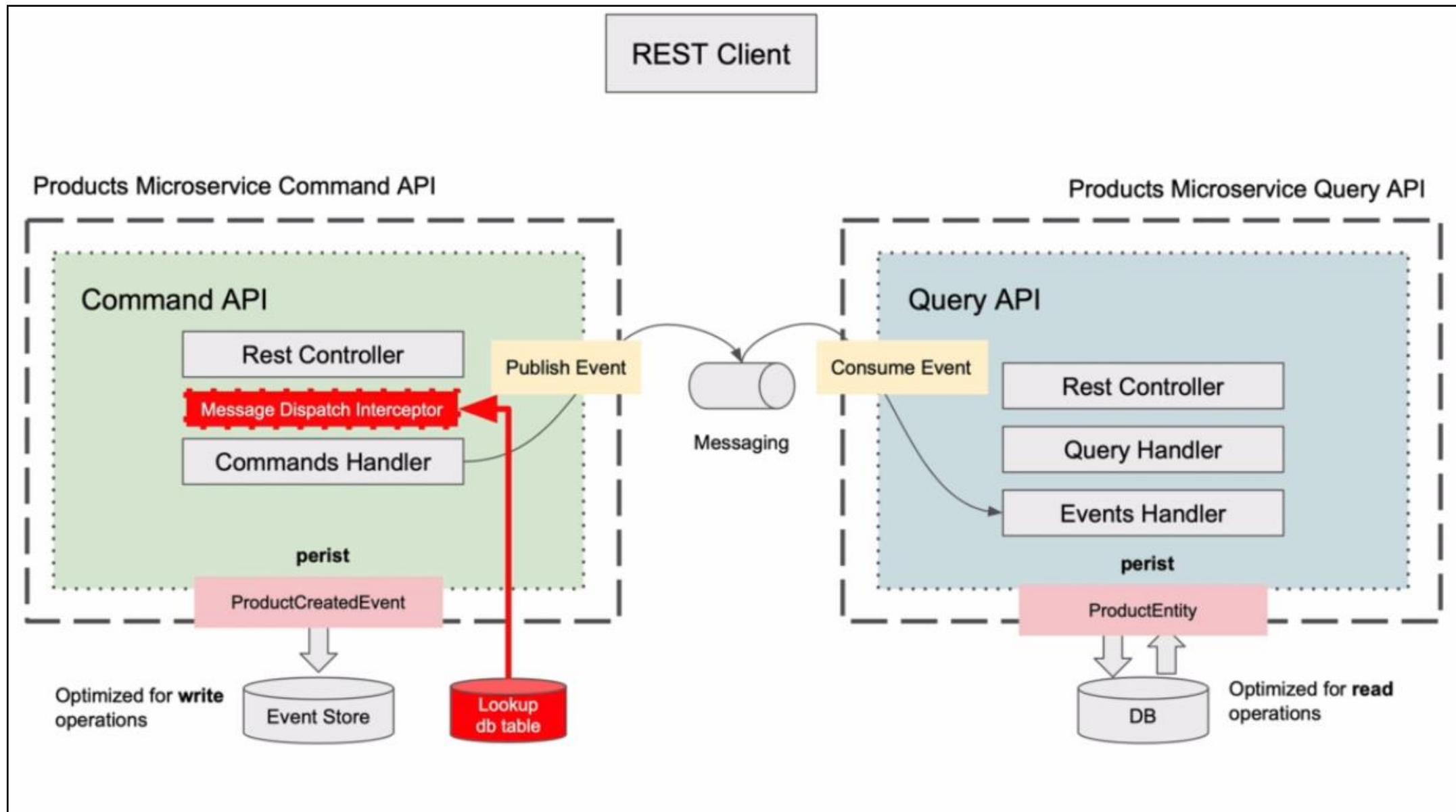


## Lookup Table

- It should not contain the exact same product details as the read database has.
- This lookup table should only store product fields that are needed to look up the record and see if it exists. Like for example, productID or the product title.
- No need to store their product price, product quantity or other product details.
- If the client application issues a command to update product title, for example, and the product title is the field by which you look up a record in this look up database table then you will need to update this field in both look up table and your query database by which you query the record in the lookup table must be consistent with the main read product database.

- How do we query this lookup table before the command handler processes the command?
- And the answer is, we use Message Dispatch Interceptor which we have already implemented.

# Set Based Consistency

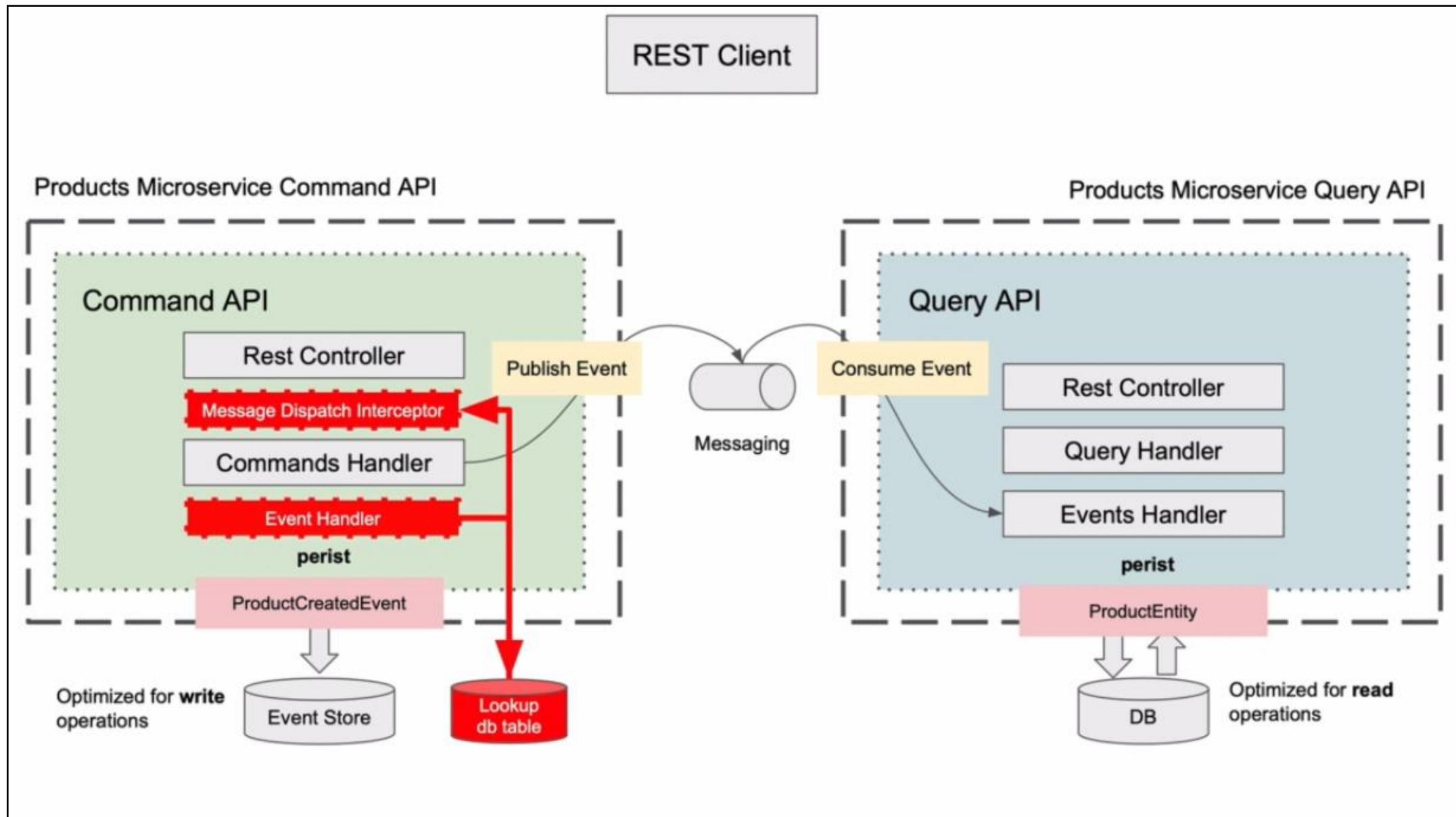




## Set Based Consistency

- How do we will write into this lookup table?
- We will implement an additional **event handler** that will persist product ID and the product title into this database table and when the command handler method publishes the product created event.

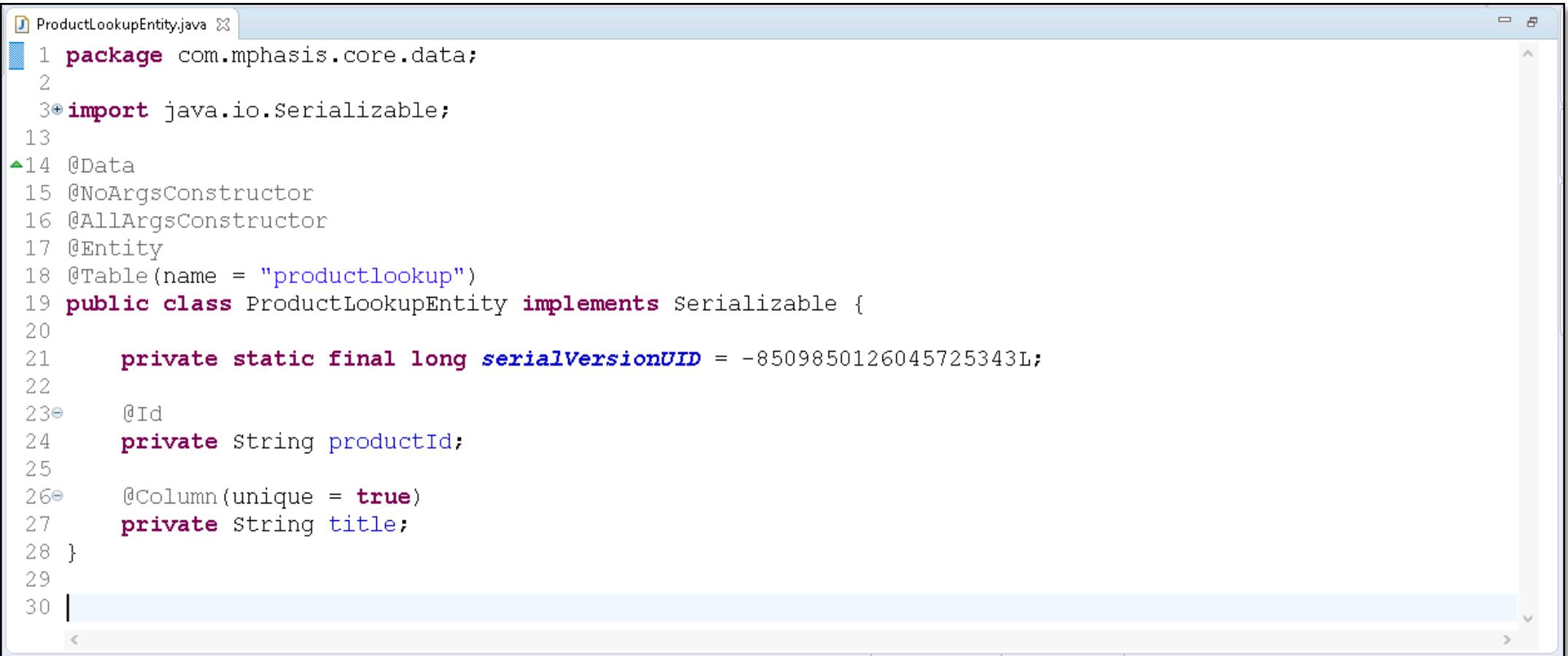
# Set Based Consistency





## Create a ProductLookupEntity class

- Create a ProductLookupEntity class:



```
ProductLookupEntity.java
1 package com.mphasis.core.data;
2
3 import java.io.Serializable;
4
5 @Data
6 @NoArgsConstructor
7 @AllArgsConstructor
8 @Entity
9 @Table(name = "productlookup")
10 public class ProductLookupEntity implements Serializable {
11
12     private static final long serialVersionUID = -8509850126045725343L;
13
14     @Id
15     private String productId;
16
17     @Column(unique = true)
18     private String title;
19
20 }
```



## Create a ProductLookupRepository class

- Create a ProductLookupRepository class:

The screenshot shows a Java code editor window with the title "ProductLookupRepository.java". The code defines a public interface named "ProductLookupRepository" that extends "JpaRepository<ProductLookupEntity, String>". It contains a single method, "findByProductIdOrTitle", which takes two String parameters: "productId" and "title".

```
1 package com.mphasis.core.data;
2
3 import org.springframework.data.jpa.repository.JpaRepository;
4
5 public interface ProductLookupRepository extends JpaRepository<ProductLookupEntity, String>{
6
7     ProductLookupEntity findByProductIdOrTitle(String productId, String title);
8 }
9 |
```



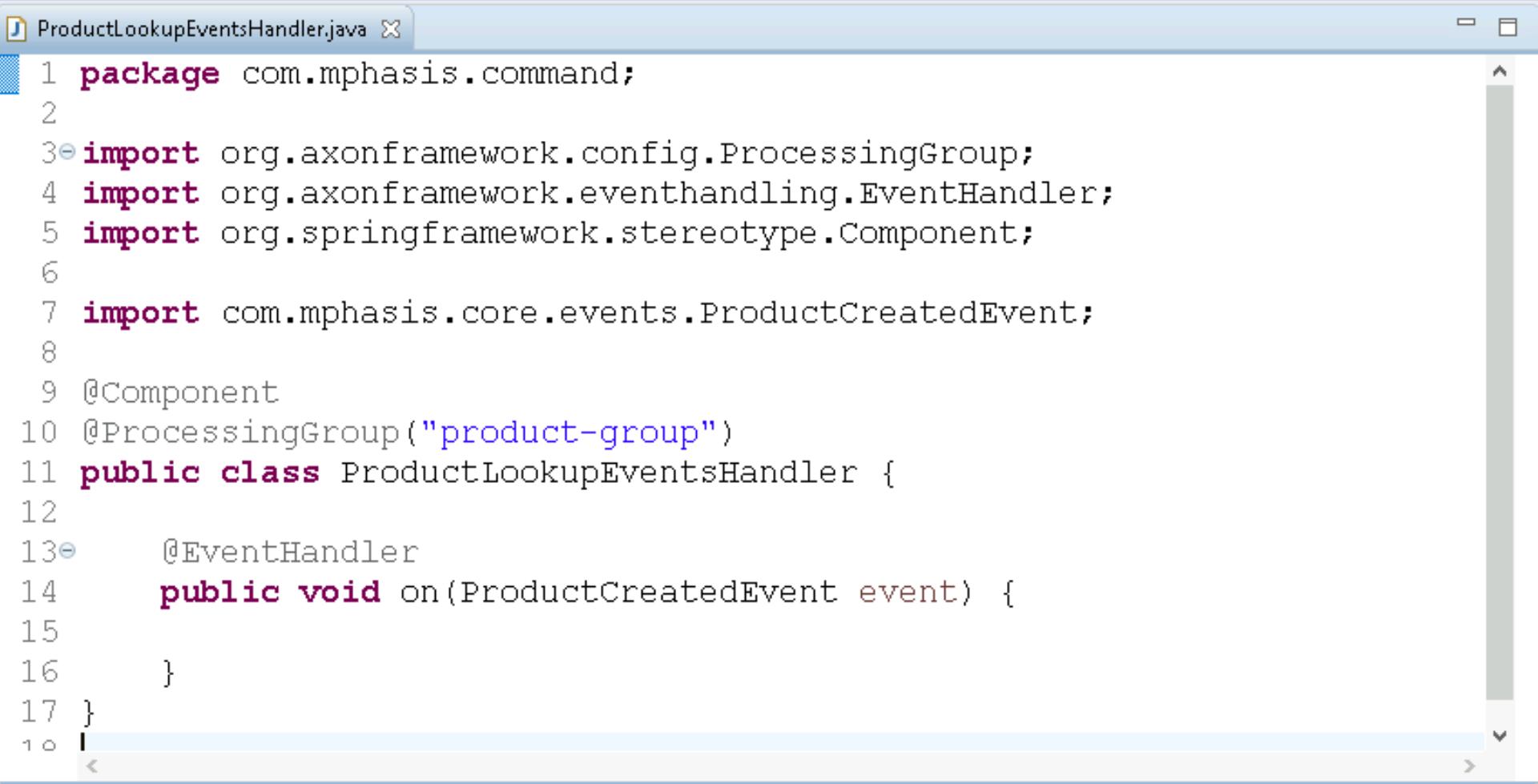
## Create a ProductLookupEventsHandler class

- Should be annotated with **@ProcessingGroup("product-group")**
- This annotation helps in logically group the events handler together.
- Will group both the events handler class – ProductLookupEventsHandler and ProductEventsHandler.
- For this ProcessingGroup, the Axon will also create separate its own TrackingEventProcessor.
- The **TrackingEventProcessor** will use a special tracking token which it will use to avoid multiple processing of the same event in different threads and nodes.
- So, by using this processing group annotation and by grouping these two event handlers in the same logical group, we will also make sure that both events are handled only once and that they're handled in the same thread.
- And this also gives us possibility to roll back the whole transaction if event processing is not successful.



## Create a ProductLookupEventsHandler class

- Create a ProductLookupEventsHandler class:

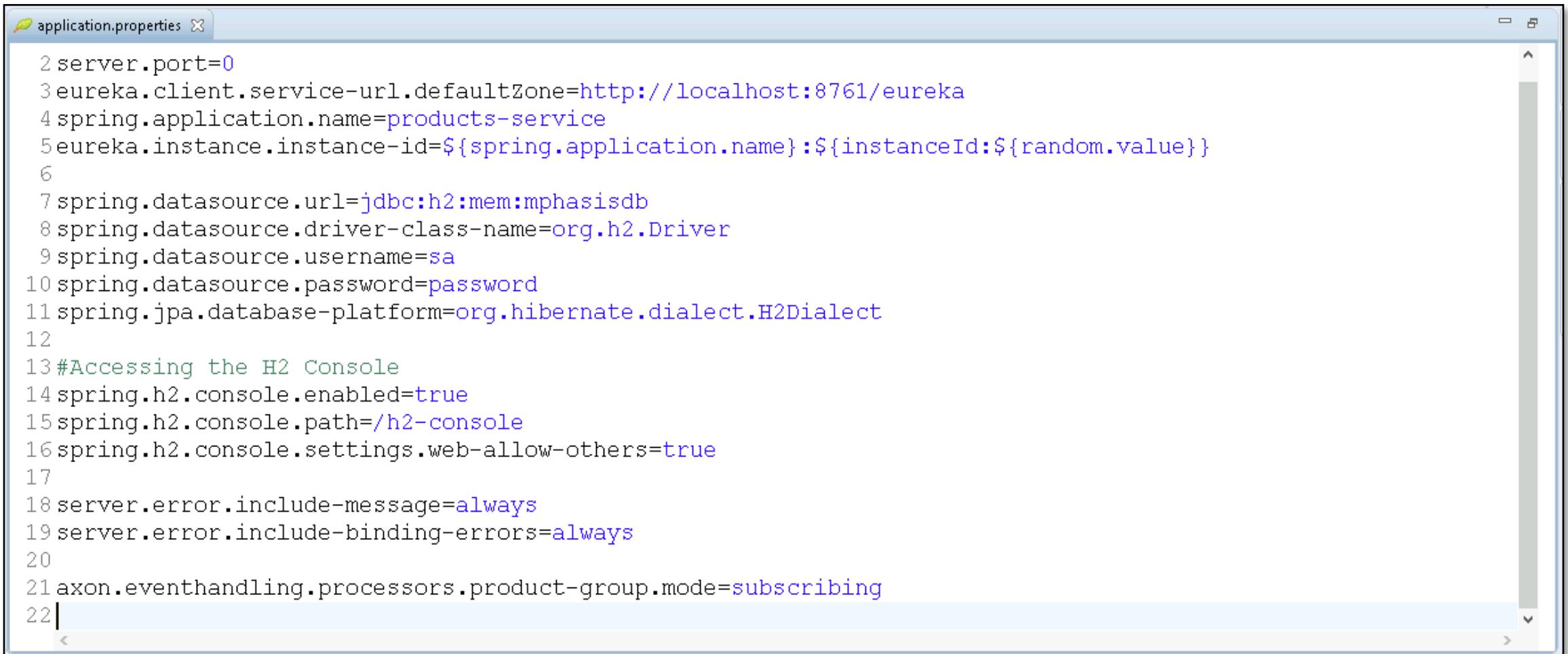


The screenshot shows a Java code editor window with the file `ProductLookupEventsHandler.java` open. The code defines a class named `ProductLookupEventsHandler` that implements the `EventHandler` interface for the `ProductCreatedEvent`. The class is annotated with `@Component` and `@ProcessingGroup("product-group")`.

```
1 package com.mphasis.command;
2
3 import org.axonframework.config.ProcessingGroup;
4 import org.axonframework.eventhandling.EventHandler;
5 import org.springframework.stereotype.Component;
6
7 import com.mphasis.core.events.ProductCreatedEvent;
8
9 @Component
10 @ProcessingGroup("product-group")
11 public class ProductLookupEventsHandler {
12
13     @EventHandler
14     public void on(ProductCreatedEvent event) {
15
16     }
17 }
```



## Add the property to application.properties file



```
application.properties
2 server.port=0
3 eureka.client.service-url.defaultZone=http://localhost:8761/eureka
4 spring.application.name=products-service
5 eureka.instance.instance-id=${spring.application.name}:${instanceId:${random.value}}
6
7 spring.datasource.url=jdbc:h2:mem:mphasisdb
8 spring.datasource.driver-class-name=org.h2.Driver
9 spring.datasource.username=sa
10 spring.datasource.password=password
11 spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
12
13 #Accessing the H2 Console
14 spring.h2.console.enabled=true
15 spring.h2.console.path=/h2-console
16 spring.h2.console.settings.web-allow-others=true
17
18 server.error.include-message=always
19 server.error.include-binding-errors=always
20
21 axon.eventhandling.processors.product-group.mode=subscribing
22 |
```



## Persisting information into a ProductLookup table

- Persisting information into a ProductLookup table:

The screenshot shows a Java code editor window with the file `ProductLookupEventsHandler.java` open. The code implements a component for handling product creation events by persisting product information into a `ProductLookupRepository`.

```
10
11 @Component
12 @ProcessingGroup("product-group")
13 public class ProductLookupEventsHandler {
14
15     private final ProductLookupRepository productLookupRepository;
16
17     public ProductLookupEventsHandler(ProductLookupRepository productLookupRepository) {
18         this.productLookupRepository = productLookupRepository;
19     }
20
21     @EventHandler
22     public void on(ProductCreatedEvent event) {
23
24         ProductLookupEntity productLookupEntity = new ProductLookupEntity(
25             event.getProductId(), event.getTitle());
26
27         productLookupRepository.save(productLookupEntity);
28     }
29 }
30
```



## Update the MessageDispatchInterceptor class

- Let's remove the if conditions of Price & Title from the CreateProductCommandInterceptor class:

```
▲28  public BiFunction<Integer, CommandMessage<?>, CommandMessage<?>> handle(
29      List<? extends CommandMessage<?>> messages) {
30
31      return (index, command) -> {
32
33          LOGGER.info("Intercepted command: " + command.getPayloadType());
34
35          if(CreateProductCommand.class.equals(command.getPayloadType())) {
36
37              CreateProductCommand createProductCommand = (CreateProductCommand) command.getPayload();
38
39              ProductLookupEntity productLookupEntity = productLookupRepository.findByIdOrTitle(
40                  createProductCommand.getProductId(), createProductCommand.getTitle());
41
42              if(productLookupEntity != null) {
43                  throw new IllegalStateException(
44                      String.format("Product with productId %s or title %s already exist",
45                      createProductCommand.getProductId(),
46                      createProductCommand.getTitle())
47                  );
48              }
49          }
50      }
51  };
```



## Trying how it works

1. Run the AxonServer using Docker command.
2. Ensure the Discovery Server (Eureka Server) and Product Service is running.
3. Ensure the ApiGateway is running.

# Send a POST request to Create Product

The screenshot shows the Postman application interface. At the top, there is a navigation bar with 'Home', 'Workspaces', 'Explore', a search bar 'Search Postman', and account options 'Sign In' and 'Create Account'. Below the navigation bar, a header bar displays 'POST http://localhost:8082/p...' and 'GET http://localhost:8082/prc...'. The main workspace shows a POST request to 'http://localhost:8082/products-service/products'. The 'Body' tab is selected, showing a JSON payload:

```
1 {
2   "title": "iPhone 8",
3   "price": 500,
4   "quantity": 5
5 }
```

The 'Headers' tab shows 8 items. The 'Tests' and 'Settings' tabs are also visible. On the right side, there are buttons for 'Cookies', 'Beautify', and 'Save Response'. Below the request details, the response section shows a status of 200 OK with a response ID: 6f0fe190-ef02-48ef-93a7-b0bd17524c46. The bottom of the screen has tabs for 'Console' and other interface controls.

# Send a GET request to Query the Products

The screenshot shows the Postman application interface. At the top, there is a navigation bar with 'Home', 'Workspaces', 'Explore', a search bar 'Search Postman', and account options 'Sign In' and 'Create Account'. Below the navigation bar, a list of requests is shown: 'POST http://localhost:8082/p...' and 'GET http://localhost:8082/pr...'. A new request button '+' and an ellipsis '...' are also present.

The main area displays a request configuration for a 'GET' method to 'http://localhost:8082/products-service/products'. The 'Params' tab is selected, showing two query parameters: 'Key' (Value: 'Value') and another 'Key' (Value: 'Value').

Below the request configuration, the 'Body' tab is selected, showing the response body in 'Pretty' format:

```
1 [  
2   {  
3     "productId": "6f0fe190-ef02-48ef-93a7-b0bd17524c46",  
4     "title": "iPhone 8",  
5     "price": 500.00,  
6     "quantity": 5  
7   }  
8 ]
```

The status bar at the bottom indicates 'Status: 200 OK' with a globe icon, 'Time: 57 ms', 'Size: 217 B', and a 'Save Response' button.

# Product Lookup Table

The screenshot shows the H2 Console interface. The title bar indicates the connection is "Not secure" to "host.docker.internal:49967/h2-console/login.do?jsessionid=6fe629166adaee826763662aa3da238e". The left sidebar lists database objects: ASSOCIATION\_VALUE\_ENTRY, PRODUCTLOOKUP, PRODUCTS, SAGA\_ENTRY, TOKEN\_ENTRY, INFORMATION\_SCHEMA, Sequences, and Users. The main area contains a SQL statement: "SELECT \* FROM PRODUCTLOOKUP". Below it, the results of the execution are shown:

```
SELECT * FROM PRODUCTLOOKUP;
```

PRODUCT_ID	TITLE
6f0fe190-ef02-48ef-93a7-b0bd17524c46	iPhone 8

(1 row, 6 ms)

**Edit**

# Send a POST request to Create Product with same JSON

The screenshot shows the Postman application interface. At the top, there is a navigation bar with 'Home', 'Workspaces', 'Explore', a search bar 'Search Postman', and account options 'Sign In' and 'Create Account'. Below the navigation bar, a header bar displays 'POST http://localhost:8082/p' and 'GET http://localhost:8082/prc'. The main workspace shows a POST request to 'http://localhost:8082/products-service/products'. The 'Body' tab is selected, showing a JSON payload:

```
1 {  
2   ... "title": "iPhone 8",  
3   ... "price": 500,  
4   ... "quantity": 5  
5 }  
6
```

The 'Body' tab also includes tabs for 'Pretty', 'Raw', 'Preview', 'Visualize', and 'Text'. The response section shows a status of '200 OK', time '18 ms', size '207 B', and a message: 'Product with productId 76e3599e-d922-4e01-8190-4272b93ba4cb or title iPhone 8 already exist'.



## Recap of Day – 4

- Validating Request Body, Bean Validation
- Bean Validation – with Hibernate Validation
- Bean Validation. Enable Bean Validation
- Bean Validation. Validating Request Body
- Validation in the @CommandHandler method
- Command Validation in the Aggregate
- Introduction to Message Dispatch Interceptor
- Creating a new Command Interceptor class
- Register Message Dispatch Interceptor



## Recap of Day – 4

- Introduction to Set Based Consistency
- Create a Product Lookup Entity
- Create a Product Lookup Repository
- Create a Product Lookup Event Handler
- Persisting information into a ProductLookup table
- Update the MessageDispatchInterceptor class



# Day - 5



## Day – 5 Agenda

- Handle Error & Rollback Transaction
- Introduction to Error Handling
- Creating a centralized Error Handler class
- Return custom error object
- @ExceptionHandler Annotation
- Creating the ListenerInvocationErrorHandler
- Register the ListenerInvocationErrorHandler
- Assignment – Orders Microservice



## Day – 5 Agenda

- The Traditional Two-phase Commit (2PC) Protocol
- The SAGA Architectural Pattern
- When to use SAGA and 2PC commit in Design?
- Order Saga: Orchestration-based Saga
- Saga Class Structure Overview

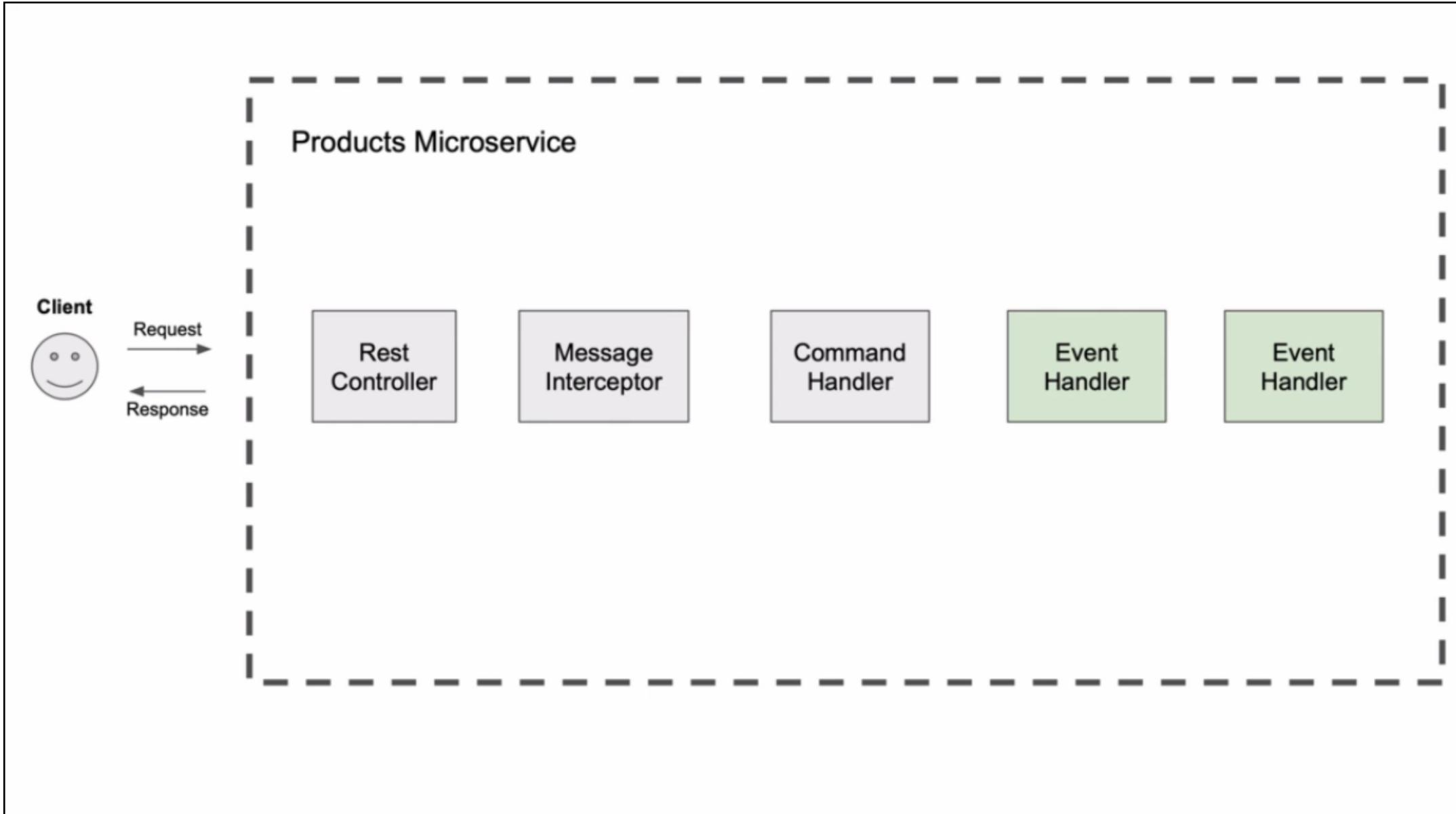


Day - 5

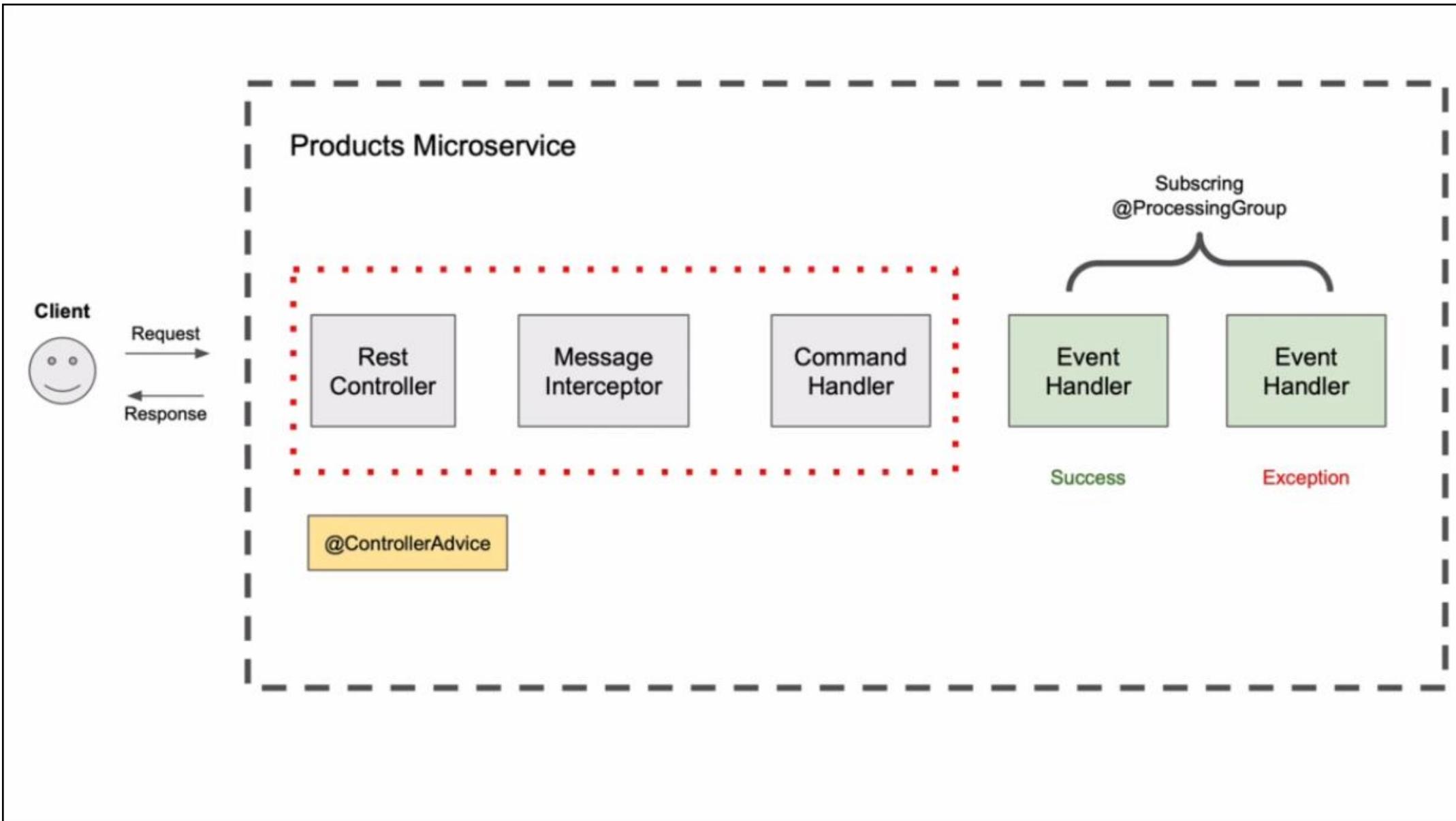
## Handle Error & Rollback Transaction



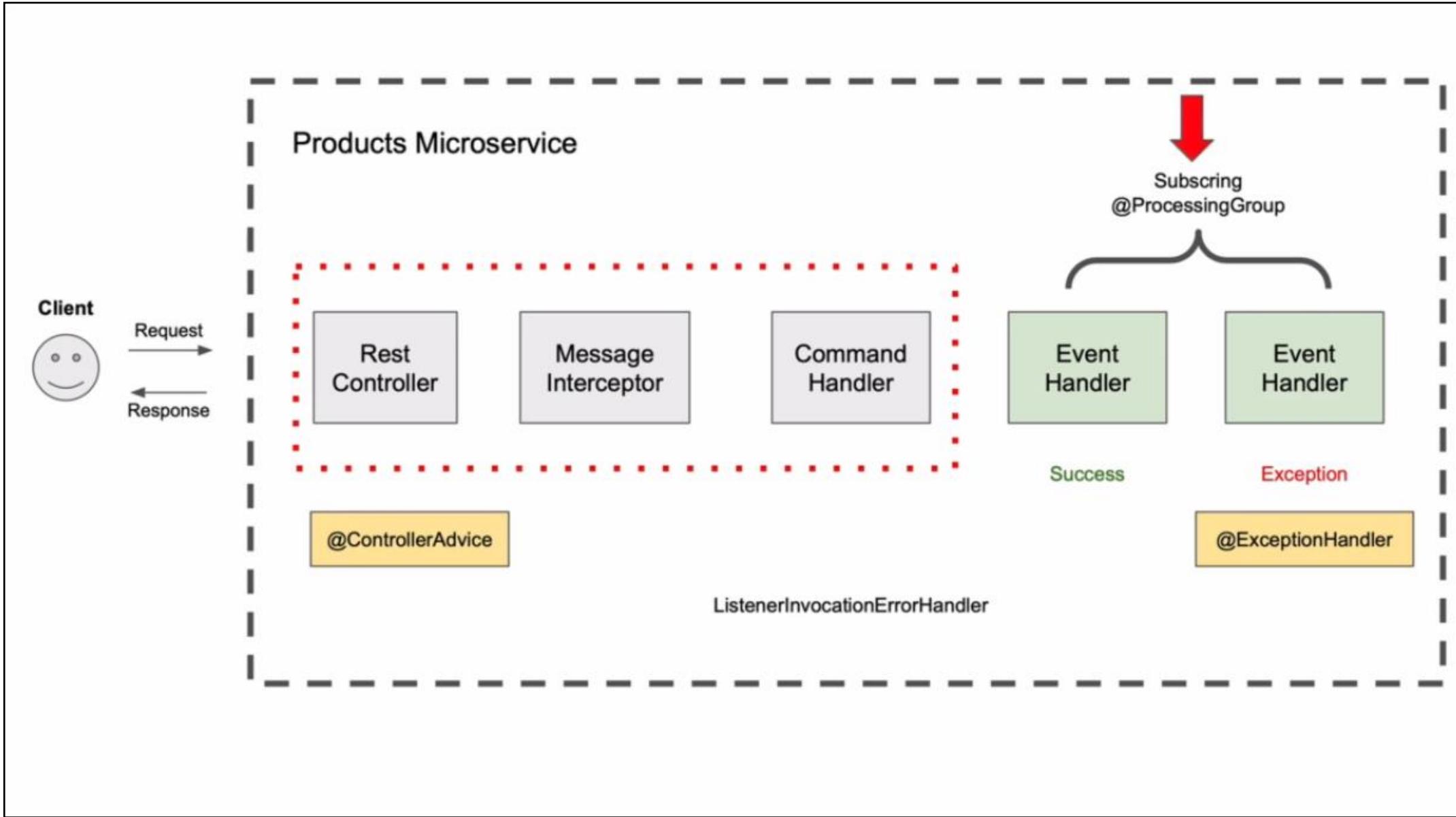
# Introduction to Error Handling



# Introduction to Error Handling



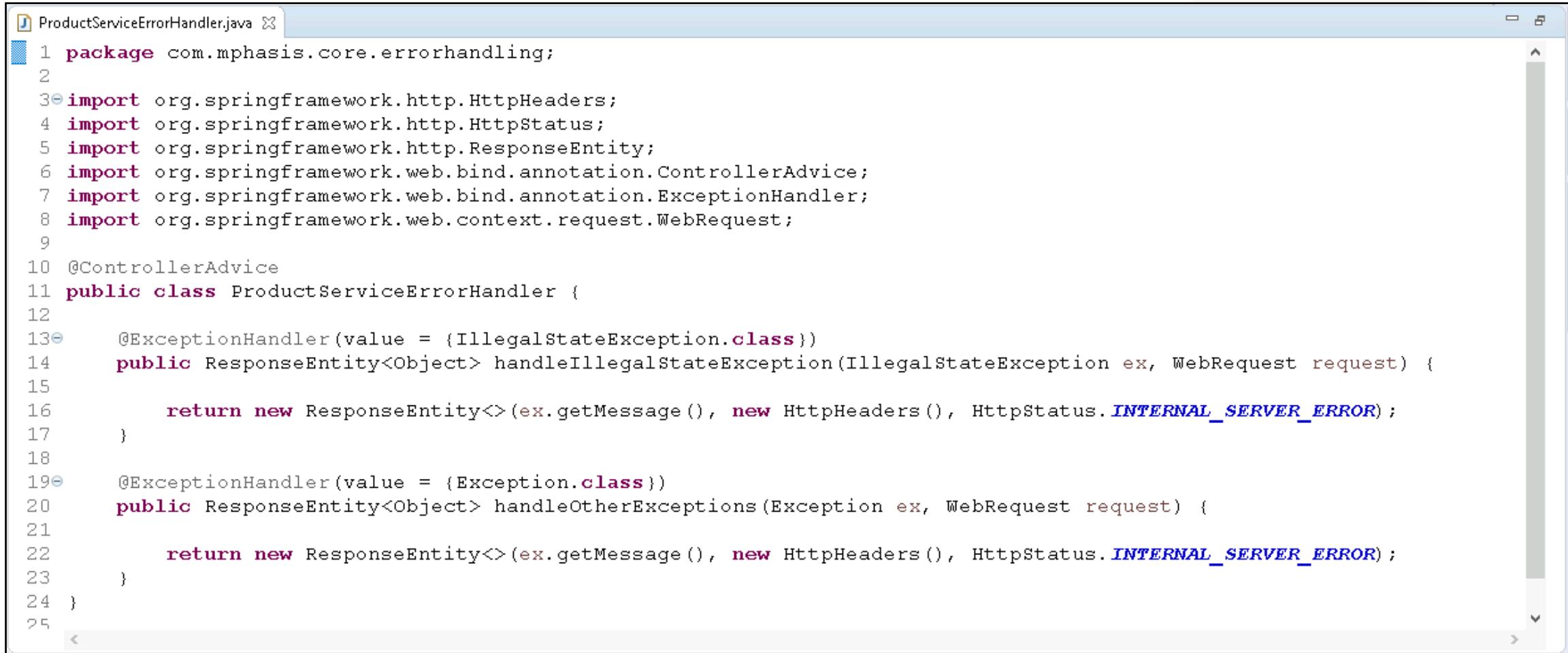
# Introduction to Error Handling





# Create a centralized Error Handler class

- Create a centralized ProductServiceErrorHandler class:



```
ProductServiceErrorHandler.java ✘
1 package com.mphasis.core.errorhandling;
2
3 import org.springframework.http.HttpHeaders;
4 import org.springframework.http.HttpStatus;
5 import org.springframework.http.ResponseEntity;
6 import org.springframework.web.bind.annotation.ControllerAdvice;
7 import org.springframework.web.bind.annotation.ExceptionHandler;
8 import org.springframework.web.context.request.WebRequest;
9
10 @ControllerAdvice
11 public class ProductServiceErrorHandler {
12
13     @ExceptionHandler(value = {IllegalStateException.class})
14     public ResponseEntity<Object> handleIllegalStateException(IllegalStateException ex, WebRequest request) {
15
16         return new ResponseEntity<>(ex.getMessage(), new HttpHeaders(), HttpStatus.INTERNAL_SERVER_ERROR);
17     }
18
19     @ExceptionHandler(value = {Exception.class})
20     public ResponseEntity<Object> handleOtherExceptions(Exception ex, WebRequest request) {
21
22         return new ResponseEntity<>(ex.getMessage(), new HttpHeaders(), HttpStatus.INTERNAL_SERVER_ERROR);
23     }
24 }
25 <
```



## Trying how it works

1. Run the AxonServer using Docker command.
2. Ensure the Discovery Server (Eureka Server) and Product Service is running.
3. Ensure the ApiGateway is running.

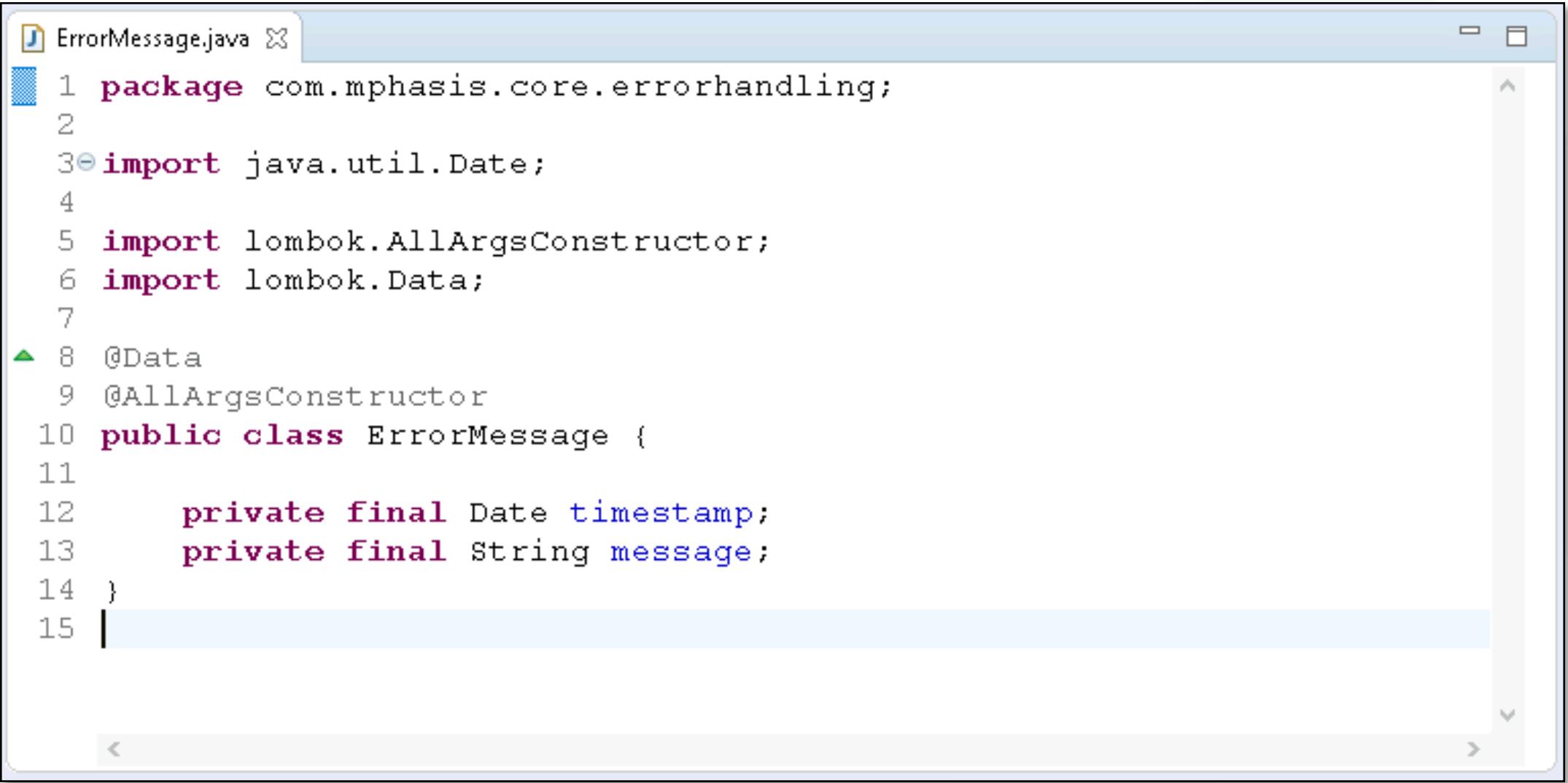
# Send a POST request to Create Product - Twice

The screenshot shows the Postman application interface. At the top, there is a navigation bar with 'Home', 'Workspaces', 'Explore', a search bar 'Search Postman', and account options 'Sign In' and 'Create Account'. Below the navigation bar, a list of requests is shown: 'POST http://localhost:8082/products' (selected), 'GET http://localhost:8082/products' (disabled), '+', and '...'. The main area displays a POST request to 'http://localhost:8082/products-service/products'. The method is set to 'POST', the URL is 'http://localhost:8082/products-service/products', and the 'Body' tab is selected. Under 'Headers', there are 8 items listed. The 'Body' section shows a table for 'Query Params' with two rows: one for 'Key' and one for 'Value'. The 'Body' tab also contains a 'Pretty' button, 'Raw' button, 'Preview' button, 'Visualize' button, and a 'Text' dropdown set to 'Text'. The response section shows a status of '500 Internal Server Error', a time of '35 ms', and a size of '226 B'. The response body is a single line: '1 Product with productId 58f66dec-b4ac-4179-ae4a-017e336138c9 or title iPhone 8 already exist'. At the bottom, there is a 'Console' tab.



## Return custom error object

- Create a Custom ErrorMessage class:



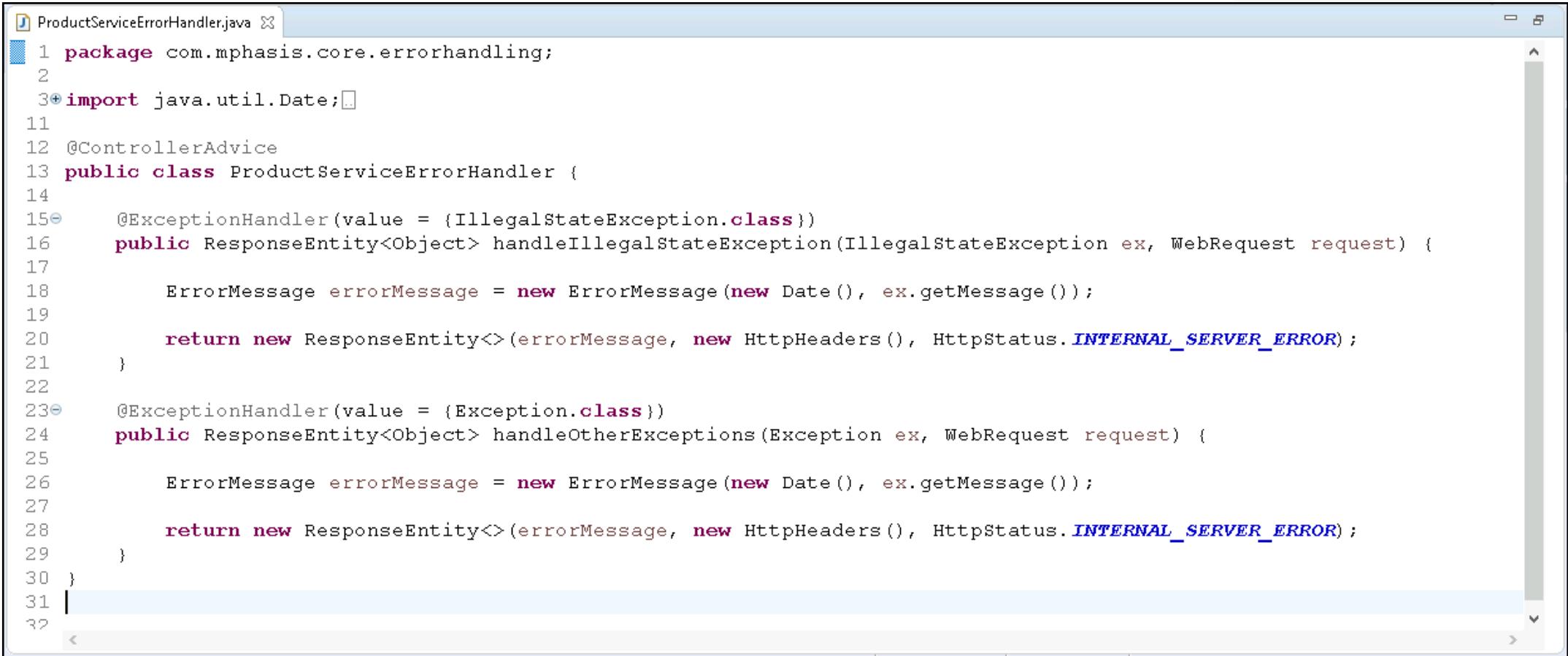
The screenshot shows a Java code editor window with the file "ErrorMessage.java" open. The code defines a class named "ErrorMessage" with two private final fields: "timestamp" and "message". The code uses Lombok annotations @Data and @AllArgsConstructor.

```
1 package com.mphasis.core.errorhandling;
2
3 import java.util.Date;
4
5 import lombok.AllArgsConstructor;
6 import lombok.Data;
7
8 @Data
9@AllArgsConstructor
10 public class ErrorMessage {
11
12     private final Date timestamp;
13     private final String message;
14 }
15
```



## Update the centralized Error Handler class

- Add the Custom ErrorMessage in the ProductServiceErrorHandler class:



```
ProductServiceErrorHandler.java
1 package com.mphasis.core.errorhandling;
2
3 import java.util.Date;
4
5 @ControllerAdvice
6 public class ProductServiceErrorHandler {
7
8     @ExceptionHandler(value = {IllegalStateException.class})
9     public ResponseEntity<Object> handleIllegalStateException(IllegalStateException ex, WebRequest request) {
10
11         ErrorMessage errorMessage = new ErrorMessage(new Date(), ex.getMessage());
12
13         return new ResponseEntity<>(errorMessage, new HttpHeaders(), HttpStatus.INTERNAL_SERVER_ERROR);
14     }
15
16     @ExceptionHandler(value = {Exception.class})
17     public ResponseEntity<Object> handleOtherExceptions(Exception ex, WebRequest request) {
18
19         ErrorMessage errorMessage = new ErrorMessage(new Date(), ex.getMessage());
20
21         return new ResponseEntity<>(errorMessage, new HttpHeaders(), HttpStatus.INTERNAL_SERVER_ERROR);
22     }
23 }
```



## Trying how it works

1. Restart the Product Service.
2. Restart the ApiGateway.

# Send a POST request to Create Product - Twice

The screenshot shows the Postman application interface. At the top, there's a navigation bar with 'Home', 'Workspaces', 'Explore', a search bar 'Search Postman', and account options 'Sign In' and 'Create Account'. Below the navigation is a header bar with 'POST http://localhost:8082/p...' and 'GET http://localhost:8082/pro...'. A collection icon and a 'Create Collection' button are also present.

The main area shows a POST request to 'http://localhost:8082/products-service/products'. The 'Params' tab is selected, showing two query parameters: 'Key' and 'Value'. The 'Body' tab is selected, showing a JSON response:

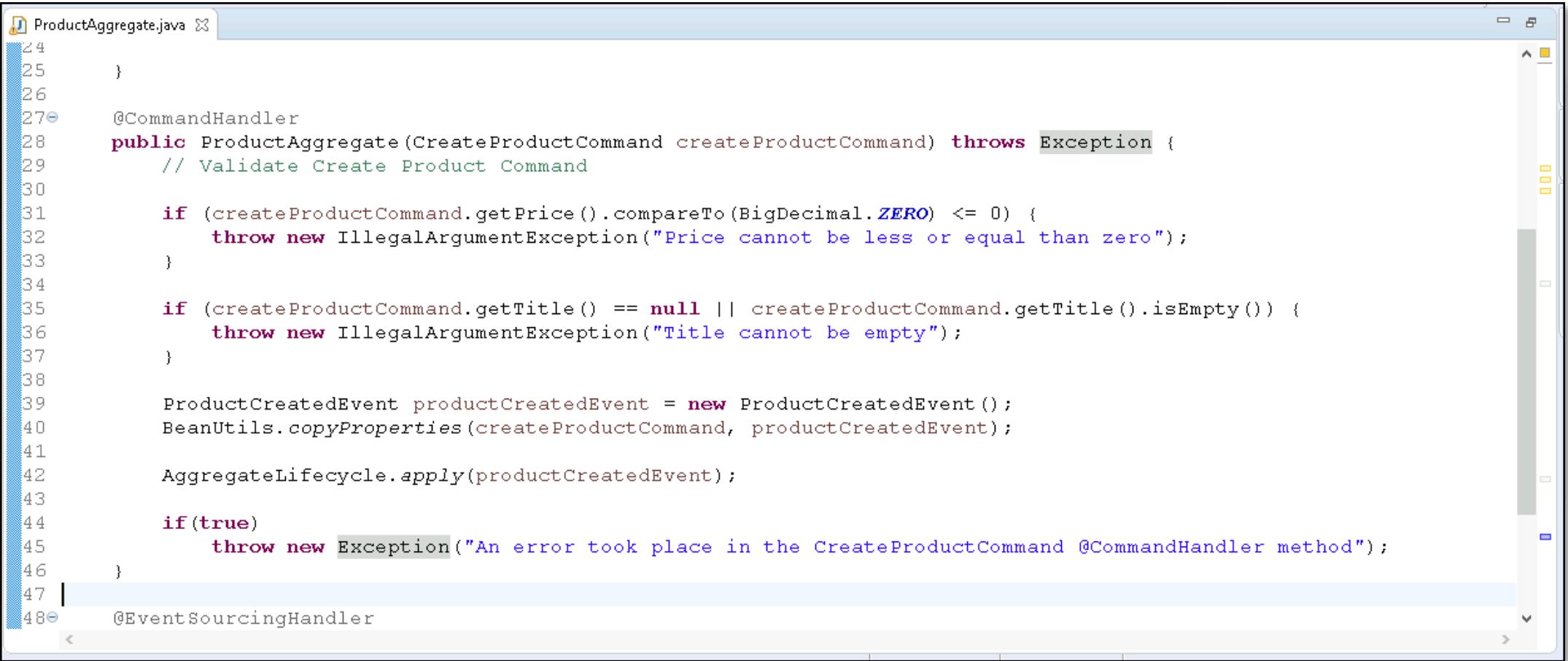
```
1 {  
2   "timestamp": "2023-07-07T10:59:13.088+00:00",  
3   "message": "Product with productId 9418a946-abe3-49e8-bafe-f5eb28834416 or title iPhone 8 already exist"  
4 }
```

The status bar at the bottom indicates 'Status: 500 Internal Server Error' with a time of '33 ms' and a size of '284 B'. There are tabs for 'Cookies', 'Headers (3)', and 'Test Results'. The bottom navigation bar includes 'Pretty', 'Raw', 'Preview', 'Visualize', 'JSON', 'Console', and other icons.



## Handle the @Command Execution Exception

- Here we will handle an Exception that is thrown by Command Handler method in the Aggregate class.

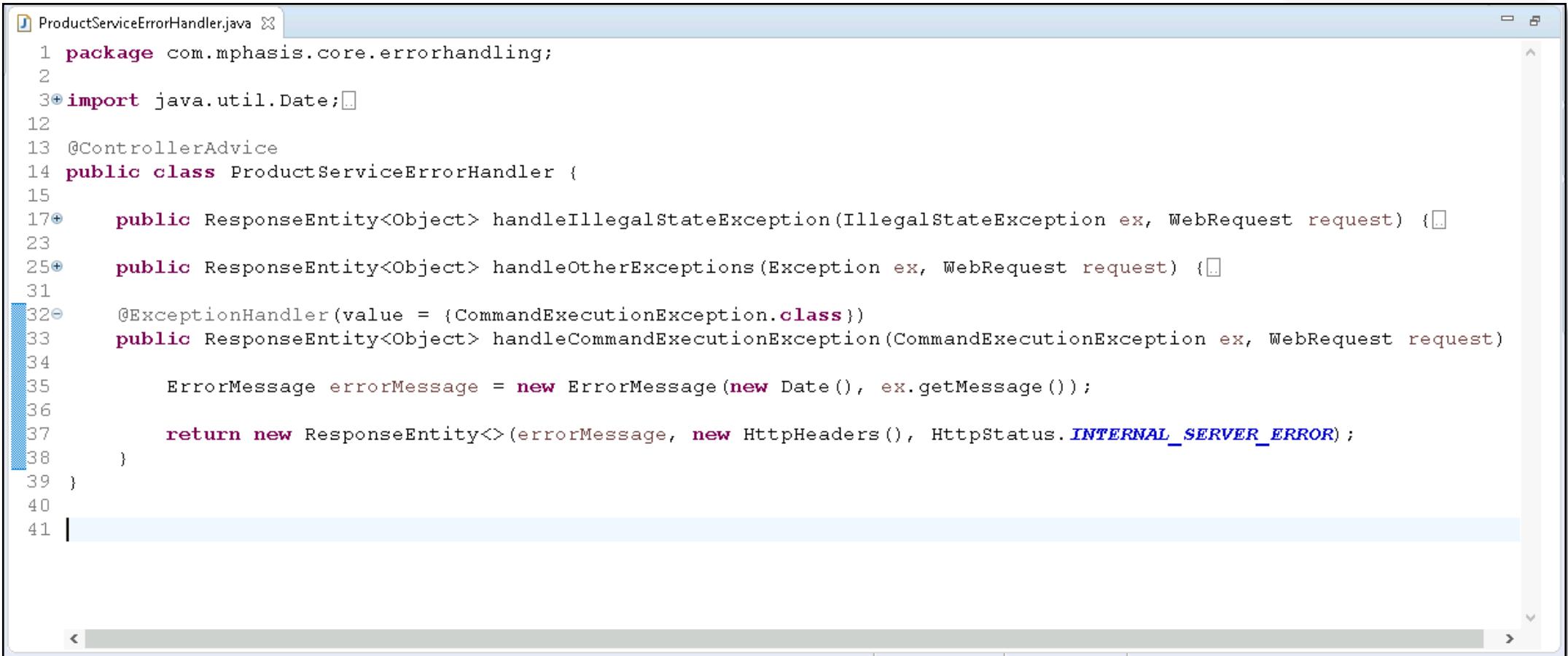


```
24
25
26
27@CommandHandler
28 public ProductAggregate(CreateProductCommand createProductCommand) throws Exception {
29     // Validate Create Product Command
30
31     if (createProductCommand.getPrice().compareTo(BigDecimal.ZERO) <= 0) {
32         throw new IllegalArgumentException("Price cannot be less or equal than zero");
33     }
34
35     if (createProductCommand.getTitle() == null || createProductCommand.getTitle().isEmpty()) {
36         throw new IllegalArgumentException("Title cannot be empty");
37     }
38
39     ProductCreatedEvent productCreatedEvent = new ProductCreatedEvent();
40     BeanUtils.copyProperties(createProductCommand, productCreatedEvent);
41
42     AggregateLifecycle.apply(productCreatedEvent);
43
44     if(true)
45         throw new Exception("An error took place in the CreateProductCommand @CommandHandler method");
46
47
48@EventSourcingHandler
```



## Handle the @Command Execution Exception

- When an Error is thrown from the command handler/query handler method then the Axon Framework wrap this Error into **CommandExecutionException/QueryExecutionException**.



```
ProductServiceErrorHandler.java ✘
1 package com.mphasis.core.errorhandling;
2
3 import java.util.Date;
4
5 @ControllerAdvice
6 public class ProductServiceErrorHandler {
7
8     public ResponseEntity<Object> handleIllegalStateException(IllegalStateException ex, WebRequest request) {
9
10    public ResponseEntity<Object> handleOtherExceptions(Exception ex, WebRequest request) {
11
12        @ExceptionHandler(value = {CommandExecutionException.class})
13        public ResponseEntity<Object> handleCommandExecutionException(CommandExecutionException ex, WebRequest request) {
14
15            ErrorMessage errorMessage = new ErrorMessage(new Date(), ex.getMessage());
16
17            return new ResponseEntity<>(errorMessage, new HttpHeaders(), HttpStatus.INTERNAL_SERVER_ERROR);
18        }
19    }
20
21 }
```



## Trying how it works

1. Restart the Product Service.
2. Restart the ApiGateway.

# Send a POST request to Create Product - Twice

The screenshot shows the Postman application interface. At the top, there are navigation links for Home, Workspaces, and Explore, along with a search bar and user authentication buttons. Below the header, a list of requests is shown: a POST request to `http://localhost:808` and a GET request to `http://localhost:8082/prc`. The main area displays a POST request to `http://localhost:8082/products-service/products`. The request method is set to POST, and the URL is `http://localhost:8082/products-service/products`. The Headers tab is selected, showing a value of 8. The Body tab is active, containing the following JSON payload:

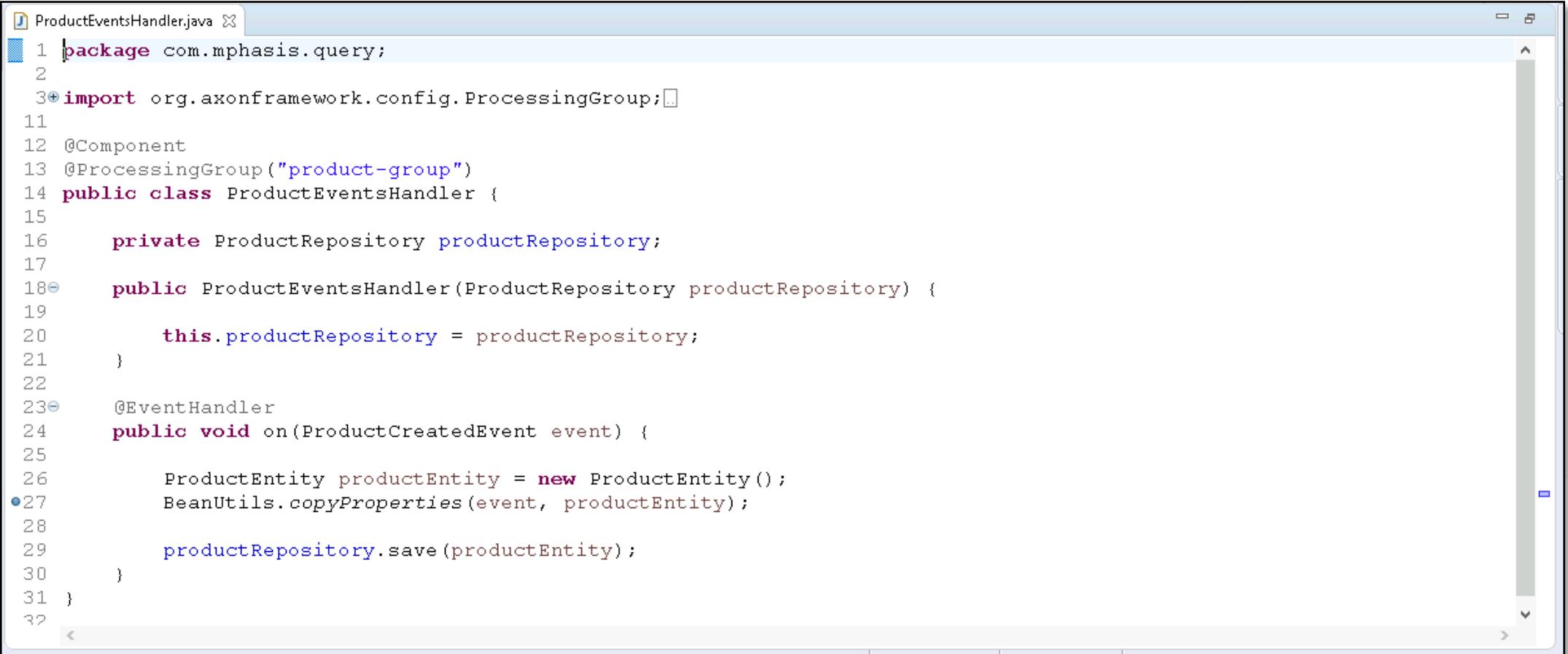
```
1 {  
2   ... "title": "iPhone 8",  
3   ... "price": 500,  
4   ... "quantity": 5
```

The response section shows a status of 500 Internal Server Error, with a timestamp of 2023-07-07T11:26:03.118+00:00 and the message: "An error took place in the CreateProductCommand @CommandHandler method".



## @ExceptionHandler

- In ProductEventsHandler, the exception can occur in the Event Handler method that handle the ProductCreatedEvent.



```
ProductEventsHandler.java
1 package com.mphasis.query;
2
3+import org.axonframework.config.ProcessingGroup;
11
12 @Component
13 @ProcessingGroup("product-group")
14 public class ProductEventsHandler {
15
16     private ProductRepository productRepository;
17
18@    public ProductEventsHandler(ProductRepository productRepository) {
19
20        this.productRepository = productRepository;
21    }
22
23@    @EventHandler
24    public void on(ProductCreatedEvent event) {
25
26        ProductEntity productEntity = new ProductEntity();
27        BeanUtils.copyProperties(event, productEntity);
28
29        productRepository.save(productEntity);
30    }
31 }
32
```



## Handle the Exception using @ExceptionHandler

- Use @ExceptionHandler annotation:

```
ProductEventsHandler.java X
1 package com.mphasis.query;
2
3+import org.axonframework.config.ProcessingGroup;□
12
13 @Component
14 @ProcessingGroup("product-group")
15 public class ProductEventsHandler {
16
17     private ProductRepository productRepository;
18
19+    public ProductEventsHandler(ProductRepository productRepository) {
20
21         this.productRepository = productRepository;
22     }
23
24+    @ExceptionHandler(resultType = Exception.class)
25    public void handle(Exception exception) {
26        // log error message
27    }
28
29+    @ExceptionHandler(resultType = IllegalArgumentException.class)
30    public void handle(IllegalArgumentException exception) {
31        // log error message
32    }

```



## Create the ListenerInvocationErrorHandler class

- Create a new class ProductsServiceEventsErrorHandler:

```
ProductsServiceEventsErrorHandler.java
1 package com.mphasis.core.errorhandling;
2
3+import org.axonframework.eventhandling.EventMessage;...
4
5 public class ProductsServiceEventsErrorHandler implements ListenerInvocationErrorHandler{
6
7     @Override
8
9     public void onException(Exception exception, EventMessage<?> event, EventMessageHandler eventHandler)
10        throws Exception {
11
12         throw exception;
13     }
14 }
15
16 |
```



## Register the ListenerInvocationErrorHandler

- Register the ProductsServiceEventsErrorHandler class:



```
ProductServiceApplication.java X
13
14 @EnableDiscoveryClient
15 @SpringBootApplication
16 public class ProductServiceApplication {
17
18     public static void main(String[] args) {
19         SpringApplication.run(ProductServiceApplication.class, args);
20     }
21
22     @Autowired
23     public void registerCreateProductCommandInterceptor(
24         ApplicationContext context, CommandBus commandBus) {
25
26         commandBus.registerDispatchInterceptor(context.getBean(CreateProductCommandInterceptor.class));
27     }
28
29     @Autowired
30     public void configure(EventProcessingConfigurer configurer) {
31
32         configurer.registerListenerInvocationErrorHandler("product-group",
33             config -> new ProductsServiceEventsErrorHandler());
34     }
35 }
36
37 |
```



## Trying how it works

1. Run the AxonServer using Docker command.
2. Ensure the Discovery Server (Eureka Server) and Product Service is running.
3. Ensure the ApiGateway is running.

# Send a POST request to Create Product

The screenshot shows the Postman application interface. At the top, there is a navigation bar with 'Home', 'Workspaces', 'Explore', a search bar 'Search Postman', and account options 'Sign In' and 'Create Account'. Below the navigation bar, a header bar shows a 'POST' request to 'http://localhost:8082/p...' and a 'GET' request to 'http://localhost:8082/prc...'. The main workspace displays an 'HTTP' request to 'http://localhost:8082/products-service/products'. The 'Body' tab is selected, showing a JSON payload:

```
1 {  
2   "title": "iPhone 9",  
3   "price": 500,  
4   "quantity": 5  
5 }  
6
```

The 'Headers' tab shows 8 items. The 'Tests' and 'Settings' tabs are also visible. On the right side, there are buttons for 'Add to collection' and 'Send'. Below the request details, the 'Body' tab is active, showing the response status: 'Status: 500 Internal Server Error', time: '400 ms', size: '235 B', and a 'Save Response' button. The 'Pretty' tab is selected, displaying the response body:

```
1 {  
2   "timestamp": "2023-07-07T17:40:06.574+00:00",  
3   "message": "Exception occurred while processing events"  
4 }
```

At the bottom, there is a 'Console' tab.

# Product Lookup Table

The screenshot shows the H2 Console interface with the following details:

- Title Bar:** AxonDashboard: query | H2 Console
- Address Bar:** Not secure | host.docker.internal:49969/h2-console/login.do?jsessionid=2d1f08f3c4fc5f0f04f0598259379839
- Toolbar:** Includes back, forward, refresh, and various configuration buttons like Auto commit (checked), Max rows: 1000, Auto complete (Off), Auto select (On), Run, Run Selected, Auto complete, Clear, and SQL statement input field.
- Left Sidebar (Schema Browser):** Shows the database structure:
  - jdbc:h2:mem:mphasisdb
  - ASSOCIATION\_VALUE\_ENTRY
  - PRODUCTLOOKUP
  - PRODUCTS
  - SAGA\_ENTRY
  - TOKEN\_ENTRY
  - INFORMATION\_SCHEMA
  - Sequences
  - UsersH2 2.1.214 (2022-06-13)
- SQL Statement Input:** SELECT \* FROM PRODUCTLOOKUP
- Result Area:** Displays the query results:

```
SELECT * FROM PRODUCTLOOKUP;
PRODUCT_ID TITLE
(no rows, 1 ms)
```

An "Edit" button is present below the result table.

# Lookup in Event Store

The screenshot shows the Axon Server dashboard interface. The top navigation bar includes tabs for 'AxonDashboard: query' (selected), 'H2 Console', and a '+' button. The URL in the address bar is 'localhost:8024/#query'. The left sidebar has a dark theme with icons for 'Settings', 'Overview', 'Search' (selected), 'Commands', 'Queries', 'Users', and 'Plugins'. The main content area features the 'Axon Server' logo and search controls. It allows searching for 'Events' or 'Snapshots' with a time window of 'last hour' and 'Live Updates' checked. A large input field 'Enter your query here' is present, along with a 'Search' button. Below this is a section titled 'About the query language'. A table header row is shown with columns: token, eventIdentifier, aggregateIdent... (truncated), aggregateType, payloadType, payload..., payloadData, timestamp, metaData. At the bottom right, there's a 'Rows per page' dropdown set to 10, a message '0-0 of 0', and navigation arrows. The footer indicates 'Axon Server 4.6.11 by AxonIQ'.



Day - 5

# Assignment – Orders Microservice



## Assignment

### 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 Dependencies:

- Add the following dependencies to your OrdersService Spring Boot project.
- spring-boot-starter-web
- Eureka (spring-cloud-starter-netflix-eureka-client)
- Google Guava
- spring-boot-starter-validation
- axon-spring-boot-starter
- Lombok
- spring-boot-starter-data-jpa
- h2



## Assignment

### 3. Create OrdersCommandController class:

- Create a new OrdersCommandController class with a request mapping "/orders" and one method that accepts the HTTP POST request.
- The method that accepts the HTTP Post request, should accept the following JSON payload as a request body.

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

- This controller class should use the CommandGateway and publish the CreateOrderCommand.



## Assignment

- The CreateOrderCommand 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()
- 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.



## Assignment

- `orderStatus` - is an `Enum` with the following content:

```
public enum OrderStatus {  
    CREATED, APPROVED, REJECTED  
}
```

- After sending the `CreateOrderCommand`, use `queryGateway.query` to publish the `FindOrderQuery` with `orderId` and fetch the first block of `OrderSummary` 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
```

### 4. Create OrderAggregate class:

- Create a new class called OrderAggregate and make it handle the CreateOrderCommand and publish the OrderCreatedEvent.
- The OrderCreatedEvent class should have the following fields:

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

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



## Assignment

### 5. Create OrderEventsHandler class:

- Create a new @Component class called OrderEventsHandler.
- Create a new JPA Repository called OrdersRepository and inject it into OrderEventsHandler using constructor-based dependency injection.
- The OrderEventHandler class should have one @EventHandler method that handles the OrderCreatedEvent and persists order details into the "read" database.
- To persist order details into the database, create a new JPA Entity class called OrderEntity. Annotate the OrderEntity class with:

```
@Entity
```

```
@Table(name = "orders")
```



## Assignment

- 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;
```



## Assignment

6. Create OrderQueriesHandler class:
  - Create a new @Component class called OrderQueriesHandler.
  - Create a new JPA Repository called OrdersRepository and inject it into OrderQueriesHandler using constructor-based dependency injection.
  - The OrderQueriesHandler class should have one @EventHandler method that handles the FindOrderQuery and fetch the order summary from the "read" database.
7. Register with Eureka:
  - Make OrdersService microservice register with Eureka as a Client.
8. Database:
  - Since each Microservice should store data in its own database, configure this Microservice to work with a new database called "orders".



## Assignment

### 9. Run and make it work:

- 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"  
}
```

# Assignment

The screenshot shows a browser window with three tabs: "Eureka" (active), "AxonDashboard: query", and "H2 Console". The URL is "localhost:8761".

**System Status**

Environment	test	Current time	2023-07-08T10:37:27 +0000
Data center	default	Uptime	00:00
		Lease expiration enabled	false
		Renews threshold	5
		Renews (last min)	0

**DS Replicas**

localhost

**Instances currently registered with Eureka**

Application	AMIs	Availability Zones	Status
API-GATEWAY	n/a (1)	(1)	UP (1) - <a href="#">host.docker.internal:api-gateway:8082</a>
ORDERS-SERVICE	n/a (1)	(1)	UP (1) - <a href="#">orders-service:30b60a986ae75cc5277de9fc73b2ea1c</a>

**General Info**

# Send a POST request to Create Order

The screenshot shows the Postman application interface. At the top, there's a navigation bar with 'Home', 'Workspaces', 'Explore', a search bar 'Search Postman', and account options 'Sign In' and 'Create Account'. Below the header, a request card is displayed for a 'POST' method to 'http://localhost:8082/orders-service/orders'. The 'Body' tab is selected, showing a JSON payload:

```
1 {  
2   ... "productId": "f24aaaf45-4854-43f4-95bc-ab5da339a29",  
3   ... "quantity": 1,  
4   ... "addressId": "afbb5881-a872-4d13-993c-faeb8350eea5"  
5 }
```

Below the body, the response section shows a status of '200 OK' with a response time of '973 ms' and a size of '203 B'. The response body is also JSON:

```
1 {  
2   ... "orderId": "328fab61-ed3e-4f59-8170-cf39fb321810",  
3   ... "orderStatus": "CREATED",  
4   ... "message": ""  
5 }
```

The bottom of the interface includes tabs for 'Body', 'Cookies', 'Headers (3)', and 'Test Results', along with a 'Save Response' button. A 'Console' tab is also visible at the very bottom.

# Orders Table

The screenshot shows the H2 Console interface running in a browser tab titled "AxonDashboard: query". The URL is `host.docker.internal:51911/h2-console/login.do?jsessionid=dfd282b67c282a2ce17fd4bf25419a70`. The left sidebar lists database objects: `ASSOCIATION_VALUE_ENTRY`, `ORDERS`, `SAGA_ENTRY`, `TOKEN_ENTRY`, `INFORMATION_SCHEMA`, `Sequences`, and `Users`. The main area displays the results of the SQL query `SELECT * FROM ORDERS`.

ORDER_ID	ADDRESS_ID	ORDER_STATUS	PRODUCT_ID	QUANTITY	USER_ID
328fab61-ed3e-4f59-8170-cf39fb321810	afbb5881-a872-4d13-993c-faeb8350eea5	CREATED	f24aaaf45-4854-43f4-95bc-ab5da339a29	1	27b95829-4f3f-4ddf-8983-151ba010e35b

(1 row, 0 ms)

[Edit](#)

## 10. Verify results:

- Check the Event Store in the Axon server and make sure that the OrderCreatedEvent gets persisted,
- Using the /h2-console connect to the Orders database and make sure that the order details are stored there as well.



## Previewing Event in the EventStore

- Go to **Search** tab and click on **Search** button:

The screenshot shows the Axon Server search interface. On the left, there is a vertical sidebar with icons for Settings, Overview, Search (which is selected), Commands, Queries, Users, and Plugins. The main area has tabs for Eureka, AxonDashboard: query (selected), and H2 Console. The URL is localhost:8024/#query. The search interface includes a sidebar with 'About the query language' and a table displaying event data.

token	eventIdentifier	aggregateIdentifier	aggregateVersion	aggregateType	payloadType	payload...	payloadData	timestamp	metaData
4	d3a6005e-d9...	328fab61-ed...	0	OrderAggreg...	com.mphasis.core.events.Order...	<com.mphasis.core.events.OrderCreatedE...	2023-07-08...	{traceId=c...	
3	81484eb2-be...	0d217720-72...	0	OrderAggreg...	com.mphasis.core.events.Order...	<com.mphasis.core.events.OrderCreatedE...	2023-07-08...	{traceId=3...	
2	0a0c9d80-a2...	f12a3c15-bc3...	0	OrderAggreg...	com.mphasis.core.events.Order...	<com.mphasis.core.events.OrderCreatedE...	2023-07-08...	{traceId=a...	
1	66fd2a10-03...	447bf419-bd...	0	OrderAggreg...	com.mphasis.core.events.Order...	<com.mphasis.core.events.OrderCreatedE...	2023-07-08...	{traceId=b...	

Rows per page: 10 | 1-4 of 4 | < >

Axon Server 4.6.11 by AxonIQ



## Previewing Event in the EventStore

- View the event details available in Event Store:

The screenshot shows the Axon Server H2 Console interface. The left sidebar contains navigation links: Eureka, Settings, Overview, Search, Commands, Queries, Users, and Plugins. The main area displays event details for a specific event. The search bar at the top indicates "Events" is selected. The table shows the following data:

Name	Value	Actions
token	4	<a href="#">Copy</a>
eventIdentifier	d3a6005e-d98e-496f-a548-5cae8863c07	<a href="#">Copy</a>
aggregateIdentifier	328fab61-ed3e-4f59-8170-cf39fb321810	<a href="#">Copy</a>
aggregateSequenceNumber	0	<a href="#">Copy</a>
aggregateType	OrderAggregate	<a href="#">Copy</a>
payloadType	com.mphasis.core.events.OrderCreatedEvent	<a href="#">Copy</a>
payloadRevision		<a href="#">Copy</a>
payloadData	<com.mphasis.core.events.OrderCreatedEvent><orderId>328fab61-ed3e-4f59-8170-cf39fb321810</orderId><productId>f24AAF45-4854-43f4-95bc-ab5da339a29</productId><userId>27b95829-4f3f-4ddf-8983-151ba010e35b</userId><quantity>1</quantity><addressId>afbb5881-a872-4d13-993c-faeb8350eea5</addressId>	<a href="#">Copy</a>

Below the table, there is a timestamped log entry:

```
OrderCreatedEvent 2023-07-08T10:30:00Z {traceId=c...}
```

At the bottom right, there are buttons for "Search" and "Copy". The footer indicates "Rows per page: 10" and "1 - 4 of 4". The bottom right corner of the window shows "Axon Server 4.6.11" and "by AxonIQ".



Day - 5

# Working with Data in Microservices

- One of the benefits of microservice architecture is that we can choose the technology stack per service.
- For instance, we can decide to use a relational database for service A and a NoSQL database for service B.
- This model lets the service manage domain data independently on a data store that best suites its data types and schema. Further, it also lets the service scale its data stores on demand and insulates it from the failures of other services.
- However, at times a transaction can span across multiple services, and ensuring data consistency across the service database is a challenge.



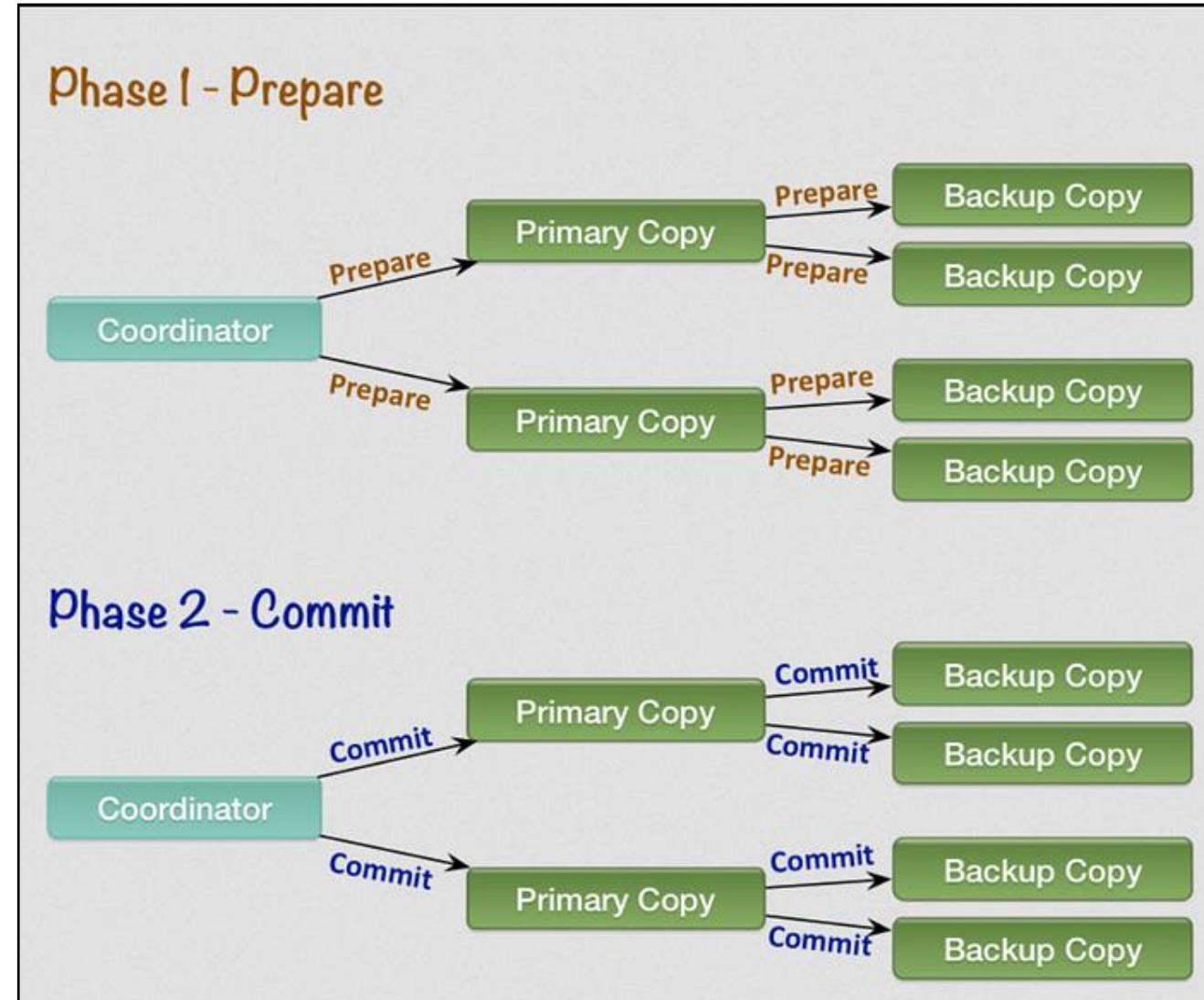
## Challenges of Distributed Transaction

- Distributed transactions in a microservice architecture pose two key challenges:
  1. The first challenge is maintaining ACID.
  2. The second challenge is managing the transaction isolation level.



## The Traditional Two-phase Commit (2PC) Protocol

- The **Two-Phase Commit (2PC) protocol** is a distributed algorithm used to ensure that all nodes in a distributed system agree to commit or abort a transaction.
- 2PC works by coordinating transactions between a coordinator node and multiple participant nodes. The coordinator sends a request to the participants to prepare for the transaction, and once all participants respond with a positive acknowledgement, the coordinator sends a commit message to each participant to commit the transaction.
- If any participant fails to respond or sends a negative acknowledgement, the coordinator sends an abort message to all participants, and the transaction is rolled back.
- **2PC guarantees that all nodes will commit or abort a transaction, but it can be slow and vulnerable to failure.**





## The SAGA Architectural Pattern

- On the other hand, SAGA Pattern is a Microservices design pattern for **managing long-lived transactions in a distributed system**.
- A saga is a **sequence of local transactions**, where each local transaction updates the state of a single service, and each service has its own database.
- Each local transaction is an atomic operation that either completes successfully or compensates for its effects. If a local transaction fails, the saga executes a **compensating transaction** that undoes the effects of the failed transaction.
- In the Saga pattern, a **compensating transaction must be *idempotent* and *retryable***. These two principles ensure that we can manage transactions without any manual intervention.
- The saga pattern allows for more flexibility than 2PC and can better handle complex, long-lived transactions, but it requires careful design to ensure data consistency.



## The SAGA Architectural Pattern

- There are two ways of coordination sagas:
- Choreography - each local transaction publishes domain events that trigger local transactions in other services
- Orchestration - an orchestrator (object) tells the participants what local transactions to execute



## When to use SAGA and 2PC commit in Design?

- The 2PC protocol is **useful in situations where all participants of the distributed transaction must commit or roll back the transaction together**. It ensures **atomicity** and **consistency** of the transaction but can lead to blocking and performance issues in highly distributed systems.
- Therefore, it's typically used in scenarios with a small number of participants, where the latency and scalability limitations of 2PC can be managed.



## When to use SAGA and 2PC commit in Design?

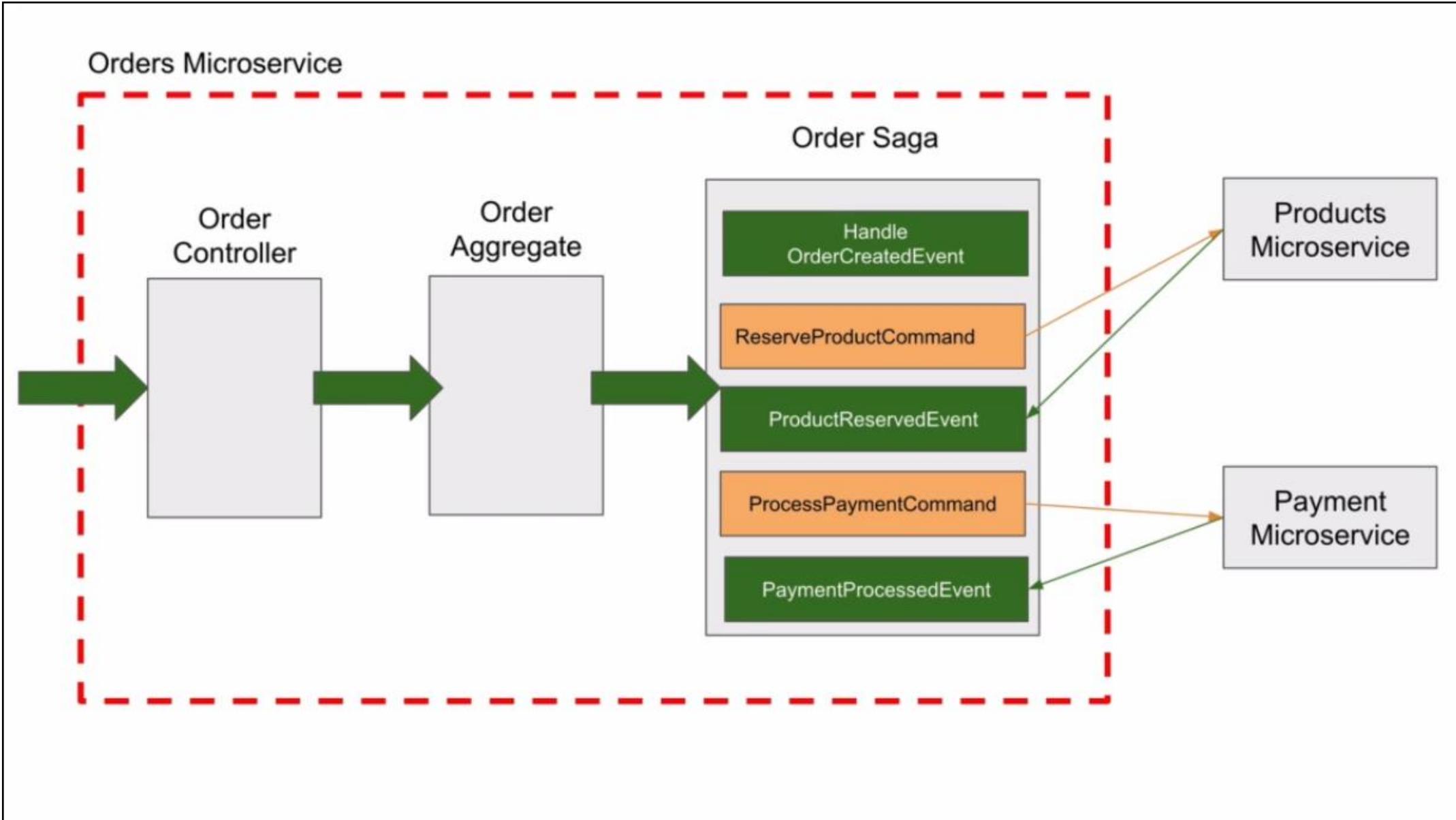
- In contrast, **SAGA pattern** is useful in situations where the transaction is too large to be managed by a single 2PC protocol. SAGA breaks the transaction down into smaller, local transactions that can be independently managed by each microservice.
- This allows for greater scalability, fault tolerance, and performance in highly distributed systems. SAGA is typically used in complex, long-running business processes that involve multiple microservices, where a transaction rollback would be expensive or not feasible.



Day - 5

# Order Saga: Orchestration-based Saga

# Order Saga: Orchestration-based Saga





## Saga Class Structure Overview

```
@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) {
        //
    }
}
```



# Saga Class Structure Overview

```
@Saga ← @Saga annotation
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) {
        //
    }
}
```



## Saga Class Structure Overview

```
@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) {
        //
    }
}
```



## Saga Class Structure Overview

```
@Saga ← @Saga annotation
public class OrderSaga {

    @Autowired
    private transient CommandGateway commandGateway;

    @StartSaga ← Start Saga
    @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 ← End Saga
    @SagaEventHandler(associationProperty = "orderId")
    public void handle(OrderApprovedEvent orderApprovedEvent) {
        //
    }
}
```



## Saga Class Structure Overview

```
@Saga ← @Saga annotation
public class OrderSaga {

    @Autowired
    private transient CommandGateway commandGateway;

    @StartSaga ← Start Saga
    @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 ← End Saga
    @SagaEventHandler(associationProperty = "orderId")
    public void handle(OrderApprovedEvent orderApprovedEvent) {
        //
    }
}
```

Event Handler method



# Saga Class Structure Overview

```
@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) {
        //
    }
}
```

Annotations and Methods:

- @Saga annotation**: Marks the class as a saga.
- @StartSaga**: Marks the first event handler method.
- first**: A red hexagonal callout pointing to the first event handler.
- Event Handler method**: A general label for the methods annotated with `@SagaEventHandler`.
- End Saga**: Marks the last event handler method.
- last**: A red hexagonal callout pointing to the last event handler.



# Saga Class Structure Overview

```
@Saga           ← @Saga annotation
public class OrderSaga {

    @Autowired
    private transient CommandGateway commandGateway;

    @StartSaga ← Start Saga
    first      @SagaEventHandler(associationProperty = "orderId") ← Association property
    public void handle(OrderCreatedEvent orderCreatedEvent) {
        //
    }

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

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

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

Event Handler method



# Saga Class Structure Overview

```
@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) {
        //
    }
}
```

Annotations and Methods:

- @Saga annotation**: Marks the class as a saga.
- @StartSaga**: Marks the first event handler.
- @SagaEventHandler(associationProperty = "orderId")**: Marks the first event handler and specifies the association property.
- first**: A red arrow points to the first event handler.
- Event Handler method**: A red arrow points to the handle method of the first event handler.
- Start Saga**: A red arrow points to the @StartSaga annotation.
- Association property**: A red arrow points to the associationProperty value in the @SagaEventHandler annotation.
- public class OrderCreatedEvent {**: A red arrow points to the start of the event class definition.
- private String orderId;**: A red arrow points to the orderId field.
- private String productId;**: A red arrow points to the productId field.
- private int quantity;**: A red arrow points to the quantity field.
- private String userId;**: A red arrow points to the userId field.
- private OrderStatus orderStatus;**: A red arrow points to the orderStatus field.
- public String getOrderId() { return orderId; }**: A red arrow points to the getOrderId method.
- // Other methods**: A red arrow points to the end of the event class definition.
- End Saga**: A red arrow points to the @EndSaga annotation.
- last**: A red arrow points to the last event handler.



## Recap of Day – 5

- Handle Error & Rollback Transaction
- Introduction to Error Handling
- Creating a centralized Error Handler class
- Return custom error object
- @ExceptionHandler Annotation
- Creating the ListenerInvocationErrorHandler
- Register the ListenerInvocationErrorHandler
- Assignment – Orders Microservice



## Recap of Day – 5

- The Traditional Two-phase Commit (2PC) Protocol
- The SAGA Architectural Pattern
- When to use SAGA and 2PC commit in Design?
- Order Saga: Orchestration-based Saga
- Saga Class Structure Overview



Day – 6,  
7, 8

- Order Saga: Orchestration-based Saga
- Create a OrderSaga class
- Create the ReserveProductCommand class
- Publish the ReserveProductCommand Instance
- Handle the ReserveProductCommand in the ProductService
- Create the ProductReservedEvent class
- Publish the ProductReservedEvent Instance
- Updating Products projection
- Handle the ProductReservedEvent in Saga
- Assignment – Users Microservice



## Day – 6, 7, 8 Agenda

- Handle the PaymentProcessedEvent
- Create and Publish the ProcessPaymentCommand
- Assignment – Payments Microservice
- Create and Publish the ApproveOrderCommand
- Handle the ApproveOrderCommand
- Create and Publish the OrderApprovedEvent
- Handle the OrderApprovedEvent and update Orders database
- Handle the OrderApprovedEvent in OrderSaga class



## Day – 6, 7, 8 Agenda

- Saga - Compensating Transactions.
- Create and Publish the CancelProductReservationCommand
- Handle the CancelProductReservationCommand in ProductService
- Create and publish the ProductReservationCancelledEvent
- Handle the ProductReservationCancelledEvent in ProductAggregate
- Create and publish the RejectOrderCommand
- Handle the RejectOrderCommand in OrderAggregate



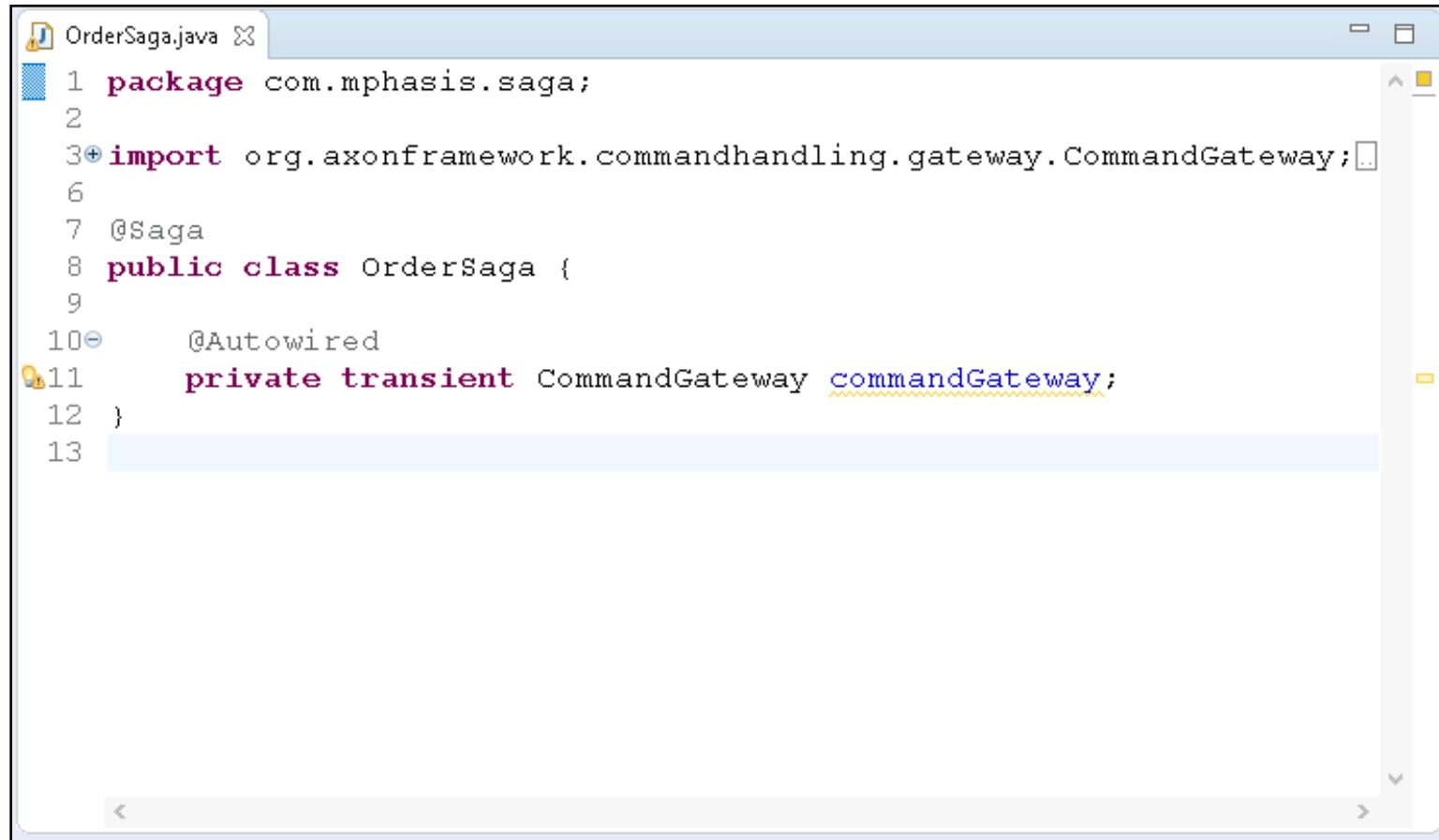
Day – 6  
& 7

# Order Saga: Orchestration-based Saga



## Create a OrderSaga class

- In OrderService, create a OrderSaga class under com.mphasis.saga package:

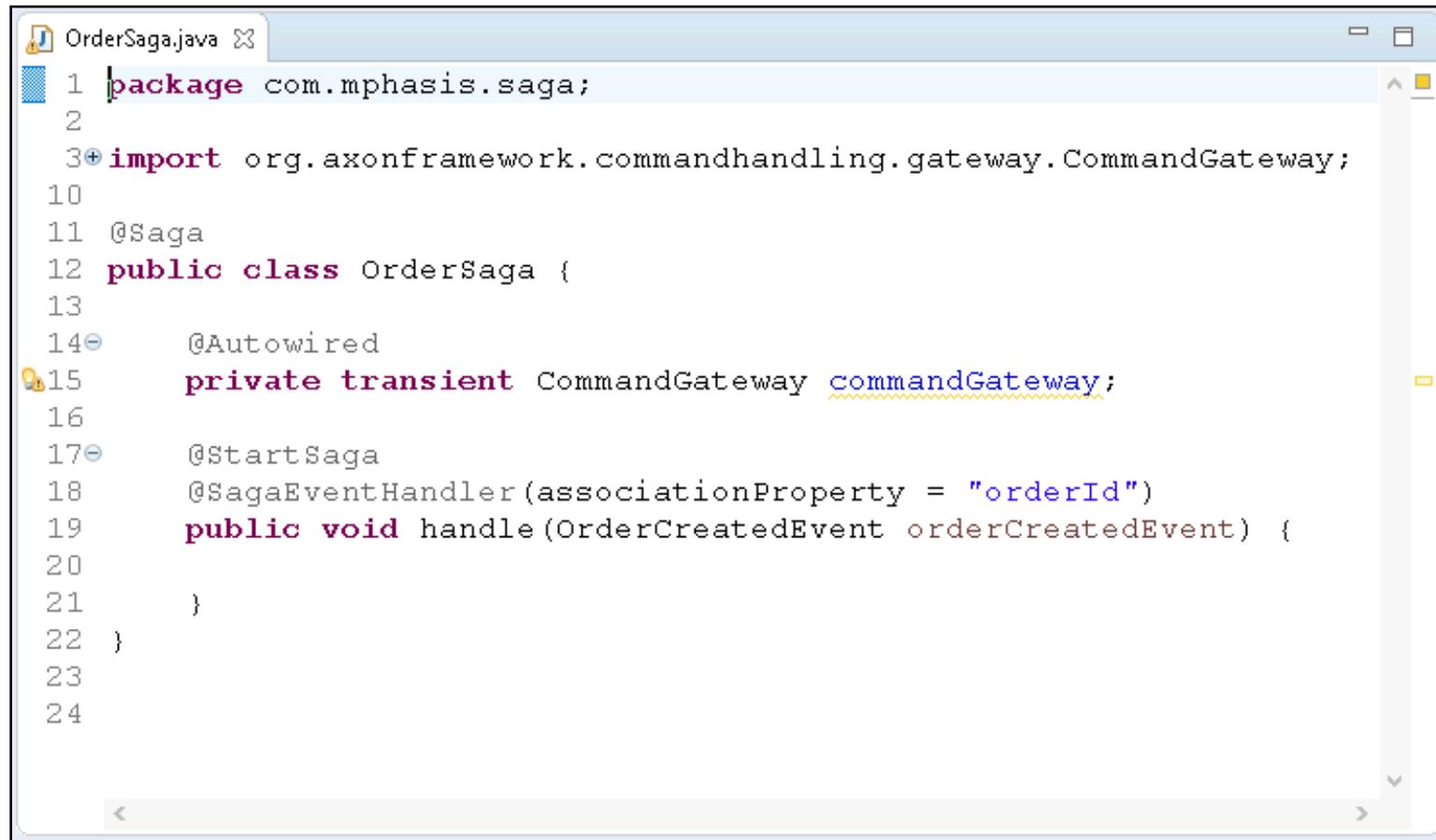


```
OrderSaga.java X
1 package com.mphasis.saga;
2
3+import org.axonframework.commandhandling.gateway.CommandGateway;□
4
5 @Saga
6 public class OrderSaga {
7
8     @Autowired
9     private transient CommandGateway commandGateway;
10 }
```



## @SagaEventHandler method

- @SagaEventHandler method for the OrderCreatedEvent:

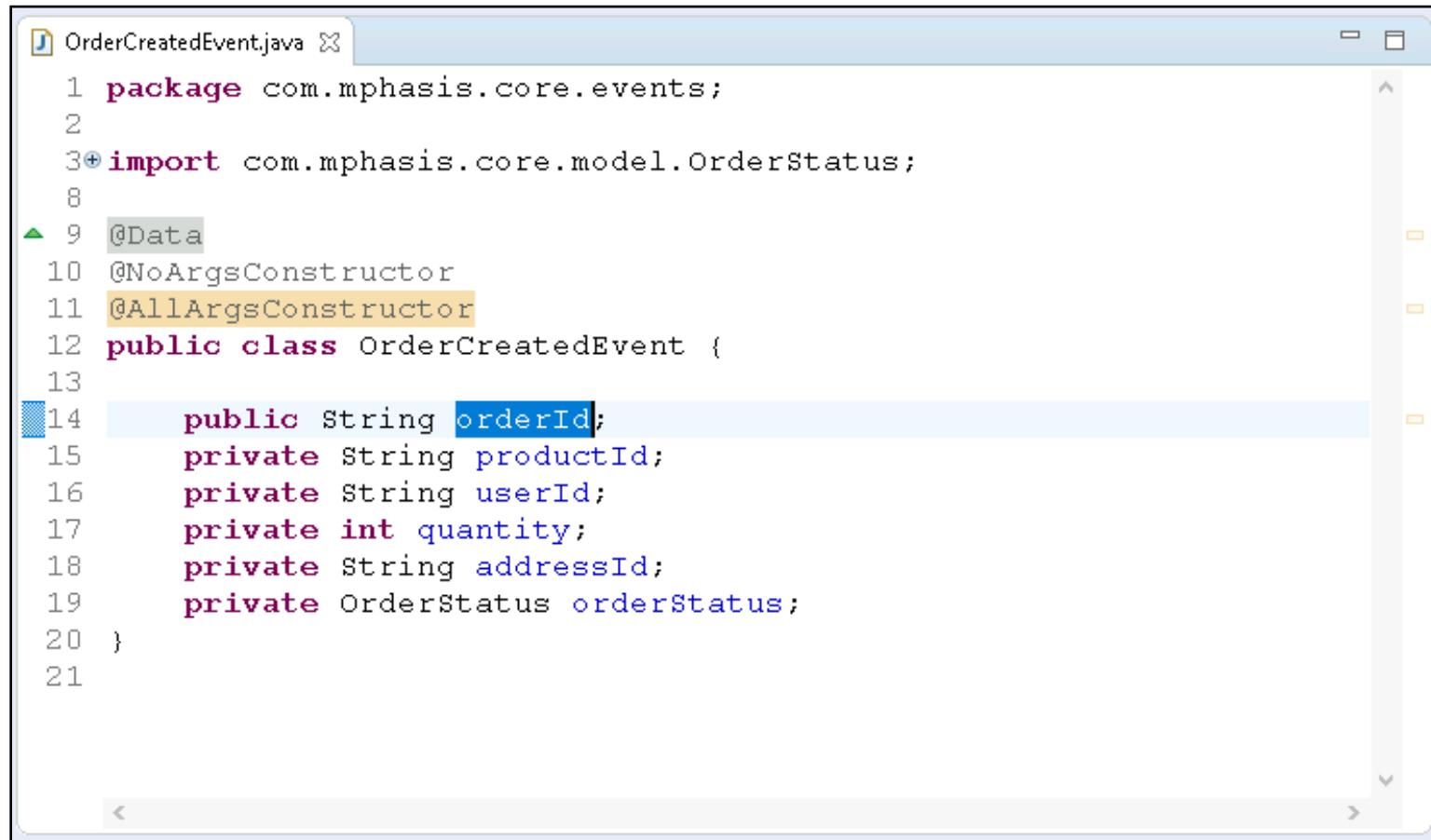


```
OrderSaga.java X
1 package com.mphasis.saga;
2
3+import org.axonframework.commandhandling.gateway.CommandGateway;
10
11 @Saga
12 public class OrderSaga {
13
14@Autowired
15 private transient CommandGateway commandGateway;
16
17@StartSaga
18 @SagaEventHandler(associationProperty = "orderId")
19 public void handle(OrderCreatedEvent orderCreatedEvent) {
20
21 }
22 }
23
24
```



## Create a OrderCreatedEvent class

- In OrderCreatedEvent, we have the associationProperty orderId with getter:

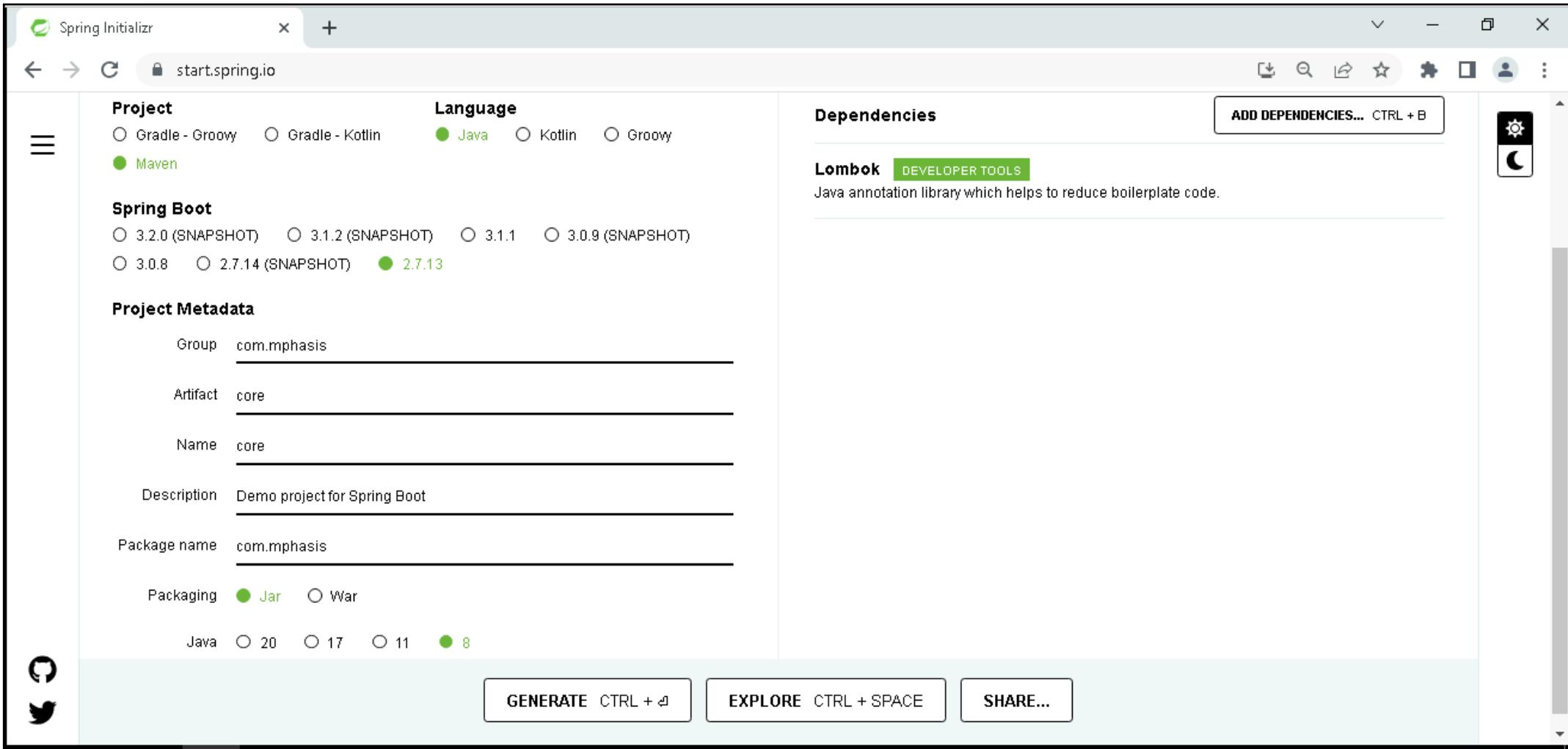


```
OrderCreatedEvent.java
1 package com.mphasis.core.events;
2
3+ import com.mphasis.core.model.OrderStatus;
4
5+ @Data
6+ @NoArgsConstructor
7+ @AllArgsConstructor
8+ public class OrderCreatedEvent {
9+
10+     public String orderId;
11+     private String productId;
12+     private String userId;
13+     private int quantity;
14+     private String addressId;
15+     private OrderStatus orderStatus;
16+
17+ }
```



# Create a new Core API module

- Create a new Spring boot project with Lombok starter with name “core”:



The screenshot shows the Spring Initializr interface at [start.spring.io](https://start.spring.io). The configuration is as follows:

- Project:** Maven (selected)
- Language:** Java (selected)
- Spring Boot:** 2.7.13 (selected)
- Project Metadata:**
  - Group: com.mphasis
  - Artifact: core
  - Name: core
  - Description: Demo project for Spring Boot
  - Package name: com.mphasis
  - Packaging: Jar (selected)
- Dependencies:** Lombok (selected)
- Java Version:** 8 (selected)

At the bottom, there are buttons for **GENERATE** (CTRL + ⌘) and **SHARE...**.



## Add axon-spring-boot-starter dependency

- 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>
```



## Adding Core project as a dependency to OrdersService

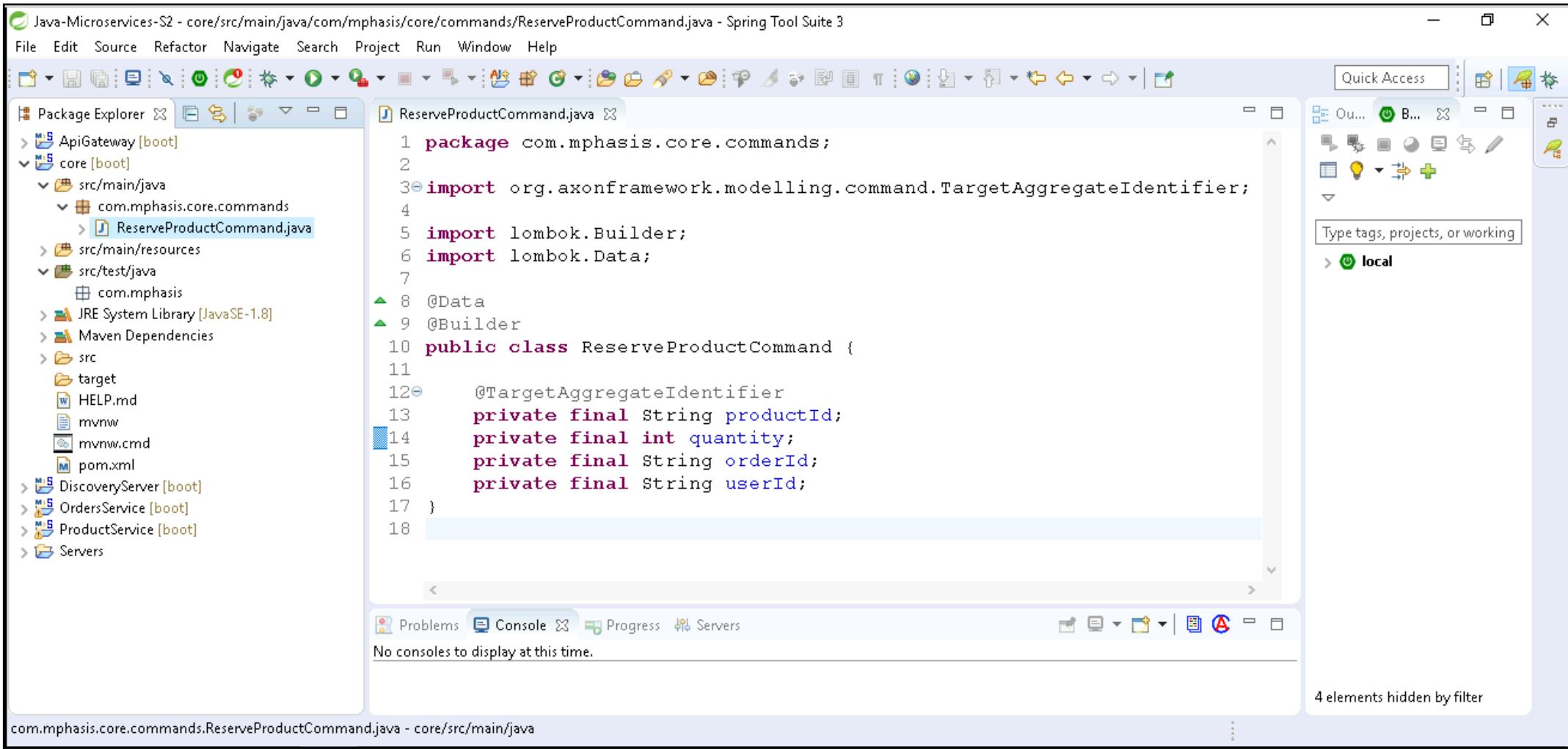
- Copy the groupId, artifactId, and version from core/pom.xml file.
- Add the new dependency into the OrderService/pom.xml file:

```
core/pom.xml OrdersService/pom.xml
60      </dependency>
61
62      <dependency>
63          <groupId>org.springframework.boot</groupId>
64          <artifactId>spring-boot-starter-validation</artifactId>
65      </dependency>
66
67      <dependency>
68          <groupId>com.mphasis</groupId>
69          <artifactId>core</artifactId>
70          <version>0.0.1-SNAPSHOT</version>
71      </dependency>
72
73      <dependency>
74          <groupId>org.springframework.boot</groupId>
75          <artifactId>spring-boot-starter-test</artifactId>
76          <scope>test</scope>
77      </dependency>
78
79  </dependencies>
80  <dependencyManagement>
81      <dependencies>
82          <dependency>
83              <groupId>org.springframework.cloud</groupId>
```



## Create the ReserveProductCommand class

- Create a new ReserveProductCommand class inside the **core** project.



The screenshot shows the Spring Tool Suite 3 interface with the following details:

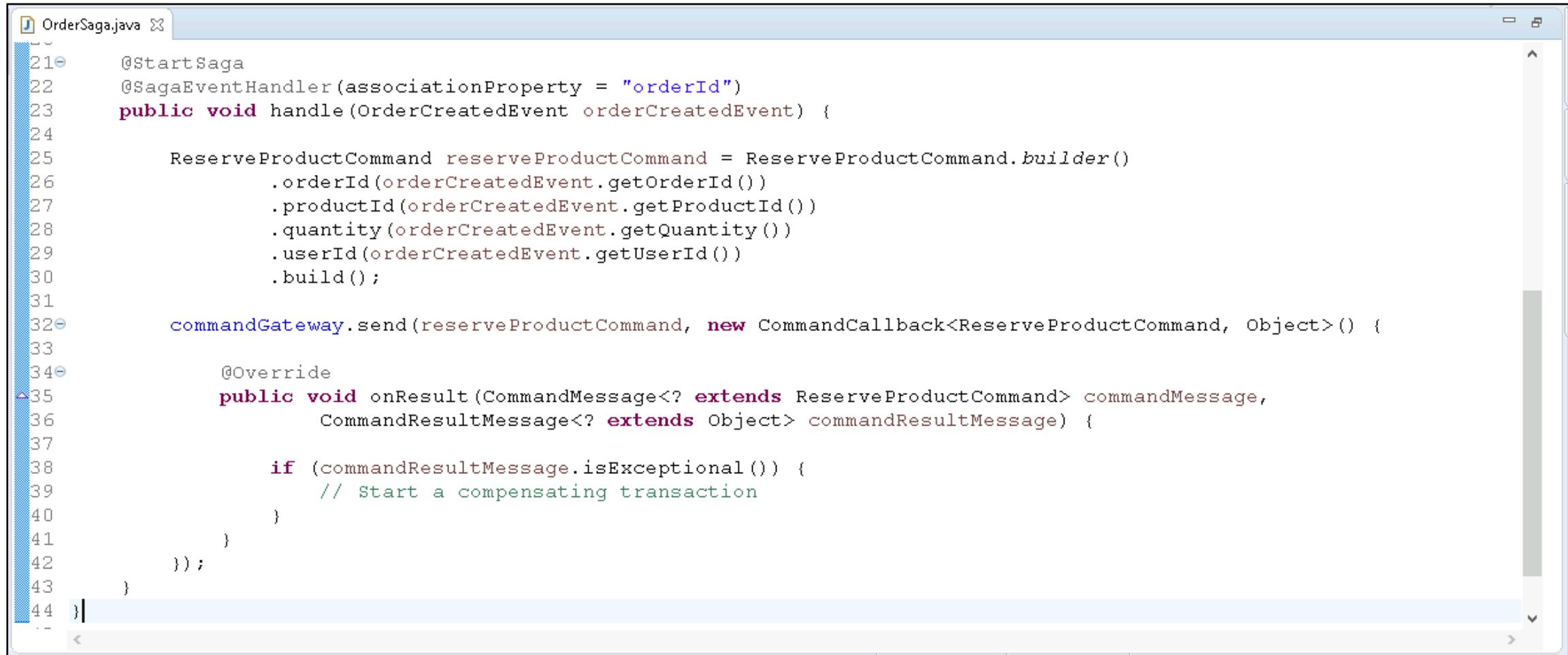
- Title Bar:** Java-Microservices-S2 - core/src/main/java/com/mpbasis/core/commands/ReserveProductCommand.java - Spring Tool Suite 3
- Menu Bar:** File Edit Source Refactor Navigate Search Project Run Window Help
- Toolbars:** Standard toolbar with various icons for file operations, search, and navigation.
- Package Explorer:** Shows the project structure:
  - ApiGateway [boot]
  - core [boot] (selected)
  - src/main/java
    - com.mphasis.core.commands (selected)
    - ReserveProductCommand.java
  - src/main/resources
  - src/test/java
    - com.mphasis
  - JRE System Library [JavaSE-1.8]
  - Maven Dependencies
  - src
    - target
    - HELP.md
    - mvnw
    - mvnw.cmd
    - pom.xml
- Editor:** Displays the code for ReserveProductCommand.java:

```
1 package com.mphasis.core.commands;
2
3 import org.axonframework.modelling.command.TargetAggregateIdentifier;
4
5 import lombok.Builder;
6 import lombok.Data;
7
8 @Data
9 @Builder
10 public class ReserveProductCommand {
11
12     @TargetAggregateIdentifier
13     private final String productId;
14     private final int quantity;
15     private final String orderId;
16     private final String userId;
17 }
18
```
- Right-hand Side Panels:**
  - Quick Access:** Shows 'local' and other project-related items.
  - Type tags, projects, or working:** A search bar with the text 'local'.
- Bottom Navigation:** Problems, Console, Progress, Servers.
- Status Bar:** com.mphasis.core.commands.ReserveProductCommand.java - core/src/main/java
- Bottom Right:** 4 elements hidden by filter



## Publish the ReserveProductCommand Instance

- Publish the ReserveProductCommand Instance:



```
OrderSaga.java
21  @StartSaga
22  @SagaEventHandler(associationProperty = "orderId")
23  public void handle(OrderCreatedEvent orderCreatedEvent) {
24
25      ReserveProductCommand reserveProductCommand = ReserveProductCommand.builder()
26          .orderId(orderCreatedEvent.getOrderId())
27          .productId(orderCreatedEvent.getProductId())
28          .quantity(orderCreatedEvent.getQuantity())
29          .userId(orderCreatedEvent.getUserId())
30          .build();
31
32      commandGateway.send(reserveProductCommand, new CommandCallback<ReserveProductCommand, Object>() {
33
34          @Override
35          public void onResult(CommandMessage<? extends ReserveProductCommand> commandMessage,
36                  CommandResultMessage<? extends Object> commandResultMessage) {
37
38              if (commandResultMessage.isExceptional()) {
39                  // Start a compensating transaction
40              }
41          }
42      });
43  }
44 }
```



## Handle the ReserveProductCommand in the ProductService

- Handle the ReserveProductCommand inside the ProductService:

```
ProductAggregate.java
46
47  @EventSourcingHandler
48  public void on(ProductCreatedEvent productCreatedEvent) {
49      this.productId = productCreatedEvent.getProductId();
50      this.price = productCreatedEvent.getPrice();
51      this.title = productCreatedEvent.getTitle();
52      this.quantity = productCreatedEvent.getQuantity();
53  }
54
55  @CommandHandler
56  public void handle(ReserveProductCommand reserverProductCommand) {
57
58      if(quantity < reserverProductCommand.getQuantity()) {
59          throw new IllegalArgumentException(
60                  "Insufficient number of items in stock");
61      }
62  }
63 }
64
65
```



## Create the ProductReservedEvent class

- Create the ProductReservedEvent class:

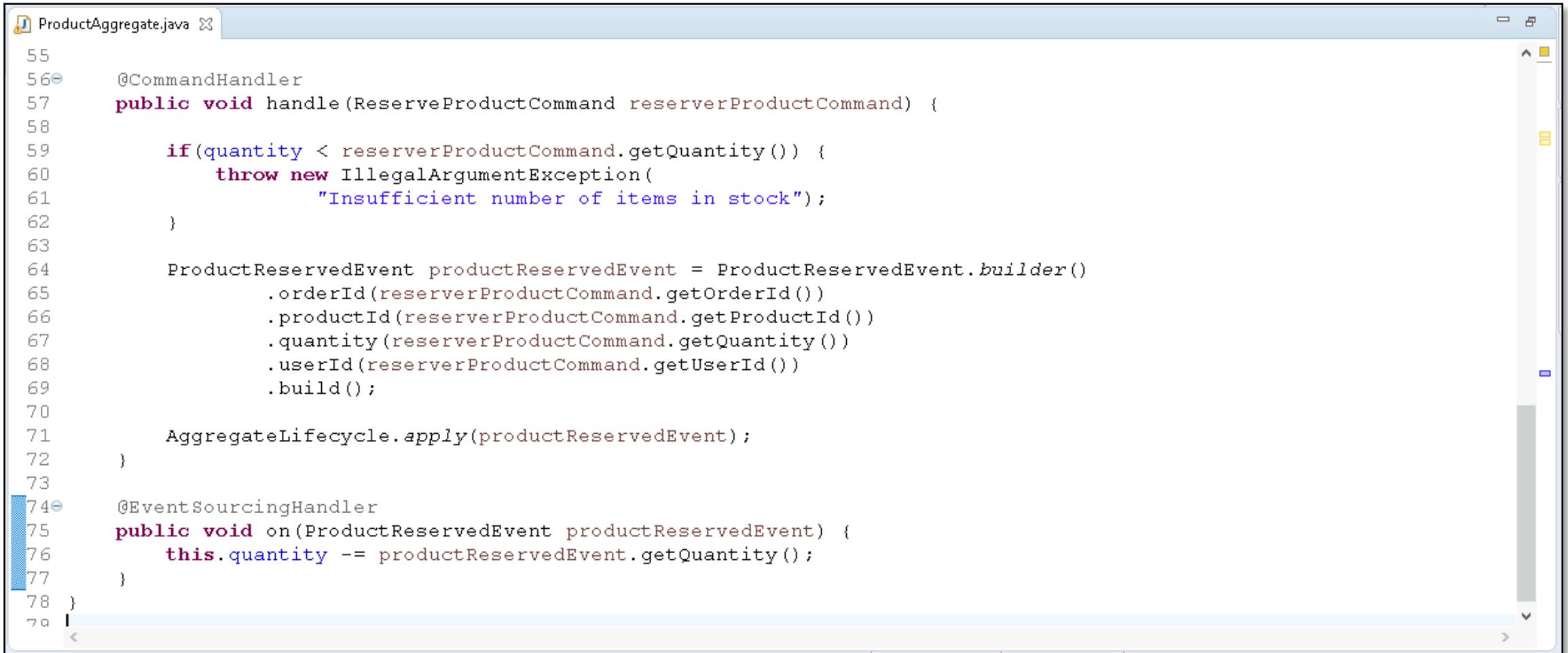
The screenshot shows a Java code editor window with the file 'ProductReservedEvent.java' open. The code defines a class named 'ProductReservedEvent' with private final fields for product ID, quantity, order ID, and user ID, annotated with @Data and @Builder from the Lombok library.

```
ProductReservedEvent.java
1 package com.mphasis.core.events;
2
3 import lombok.Builder;
4 import lombok.Data;
5
6 @Data
7 @Builder
8 public class ProductReservedEvent {
9
10     private final String productId;
11     private final int quantity;
12     private final String orderId;
13     private final String userId;
14 }
15
```



## Publish the ProductReservedEvent Instance

- Let's go to our Productservice – ProductAggregate class and publish this event.



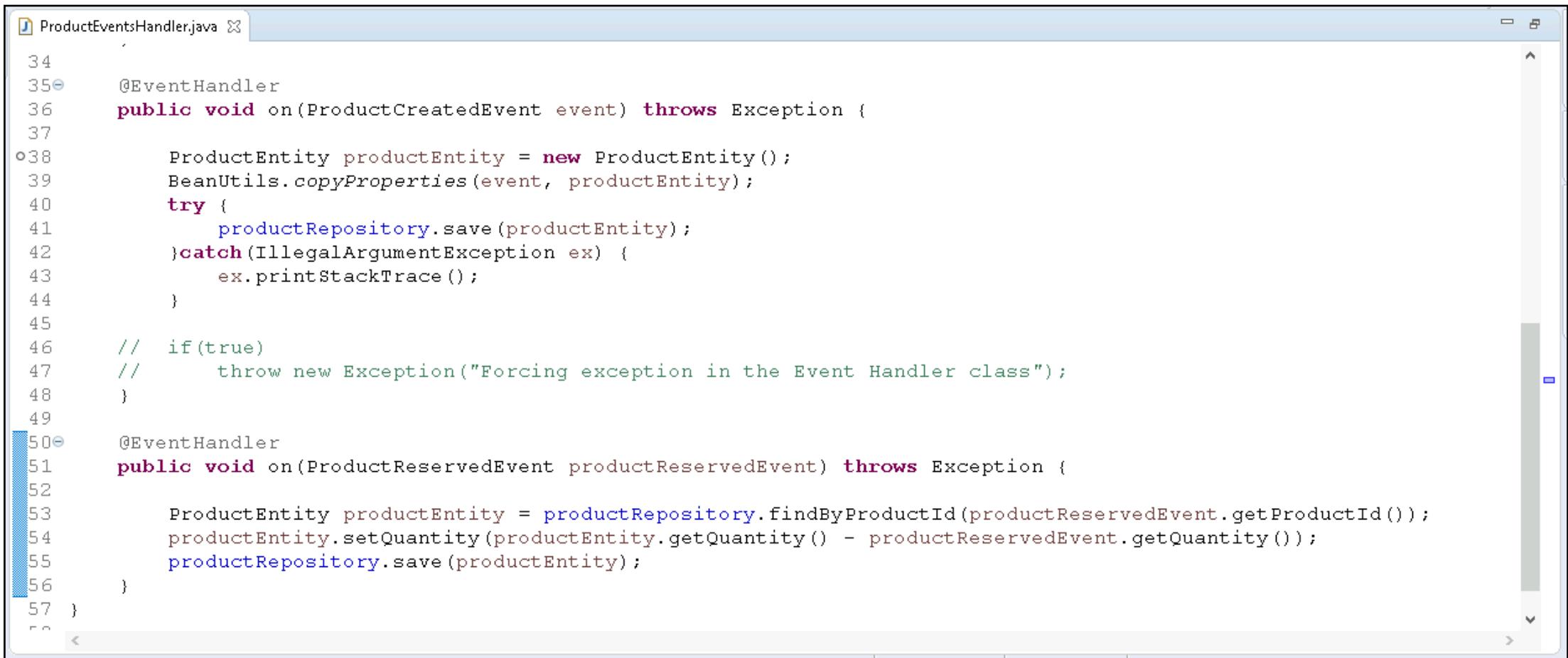
The screenshot shows a Java code editor window with the file `ProductAggregate.java` open. The code implements the `ProductAggregate` class with two methods: `handle` and `on`. The `handle` method handles a `ReserveProductCommand`, checks if there are enough items in stock, creates a `ProductReservedEvent` with the required details, and applies it to the aggregate. The `on` method is an event sourcing handler that updates the aggregate's quantity by subtracting the quantity from the reserved event.

```
55
56@CommandHandler
57public void handle(ReserveProductCommand reserverProductCommand) {
58
59    if(quantity < reserverProductCommand.getQuantity()) {
60        throw new IllegalArgumentException(
61            "Insufficient number of items in stock");
62    }
63
64    ProductReservedEvent productReservedEvent = ProductReservedEvent.builder()
65        .orderId(reserverProductCommand.getOrderId())
66        .productId(reserverProductCommand.getProductId())
67        .quantity(reserverProductCommand.getQuantity())
68        .userId(reserverProductCommand.getUserId())
69        .build();
70
71    AggregateLifecycle.apply(productReservedEvent);
72}
73
74@EventSourcingHandler
75public void on(ProductReservedEvent productReservedEvent) {
76    this.quantity -= productReservedEvent.getQuantity();
77}
78}
```



## Updating Products projection

- 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.



```
ProductEventsHandler.java

34
35@EventHandler
36    public void on(ProductCreatedEvent event) throws Exception {
37
38        ProductEntity productEntity = new ProductEntity();
39        BeanUtils.copyProperties(event, productEntity);
40        try {
41            productRepository.save(productEntity);
42        }catch(InvalidArgumentException ex) {
43            ex.printStackTrace();
44        }
45
46        // if(true)
47        //     throw new Exception("Forcing exception in the Event Handler class");
48    }
49
50@EventHandler
51    public void on(ProductReservedEvent productReservedEvent) throws Exception {
52
53        ProductEntity productEntity = productRepository.findById(productReservedEvent.getProductId());
54        productEntity.setQuantity(productEntity.getQuantity() - productReservedEvent.getQuantity());
55        productRepository.save(productEntity);
56    }
57}
```



## Handle the ProductReservedEvent in Saga

- In OrderService – OrderSaga, create a new handler method with the same associationProperty i.e., orderId:

The screenshot shows a Java code editor window with the title bar "OrderSaga.java". The code is annotated with line numbers (45, 46, 47, 48, 49, 50) on the left. The code itself is as follows:

```
45
46     @SagaEventHandler(associationProperty = "orderId")
47     public void handle(ProductReservedEvent productReservedEvent) {
48         // Process user payment
49     }
50 }
```



## Trying how it works

1. Run the AxonServer using Docker command.
2. Ensure the Discovery Server (Eureka Server), Product Service and OrdersService is running.
3. Ensure the ApiGateway is running.



## Instances currently registered with Eureka

The screenshot shows the Eureka UI interface running on localhost:8761. It includes sections for 'DS Replicas' (localhost), 'Instances currently registered with Eureka', and 'General Info'.

**DS Replicas**  
localhost

**Instances currently registered with Eureka**

Application	AMIs	Availability Zones	Status
API-GATEWAY	n/a (1)	(1)	UP (1) - <a href="#">host.docker.internal:api-gateway:8082</a>
ORDERS-SERVICE	n/a (1)	(1)	UP (1) - <a href="#">orders-service:8c7b14f494fc1de25f79ea1a90204c61</a>
PRODUCTS-SERVICE	n/a (1)	(1)	UP (1) - <a href="#">products-service:7a089446e070ce9570948110812fcdb8c</a>

**General Info**

Name	Value
------	-------

# Send a POST request to Create Product

The screenshot shows the Postman application interface. At the top, there is a navigation bar with 'Home', 'Workspaces', 'Explore', a search bar 'Search Postman', and account options 'Sign In' and 'Create Account'. Below the navigation bar, a request card is displayed with the method 'POST' and URL 'http://localhost:8082/p'. The main workspace shows a POST request to 'http://localhost:8082/products-service/products'. The 'Body' tab is selected, showing a JSON payload:

```
1 {  
2   ... "title": "iPhone 9",  
3   ... "price": 500,  
4   ... "quantity": 5  
5 }  
6
```

The 'Body' tab also includes tabs for 'Pretty', 'Raw', 'Preview', 'Visualize', and 'Text'. Below the body, the response status is shown as 'Status: 200 OK' with a timestamp of 'Time: 1113 ms' and a size of 'Size: 152 B'. The response content is a single line: '39e75b3f-5f09-4b2b-8fb2-47593b49599b'. At the bottom, there are tabs for 'Console' and other interface elements.

# Products Table

The screenshot shows the H2 Console interface running in a browser tab titled "AxonDashboard: query". The left sidebar lists database objects: ASSOCIATION\_VALUE\_ENTRY, PRODUCTLOOKUP, PRODUCTS, SAGA\_ENTRY, TOKEN\_ENTRY, INFORMATION\_SCHEMA, Sequences, and Users. The main area contains a SQL statement editor with the query "SELECT \* FROM PRODUCTS" and its execution results.

SQL statement:

```
SELECT * FROM PRODUCTS;
```

Execution results:

PRODUCT_ID	PRICE	QUANTITY	TITLE
39e75b3f-5f09-4b2b-8fb2-47593b49599b	500.00	5	iPhone 9

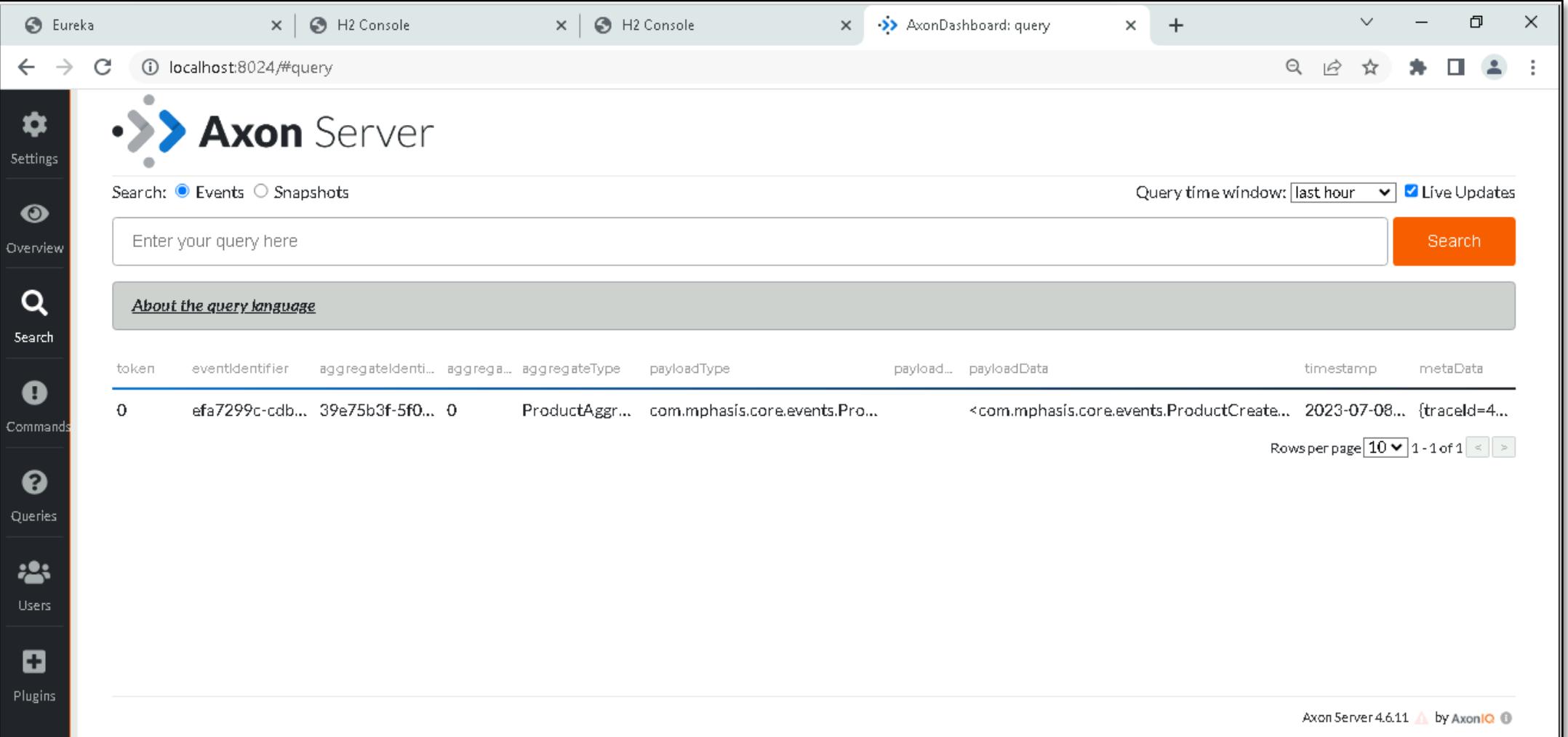
(1 row, 1 ms)

Buttons: Run, Run Selected, Auto complete, Clear, SQL statement.



# Previewing Event in the EventStore

- Go to **Search** tab and click on **Search** button:



The screenshot shows the Axon Server dashboard with the "Search" tab selected. The main interface is titled "Axon Server". On the left, there is a sidebar with icons for Settings, Overview, Search (which is selected), Commands, Queries, Users, and Plugins. The search interface includes a "Search: Events" radio button, a "Query time window" dropdown set to "last hour", and a "Live Updates" checkbox. A search bar contains the placeholder "Enter your query here" and an orange "Search" button. Below the search bar, a section titled "About the query language" provides information on the query language used for searching events. The main content area displays a table of event logs. The columns are: token, eventIdentifier, aggregateIdentifier, aggregateType, payloadType, payload..., payloadData, timestamp, and metaData. One event is listed in the table:

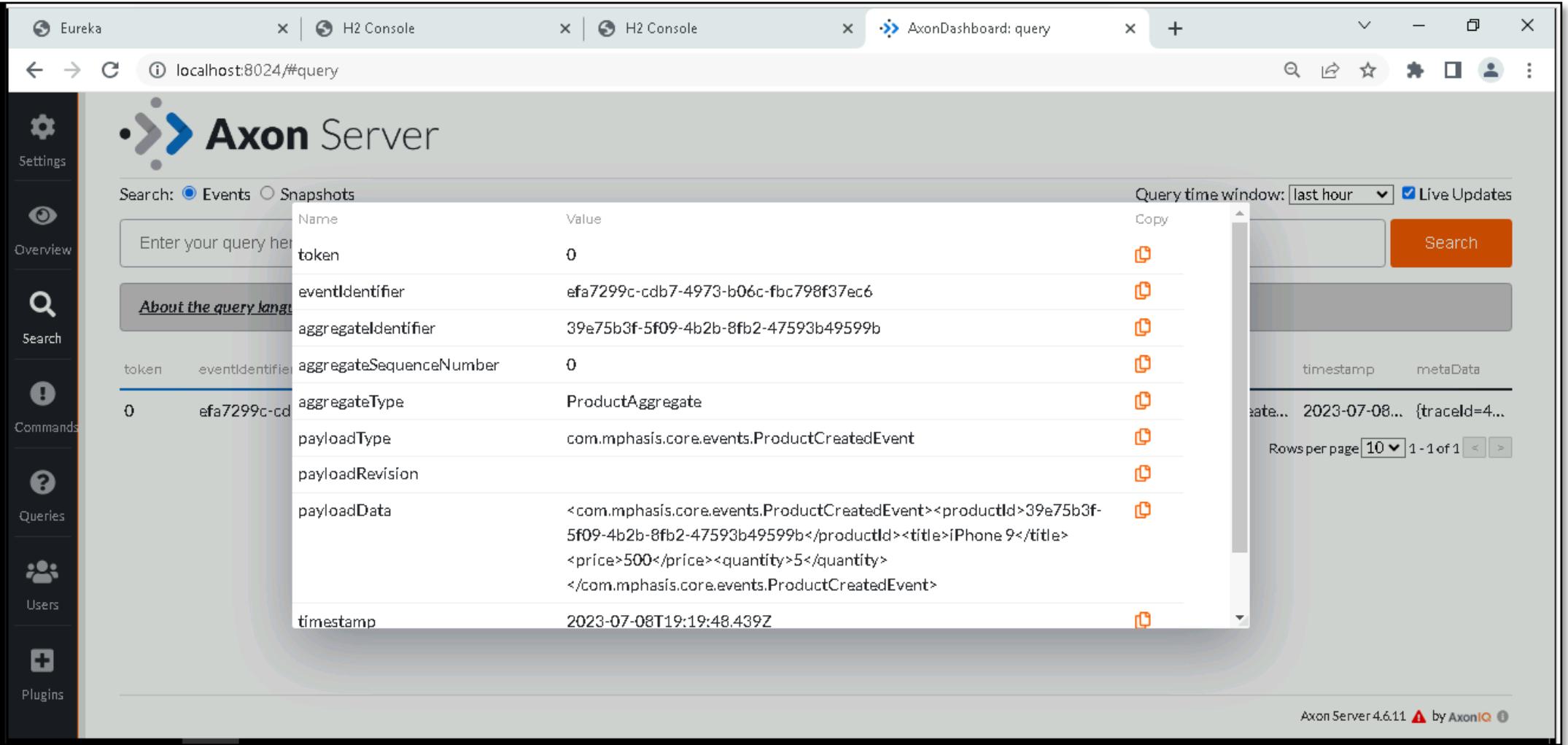
token	eventIdentifier	aggregateIdentifier	aggregateType	payloadType	payload...	payloadData	timestamp	metaData
0	efa7299c-cdb...	39e75b3f-5f0...	0	ProductAggr...	com.mphasis.core.events.Pro...	<com.mphasis.core.events.ProductCreate...	2023-07-08...	{traceId=4...

Below the table, there are buttons for "Rows per page" (set to 10), "1 - 1 of 1", and navigation arrows. At the bottom right, the footer reads "Axon Server 4.6.11 by AxonIQ".



# Previewing Event in the EventStore

- View the event details available in Event Store:



The screenshot shows the Axon Server dashboard interface. On the left, there is a vertical sidebar with icons for Settings, Overview, Search, Commands, Queries, Users, and Plugins. The main area displays a table of event details. The table has columns for Name and Value. The event details are as follows:

Name	Value
token	0
eventIdentifier	efa7299c-cdb7-4973-b06c-fbc798f37ec6
aggregateIdentifier	39e75b3f-5f09-4b2b-8fb2-47593b49599b
aggregateSequenceNumber	0
aggregateType	ProductAggregate
payloadType	com.mphasis.core.events.ProductCreatedEvent
payloadRevision	
payloadData	<com.mphasis.core.events.ProductCreatedEvent><productId>39e75b3f-5f09-4b2b-8fb2-47593b49599b</productId><title>iPhone 9</title><price>500</price><quantity>5</quantity></com.mphasis.core.events.ProductCreatedEvent>
timestamp	2023-07-08T19:19:48.439Z

On the right side of the dashboard, there is a search bar with a "Search" button, a timestamp range selector, and a "Rows per page" dropdown set to 10. The status bar at the bottom right indicates "Axon Server 4.6.11" and "by AxonIQ".

# Send a POST request to Create Orders using productId

The screenshot shows the Postman application interface. At the top, there are navigation tabs for Home, Workspaces, and Explore, along with a search bar and user authentication links. Below the header, two recent requests are listed: a POST to http://localhost:8082/p and a POST to http://localhost:8082/o. The main workspace displays a POST request to http://localhost:8082/orders-service/orders. The request method is set to POST, and the URL is http://localhost:8082/orders-service/orders. The Headers tab shows 8 items, and the Body tab is selected, displaying a JSON payload:

```
1 {  
2   ... "productId": "39e75b3f-5f09-4b2b-8fb2-47593b49599b",  
3   ... "quantity": 1,  
4   ... "addressId": "afbb5881-a872-4d13-993c-faeb8350eea5"  
5 }
```

The response section shows a status of 200 OK with a response time of 451 ms and a size of 203 B. The response body is displayed in Pretty format:

```
1 {  
2   "orderId": "c324b562-5d5f-458d-abfe-f419a5e45773",  
3   "orderStatus": "CREATED",  
4   "message": ""  
5 }
```

At the bottom, there are tabs for Body, Cookies, Headers, Test Results, and a console window.

# Orders Table

The screenshot shows the H2 Console interface with a single tab open. The URL is `host.docker.internal:50530/h2-console/login.do?jsessionid=51bafa48f593bf831f8d66d5c1f45a0b`. The left sidebar lists database objects: `ASSOCIATION_VALUE_ENTRY`, `ORDERS`, `SAGA_ENTRY`, `TOKEN_ENTRY`, `INFORMATION_SCHEMA`, `Sequences`, and `Users`. The main area contains a SQL statement editor with the query `SELECT * FROM ORDERS` and a results table.

SQL statement:

```
SELECT * FROM ORDERS;
```

ORDER_ID	ADDRESS_ID	ORDER_STATUS	PRODUCT_ID	QUANTITY	USER_ID
c324b562-5d5f-458d-abfe-f419a5e45773	afbb5881-a872-4d13-993c-faeb8350eea5	CREATED	39e75b3f-5f09-4b2b-8fb2-47593b49599b	1	27b95829-4f3f-4ddf-8983-151ba010e3

(1 row, 2 ms)

Edit



## Previewing Event in the EventStore

- Go to **Search** tab and click on **Search** button:

The screenshot shows the Axon Server dashboard with the "Search" tab selected. The main interface includes a sidebar with icons for Settings, Overview, Search (which is selected), Commands, Queries, Users, and Plugins. The main content area features the Axon Server logo and search controls for "Events" and "Snapshots". A query input field contains "Enter your query here", and a "Search" button is highlighted in orange. Below this is a section titled "About the query language". The main table displays event data with columns: token, eventIdentifier, aggregateIdentifier, aggregateType, payloadType, payloadData, timestamp, and metaData. Three rows of data are shown:

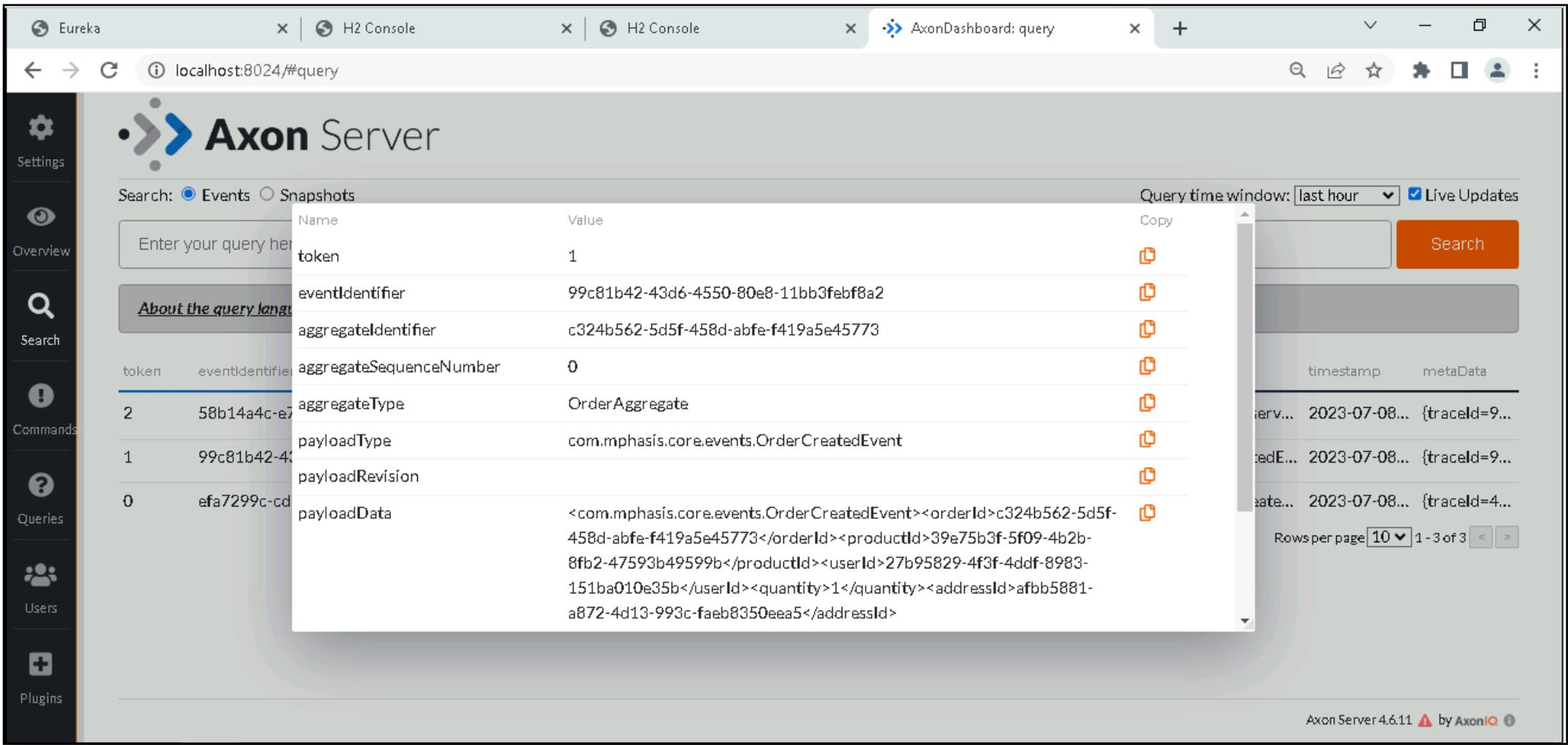
token	eventIdentifier	aggregateIdentifier	aggregateType	payloadType	payloadData	timestamp	metaData
2	58b14a4c-e7...	39e75b3f-5f0...	1	ProductAggr...	<com.mphasis.core.events.Pro...	2023-07-08...	{traceId=9...
1	99c81b42-43...	c324b562-5d...	0	OrderAggreg...	<com.mphasis.core.events.OrderCreatedE...	2023-07-08...	{traceId=9...
0	efa7299c-cdb...	39e75b3f-5f0...	0	ProductAggr...	<com.mphasis.core.events.ProductCreate...	2023-07-08...	{traceId=4...

At the bottom, there are buttons for "Rows per page" (set to 10), "1 - 3 of 3", and navigation arrows. The footer indicates "Axon Server 4.6.11 by AxonIQ".



# Previewing Event in the EventStore

- View the event details available in Event Store:



The screenshot shows the Axon Server dashboard interface. On the left, a sidebar menu includes: Eureka, H2 Console, H2 Console, AxonDashboard: query, Settings, Overview, Search, Commands, Queries, Users, and Plugins. The main area displays the "Axon Server" title and search filters for "Events" or "Snapshots". A search bar contains the placeholder "Enter your query here". Below the search bar, a link "About the query language" is visible. The main content area shows a table of event details:

Name	Value	Copy
token	1	
eventIdentifier	99c81b42-43d6-4550-80e8-11bb3febf8a2	
aggregateIdentifier	c324b562-5d5f-458d-abfe-f419a5e45773	
aggregateSequenceNumber	0	
aggregateType	OrderAggregate	
payloadType	com.mphasis.core.events.OrderCreatedEvent	
payloadRevision		
payloadData	<com.mphasis.core.events.OrderCreatedEvent><orderId>c324b562-5d5f-458d-abfe-f419a5e45773</orderId><productId>39e75b3f-5f09-4b2b-8fb2-47593b49599b</productId><userId>27b95829-4f3f-4ddf-8983-151ba010e35b</userId><quantity>1</quantity><addressId>afbb5881-a872-4d13-993c-faeb8350eea5</addressId>	

On the right side, there is a preview of event logs with columns for timestamp and metaData. The timestamp column shows dates like 2023-07-08... and the metaData column shows trace IDs. A "Search" button is also present. The bottom right corner of the dashboard displays the text "Axon Server 4.6.11" and "by AxonIQ".



# Previewing ProductReservedEvent Payload in the EventStore

- Go to **Search** tab and click on **Search** button:

The screenshot shows the Axon Server dashboard with the "Search" tab selected. The main interface includes a sidebar with icons for Settings, Overview, Search (which is highlighted in orange), Commands, Queries, Users, and Plugins. The search interface features a query input field, a "Search" button, and a "About the query language" section. The results table displays three event entries:

token	eventIdentifier	aggregateIdentifier	aggregateVersion	aggregateType	payloadType	payloadData	timestamp	metaData
2	58b14a4c-e7...	39e75b3f-5f0...	1	ProductAggr...	com.mphasis.core.events.Pro...	<com.mphasis.core.events.ProductReserv...	2023-07-08...	{traceId=9...
1	99c81b42-43...	c324b562-5d...	0	OrderAggreg...	com.mphasis.core.events.Org...	<com.mphasis.core.events.OrderCreatedE...	2023-07-08...	{traceId=9...
0	efa7299c-cdb...	39e75b3f-5f0...	0	ProductAggr...	com.mphasis.core.events.Pro...	<com.mphasis.core.events.ProductCreate...	2023-07-08...	{traceId=4...

At the bottom right, there are buttons for "Rows per page" (set to 10), "1 - 3 of 3", and navigation arrows. The footer indicates "Axon Server 4.6.11" and "by AxonIQ".



# Previewing ProductReservedEvent Payload in the EventStore

- View the event details available in Event Store:

The screenshot shows the Axon Server dashboard with the "Events" search selected. A query has been entered: "token". The results table displays the following data:

Name	Value	Copy
token	2	
eventIdentifier	58b14a4c-e7cf-46f4-8551-cc15630738ff	
aggregateIdentifier	39e75b3f-5f09-4b2b-8fb2-47593b49599b	
aggregateSequenceNumber	1	
aggregateType	ProductAggregate	
payloadType	com.mphasis.core.events.ProductReservedEvent	
payloadRevision		
payloadData	<com.mphasis.core.events.ProductReservedEvent><productId>39e75b3f-5f09-4b2b-8fb2-47593b49599b</productId><quantity>1</quantity><orderId>c324b562-5d5f-458d-abfe-f419a5e45773</orderId><userId>27b95829-4f3f-4ddf-8983-151ba010e35b</userId></com.mphasis.core.events.ProductReservedEvent>	

On the right side of the dashboard, there is a preview pane showing the timestamp and metaData for the events. The first event is from 2023-07-08... (traceId=9...) and the second is from 2023-07-08... (traceId=9...). The third event is from 2023-07-08... (traceId=4...).

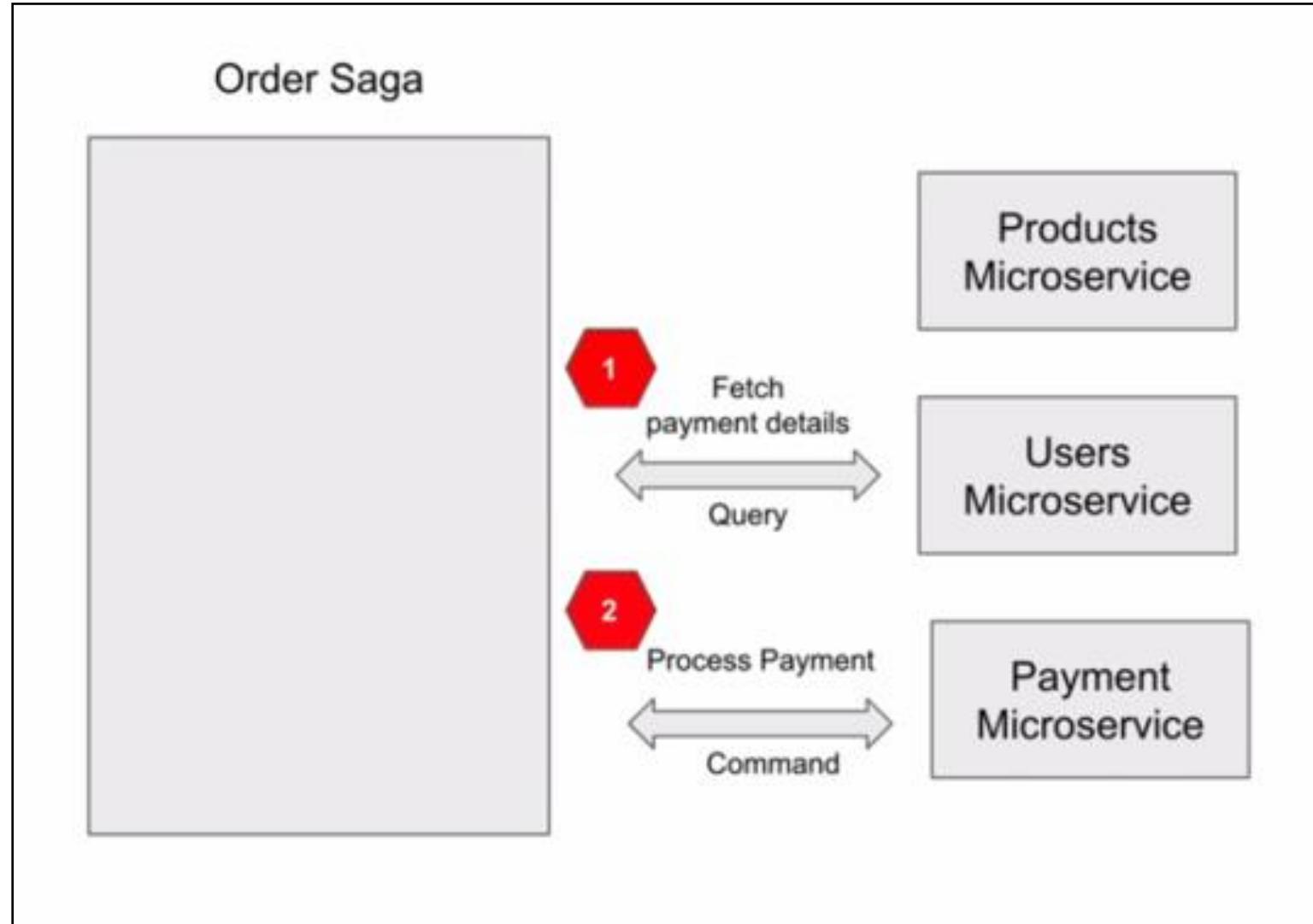


Day – 6  
& 7

Orchestration based Saga.  
Part 2. Fetch Payment Details.



# Fetch Payment Details





Day – 6  
& 7

## Assignment – Users Microservice



# Assignment

## 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 "UserService".

## 2. Dependencies

- Add the following dependencies to your UserService Spring Boot project.
- spring-boot-starter-web
- axon-spring-boot-starter
- Core
- Lombok
- guava

### 3. Create PaymentDetails class.

- 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 `core.model` package.

## 4. Create User class.

- 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 `core.model` package.



## Assignment

### 5. Create FetchUserPaymentDetailsQuery class.

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

```
private String userId;
```

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



## Assignment

### 6. Create UserEventsHandler class.

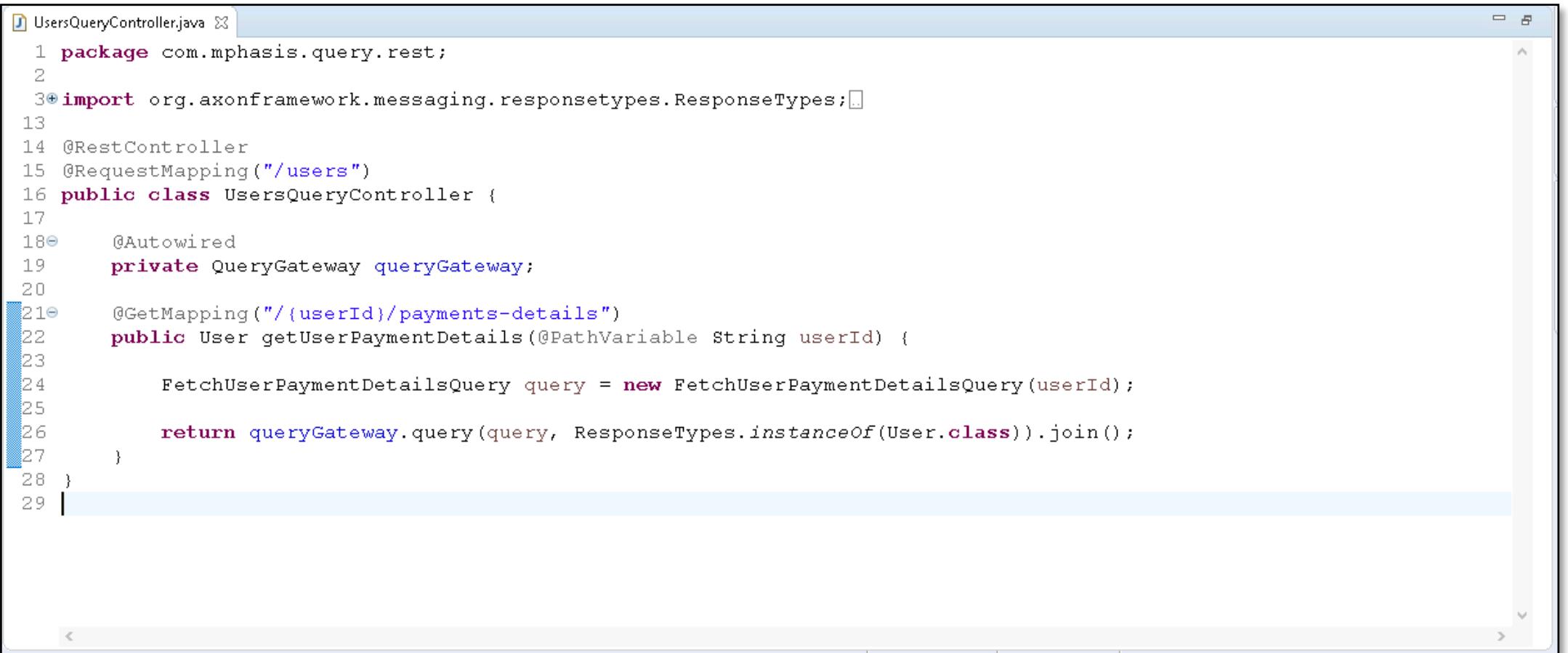
- In the UserService project, create a new UserEventsHandler class annotated with @Component annotation. In this class create a single method annotated with @QueryHandler. Make this method accept FetchUserPaymentDetailsQuery as a method argument and return an instance of a User object with hard-coded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

```
PaymentDetails paymentDetails = PaymentDetails.builder()  
    .cardNumber("123Card")  
    .cvv("123")  
    .name("Manpreet Singh Bindra")  
    .validUntilMonth(12)  
    .validUntilYear(2030)  
    .build();  
  
User user = User.builder()  
    .firstName("Manpreet Singh")  
    .lastName("Bindra")  
    .userId(query.getUserId())  
    .paymentDetails(paymentDetails)  
    .build();
```



# Assignment

## 7. Create a UsersQueryController class.

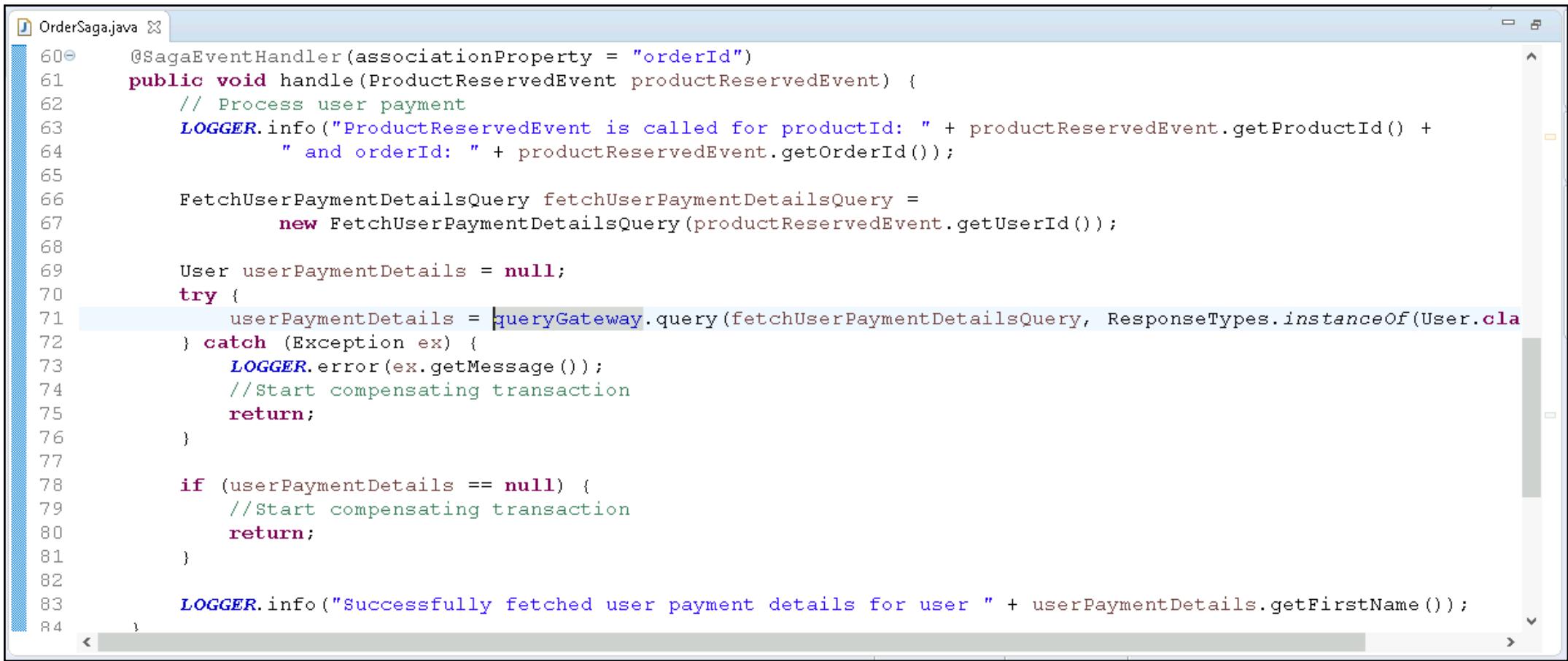


```
UsersQueryController.java X
1 package com.mphasis.query.rest;
2
3 import org.axonframework.messaging.responsetypes.ResponseTypes;
4
5 @RestController
6 @RequestMapping("/users")
7 public class UsersQueryController {
8
9     @Autowired
10    private QueryGateway queryGateway;
11
12    @GetMapping("/{userId}/payments-details")
13    public User getUserPaymentDetails(@PathVariable String userId) {
14
15        FetchUserPaymentDetailsQuery query = new FetchUserPaymentDetailsQuery(userId);
16
17        return queryGateway.query(query, ResponseTypes.instanceOf(User.class)).join();
18    }
19}
```



## Fetching User Payment Details

### 8. Fetching UserPaymentDetails inside OrderSaga:



```
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.class));
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     }
```



## Assignment

9. Register with Eureka:

- Make UserService microservice register with Eureka as a Client.

10. Run and make it work:

- Run your UserService microservice and make it work.

# Send a POST request to Create Product

The screenshot shows the Postman application interface. At the top, there are navigation links for Home, Workspaces, Explore, and a search bar labeled "Search Postman". On the right side of the header are "Sign In", "Create Account", and window control buttons.

In the main workspace, there are two tabs: "POST http://localhost:8082/p" and "POST http://localhost:8082/o". The "o" tab is currently active, showing the URL "http://localhost:8082/products-service/products". There are buttons for "Add to collection" and "Send".

The request configuration panel includes:

- Method: POST
- URL: http://localhost:8082/products-service/products ...
- Headers: (8)
- Body (highlighted): JSON
- Params, Authorization, Pre-request Script, Tests, Settings, Cookies, Beautify, and other tabs are also visible.

The Body section contains the following JSON payload:

```
1 {  
2   "title": "iPad 2",  
3   "price": 500,  
4   "quantity": 5  
5 }  
6
```

The response panel at the bottom shows:

- Status: 200 OK
- Time: 769 ms
- Size: 152 B
- Save Response
- Pretty, Raw, Preview, Visualize, Text (dropdown), and other visualization options.

The response body is displayed as:

```
1 ddb6d187-d094-4895-b9bc-c986964766ec
```

At the bottom left, there are "Console" and other interface buttons.

# Products Table

The screenshot shows a web-based H2 Database Console interface. At the top, there are three tabs: "AxonDashboard: query", "H2 Console", and another "H2 Console". The URL in the address bar is "Not secure | host.docker.internal:50347/h2-console/login.do?sessionid=83f5824b04eaefbaae0010ecfc706cae". The main area contains a sidebar with database schema navigation and a central SQL editor and results pane.

**Left Sidebar:**

- jdbc:h2:mem:productdb
- + ASSOCIATION\_VALUE\_ENTRY
- + PRODUCTLOOKUP
- + PRODUCTS
- + SAGA\_ENTRY
- + TOKEN\_ENTRY
- + INFORMATION\_SCHEMA
- + Sequences
- + Users
- ① H2 2.1.214 (2022-06-13)

**SQL Statement:**

```
SELECT * FROM PRODUCTS;
```

**Results:**

PRODUCT_ID	PRICE	QUANTITY	TITLE
ddb6d187-d094-4895-b9bc-c986964766ec	500.00	5	iPad 2

(1 row, 1 ms)

**Buttons:**

- Run
- Run Selected
- Auto complete
- Clear
- SQL statement:



## Previewing Event in the EventStore

- Go to **Search** tab and click on **Search** button:

The screenshot shows the Axon Server dashboard with the "Search" tab selected. The main interface includes a sidebar with icons for Settings, Overview, Search (selected), Commands, Queries, Users, and Plugins. The search interface features a logo, navigation buttons, and tabs for "Events" (selected) and "Snapshots". It includes a query input field, a "Search" button, and a link to "About the query language". The search results table has columns: token, eventIdentifier, aggregateIdentifier, aggregateType, payloadType, payloadData, timestamp, and metaData. A single row is shown with values: 5, 7048324e-e9..., ddb6d187-d0..., 0, ProductAggr..., com.mphasis.core.events.Pro..., <com.mphasis.core.events.ProductCreate..., 2023-07-09..., {traceId=3...}. The bottom of the screen shows pagination controls for "Rows per page" (10) and "1 - 1 of 1". The footer indicates "Axon Server 4.6.11 by AxonIQ".

token	eventIdentifier	aggregateIdentifier	aggregateType	payloadType	payloadData	timestamp	metaData	
5	7048324e-e9...	ddb6d187-d0...	0	ProductAggr...	com.mphasis.core.events.Pro...	<com.mphasis.core.events.ProductCreate...	2023-07-09...	{traceId=3...}

# Send a POST request to Create Order using productId

The screenshot shows the Postman application interface. At the top, there's a navigation bar with 'Home', 'Workspaces', 'Explore', a search bar 'Search Postman', and account options 'Sign In' and 'Create Account'. Below the header, the URL 'p://localhost:8082/p' is in the address bar, and the method 'POST http://localhost:8082/orders-service/orders' is selected. To the right of the URL are buttons for '+', '...', 'Add to collection', and 'X'. The main area shows a POST request to 'http://localhost:8082/orders-service/orders'. The 'Body' tab is active, showing JSON input:

```
1 {  
2   "productId": "ddb6d187-d094-4895-b9bc-c986964766ec",  
3   "quantity": 1,  
4   "addressId": "afbb5881-a872-4d13-993c-faeb8350eea5"  
5 }
```

Below the body, the 'Body' tab is selected, followed by 'Cookies', 'Headers (3)', and 'Test Results'. The status bar at the bottom indicates 'Status: 200 OK Time: 410 ms Size: 203 B Save Response'. The 'Pretty' tab is selected in the body preview, showing the response JSON:

```
1 {  
2   "orderId": "5f1b2c28-5d91-4aed-a2a6-0070fde1d2e9",  
3   "orderStatus": "CREATED",  
4   "message": ""  
5 }
```

At the bottom, there are tabs for 'Console' and other interface elements.

# Orders Table

The screenshot shows the AxonDashboard query interface with three tabs: AxonDashboard: query, H2 Console, and H2 Console. The main area displays the results of a SQL query against the ORDERS table.

SQL statement:

```
SELECT * FROM ORDERS;
```

Results:

ORDER_ID	ADDRESS_ID	ORDER_STATUS	PRODUCT_ID	QUANTITY	USER_ID
5f1b2c28-5d91-4aed-a2a6-0070fde1d2e9	afbb5881-a872-4d13-993c-faeb8350eea5	CREATED	ddb6d187-d094-4895-b9bc-c986964766ec	1	27b95829-4f3f-4ddf-8983-151ba010

(1 row, 0 ms)

Buttons: Run, Run Selected, Auto complete, Clear, SQL statement: SELECT \* FROM ORDERS; Edit

Schemas listed on the left: ASSOCIATION\_VALUE\_ENTRY, ORDERS, SAGA\_ENTRY, TOKEN\_ENTRY, INFORMATION\_SCHEMA, Sequences, Users. Version: H2 2.1.214 (2022-06-13).



## Previewing Event in the EventStore

- Go to **Search** tab and click on **Search** button:

The screenshot shows the Axon Server dashboard with the "Search" tab selected. The main interface includes a sidebar with icons for Settings, Overview, Search (which is selected), Commands, Queries, Users, and Plugins. The search interface has tabs for "Events" (selected) and "Snapshots". It features a search bar with placeholder "Enter your query here" and an orange "Search" button. Below the search bar is a section titled "About the query language". The main content area displays a table of event logs with the following columns: token, eventIdentifier, aggregateIdentifier, aggregateVersion, aggregateType, payloadType, payloadData, timestamp, and metaData. Three events are listed:

token	eventIdentifier	aggregateIdentifier	aggregateVersion	aggregateType	payloadType	payloadData	timestamp	metaData
7	e98765e7-66...	ddb6d187-d0...	1	ProductAggr...	com.mphasis.core.events.Pro...	<com.mphasis.core.events.ProductReserv...	2023-07-09...	{traceId=d...
6	ca10fdc2-04e...	5f1b2c28-5d...	0	OrderAggreg...	com.mphasis.core.events.Org...	<com.mphasis.core.events.OrderCreatedE...	2023-07-09...	{traceId=d...
5	7048324e-e9...	ddb6d187-d0...	0	ProductAggr...	com.mphasis.core.events.Pro...	<com.mphasis.core.events.ProductCreate...	2023-07-09...	{traceId=3...

At the bottom, there are buttons for "Rows per page" (set to 10), "1 - 3 of 3", and navigation arrows. The footer indicates "Axon Server 4.6.11" and "by AxonIQ".



## Review the OrderService Console

- Check the successful message on the OrderService console:

```
com.mphasis.saga.OrderSaga
```

```
: Successfully fetched user payment details for user Manpreet Singh
```



Day – 6  
& 7

# Orchestration based Saga. Part 3. Process User Payment.



## Create the ProcessPaymentCommand class

- In the **core** project, we will create the **ProcessPaymentCommand** class:

The screenshot shows a Java code editor window with the title bar "ProcessPaymentCommand.java". The code itself is as follows:

```
1 package com.mphasis.core.commands;
2
3 import org.axonframework.modelling.command.TargetAggregateIdentifier;
4 import com.mphasis.core.model.PaymentDetails;
5 import lombok.Builder;
6 import lombok.Data;
7
8 @Data
9 @Builder
10 public class ProcessPaymentCommand {
11
12     @TargetAggregateIdentifier
13     private final String paymentId;
14     private final String orderId;
15     private final PaymentDetails paymentDetails;
16 }
17
```



## Publish the ProcessPaymentCommand Instance

- As the ProcessPaymentCommand is ready, we will use OrderSaga to publish:

```
OrderSaga.java
85
86
87     LOGGER.info("Successfully fetched user payment details for user " + userPaymentDetails.getFirstName());
88
89     ProcessPaymentCommand processPaymentCommand = ProcessPaymentCommand.builder()
90         .orderId(productReservedEvent.getOrderId())
91         .paymentDetails(userPaymentDetails.getPaymentDetails())
92         .paymentId(UUID.randomUUID().toString())
93         .build();
94
95     String result = null;
96     try {
97         result = commandGateway.sendAndWait(processPaymentCommand, 10, TimeUnit.SECONDS);
98     } catch (Exception ex) {
99         LOGGER.error(ex.getMessage());
100        // Start compensating transaction
101    }
102
103    if(result == null) {
104        LOGGER.info("The ProcessPaymentCommand resulted in NULL. Initiating a compensating transaction");
105        // Start compensating transaction
106    }
107}
108
109
```



Day – 6  
& 7

## Assignment – Payments Microservice



## Assignment

### 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 "PaymentsService".

### 2. Dependencies

- Add the following dependencies to your PaymentsService Spring Boot project.
- spring-boot-starter-web
- axon-spring-boot-starter
- Lombok
- spring-boot-starter-data-jpa
- H2
- guava
- core

### 3. Create PaymentAggregate class

- Create a new class called `PaymentAggregate`, make it handle the `ProcessPaymentCommand`, and publish the `PaymentProcessedEvent`.

- The `PaymentProcessedEvent` class will have the following fields:

`private final String orderId;`

`private final String paymentId;`

- The `PaymentAggregate` class should also have an `@EventSourcingHandler` method that sets values for all fields in the `PaymentAggregate`.

- **Note:** Place the `PaymentProcessedEvent` into a shared **core** project.

## 4. Validate the ProcessPaymentCommand.

- In the PaymentAggregate class, add a little code to validate the ProcessPaymentCommand. If one of the required fields contains an invalid value, then throw an IllegalArumentException.

```
21@CommandHandler
22public 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
```



## Assignment

5. Create the PaymentEventsHandler class.
  - Create a new @Component class called PaymentEventsHandler.
  - Create a new JPA Repository called PaymentsRepository and inject it into PaymentEventsHandler using constructor-based dependency injection.
  - The PaymentEventsHandler class should have one @EventHandler method that handles the PaymentProcessedEvent and persists payment details into the "read" database.
  - To persist payment details into the database, create a new JPA Entity class called PaymentEntity. Annotate the PaymentEntity class with:

```
@Entity  
@Table(name = "payments")
```

- and make the PaymentEntity class have the following fields:

```
@Id  
private String paymentId;  
@Column  
public String orderId;
```



# Assignment

## 6. Database

- Since each Microservice should store data in its own database, configure Payments Microservice to work with a new database called "payments".

## 7. Register with Eureka:

- Make PaymentsService microservice register with Eureka as a Client.



## Handle the PaymentProcessedEvent

- Let's handle the PaymentProcessedEvent in OrderSaga.
- Need to add new **@SagaEventHandler** methods.

```
OrderSaga.java
109
110     @SagaEventHandler(associationProperty = "orderId")
111     public void handle(PaymentProcessedEvent paymentProcessedEvent) {
112         // Send an ApproveOrderCommand
113     }
114 }
115 |
```



Day – 6  
& 7

# Orchestration based Saga. Part 3. Process User Payment.



## Create and Publish the ApproveOrderCommand

- Create the ApproveOrderCommand class in OrdersService:

The screenshot shows a code editor window with the title bar "ApproveOrderCommand.java". The code itself is as follows:

```
1 package com.mphasis.command.commands;
2
3 import org.axonframework.modelling.command.TargetAggregateIdentifier;
4
5 import lombok.AllArgsConstructor;
6 import lombok.Data;
7
8 @Data
9@AllArgsConstructor
10 public class ApproveOrderCommand {
11
12     @TargetAggregateIdentifier
13     private final String orderId;
14 }
15 |
```



## Create and Publish the ApproveOrderCommand

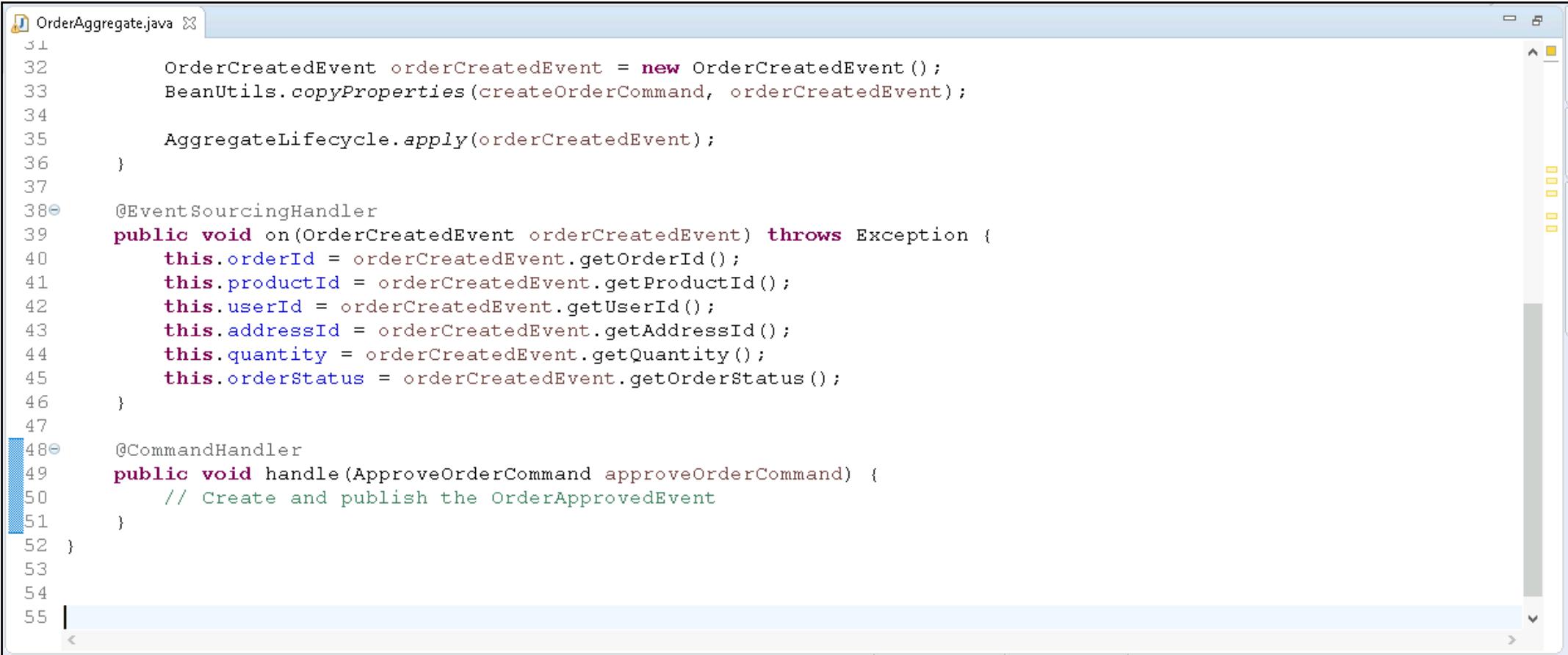
- Publish the ApproveOrderCommand Instance from OrderSaga:

```
OrderSaga.java
110
111     @SagaEventHandler(associationProperty = "orderId")
112     public void handle(PaymentProcessedEvent paymentProcessedEvent) {
113
114         // Send an ApproveOrderCommand
115         ApproveOrderCommand approveOrderCommand =
116             new ApproveOrderCommand(paymentProcessedEvent.getOrderId());
117
118         commandGateway.send(approveOrderCommand);
119     }
120 }
121
122
```



## Handle the ApproveOrderCommand

- Handle the ApproveOrderCommand in the OrderAggregate class:



The screenshot shows a Java code editor window with the file `OrderAggregate.java` open. The code implements the Command pattern, handling two types of commands: `OrderCreatedEvent` and `ApproveOrderCommand`.

```
OrderAggregate.java
31
32     OrderCreatedEvent orderCreatedEvent = new OrderCreatedEvent();
33     BeanUtils.copyProperties(createOrderCommand, orderCreatedEvent);
34
35     AggregateLifecycle.apply(orderCreatedEvent);
36 }
37
38@EventSourcingHandler
39 public void on(OrderCreatedEvent orderCreatedEvent) throws Exception {
40     this.orderId = orderCreatedEvent.getOrderId();
41     this.productId = orderCreatedEvent.getProductId();
42     this.userId = orderCreatedEvent.getUserId();
43     this.addressId = orderCreatedEvent.getAddressId();
44     this.quantity = orderCreatedEvent.getQuantity();
45     this.orderStatus = orderCreatedEvent.getOrderStatus();
46 }
47
48@CommandHandler
49 public void handle(ApproveOrderCommand approveOrderCommand) {
50     // Create and publish the OrderApprovedEvent
51 }
52 }
```



## Create and Publish the OrderApprovedEvent

- Create a new OrderApprovedEvent class in com.mphasis.core.events package:

The screenshot shows a Java code editor window with the following details:

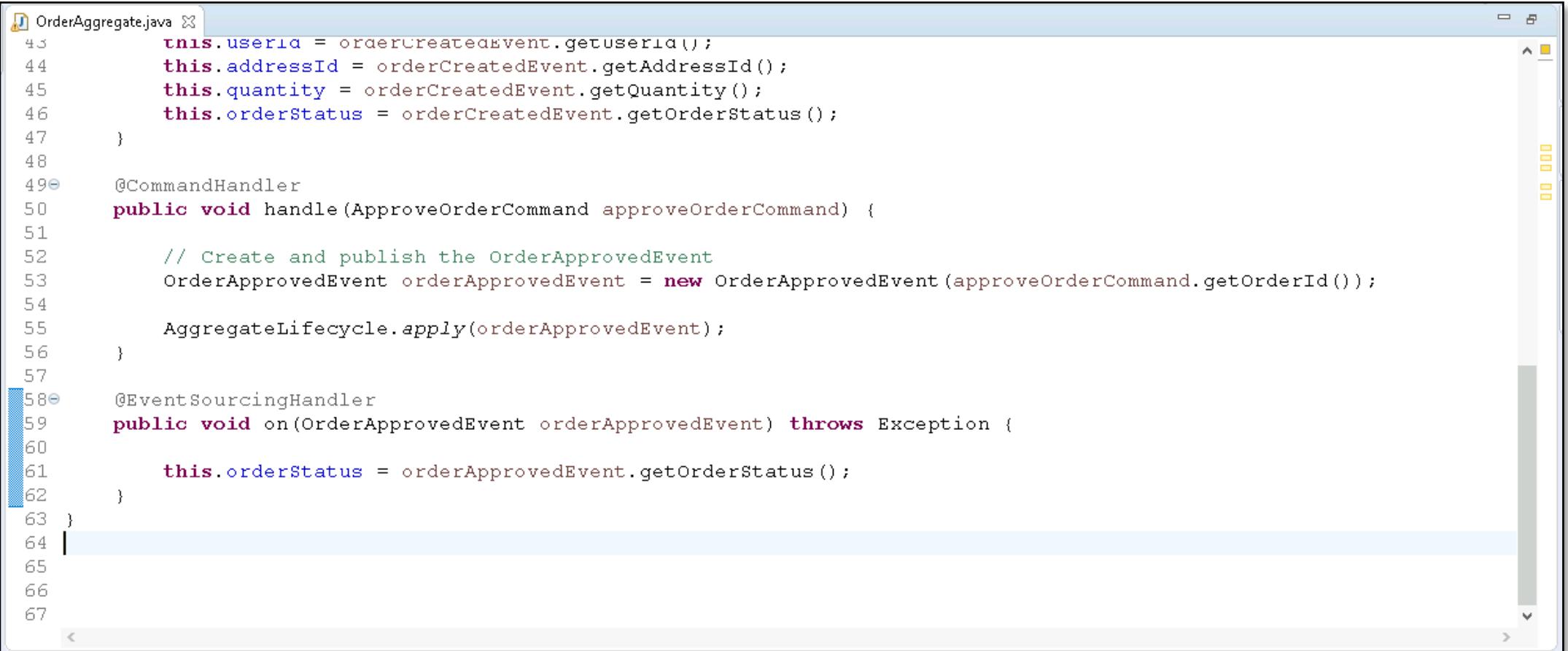
- Title Bar:** OrderApprovedEvent.java
- Code Content:**

```
1 package com.mphasis.core.events;
2
3 import com.mphasis.core.model.OrderStatus;
4
5 import lombok.Value;
6
7 @Value
8 public class OrderApprovedEvent {
9
10     private final String orderId;
11     private final OrderStatus orderStatus = OrderStatus.APPROVED;
12 }
13
```
- Code Editor Features:** The code editor has standard features like scroll bars, tabs, and a status bar at the bottom.



## Create and Publish the OrderApprovedEvent

- Publish the OrderApprovedEvent Instance from OrderAggregate:

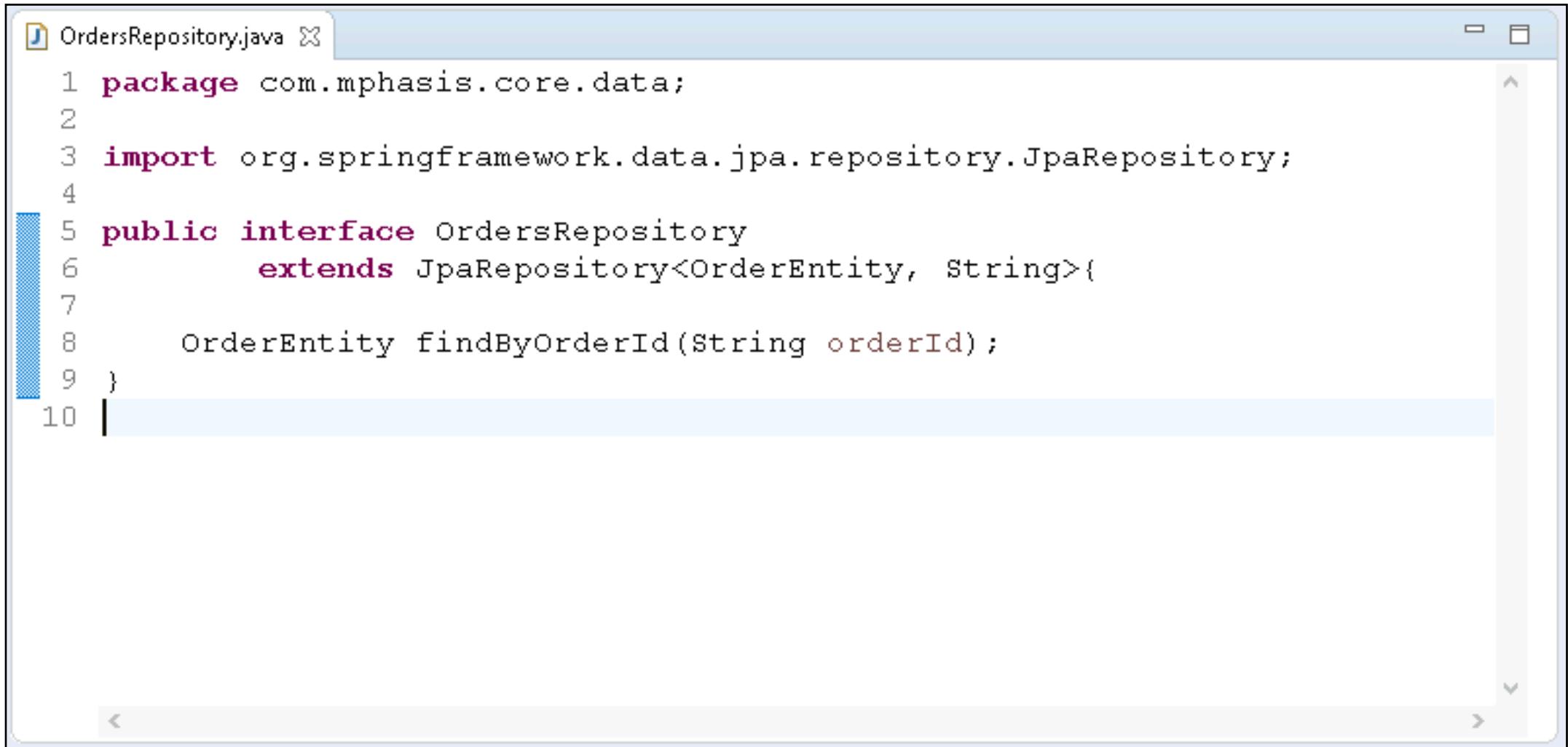


```
OrderAggregate.java X
43     this.userId = orderCreatedEvent.getUserId();
44     this.addressId = orderCreatedEvent.getAddressId();
45     this.quantity = orderCreatedEvent.getQuantity();
46     this.orderStatus = orderCreatedEvent.getOrderStatus();
47 }
48
49@CommandHandler
50 public void handle(ApproveOrderCommand approveOrderCommand) {
51
52     // Create and publish the OrderApprovedEvent
53     OrderApprovedEvent orderApprovedEvent = new OrderApprovedEvent(approveOrderCommand.getOrderId());
54
55     AggregateLifecycle.apply(orderApprovedEvent);
56 }
57
58@EventSourcingHandler
59 public void on(OrderApprovedEvent orderApprovedEvent) throws Exception {
60
61     this.orderStatus = orderApprovedEvent.getOrderStatus();
62 }
63
64
65
66
67 }
```



## Handle the OrderApprovedEvent and update Orders database

- Create the OrdersRepository class:



The screenshot shows a Java code editor window with a title bar "OrdersRepository.java". The code is as follows:

```
1 package com.mphasis.core.data;
2
3 import org.springframework.data.jpa.repository.JpaRepository;
4
5 public interface OrdersRepository
6     extends JpaRepository<OrderEntity, String>{
7
8     OrderEntity findByOrderId(String orderId);
9 }
10
```



## Handle the OrderApprovedEvent and update Orders database

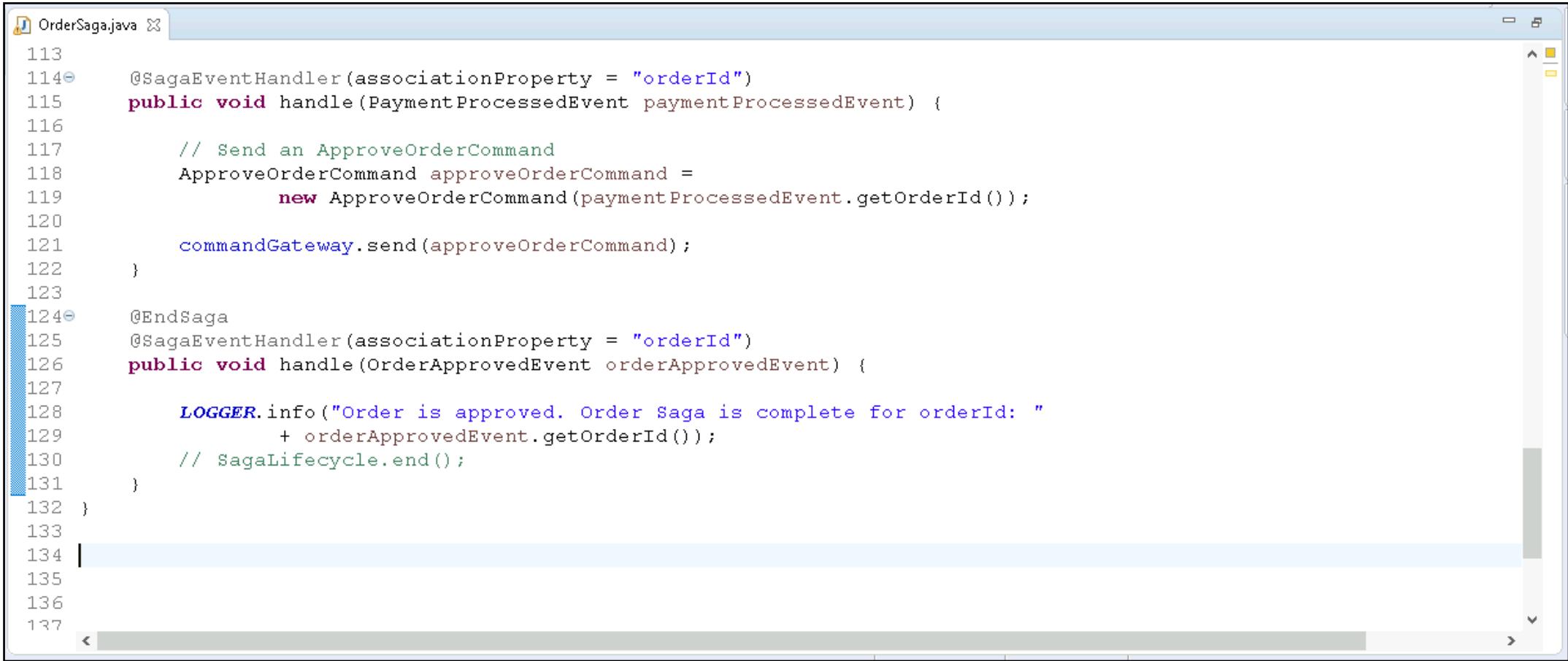
- Create an event handler for OrderApprovedEvent inside the OrderEventsHandler class:

```
OrderEventsHandler.java
1  public void on(OrderCreateEvent event) throws Exception {
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32 @EventHandler
33 public void on(OrderApprovedEvent orderApprovedEvent) throws Exception {
34
35
36
37
38
39
40
41
42
43
44
45
46 }
```



## Handle the OrderApprovedEvent in OrderSaga class

- Create a saga event handler for OrderApprovedEvent inside the OrderSaga class:



```
OrderSaga.java
113
114@SagaEventHandler(associationProperty = "orderId")
115public void handle(PaymentProcessedEvent paymentProcessedEvent) {
116
117    // Send an ApproveOrderCommand
118    ApproveOrderCommand approveOrderCommand =
119        new ApproveOrderCommand(paymentProcessedEvent.getOrderId());
120
121    commandGateway.send(approveOrderCommand);
122}
123
124@EndSaga
125@SagaEventHandler(associationProperty = "orderId")
126public void handle(OrderApprovedEvent orderApprovedEvent) {
127
128    LOGGER.info("Order is approved. Order Saga is complete for orderId: "
129                + orderApprovedEvent.getOrderId());
130    // SagaLifecycle.end();
131}
132}
133
134
135
136
137
```



## Trying how it works

1. Run the AxonServer using Docker command.
2. Ensure the Discovery Server (Eureka Server), Product Service, OrdersService, UsersService, and PaymentsService is running.
3. Ensure the ApiGateway is running.

# Send a POST request to Create Product

The screenshot shows the Postman application interface. At the top, there is a navigation bar with 'Home', 'Workspaces', 'Explore', a search bar 'Search Postman', and account options 'Sign In' and 'Create Account'. Below the navigation bar, there are two tabs: 'POST http://localhost:8082/p' and 'POST http://localhost:8082/o'. The main workspace displays a POST request to 'http://localhost:8082/products-service/products'. The 'Body' tab is selected, showing a JSON payload:

```
1 {  
2   ... "title": "iPad Pro",  
3   ... "price": 500,  
4   ... "quantity": 5  
5 }  
6
```

The 'Headers' tab shows 8 items. The 'Tests' and 'Settings' tabs are also visible. On the right side, there are buttons for 'Add to collection' and 'Send'. Below the request details, the response status is shown as 'Status: 200 OK' with a response time of '1190 ms' and a size of '152 B'. The response body is displayed in 'Pretty' format, showing the generated ID: 'e26ff077-45a1-46fa-a279-e80b362777f2'. There are tabs for 'Body', 'Cookies', 'Headers (3)', and 'Test Results'. The bottom of the interface includes a 'Console' tab and various window control buttons.

# Send a POST request to Create Order using productId

The screenshot shows the Postman application interface. At the top, there are navigation tabs: Home, Workspaces, Explore, a search bar labeled "Search Postman", and account options like "Sign In" and "Create Account". Below the header, there are two tabs: "POST http://localhost:8082/p" and "POST http://localhost:8082/o". The second tab is selected and shows the URL "http://localhost:8082/orders-service/orders". On the right side of this tab, there are buttons for "Add to collection" and "Send". The main body of the request window shows the method "POST" and the URL "http://localhost:8082/orders-service/orders". Below this, under the "Body" tab, the request body is defined as follows:

```
1 {  
2   "productId": "e26ff077-45a1-46fa-a279-e80b362777f2",  
3   "quantity": 1,  
4   "addressId": "afbb5881-a872-4d13-993c-faeb8350eea5"  
5 }
```

At the bottom of the request window, there are tabs for "Body", "Cookies", "Headers (3)", and "Test Results". The "Test Results" tab is currently active, showing the response status: "Status: 200 OK Time: 575 ms Size: 203 B" and a "Save Response" button. Below this, there are three tabs: "Pretty", "Raw", and "Visualize", with "Pretty" selected. The response body is displayed as:

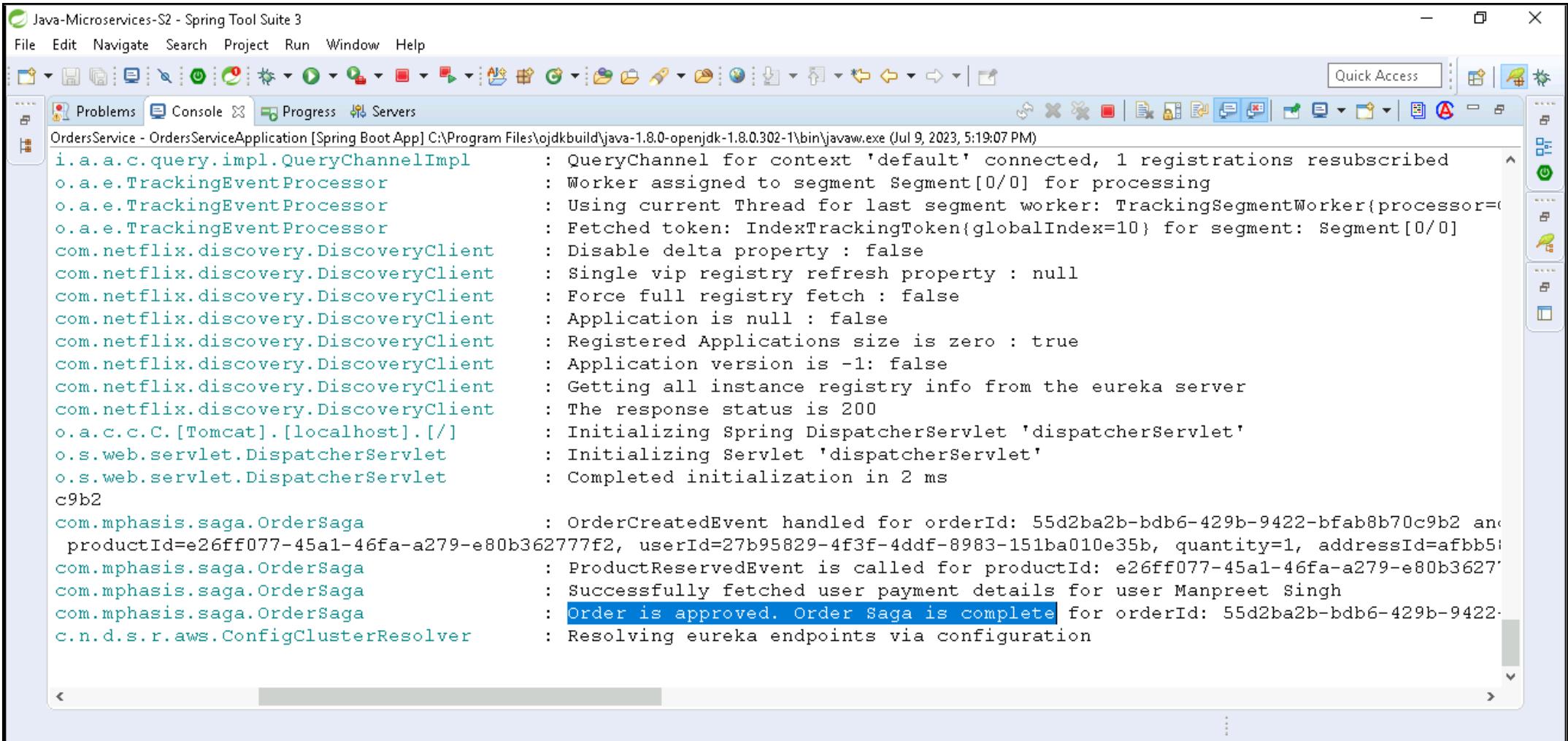
```
1 {  
2   "orderId": "55d2ba2b-bdb6-429b-9422-bfab8b70c9b2",  
3   "orderStatus": "CREATED",  
4   "message": ""  
5 }
```

At the very bottom of the interface, there are buttons for "Console" and other application controls.



## Review the OrderService Console

- Check the successful message on the OrderService console:



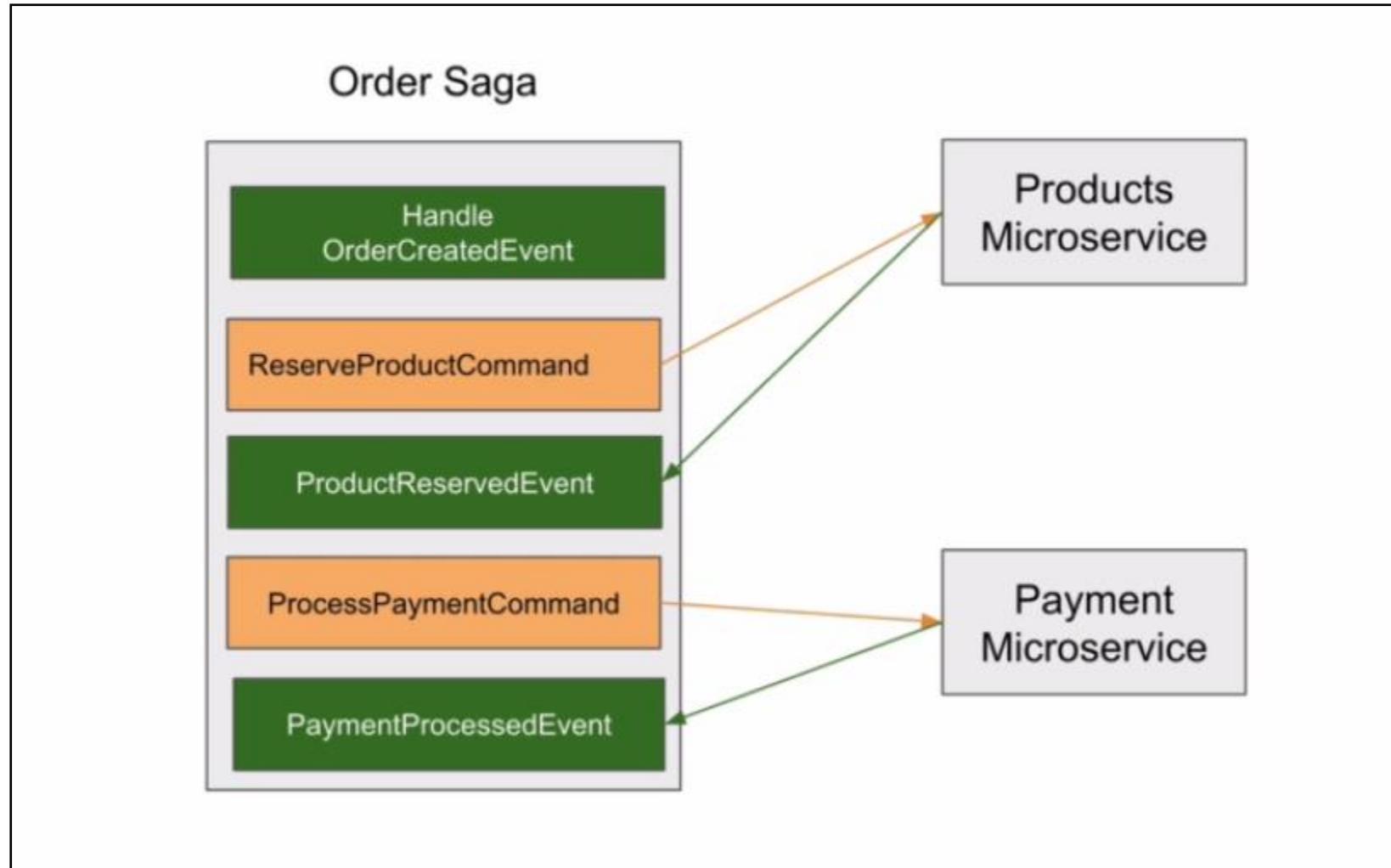
```
Java-Microservices-S2 - Spring Tool Suite 3
File Edit Navigate Search Project Run Window Help
Problems Console Progress Servers
OrdersService - OrdersServiceApplication [Spring Boot App] C:\Program Files\ojdkbuild\java-1.8.0-openjdk-1.8.0.302-1\bin\javaw.exe (Jul 9, 2023, 5:19:07 PM)
i.a.a.c.query.impl.QueryChannelImpl : QueryChannel for context 'default' connected, 1 registrations resubscribed
o.a.e.TrackingEventProcessor : Worker assigned to segment Segment[0/0] for processing
o.a.e.TrackingEventProcessor : Using current Thread for last segment worker: TrackingSegmentWorker{processor=(null)}
o.a.e.TrackingEventProcessor : Fetched token: IndexTrackingToken{globalIndex=10} for segment: Segment[0/0]
com.netflix.discovery.DiscoveryClient : Disable delta property : false
com.netflix.discovery.DiscoveryClient : Single vip registry refresh property : null
com.netflix.discovery.DiscoveryClient : Force full registry fetch : false
com.netflix.discovery.DiscoveryClient : Application is null : false
com.netflix.discovery.DiscoveryClient : Registered Applications size is zero : true
com.netflix.discovery.DiscoveryClient : Application version is -1: false
com.netflix.discovery.DiscoveryClient : Getting all instance registry info from the eureka server
com.netflix.discovery.DiscoveryClient : The response status is 200
o.a.c.c.C.[Tomcat].[localhost].[] : Initializing Spring DispatcherServlet 'dispatcherServlet'
o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
o.s.web.servlet.DispatcherServlet : Completed initialization in 2 ms
c9b2
com.mphasis.saga.OrderSaga : OrderCreatedEvent handled for orderId: 55d2ba2b-bdb6-429b-9422-bfab8b70c9b2 and
product Id=e26ff077-45a1-46fa-a279-e80b362777f2, userId=27b95829-4f3f-4ddf-8983-151ba010e35b, quantity=1, addressId=afbb51
com.mphasis.saga.OrderSaga : ProductReservedEvent is called for productId: e26ff077-45a1-46fa-a279-e80b3627
com.mphasis.saga.OrderSaga : Successfully fetched user payment details for user Manpreet Singh
com.mphasis.saga.OrderSaga : Order is approved. Order Saga is complete for orderId: 55d2ba2b-bdb6-429b-9422-
c.n.d.s.r.aws.ConfigClusterResolver : Resolving eureka endpoints via configuration
```



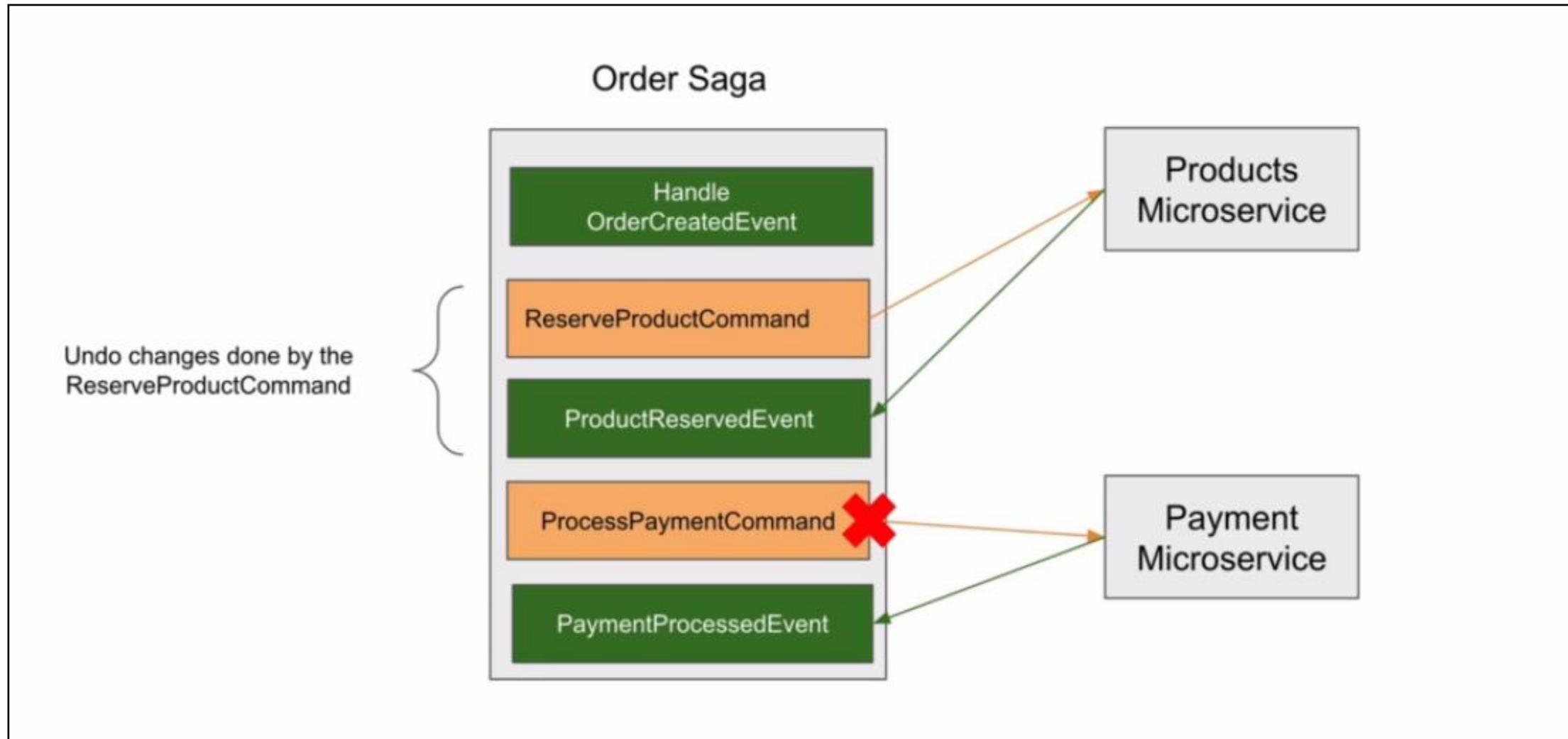
Day - 8

# Saga. Compensating Transactions.

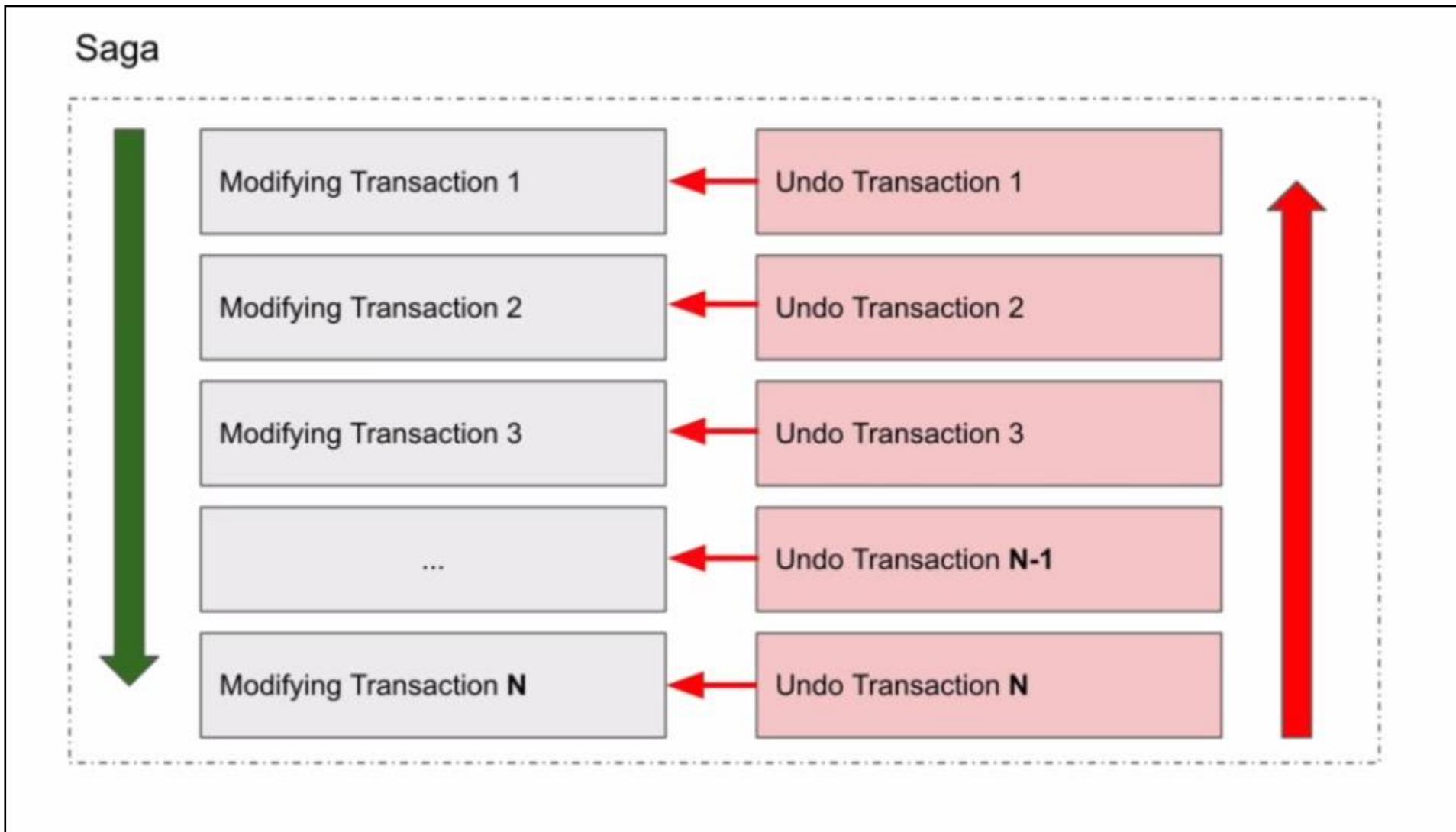
# Saga. Compensating Transactions.



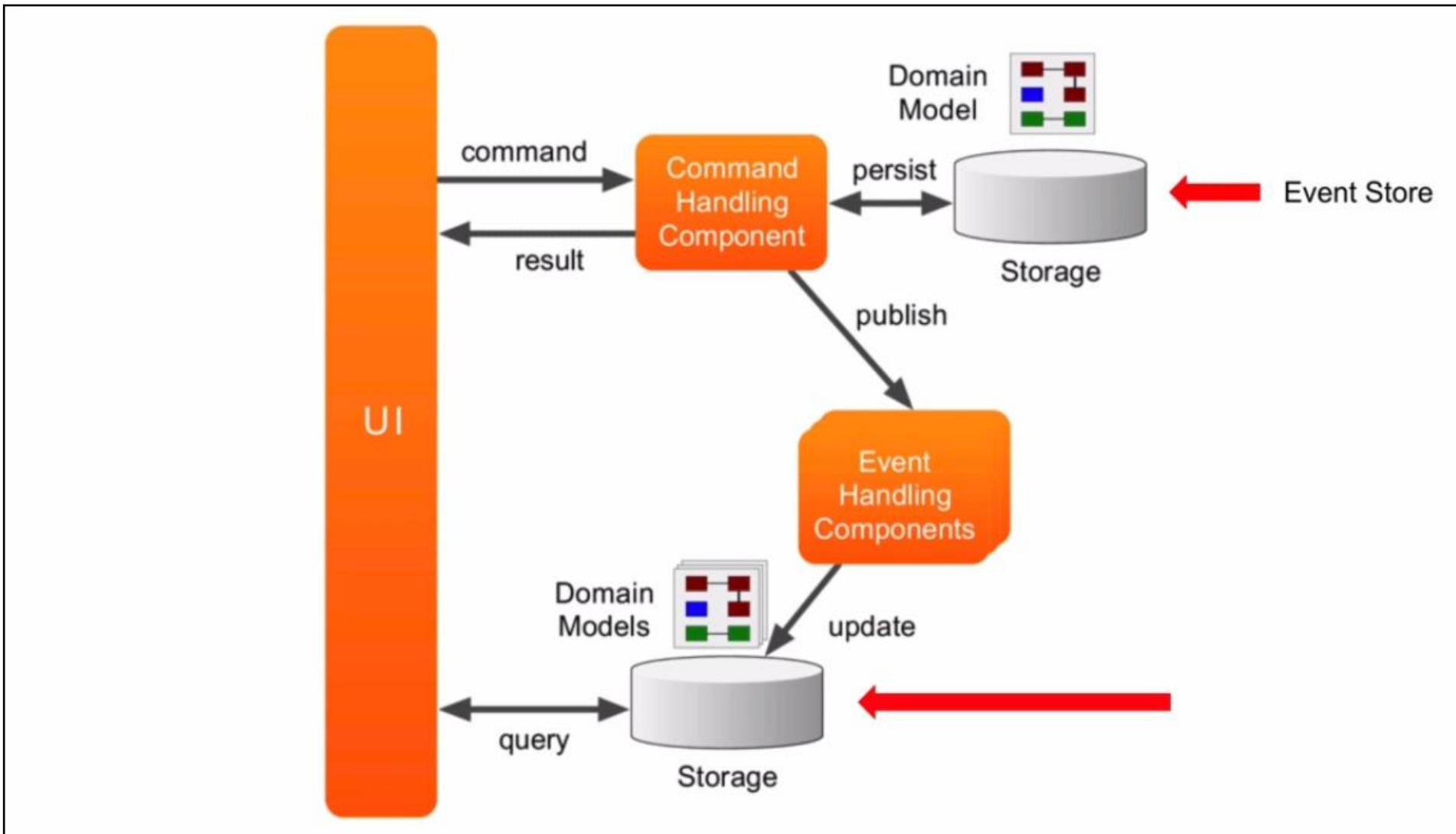
# Saga. Compensating Transactions.



# Saga. Compensating Transactions.



# Saga. Compensating Transactions.





## Create and Publish the CancelProductReservationCommand

- Let's create the CancelProductReservationCommand in the core project:

The screenshot shows a Java code editor window with the title "CancelProductReservationCommand.java". The code defines a command class with private final fields for product ID, quantity, order ID, user ID, and reason, along with a builder pattern.

```
1 package com.mphasis.core.commands;
2
3 import org.axonframework.modelling.command.TargetAggregateIdentifier;
4
5
6 @Data
7
8 @Builder
9
10 public class CancelProductReservationCommand {
11
12     @TargetAggregateIdentifier
13     private final String productId;
14
15     private final int quantity;
16     private final String orderId;
17     private final String userId;
18     private final String reason;
19 }
20
```



## Create and Publish the CancelProductReservationCommand

- Let's create a private method for cancelProductReservation in OrderSaga:

```
120④    private void cancelProductReservation(ProductReservedEvent productReservedEvent, String reason) {  
121  
122        CancelProductReservationCommand publishProductReservationCommand =  
123            CancelProductReservationCommand.builder()  
124                .orderId(productReservedEvent.getOrderId())  
125                .productId(productReservedEvent.getProductId())  
126                .quantity(productReservedEvent.getQuantity())  
127                .userId(productReservedEvent.getUserId())  
128                .reason(reason)  
129                .build();  
130  
131        commandGateway.send(publishProductReservationCommand);  
132    }
```



## Create and Publish the CancelProductReservationCommand

- Now publish from 4 places in the OrderSaga class:

```
70@SagaEventHandler(associationProperty = "orderId")
71public 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.class));
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    }
```



## Create and Publish the CancelProductReservationCommand

- Now publish from 4 places in the OrderSaga class:

```
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 }
```



## Handle the CancelProductReservationCommand in ProductService

- We will update the ProductAggregate class in ProductService to handle the CancelProductReservationCommand:

```
ProductAggregate.java
19
80     @CommandHandler
81     public void handle(CancelProductReservationCommand cancelProductReservationCommand) {
82
83     }
84 }
85
```



## Create and publish the ProductReservationCancelledEvent

- Let's create the ProductReservationCancelledEvent class in the core project:

The screenshot shows a Java code editor window with the following details:

- Title Bar:** The title bar displays the file name "ProductReservationCancelledEvent.java".
- Code Content:** The code is a Java class definition named "ProductReservationCancelledEvent". It includes imports for the package and Lombok annotations (@Data and @Builder). The class has five private final fields: productID, quantity, orderId, userId, and reason.
- Annotations:** Lines 6 and 7 are annotated with green arrows pointing upwards, indicating these are annotations for the class definition.
- Editor UI:** The window includes standard Java IDE features like scroll bars, a status bar at the bottom, and a toolbar area above the code editor.



## Create and publish the ProductReservationCancelledEvent

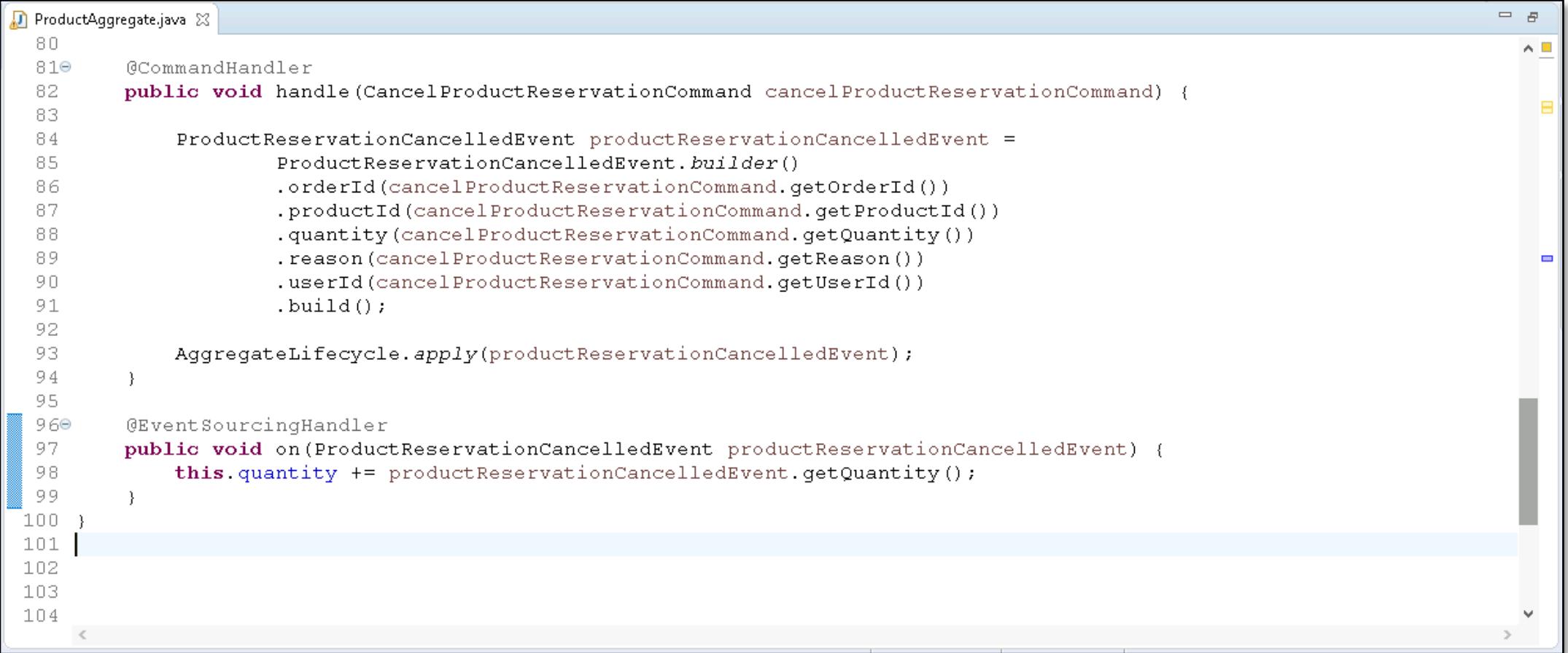
- Let's publish the ProductReservationCancelledEvent from the ProductAggregate class:

```
ProductAggregate.java
80
81  @CommandHandler
82  public void handle(CancelProductReservationCommand cancelProductReservationCommand) {
83
84      ProductReservationCancelledEvent productReservationCancelledEvent =
85          ProductReservationCancelledEvent.builder()
86              .orderId(cancelProductReservationCommand.getOrderId())
87              .productId(cancelProductReservationCommand.getProductId())
88              .quantity(cancelProductReservationCommand.getQuantity())
89              .reason(cancelProductReservationCommand.getReason())
90              .userId(cancelProductReservationCommand.getUserId())
91              .build();
92
93      AggregateLifecycle.apply(productReservationCancelledEvent);
94  }
95 }
```



## Handle the ProductReservationCancelledEvent

- Let's create the eventsourcinghandler in the ProductAggregate class:



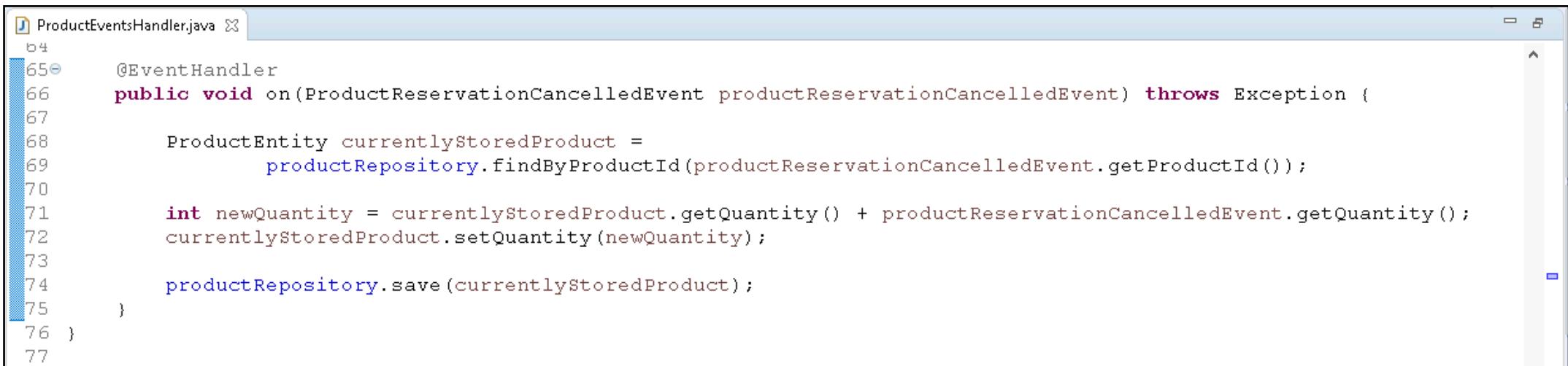
The screenshot shows a Java code editor window with the file `ProductAggregate.java` open. The code implements the Command Handler pattern for handling a `CancelProductReservationCommand` and the Event Sourcing pattern for handling a `ProductReservationCancelledEvent`.

```
ProductAggregate.java
80
81@CommandHandler
82public void handle(CancelProductReservationCommand cancelProductReservationCommand) {
83
84    ProductReservationCancelledEvent productReservationCancelledEvent =
85        ProductReservationCancelledEvent.builder()
86        .orderId(cancelProductReservationCommand.getOrderId())
87        .productId(cancelProductReservationCommand.getProductId())
88        .quantity(cancelProductReservationCommand.getQuantity())
89        .reason(cancelProductReservationCommand.getReason())
90        .userId(cancelProductReservationCommand.getUserId())
91        .build();
92
93    AggregateLifecycle.apply(productReservationCancelledEvent);
94}
95
96@EventSourcingHandler
97public void on(ProductReservationCancelledEvent productReservationCancelledEvent) {
98    this.quantity += productReservationCancelledEvent.getQuantity();
99}
100}
101
102
103
104
```



## Handle the ProductReservationCancelledEvent

- Let's create the eventhandler in the ProductEventsHandler class:



```
ProductEventsHandler.java
64
65 @EventHandler
66 public void on(ProductReservationCancelledEvent productReservationCancelledEvent) throws Exception {
67
68     ProductEntity currentlyStoredProduct =
69         productRepository.findById(productReservationCancelledEvent.getProductId());
70
71     int newQuantity = currentlyStoredProduct.getQuantity() + productReservationCancelledEvent.getQuantity();
72     currentlyStoredProduct.setQuantity(newQuantity);
73
74     productRepository.save(currentlyStoredProduct);
75 }
76
77 }
```



## Handle the ProductReservationCancelledEvent

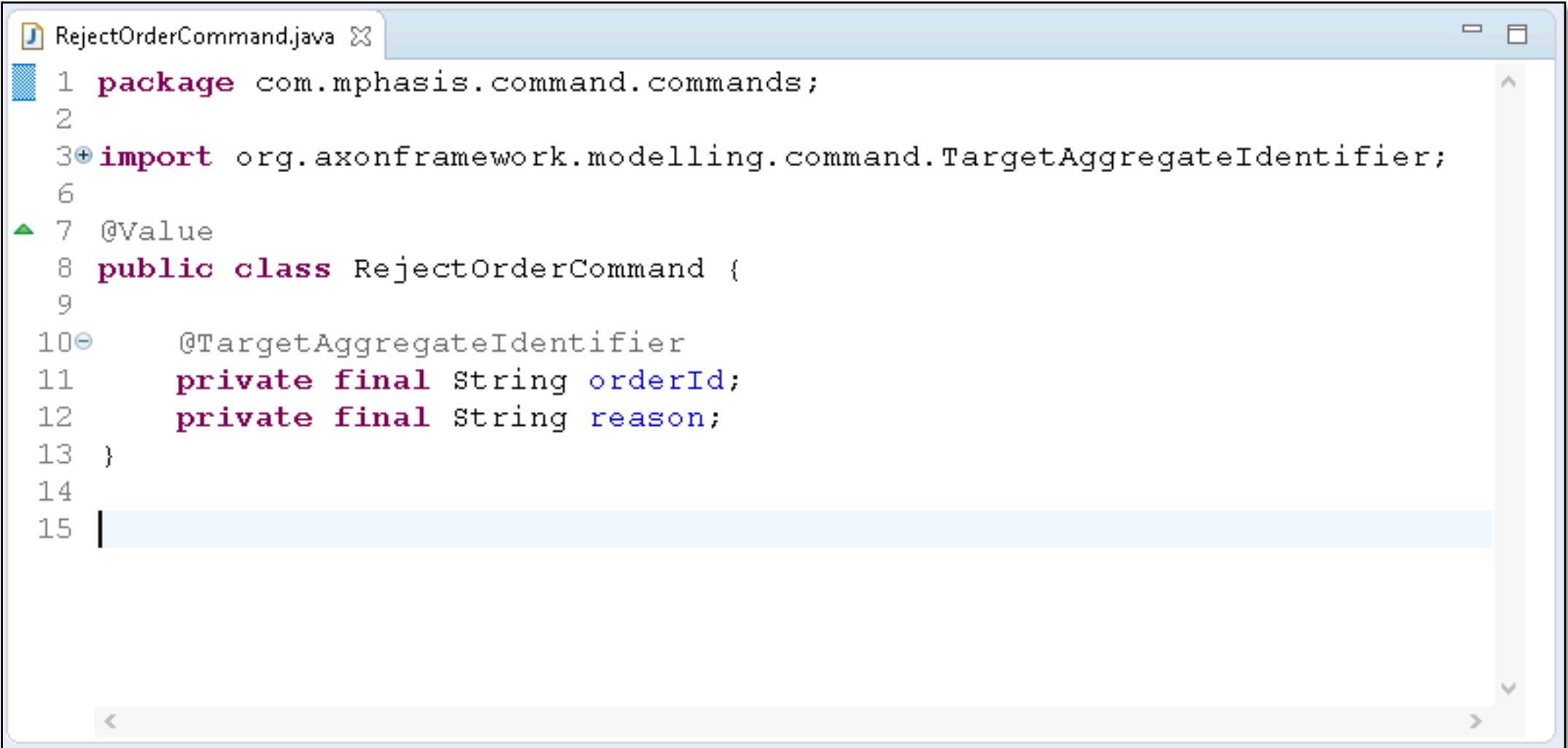
- Let's create the sageeventhandler in the OrderSaga class:

```
OrderSaga.java
144
145@EndSaga
146 @SagaEventHandler(associationProperty = "orderId")
147 public void handle(OrderApprovedEvent orderApprovedEvent) {
148
149     LOGGER.info("Order is approved. Order Saga is complete for orderId: "
150                 + orderApprovedEvent.getOrderId());
151     // SagaLifecycle.end();
152 }
153
154@SagaEventHandler(associationProperty = "orderId")
155 public void handle(ProductReservationCancelledEvent productReservationCancelledEvent) {
156
157     // Create and send a RejectOrderCommand
158 }
159
160
```



## Create and Publish the RejectOrderCommand

- Let's create RejectOrderCommand in local commands package of OrdersService:



The screenshot shows a Java code editor window with the file 'RejectOrderCommand.java' open. The code defines a class 'RejectOrderCommand' with private final fields for target aggregate identifier, order ID, and reason. The code editor interface includes tabs, status bars, and scroll bars.

```
1 package com.mphasis.command.commands;
2
3+import org.axonframework.modelling.command.TargetAggregateIdentifier;
4
5
6
7 @Value
8 public class RejectOrderCommand {
9
10@TargetAggregateIdentifier
11    private final String orderId;
12    private final String reason;
13 }
14
15 |
```



## Create and Publish the RejectOrderCommand

- Let's go to OrderSaga class and will publish the RejectOrderCommand:

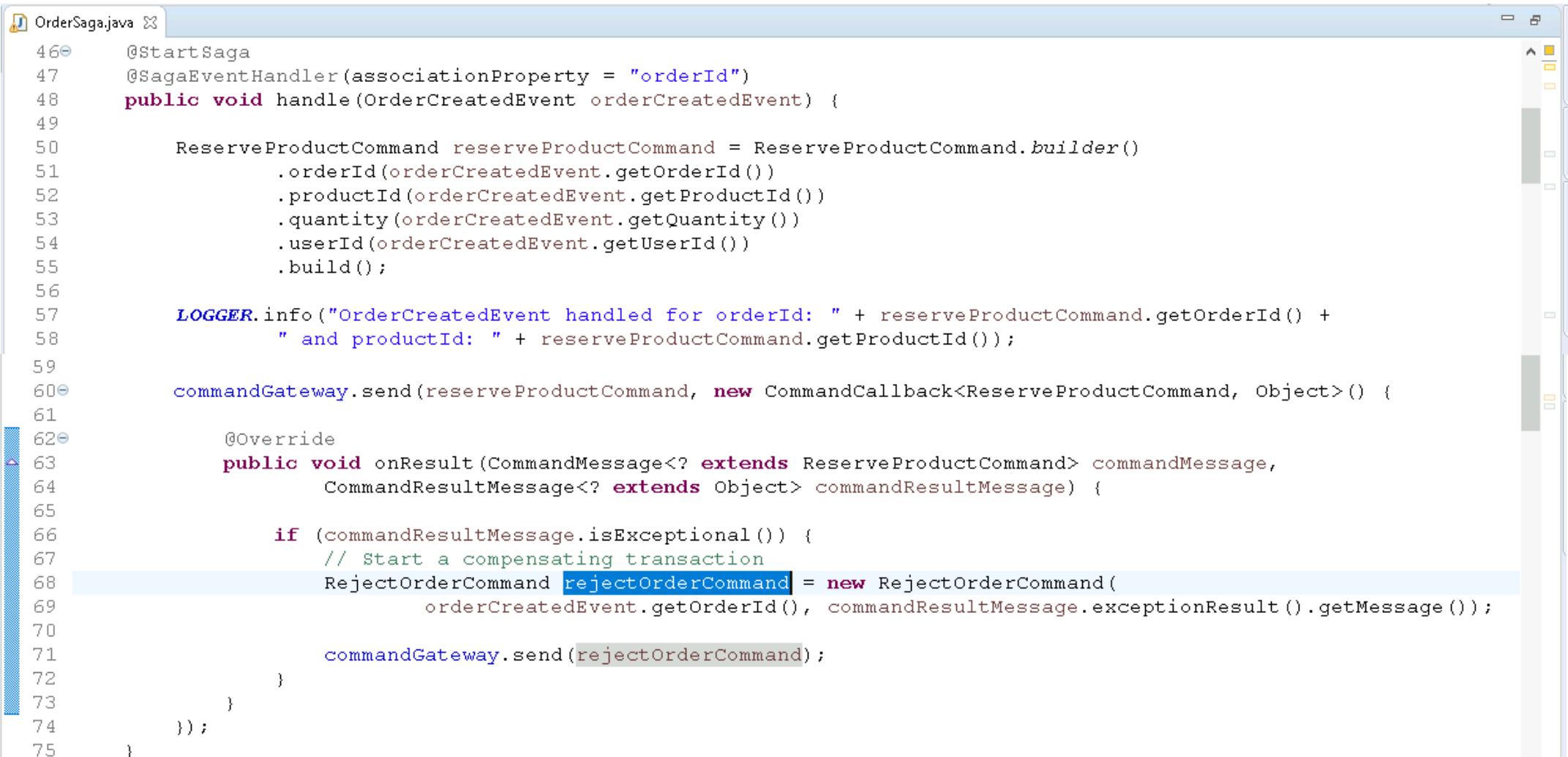
The screenshot shows a Java code editor window with the title "OrderSaga.java". The code is annotated with line numbers from 154 to 165. The code itself is as follows:

```
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
```



## Create and Publish the RejectOrderCommand

- Also publish the RejectOrderCommand from OrderSaga class if order is not created:



```
OrderSaga.java
46  @StartSaga
47  @SagaEventHandler(associationProperty = "orderId")
48  public void handle(OrderCreatedEvent orderCreatedEvent) {
49
50      ReserveProductCommand reserveProductCommand = ReserveProductCommand.builder()
51          .orderId(orderCreatedEvent.getOrderId())
52          .productId(orderCreatedEvent.getProductId())
53          .quantity(orderCreatedEvent.getQuantity())
54          .userId(orderCreatedEvent.getUserId())
55          .build();
56
57      LOGGER.info("OrderCreatedEvent handled for orderId: " + reserveProductCommand.getOrderId() +
58                  " and productId: " + reserveProductCommand.getProductId());
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  }
```



## Handle the RejectOrderCommand on the query side

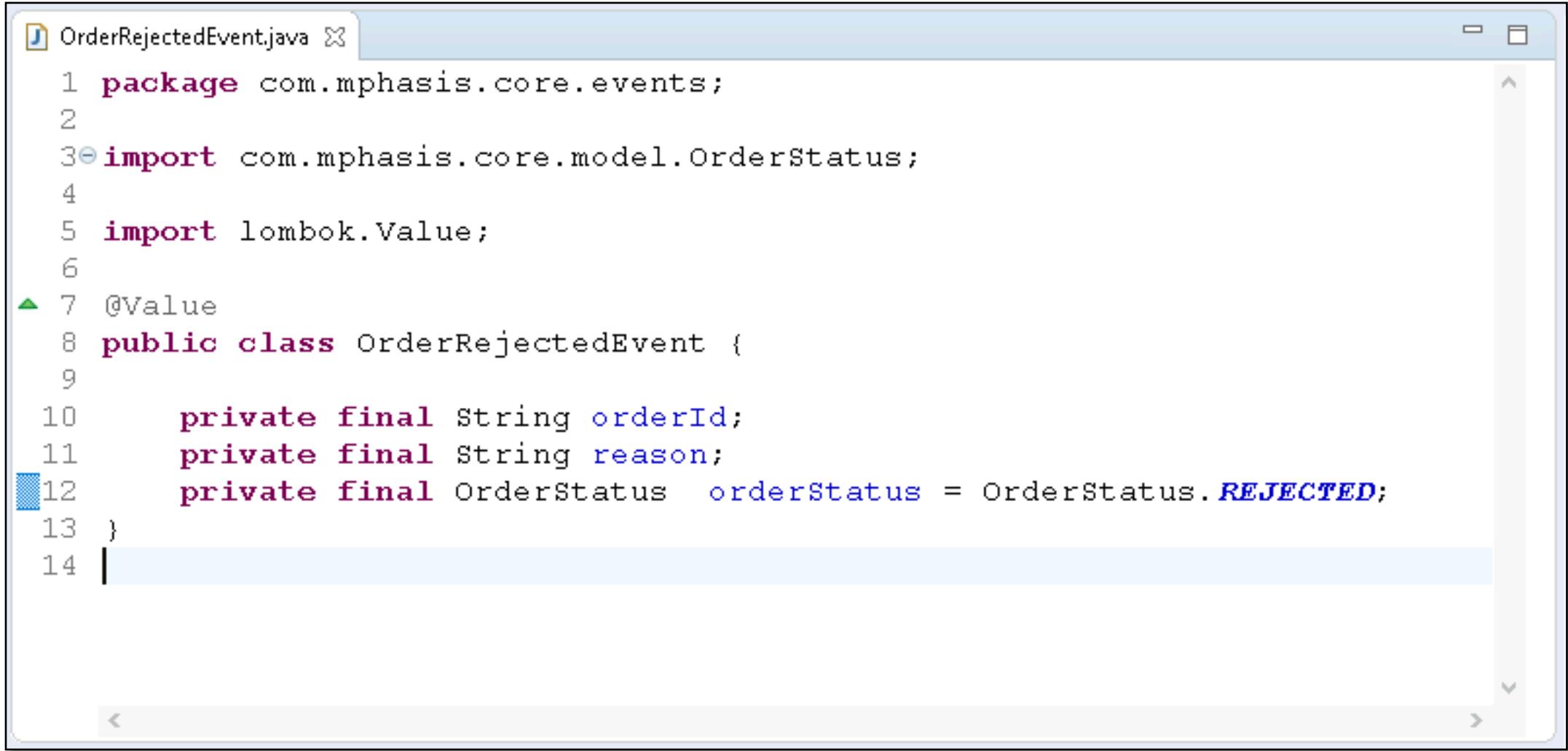
- Let's create CommandHandler in OrderAggregate class:

```
OrderAggregate.java
49
50     @CommandHandler
51     public void handle(ApproveOrderCommand approveOrderCommand) {
52
53         // Create and publish the OrderApprovedEvent
54         OrderApprovedEvent orderApprovedEvent = new OrderApprovedEvent(approveOrderCommand.getOrderId());
55
56         AggregateLifecycle.apply(orderApprovedEvent);
57     }
58
59     @EventSourcingHandler
60     public void on(OrderApprovedEvent orderApprovedEvent) throws Exception {
61
62         this.orderStatus = orderApprovedEvent.getOrderStatus();
63     }
64
65     @CommandHandler
66     public void handle(RejectOrderCommand rejectOrderCommand) {
67
68     }
69 }
```



## Create, Publish, and Handle the OrderRejectedEvent

- Let's create the OrderRejectedEvent in the events package:



The screenshot shows a Java code editor window with the file `OrderRejectedEvent.java` open. The code defines a class `OrderRejectedEvent` with private final fields `orderId`, `reason`, and `orderStatus` set to `REJECTED`. The `orderStatus` field is annotated with `@Value` from the Lombok library.

```
1 package com.mphasis.core.events;
2
3 import com.mphasis.core.model.OrderStatus;
4
5 import lombok.Value;
6
7 @Value
8 public class OrderRejectedEvent {
9
10     private final String orderId;
11     private final String reason;
12     private final OrderStatus orderStatus = OrderStatus.REJECTED;
13 }
14 |
```



## Create, Publish, and Handle the OrderRejectedEvent

- Let's publish the OrderRejectedEvent in the OrderAggregate class:

```
OrderAggregate.java X
66@CommandHandler
67 public void handle(RejectOrderCommand rejectOrderCommand) {
68
69     OrderRejectedEvent orderRejectedEvent = new OrderRejectedEvent(
70         rejectOrderCommand.getOrderId(), rejectOrderCommand.getReason());
71
72     AggregateLifecycle.apply(orderRejectedEvent);
73 }
74
75@EventSourcingHandler
76 public void on(OrderRejectedEvent orderRejectedEvent) throws Exception {
77
78     this.orderStatus = orderRejectedEvent.getOrderStatus();
79 }
80
```



## Create, Publish, and Handle the OrderRejectedEvent

- Let's handle the OrderRejectedEvent in the OrderEventsHandler class:

```
OrderEventsHandler.java
33@EventHandler
34 public void on(OrderApprovedEvent orderApprovedEvent) throws Exception {
35
36     OrderEntity orderEntity = ordersRepository.findById(orderApprovedEvent.getOrderId());
37
38     if(orderEntity == null) {
39         //TODO: Do something about id
40         return;
41     }
42
43     orderEntity.setOrderStatus(orderApprovedEvent.getOrderStatus());
44
45     ordersRepository.save(orderEntity);
46 }
47
48@EventHandler
49 public void on(OrderRejectedEvent orderRejectedEvent) throws Exception {
50
51     OrderEntity orderEntity = ordersRepository.findById(orderRejectedEvent.getOrderId());
52     orderEntity.setOrderStatus(orderRejectedEvent.getOrderStatus());
53     ordersRepository.save(orderEntity);
54 }
55 }
```



## Create, Publish, and Handle the OrderRejectedEvent

- Let's handle the OrderRejectedEvent in the OrderSaga class:

The screenshot shows a Java code editor window with the title bar "OrderSaga.java". The code is annotated with line numbers from 155 to 172. It defines two methods: a saga event handler for a ProductReservationCancelledEvent and an end saga event handler for an OrderRejectedEvent. The first method creates and sends a RejectOrderCommand. The second method logs a success message.

```
155
156 @SagaEventHandler(associationProperty = "orderId")
157 public void handle(ProductReservationCancelledEvent productReservationCancelledEvent) {
158
159     // Create and send a RejectOrderCommand
160     RejectOrderCommand rejectOrderCommand = new RejectOrderCommand(
161         productReservationCancelledEvent.getOrderId(), productReservationCancelledEvent.getReason());
162
163     commandGateway.send(rejectOrderCommand);
164 }
165
166 @EndSaga
167 @SagaEventHandler(associationProperty = "orderId")
168 public void handle(OrderRejectedEvent orderRejectedEvent) {
169
170     LOGGER.info("Successfully rejected order with id " + orderRejectedEvent.getOrderId());
171 }
172 }
```



## Trying how it works

1. Run the AxonServer using Docker command.
2. Ensure the Discovery Server (Eureka Server), Product Service, OrdersService, UsersService, and PaymentsService is running.
3. Ensure the ApiGateway is running.

# Send a POST request to Create Product

The screenshot shows the Postman application interface. At the top, there is a navigation bar with 'Home', 'Workspaces', 'Explore', a search bar 'Search Postman', and account options 'Sign In' and 'Create Account'. Below the navigation bar, a header bar displays two POST requests: 'POST http://localhost:8082/p...' and 'POST http://localhost:8082/o...'. The main workspace shows a POST request to 'http://localhost:8082/products-service/products'. The 'Body' tab is selected, showing a JSON payload:

```
1 {  
2   "title": "iPad Pro2",  
3   "price": 500,  
4   "quantity": 5  
5 }  
6
```

The 'Body' tab also includes tabs for 'Pretty', 'Raw', 'Preview', 'Visualize', and 'Text'. Below the body, the response status is shown as 'Status: 200 OK Time: 864 ms Size: 152 B'. The bottom of the interface includes a 'Console' tab.

# Send a POST request to Create Order using productId

The screenshot shows the Postman application interface. At the top, there are navigation tabs: Home, Workspaces, Explore, a search bar labeled "Search Postman", and account options like "Sign In" and "Create Account". Below the header, there are two tabs: "POST http://localhost:8082/p" and "POST http://localhost:8082/o". The "o" tab is selected, showing the URL "http://localhost:8082/orders-service/orders". On the right side of this tab, there are buttons for "Add to collection" and "Send". The main body of the screen shows a "POST" method selected, pointing to "http://localhost:8082/orders-service/orders". Below this, under the "Body" tab, the "JSON" option is chosen, and the request body is displayed as follows:

```
1 {
2   "productId": "376b5347-a7d7-42a0-9d89-f645354b936b",
3   "quantity": 1,
4   "addressId": "afbb5881-a872-4d13-993c-faeb8350eea5"
5 }
```

At the bottom of the request body editor, there are tabs for "Body", "Cookies", "Headers (3)", and "Test Results". The "Test Results" tab is currently active, showing the response status: "Status: 200 OK", "Time: 340 ms", and "Size: 203 B". The response body is also displayed in JSON format:

```
1 {
2   "orderId": "f03d0394-9a7e-429f-86ff-102cdd857837",
3   "orderStatus": "CREATED",
4   "message": ""
5 }
```

At the very bottom of the interface, there is a "Console" tab.

# Products Table

The screenshot shows the H2 Console interface running in a browser window titled "Eureka". The URL is "Not secure | host.docker.internal:51127/h2-console/login.do?jsessionid=e0879461f64258ed954c616558e535a4". The left sidebar lists database objects: ASSOCIATION\_VALUE\_ENTRY, PRODUCTLOOKUP, PRODUCTS, SAGA\_ENTRY, TOKEN\_ENTRY, INFORMATION\_SCHEMA, Sequences, and Users. The main area contains a SQL statement editor with the query "SELECT \* FROM PRODUCTS" and its execution results.

SQL statement:

```
SELECT * FROM PRODUCTS;
```

Execution results:

PRODUCT_ID	PRICE	QUANTITY	TITLE
376b5347-a7d7-42a0-9d89-f645354b936b	500.00	4	iPad Pro2

(1 row, 1 ms)

Buttons: Run, Run Selected, Auto complete, Clear, SQL statement.

# Orders Table

The screenshot shows the H2 Console interface with three tabs: Eureka, H2 Console, and another H2 Console tab. The second H2 Console tab is active, displaying a database named 'orderdb'.

The left sidebar lists database objects:

- jdbc:h2:mem:orderdb
- + ASSOCIATION\_VALUE\_ENTRY
- + ORDERS
- + SAGA\_ENTRY
- + TOKEN\_ENTRY
- + INFORMATION\_SCHEMA
- + Sequences
- + Users
- (i) H2 2.1.214 (2022-06-13)

The main area contains the following SQL statement and its results:

```
SELECT * FROM ORDERS;
```

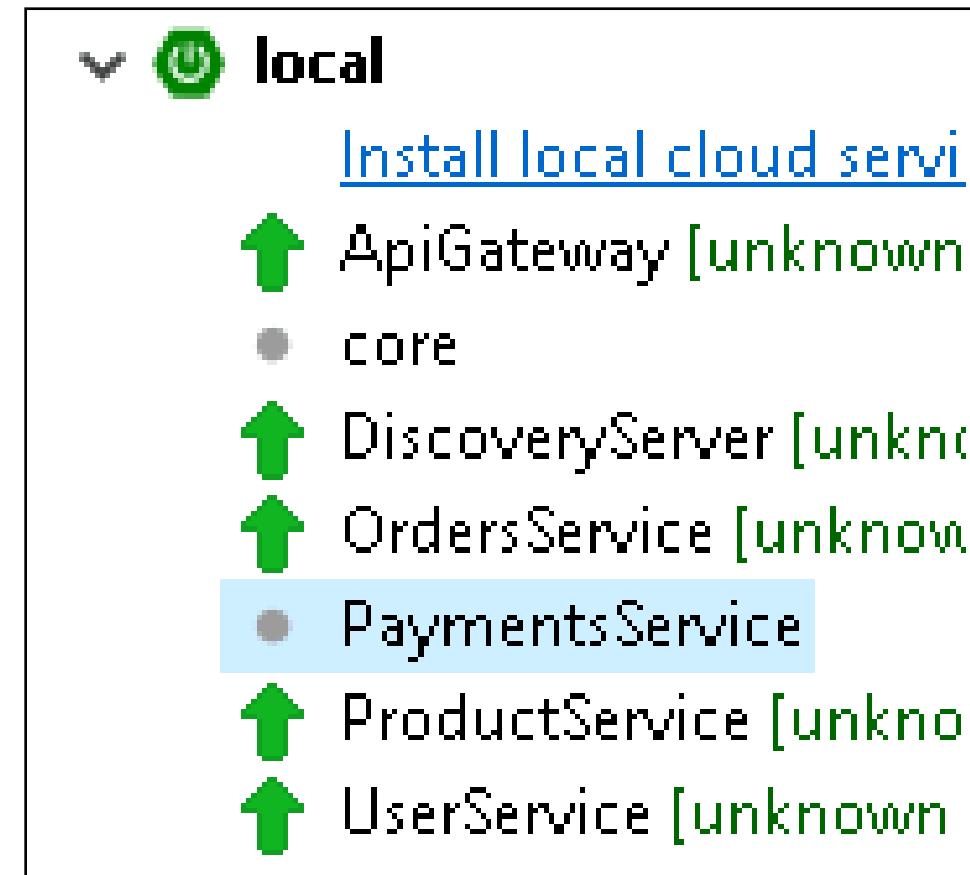
ORDER_ID	ADDRESS_ID	ORDER_STATUS	PRODUCT_ID	QUANTITY	USER_ID
f03d0394-9a7e-429f-86ff-102cdd857837	afbb5881-a872-4d13-993c-faeb8350eea5	APPROVED	376b5347-a7d7-42a0-9d89-f645354b936b	1	27b95829-4f3f-4ddf-8983-151ba010e3

(1 row, 1 ms)

An 'Edit' button is present below the table.



## Stop the PaymentsService





## Create Order using same productId - REJECTED

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

# Products Table

The screenshot shows the H2 Console interface running in a browser window titled "Eureka". The URL is "Not secure | host.docker.internal:51127/h2-console/login.do?jsessionid=e0879461f64258ed954c616558e535a4". The left sidebar lists database objects: ASSOCIATION\_VALUE\_ENTRY, PRODUCTLOOKUP, PRODUCTS, SAGA\_ENTRY, TOKEN\_ENTRY, INFORMATION\_SCHEMA, Sequences, and Users. The main area contains a SQL statement editor with the query "SELECT \* FROM PRODUCTS" and its execution results.

SQL statement:

```
SELECT * FROM PRODUCTS;
```

Execution results:

PRODUCT_ID	PRICE	QUANTITY	TITLE
376b5347-a7d7-42a0-9d89-f645354b936b	500.00	4	iPad Pro2

(1 row, 1 ms)

Buttons: Run, Run Selected, Auto complete, Clear, SQL statement.



## Recap of Day – 6,7,8

- Order Saga: Orchestration-based Saga
- Create a OrderSaga class
- Create the ReserveProductCommand class
- Publish the ReserveProductCommand Instance
- Handle the ReserveProductCommand in the ProductService
- Create the ProductReservedEvent class
- Publish the ProductReservedEvent Instance
- Updating Products projection
- Handle the ProductReservedEvent in Saga
- Assignment – Users Microservice



## Recap of Day – 6,7,8

- Handle the PaymentProcessedEvent
- Create and Publish the ProcessPaymentCommand
- Assignment – Payments Microservice
- Create and Publish the ApproveOrderCommand
- Handle the ApproveOrderCommand
- Create and Publish the OrderApprovedEvent
- Handle the OrderApprovedEvent and update Orders database
- Handle the OrderApprovedEvent in OrderSaga class



## Recap of Day – 6,7,8

- Saga - Compensating Transactions.
- Create and Publish the CancelProductReservationCommand
- Handle the CancelProductReservationCommand in ProductService
- Create and publish the ProductReservationCancelledEvent
- Handle the ProductReservationCancelledEvent in ProductAggregate
- Create and publish the RejectOrderCommand
- Handle the RejectOrderCommand in OrderAggregate



## Recap of overall Agenda

- Understanding CQRS and Event Sourcing Pattern
- Implementing CQRS with Event Sourcing with Axon Framework
- Distributed Transaction in Java Microservices using SAGA Pattern
- Introduce Docker and state its benefit over VM
- Understanding the Architecture of Docker and terminologies
- Describe what is Container in Docker, why to use it, and its scopes
- Deploy Microservice in Docker
- Deploy Microservice with H2 to AWS Fargate
- Implement Centralized Configuration Management with AWS Parameter Store
- Implement Auto Scaling and Load Balancing with AWS Fargate



## Additional Recommended Reading

- [Spring Microservices in Action](#)
- [Spring Boot Intermediate Microservices](#)



## THANK YOU

### About Mphasis

**Mphasis** (BSE: 526299; NSE: MPHASIS) applies next-generation technology to help enterprises transform businesses globally. Customer centricity is foundational to Mphasis and is reflected in the Mphasis' Front2Back™ Transformation approach. Front2Back™ uses the exponential power of cloud and cognitive to provide hyper-personalized ( $C=X^2C^2$ ) digital experience to clients and their end customers. Mphasis' Service Transformation approach helps 'shrink the core' through the application of digital technologies across legacy environments within an enterprise, enabling businesses to stay ahead in a changing world. Mphasis' core reference architectures and tools, speed and innovation with domain expertise and specialization are key to building strong relationships with marquee clients. Click [here](#) to know

### Important Confidentiality Notice

This document is the property of, and is proprietary to Mphasis, and identified as "Confidential". Those parties to whom it is distributed shall exercise the same degree of custody and care afforded their own such information. It is not to be disclosed, in whole or in part to any third parties, without the express written authorization of Mphasis. It is not to be duplicated or used, in whole or in part, for any purpose other than the evaluation of, and response to, Mphasis' proposal or bid, or the performance and execution of a contract awarded to Mphasis. This document will be returned to Mphasis upon request.



## Any Questions?

Manpreet.Bindra@mphasis.com

FOLLOW US IN THE LINK BELOW

@Mphasis

