Mphasis

The Next Applied

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

© Mphasis 2020 Proprietary and confidential information

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

© Mphasis 2020 Proprietary and confidential information

Day - 1

Day – 1 Agenda

- 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

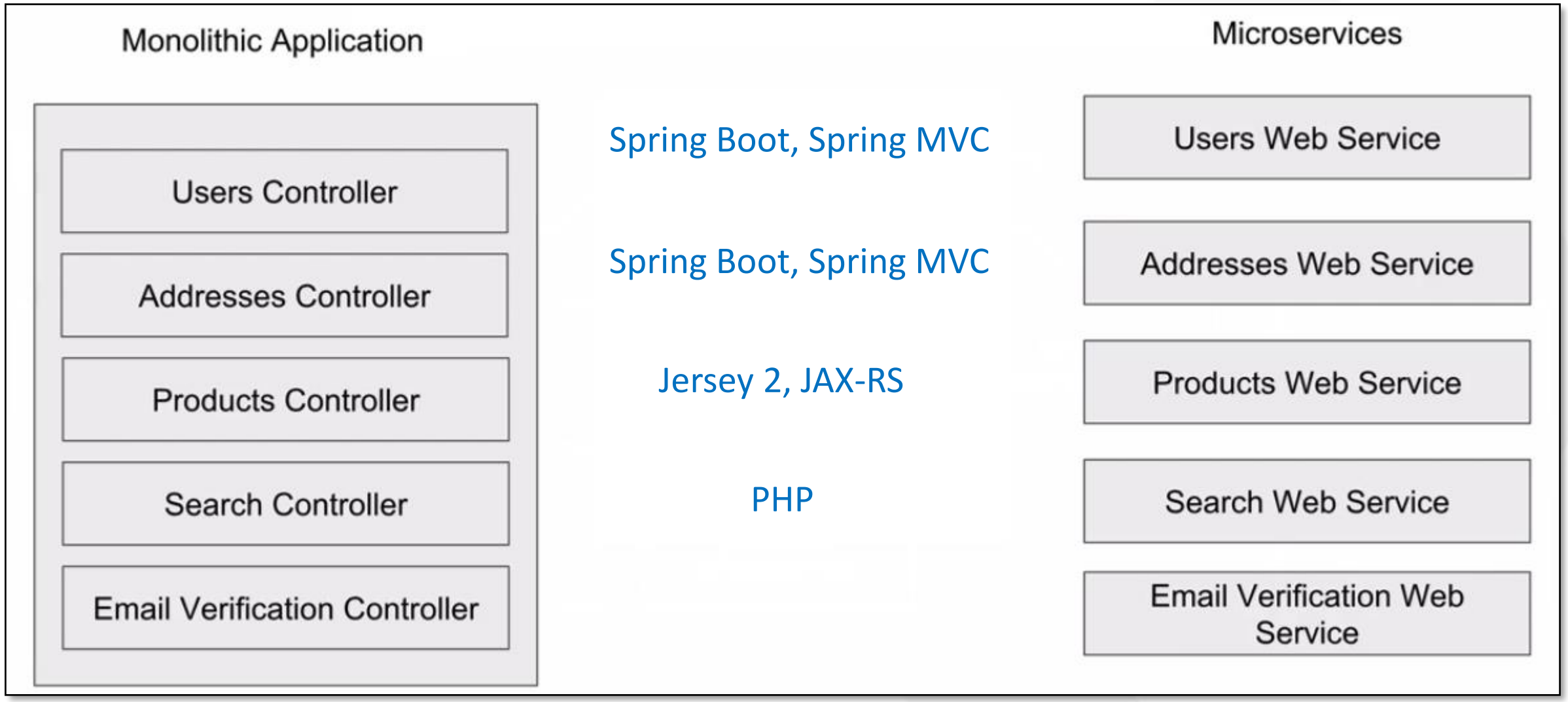
What are Microservice?

- 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.

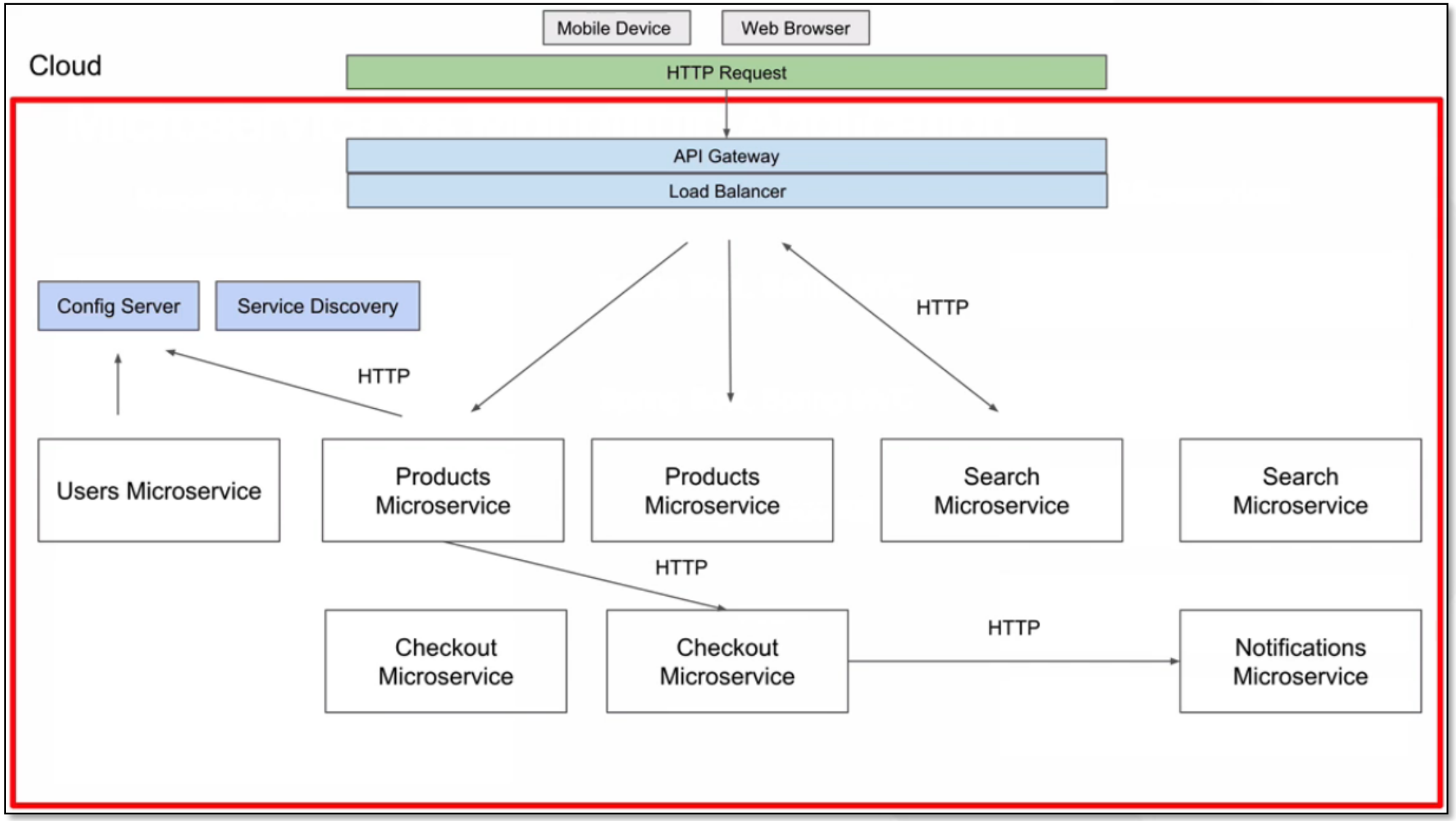
What are Microservice?

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

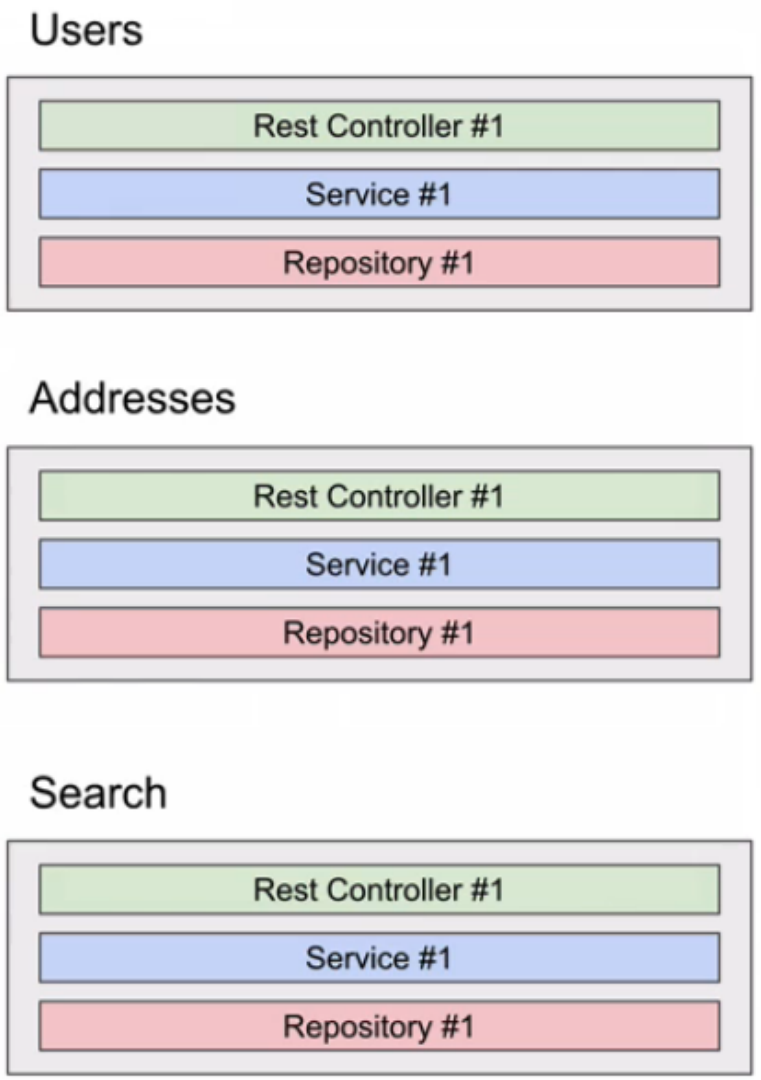
Microservice vs Monolithic Application



Application Diagram



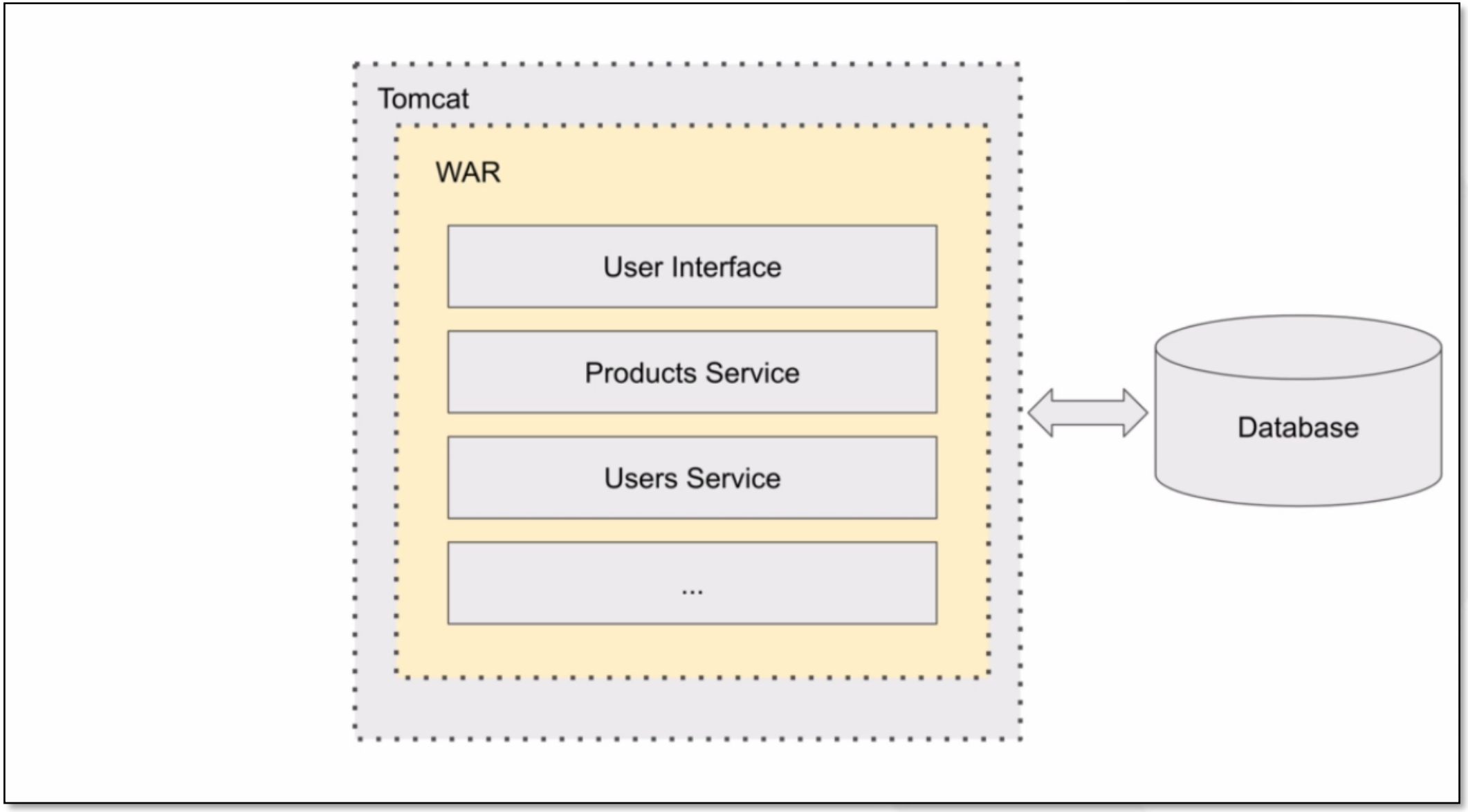
Rest Controller - Services



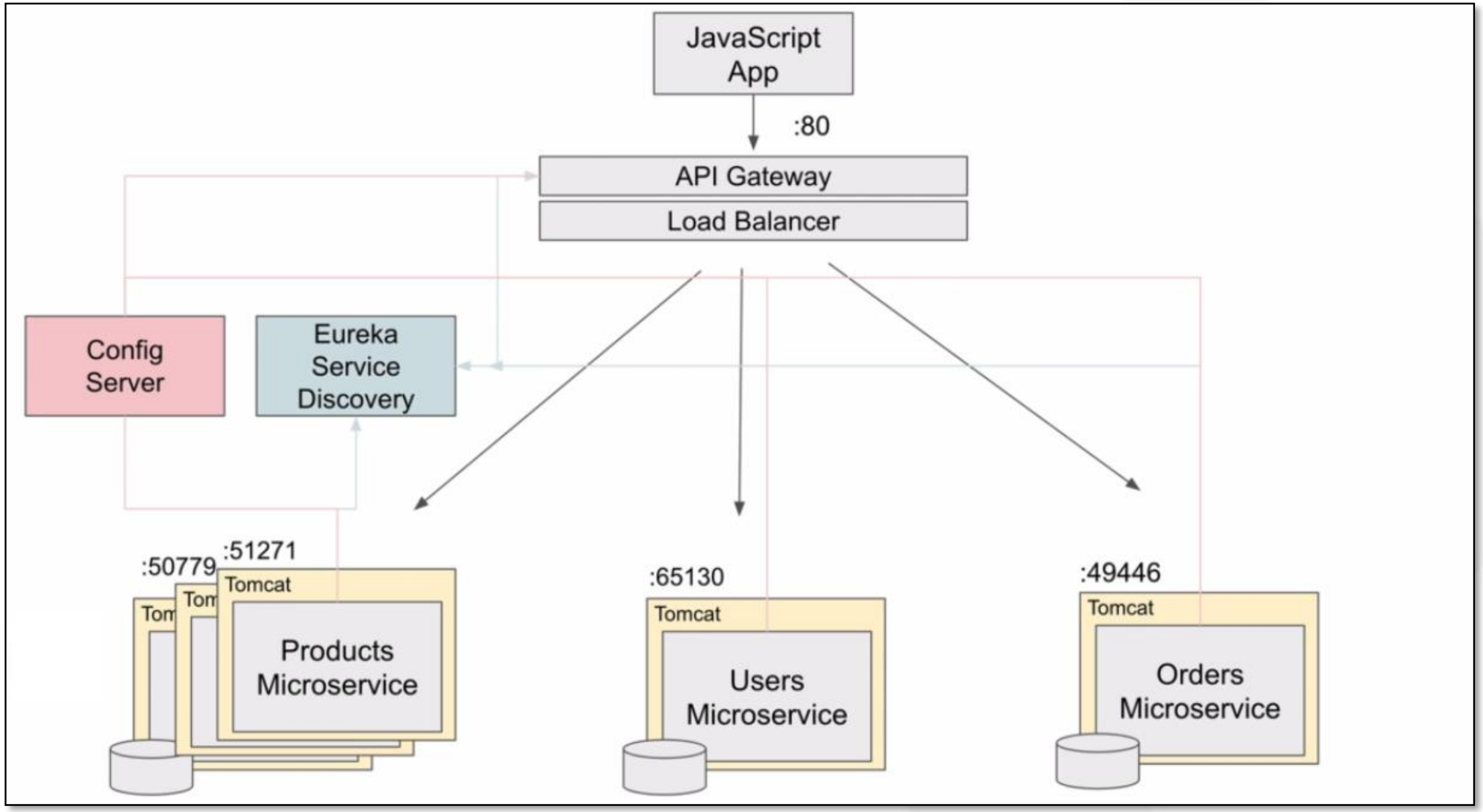
Day - 1

Microservices Architecture Overview

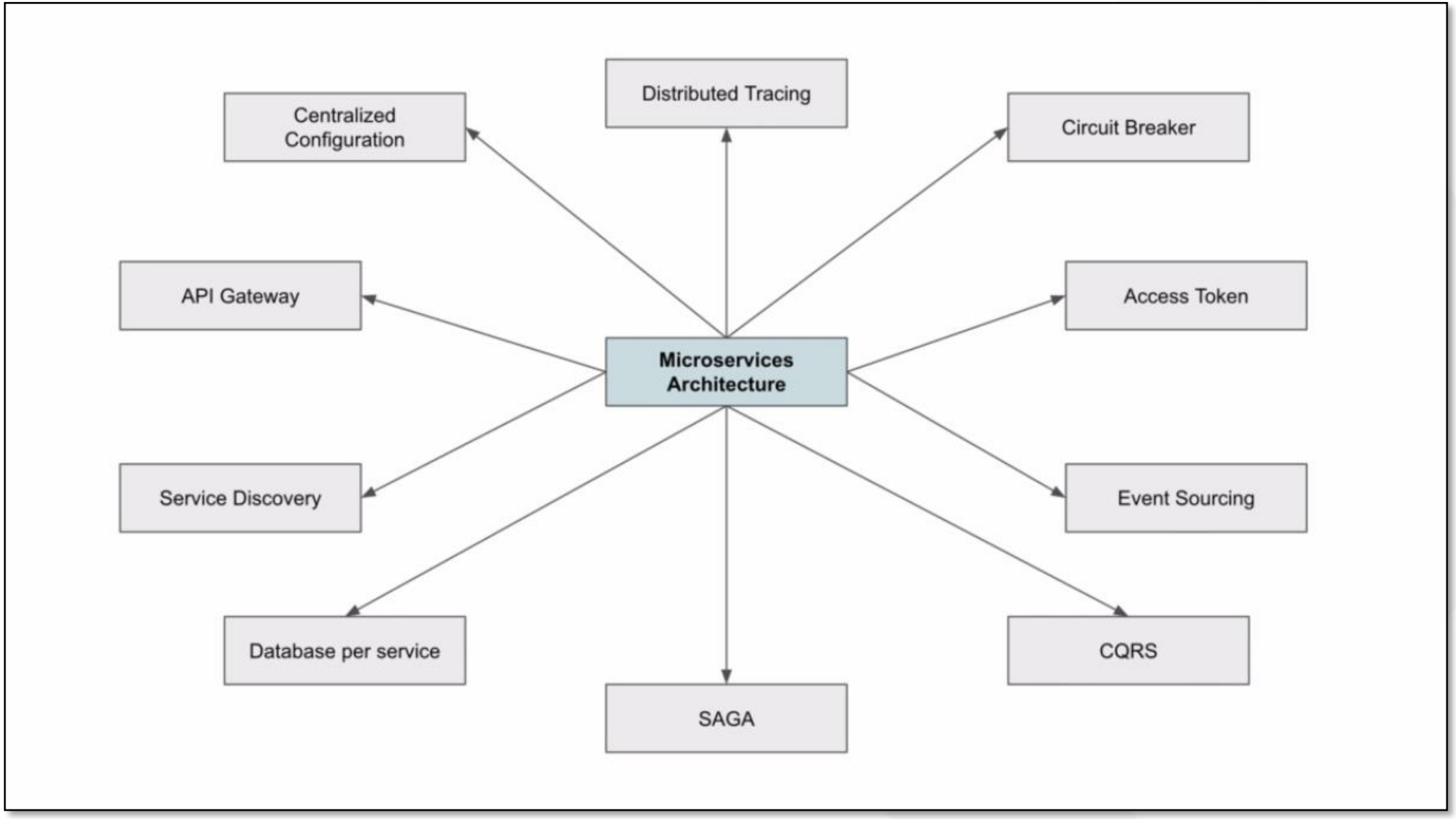
Monolithic Architecture



Microservices Architecture



Pattern Diagram



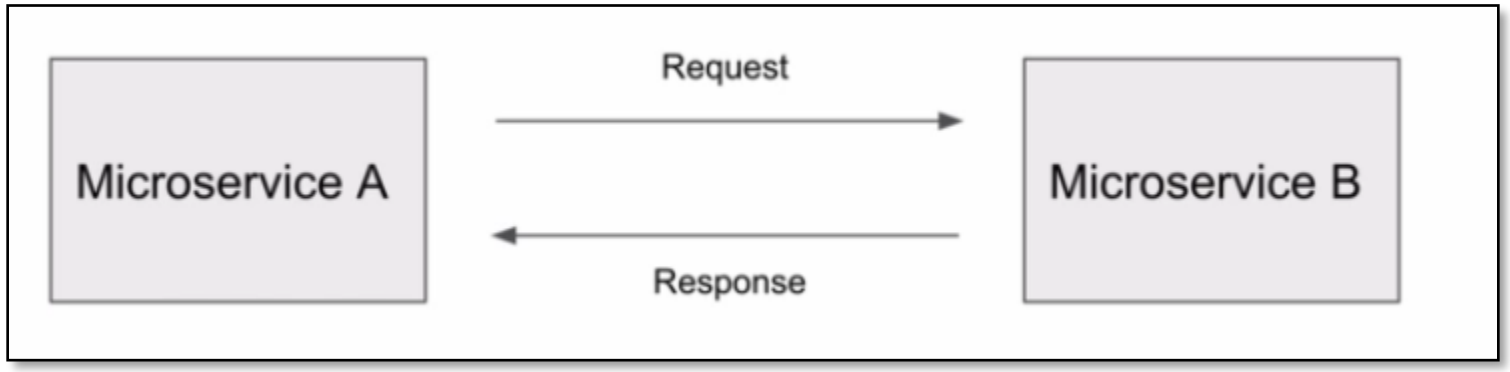
Day - 1

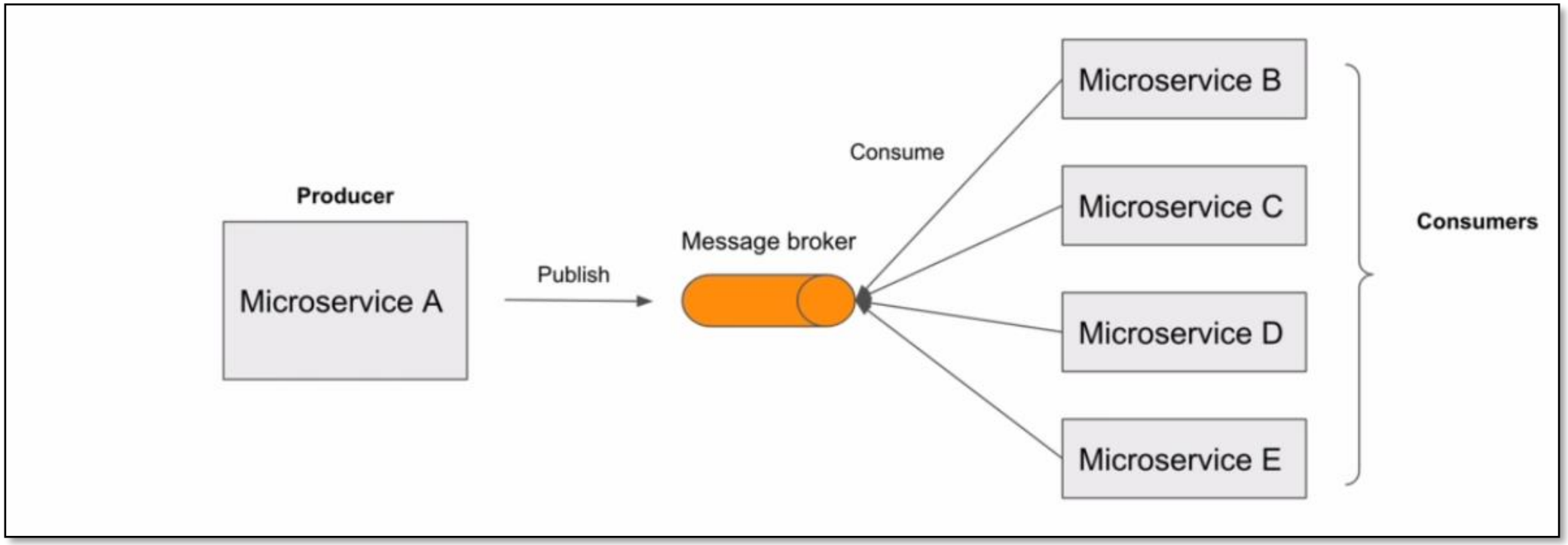
Event-Driven Microservices

Event-Driven Microservices

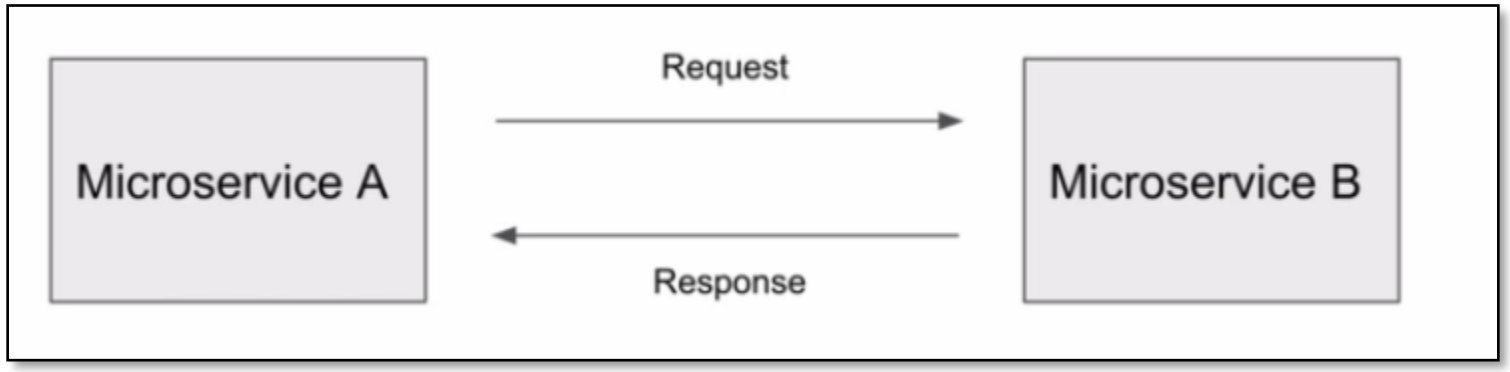
- It’s asynchronous.
- If your microservices are simple, follow request-response approach.
- If you don’t know how many microservices can be added/removed as consumer, we can prefer the event-driven microservices/message-driven microservices.

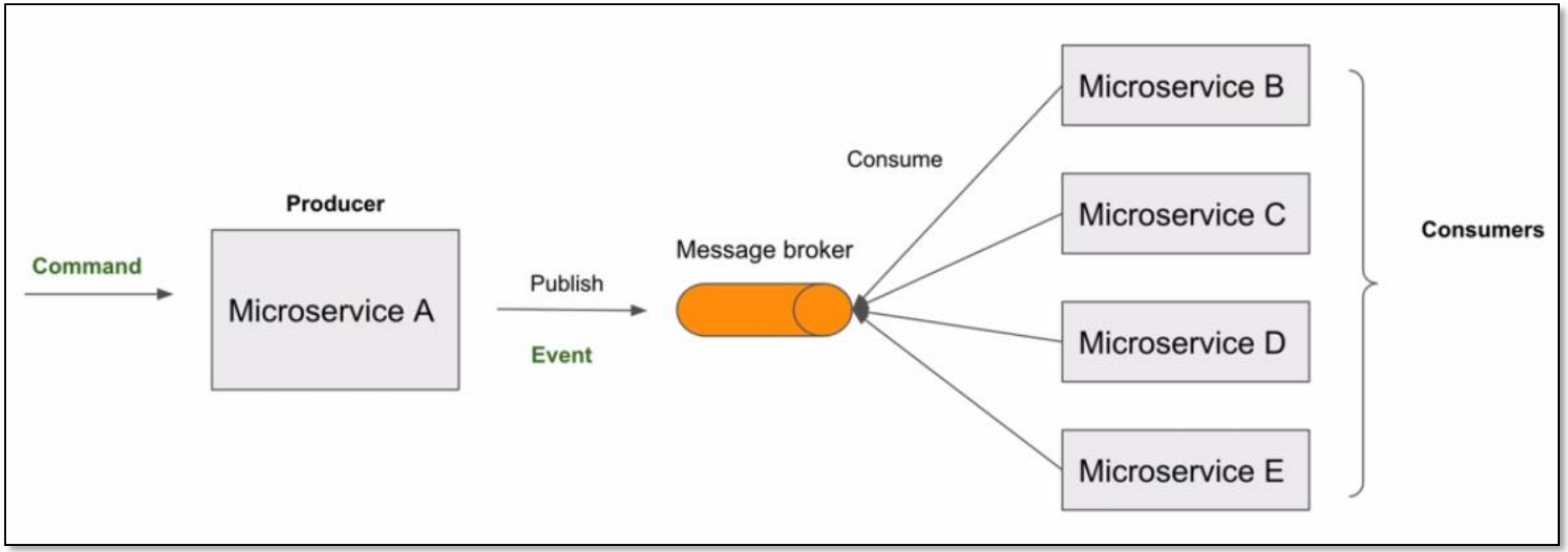
Event-Driven Microservices



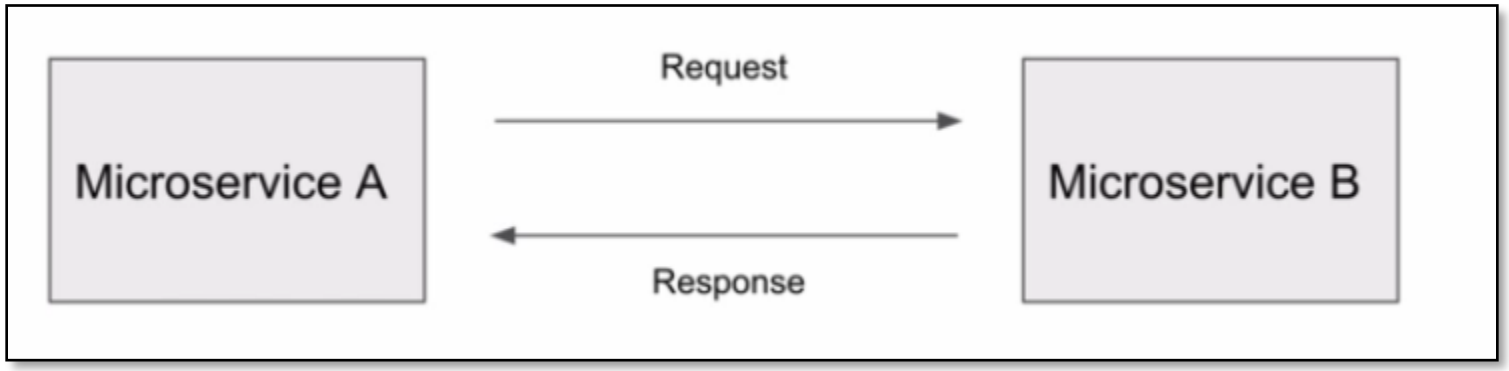


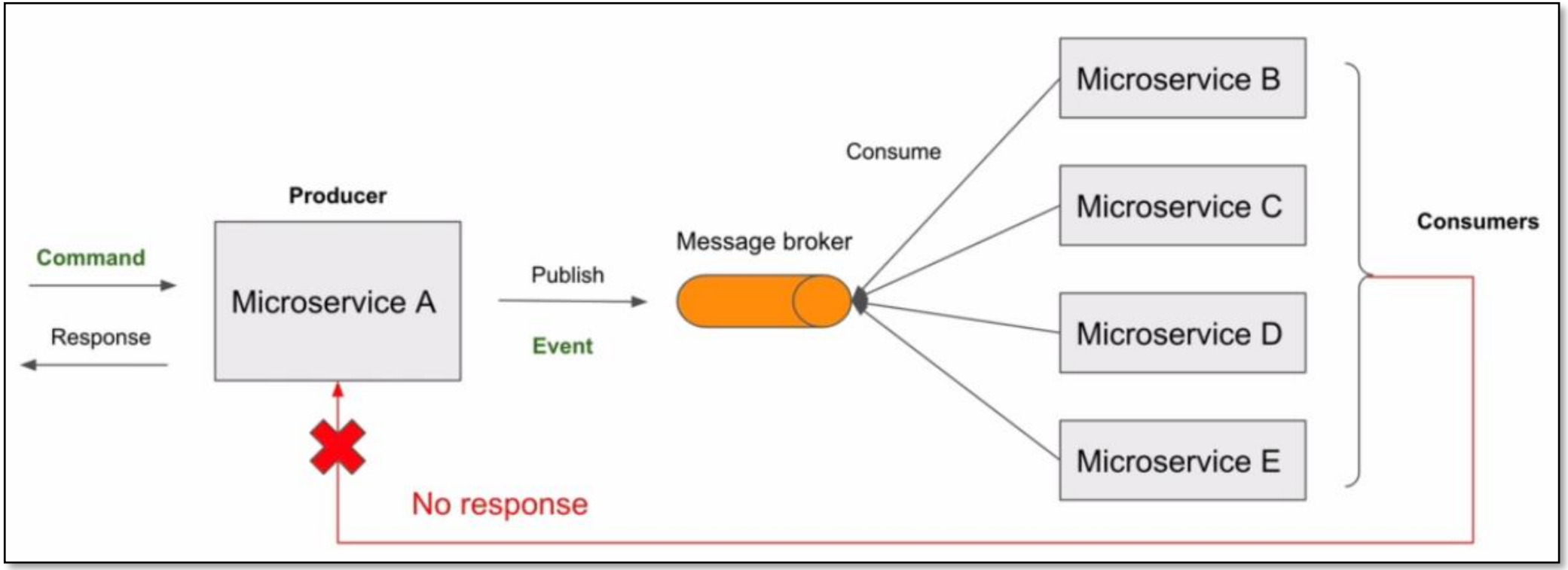
Event-Driven Microservices





Event-Driven Microservices





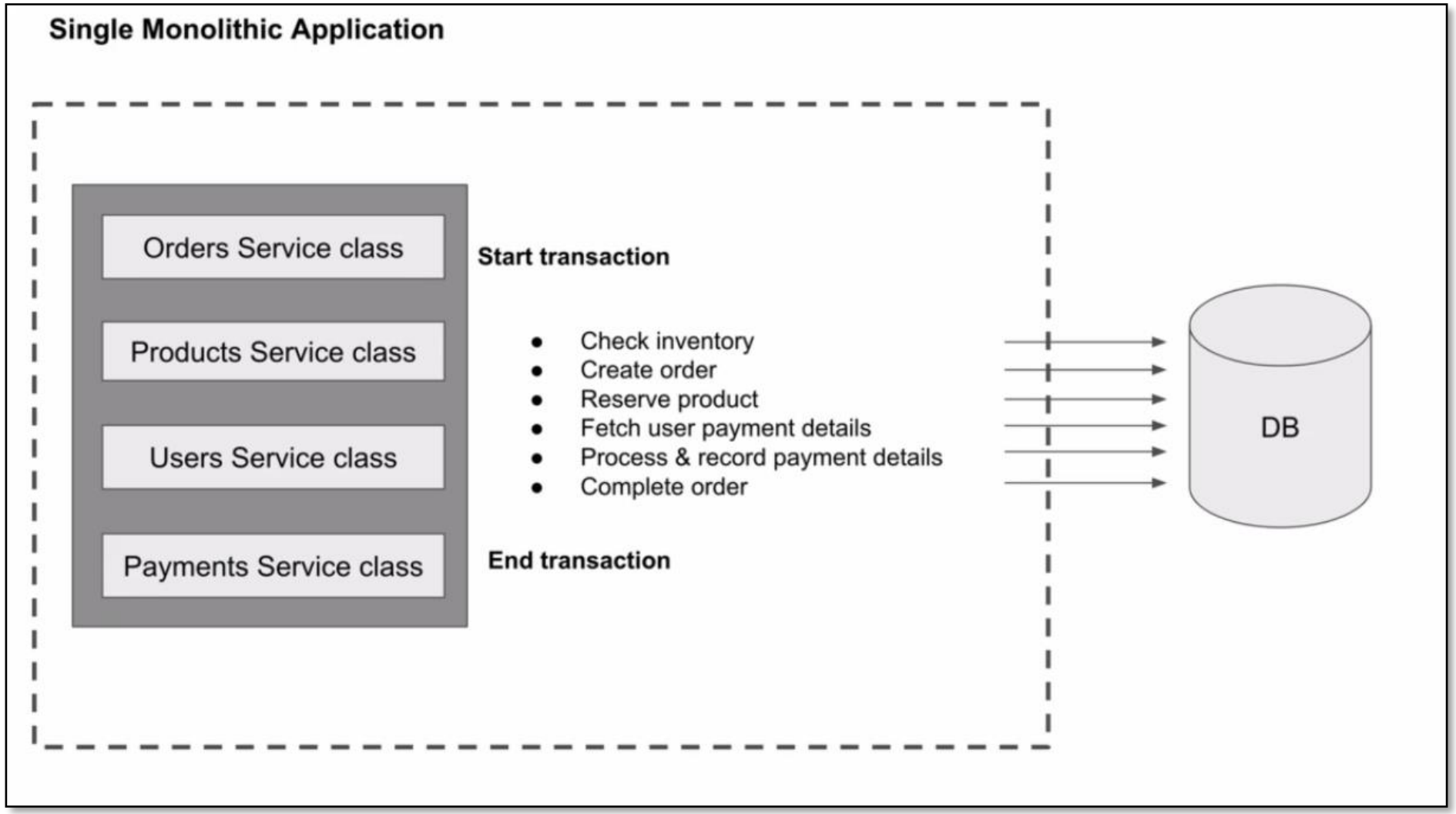
Day - 1

Transactions in Microservices

© Mphasis 2020 Proprietary and confidential information

Transactions in Microservices

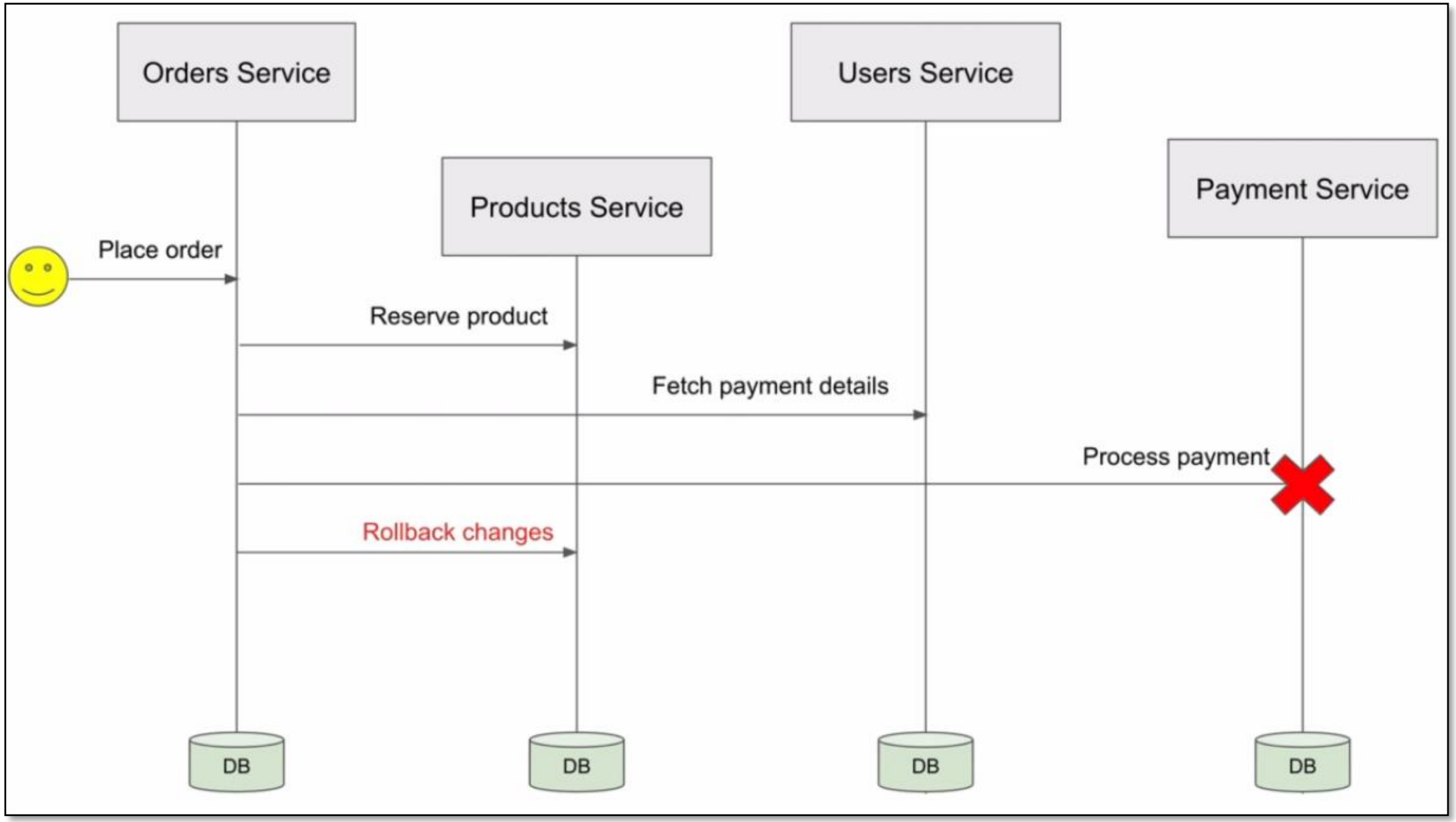
- In monolithic applications, implementing the transaction is very easy.



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

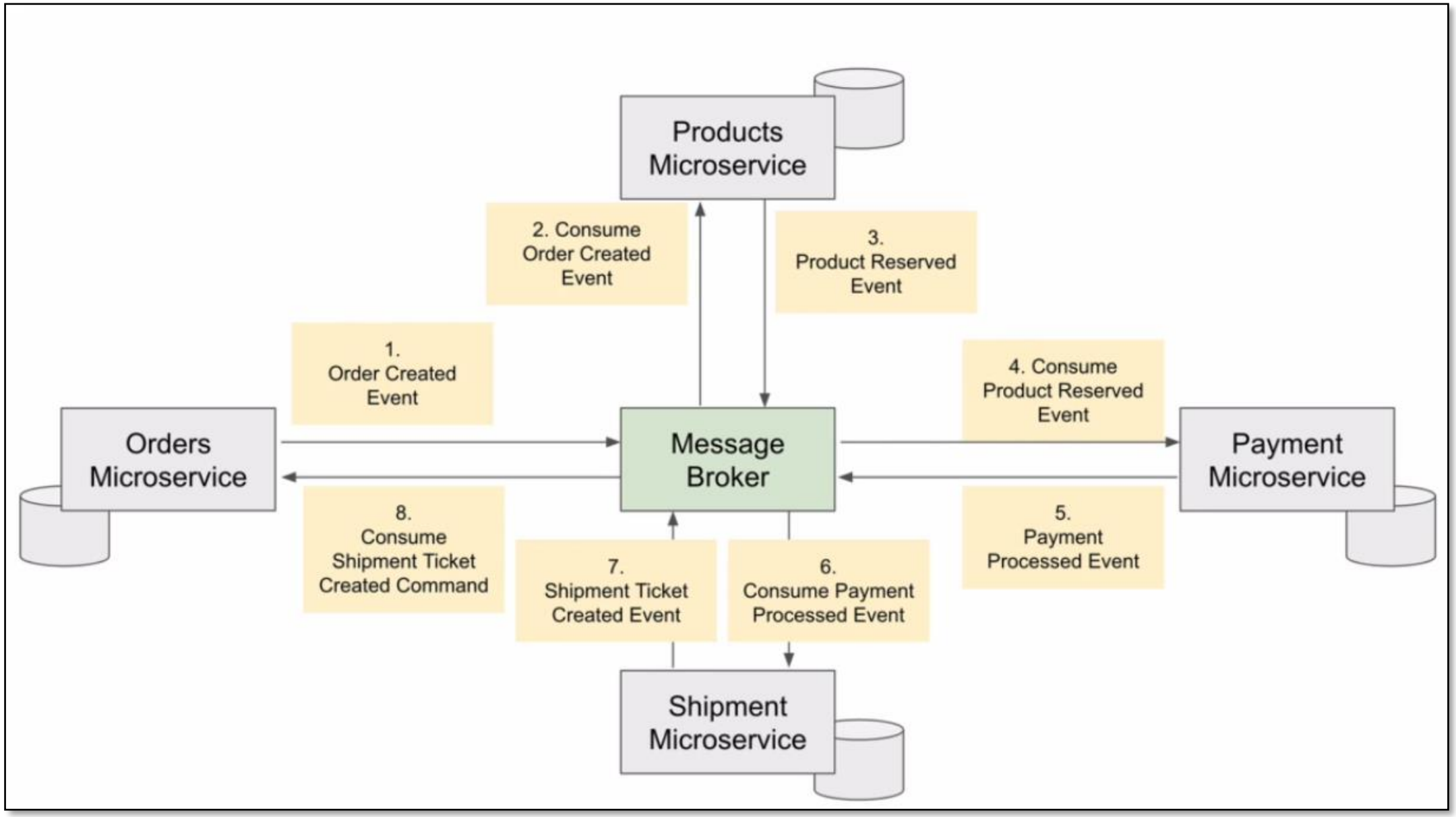
Day - 1

Choreography-Based SAGA

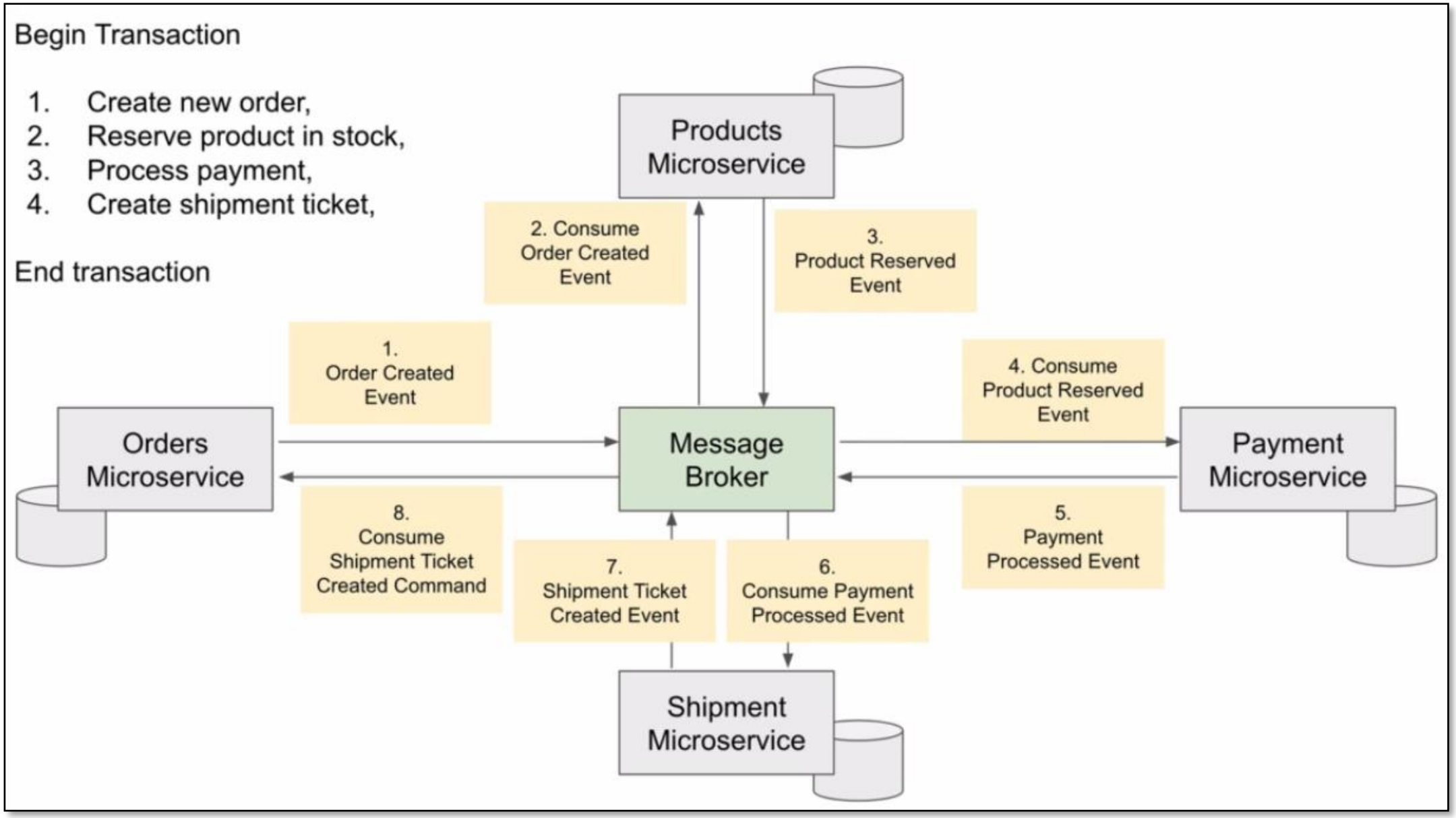
Choreography-Based SAGA

- 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.

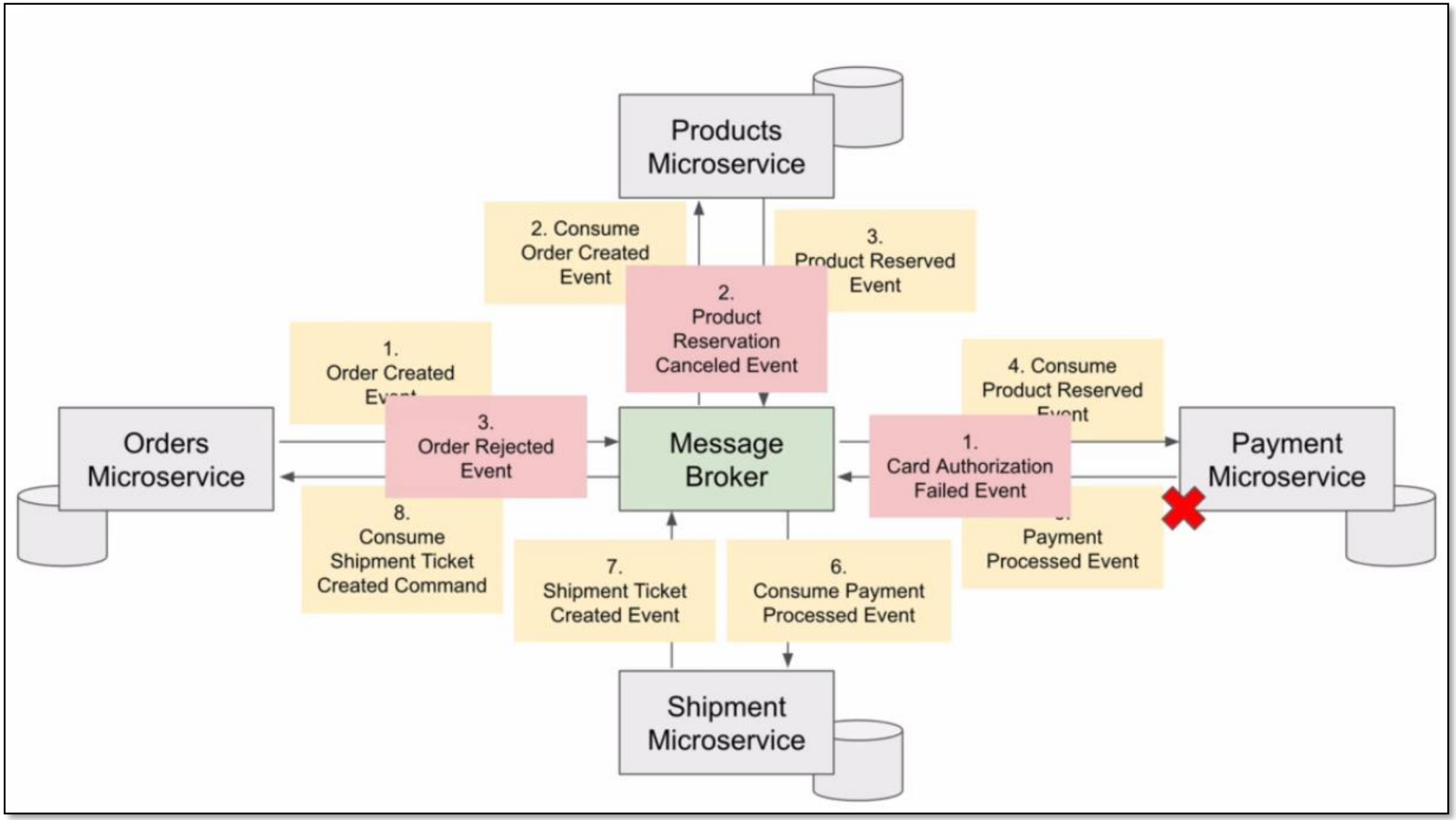
Choreography-Based SAGA



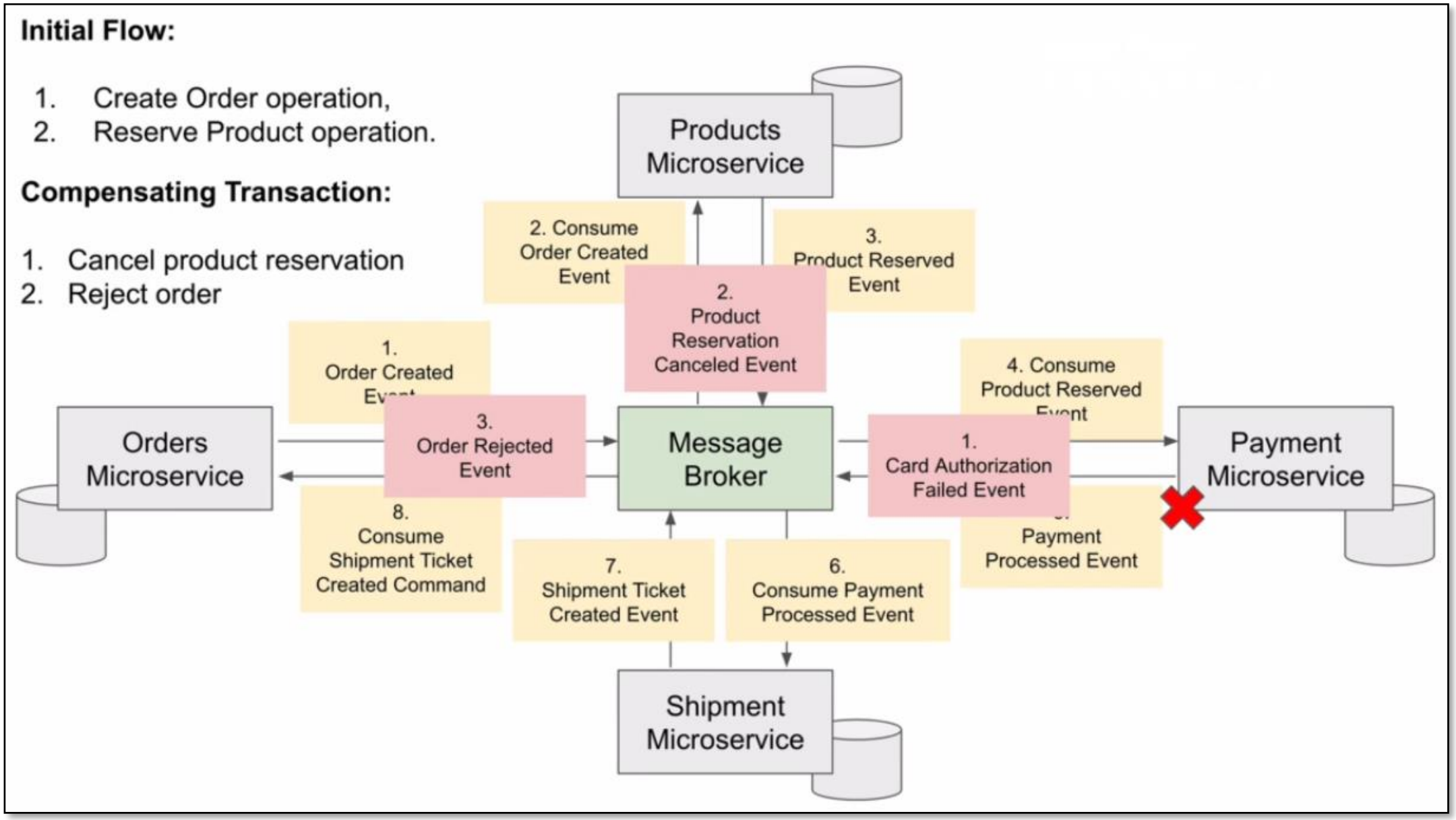
Choreography-Based SAGA



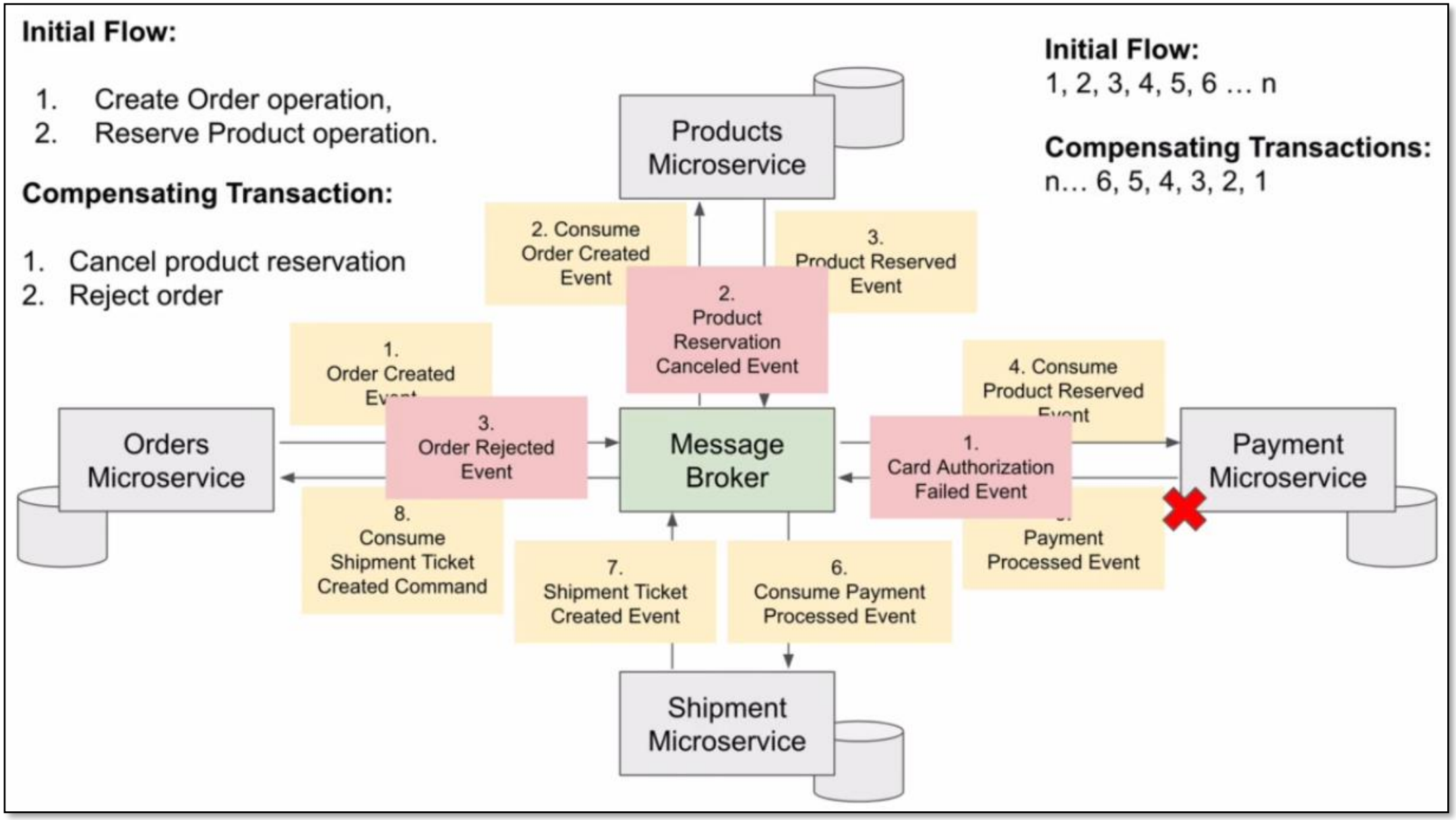
Choreography-Based SAGA



Choreography-Based SAGA



Choreography-Based SAGA



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

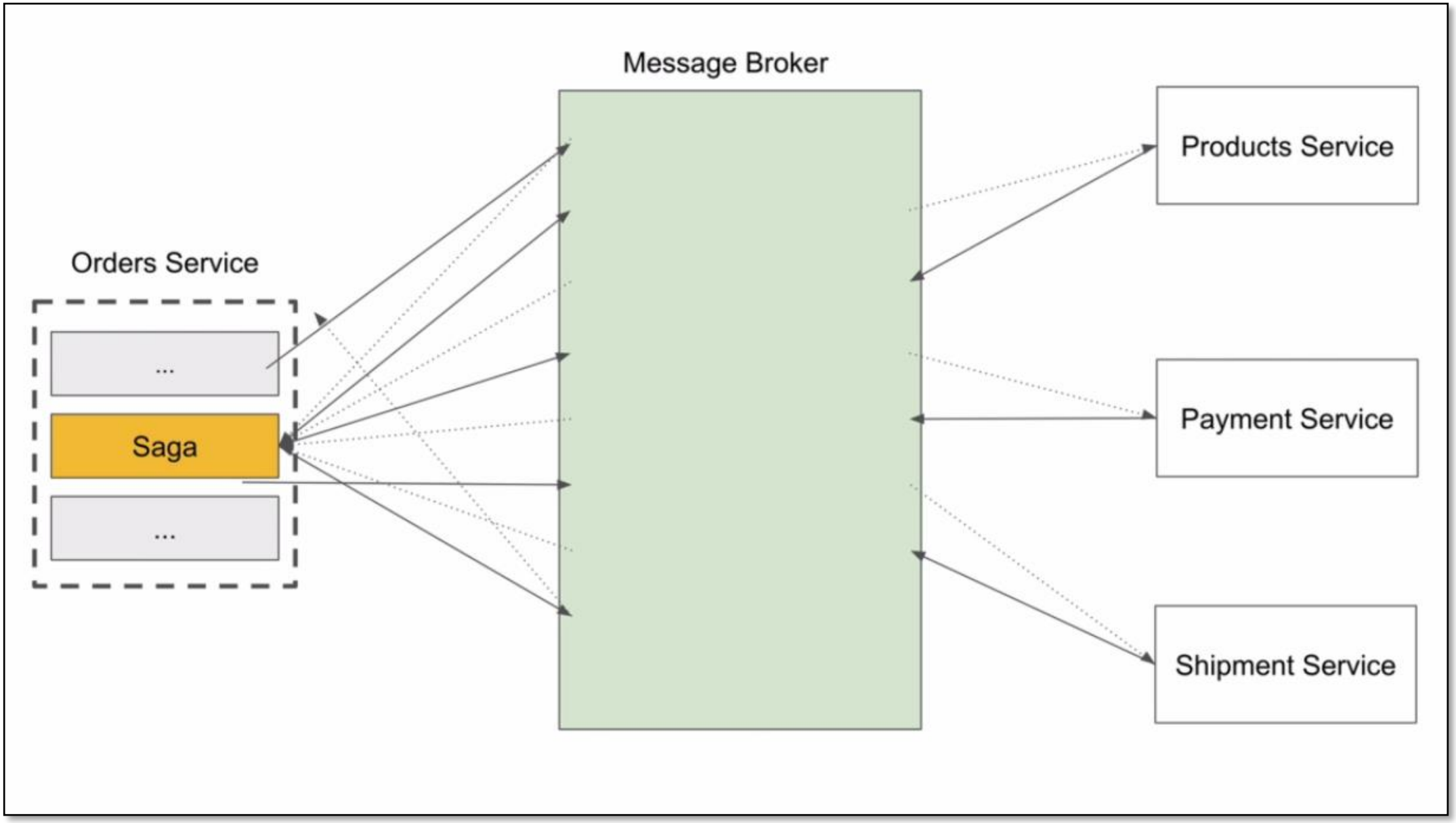
Day - 1

Orchestration-Based SAGA

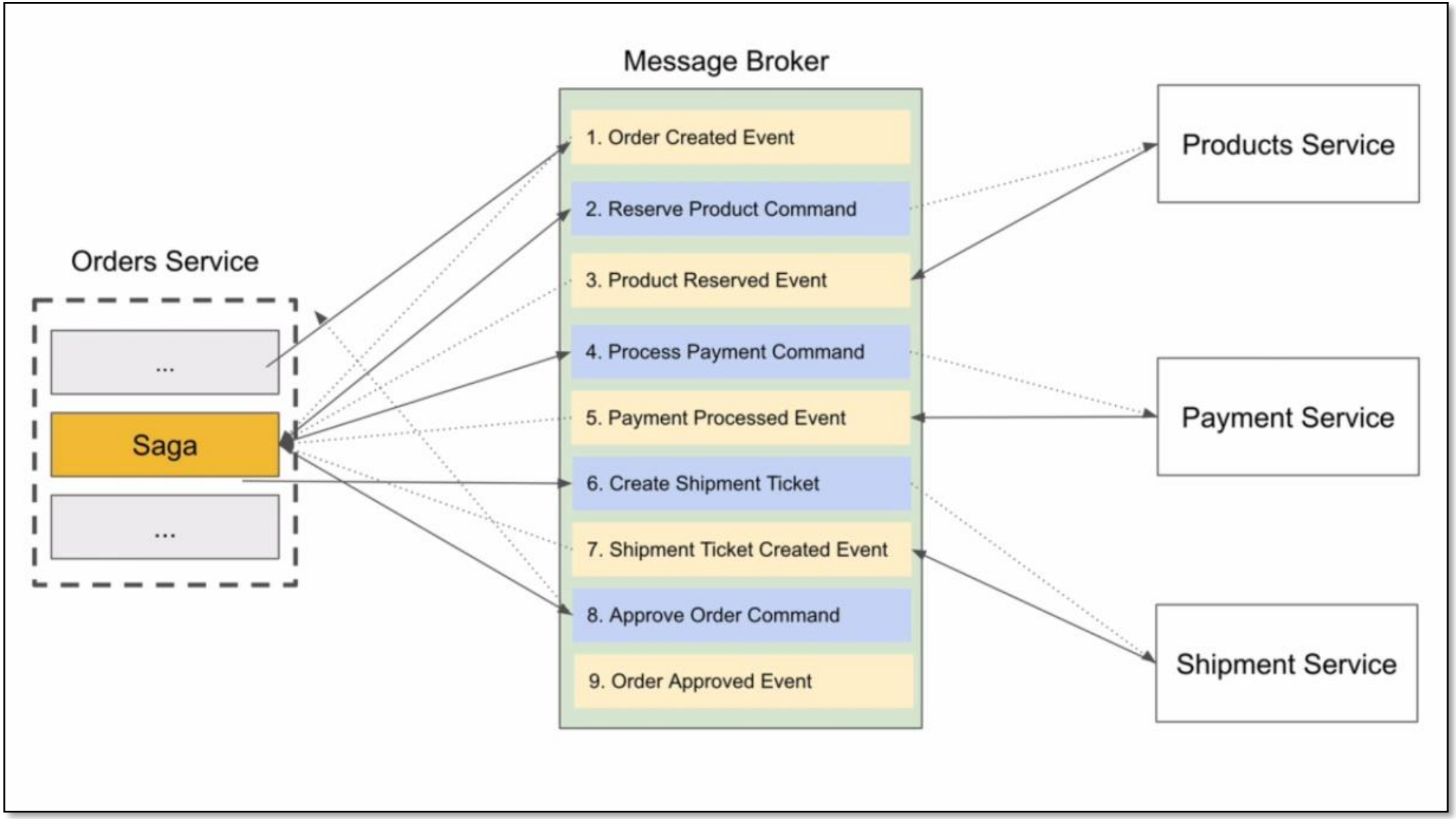
Orchestration-Based SAGA

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

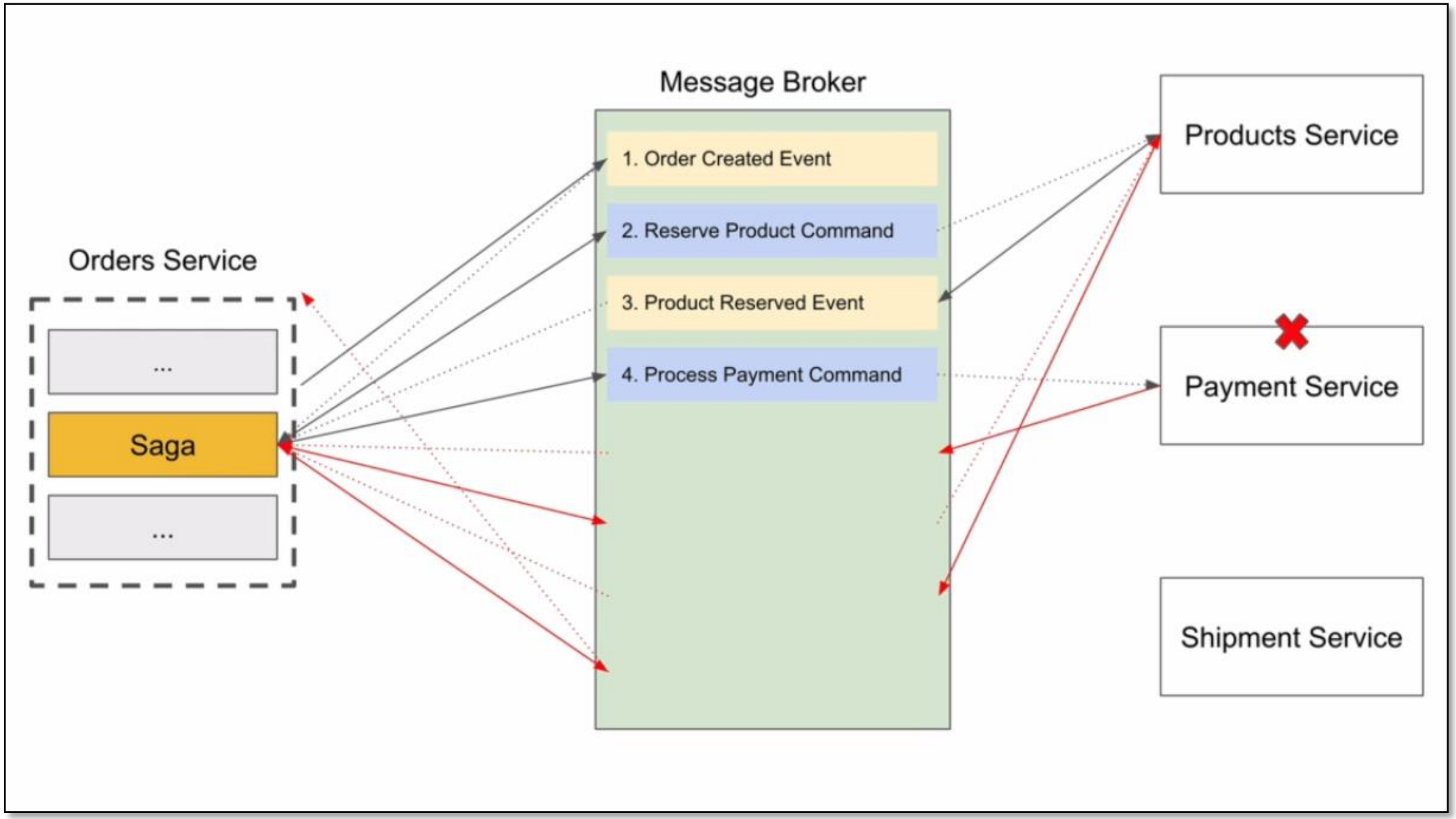
Orchestration-Based SAGA



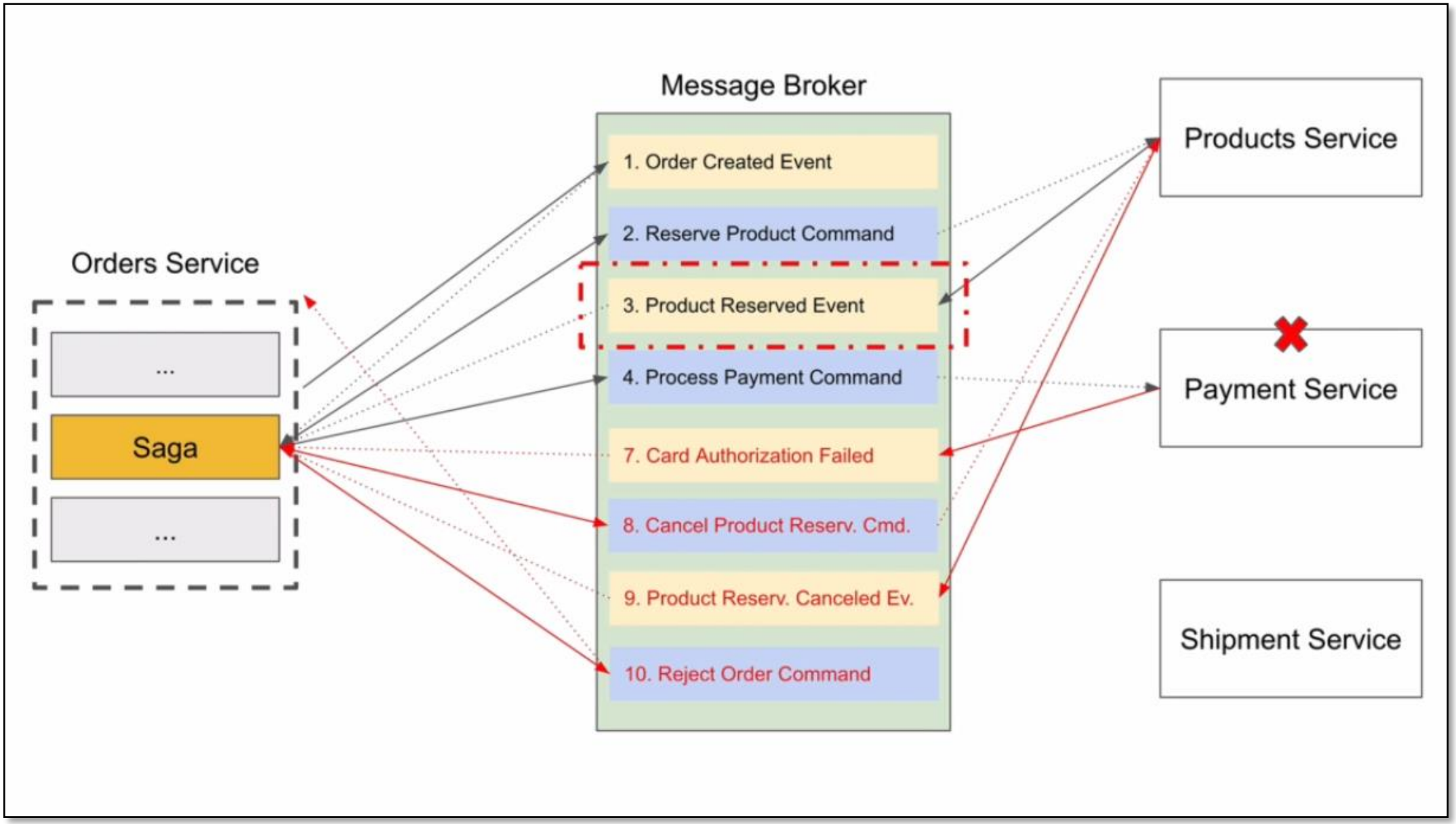
Orchestration-Based SAGA



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.

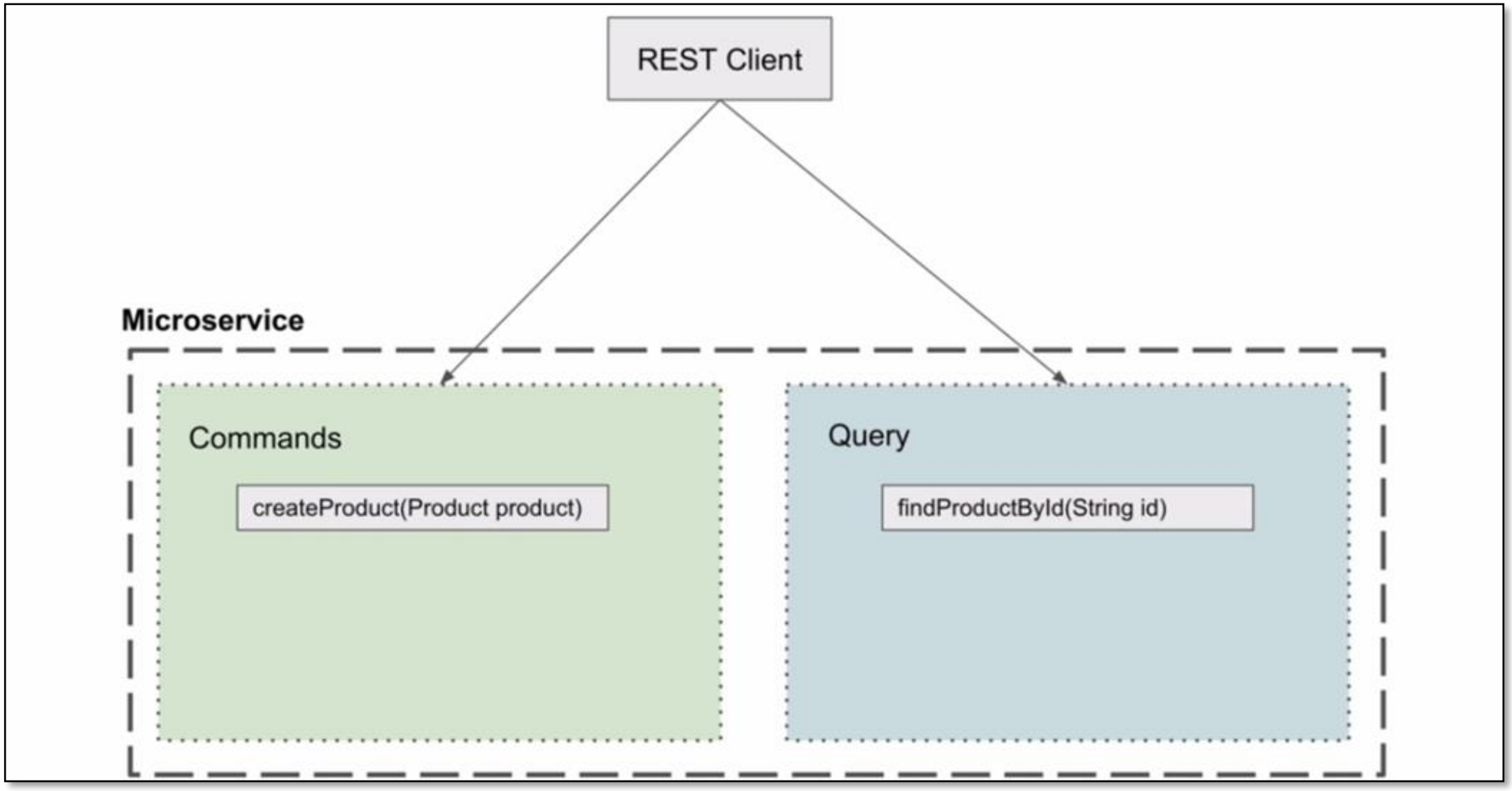
Day - 1

Command Query Responsibility Segregation

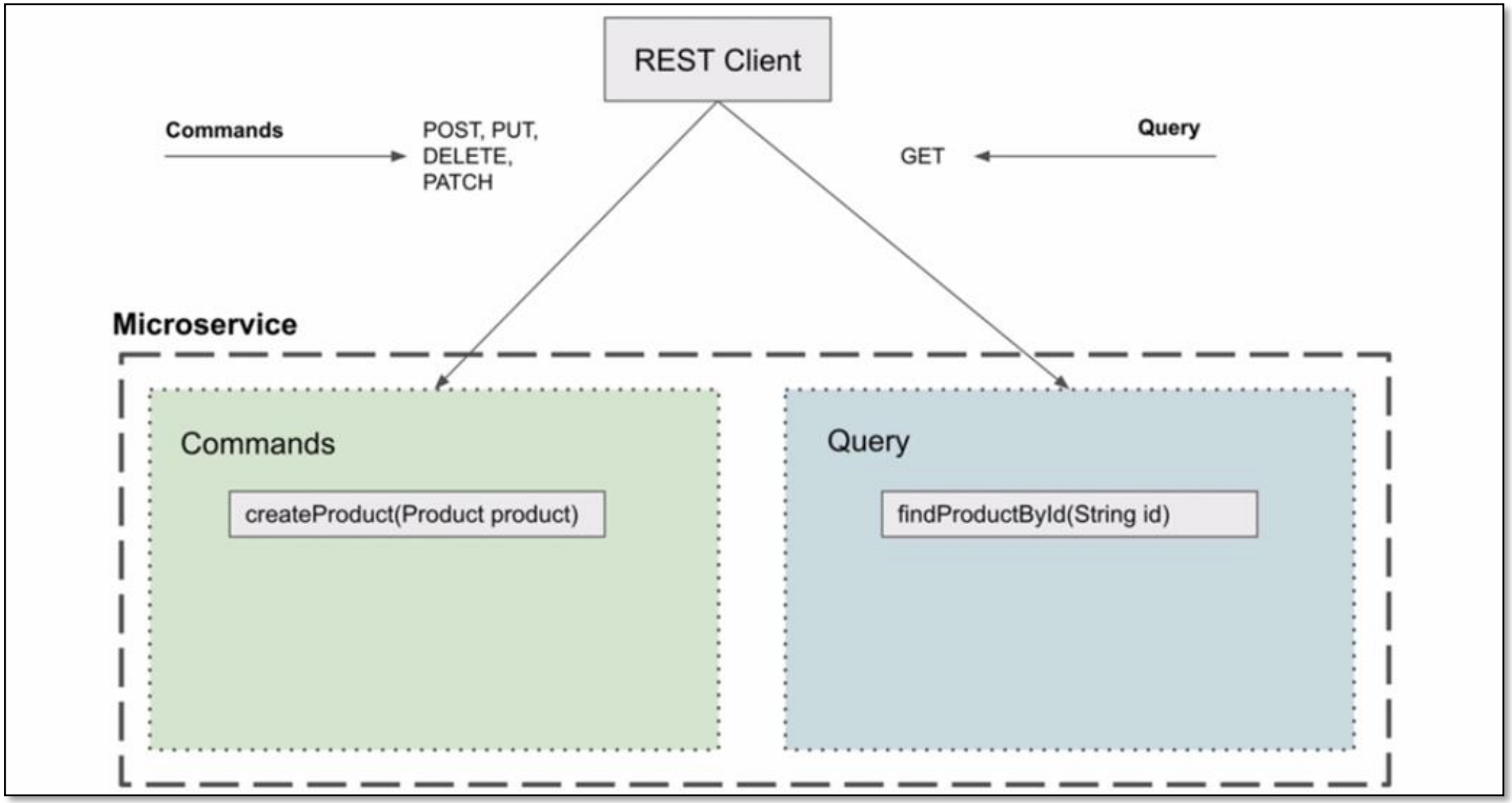
CQRS pattern

- 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.

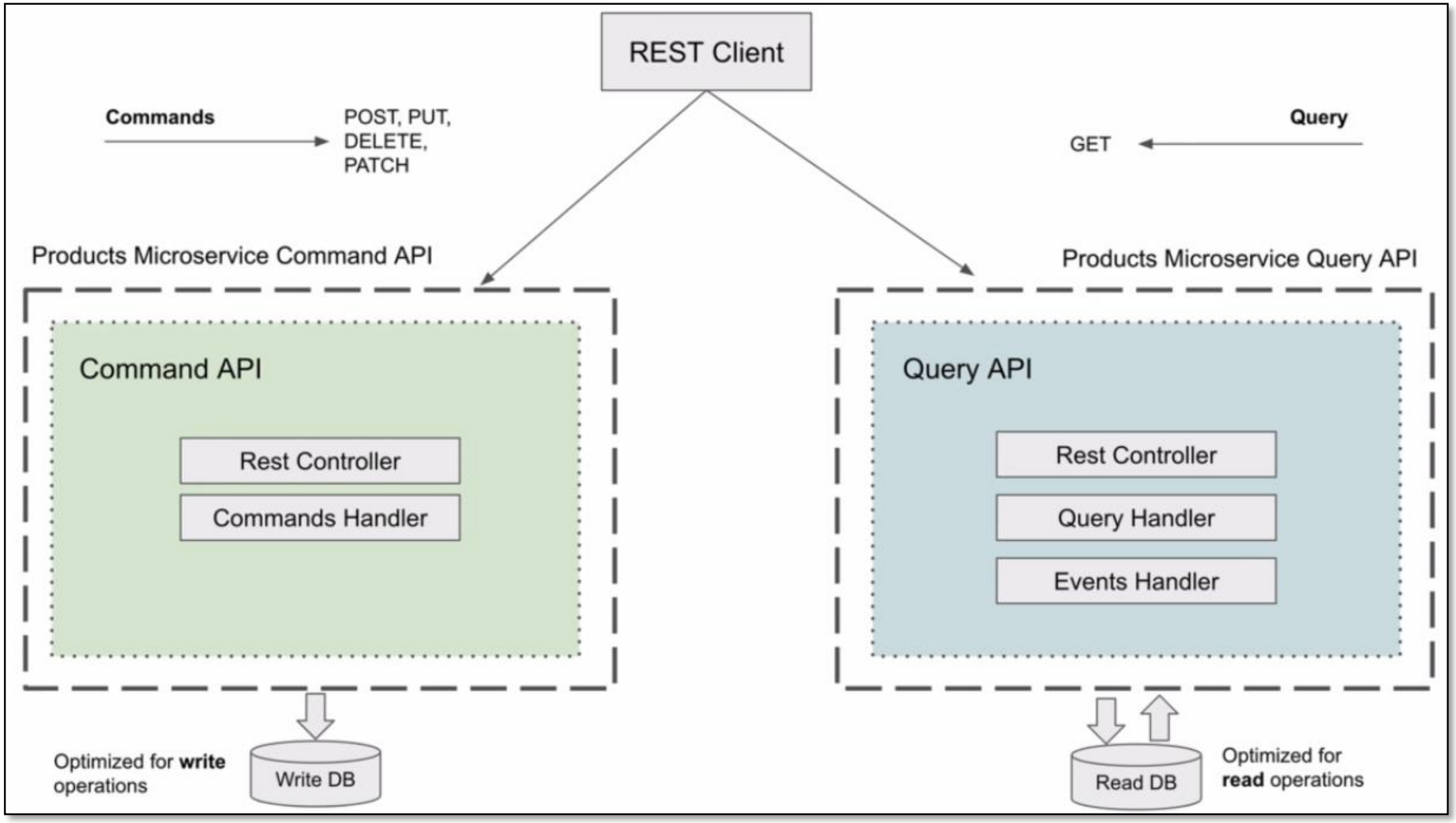
Command Query Responsibility Segregation



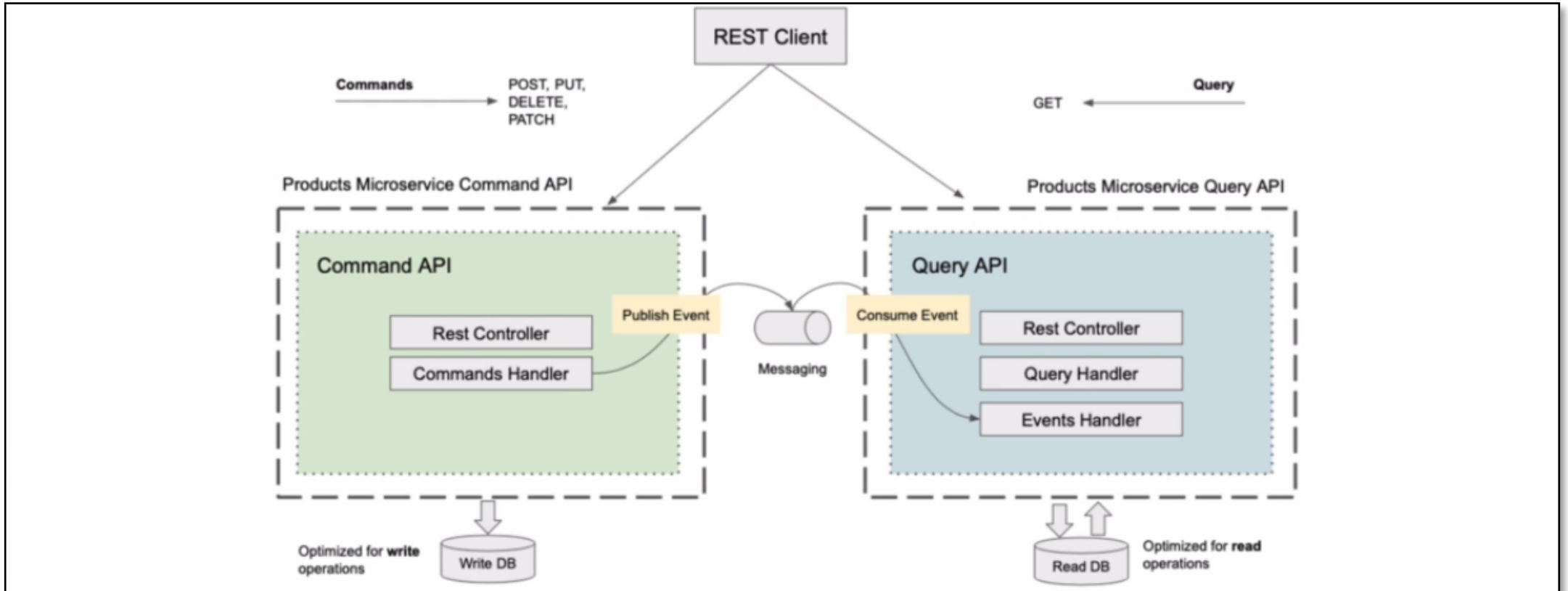
Command vs Query – Controllers



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*.

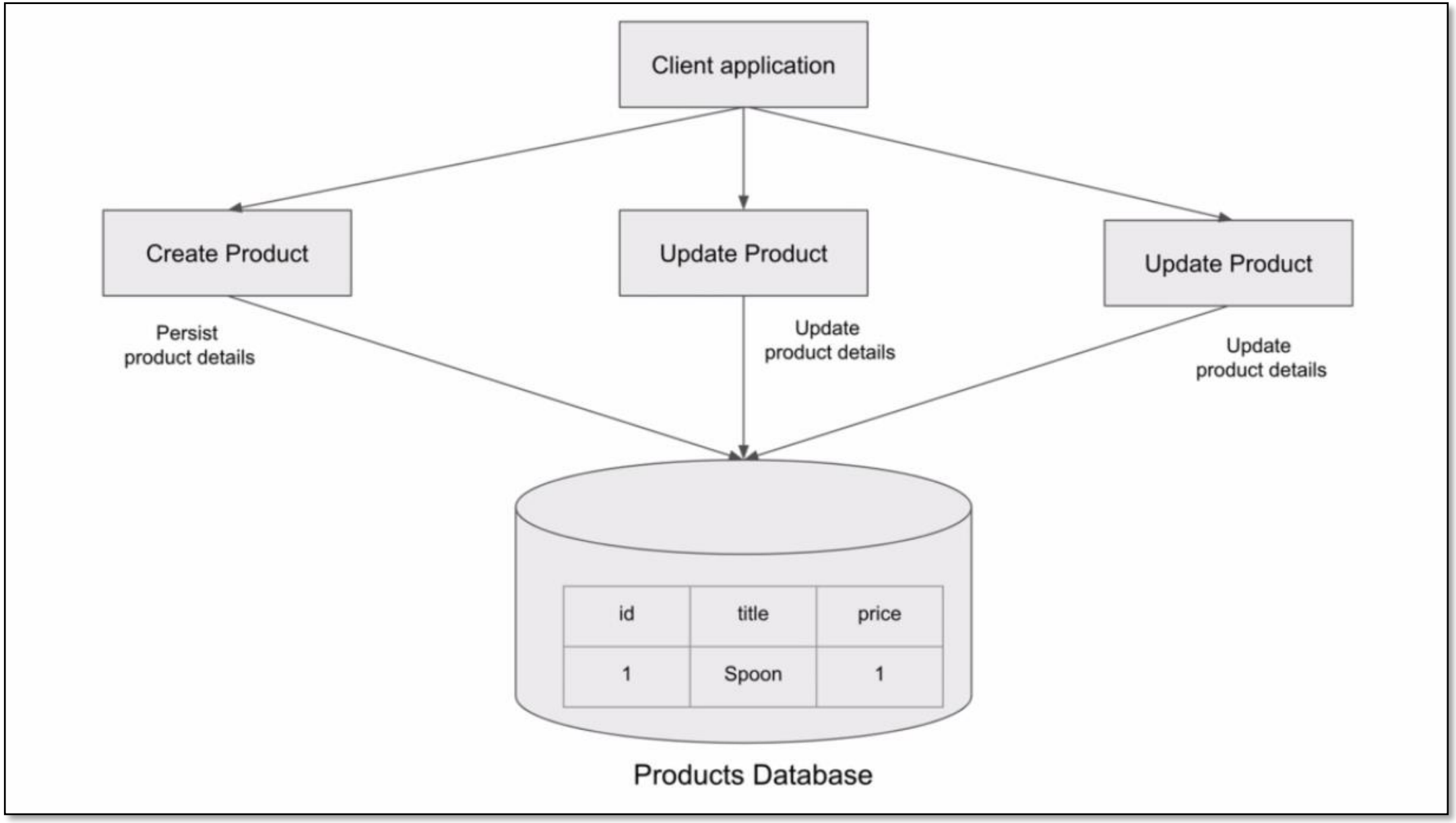
Day - 1

CQRS and Event Sourcing

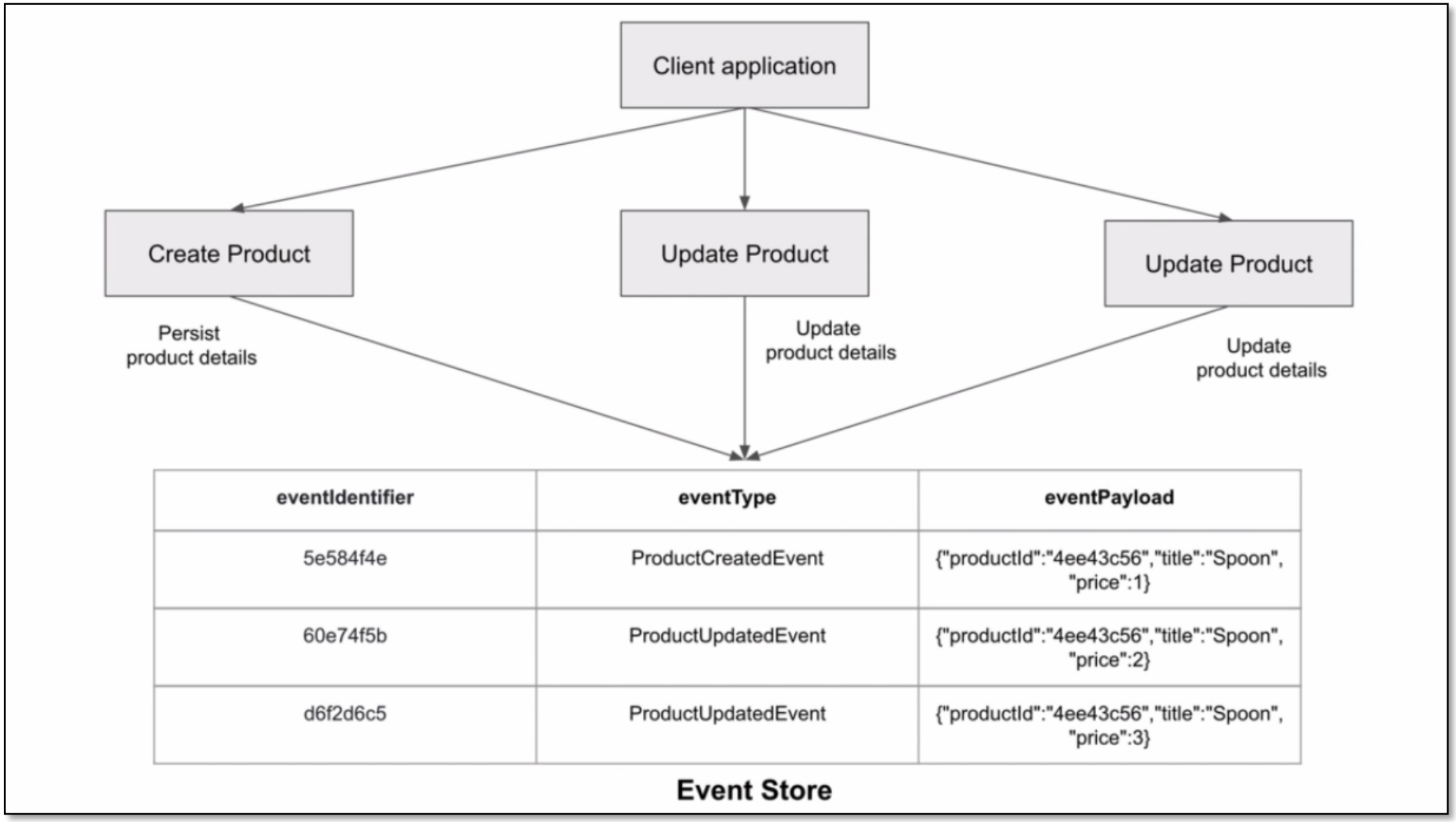
Event Sourcing

- 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

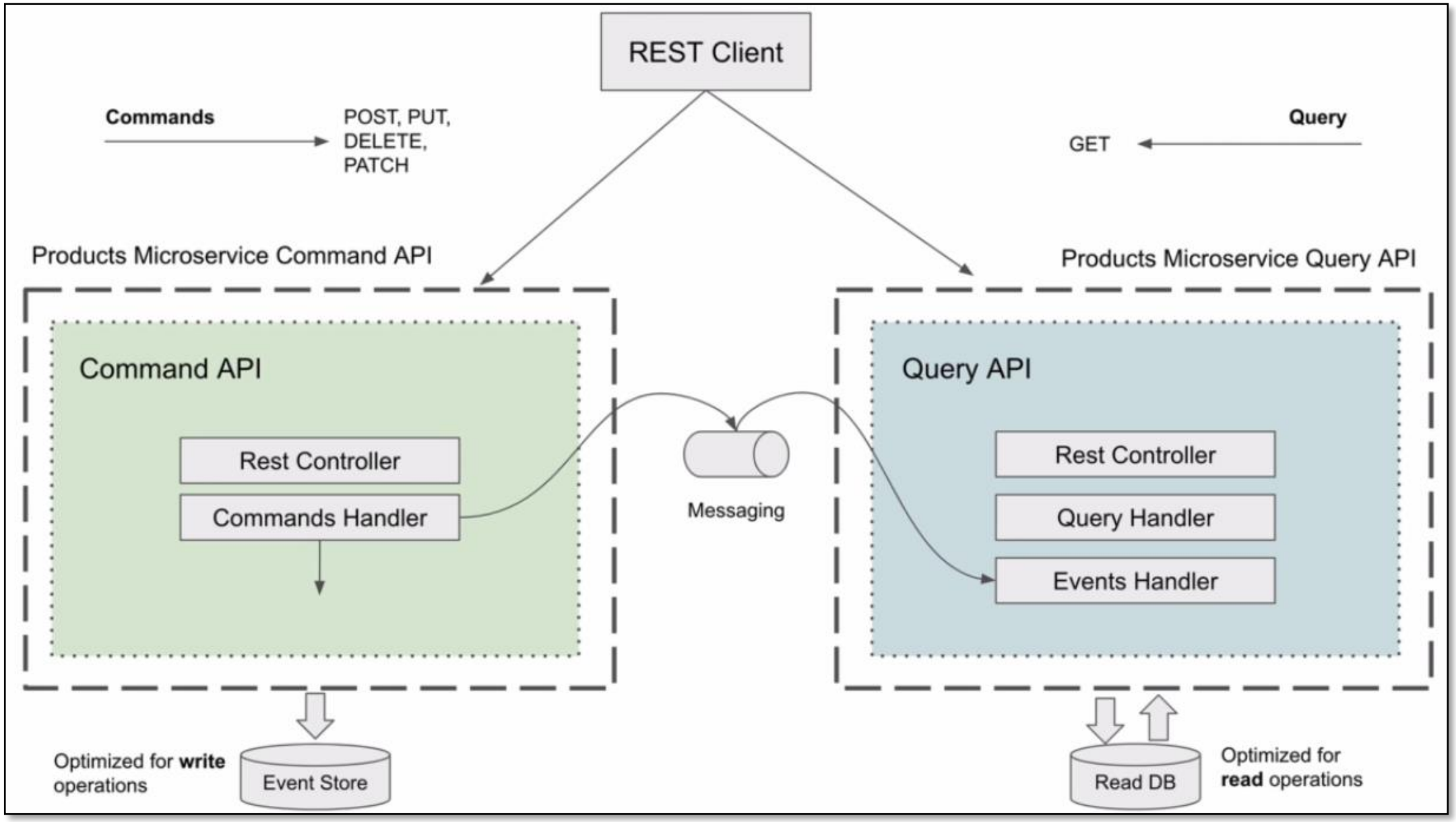


eventIdentifier	eventType	eventPayload
5e584f4e	ProductCreatedEvent	{"productId":"4ee43c56","title":"Spoon", "price":1}
60e74f5b	ProductUpdatedEvent	{"productId":"4ee43c56","title":"Spoon", "price":2}
d6f2d6c5	ProductUpdatedEvent	{"productId":"4ee43c56","title":"Spoon", "price":3}

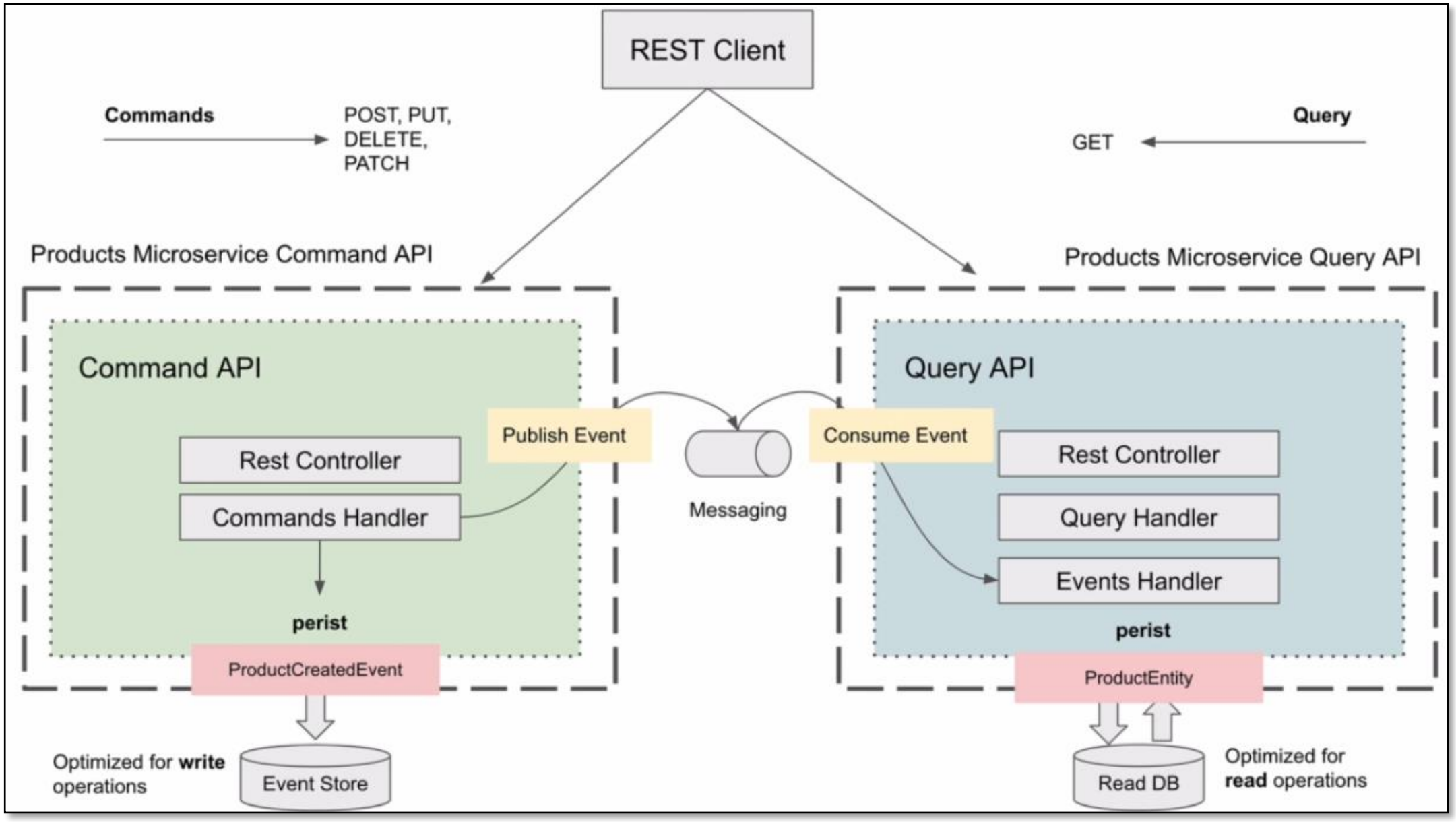
Event Store

© Mphasis 2020 Proprietary and confidential information I 5/10/2024

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

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

Configure Microservices with Eureka Service Registry Server

1. Create a new Project for **DiscoveryServer** using the Spring Initializr (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
2 server.port=8761
3 eureka.client.register-with-eureka=false
4 eureka.client.fetch-registry=false
5 eureka.instance.prefer-ip-address=true
6 #eureka.instance.hostname=localhost
7 eureka.client.service-url.defaultZone=http://localhost:8761/eureka
8
```

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.

application.properties

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

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

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

© Mphasis 2020 Proprietary and confi

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

Recap of 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 - 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

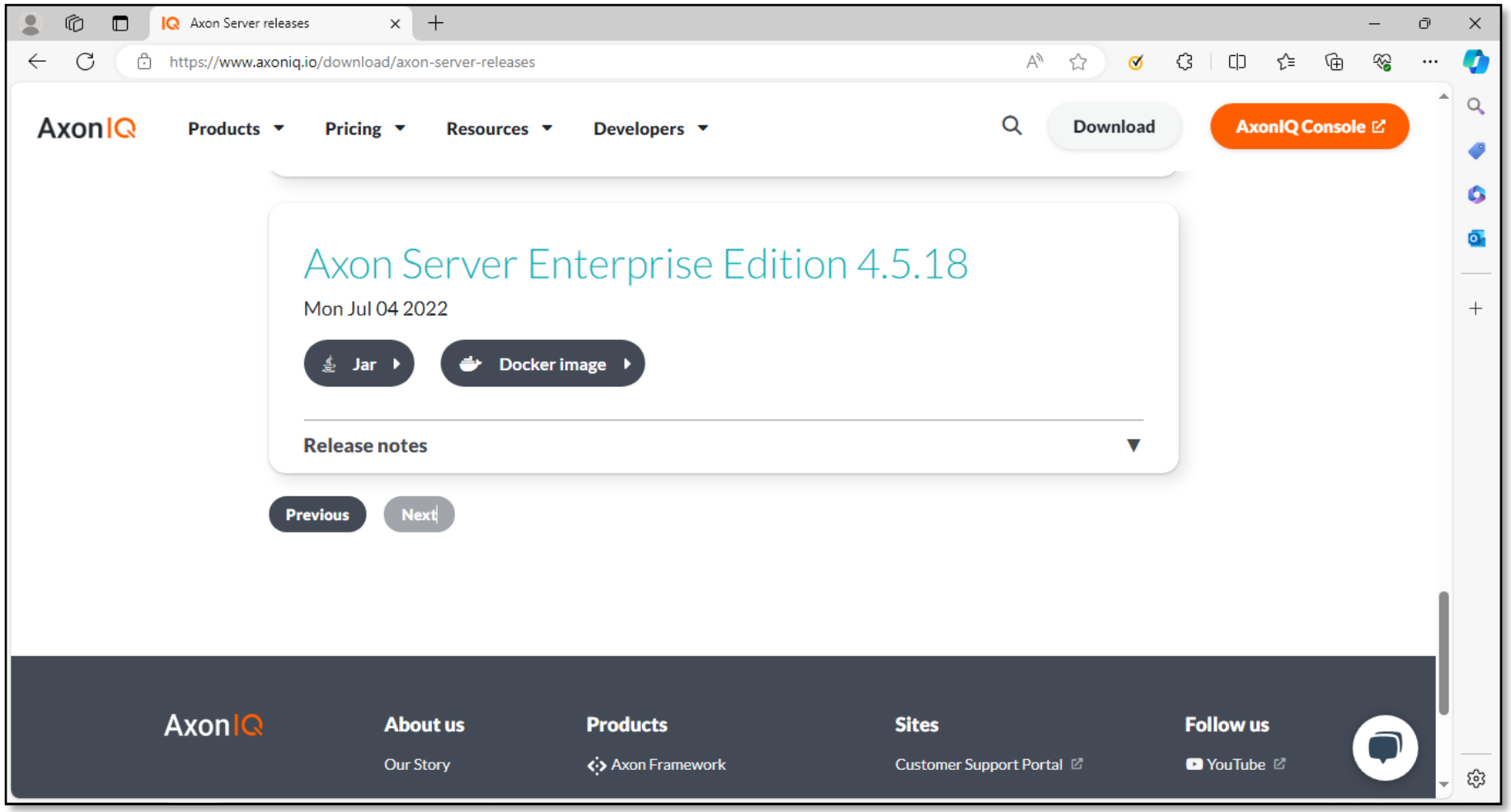
- Axon provides the **Axon Framework** and the **Axon Server** to help build applications centered on three core concepts - **CQRS** (Command Query Responsibility Segregation) / **Event Sourcing** and **DDD** (Domain Driven Design).
- While many types of applications can be built using Axon, it has proven 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.

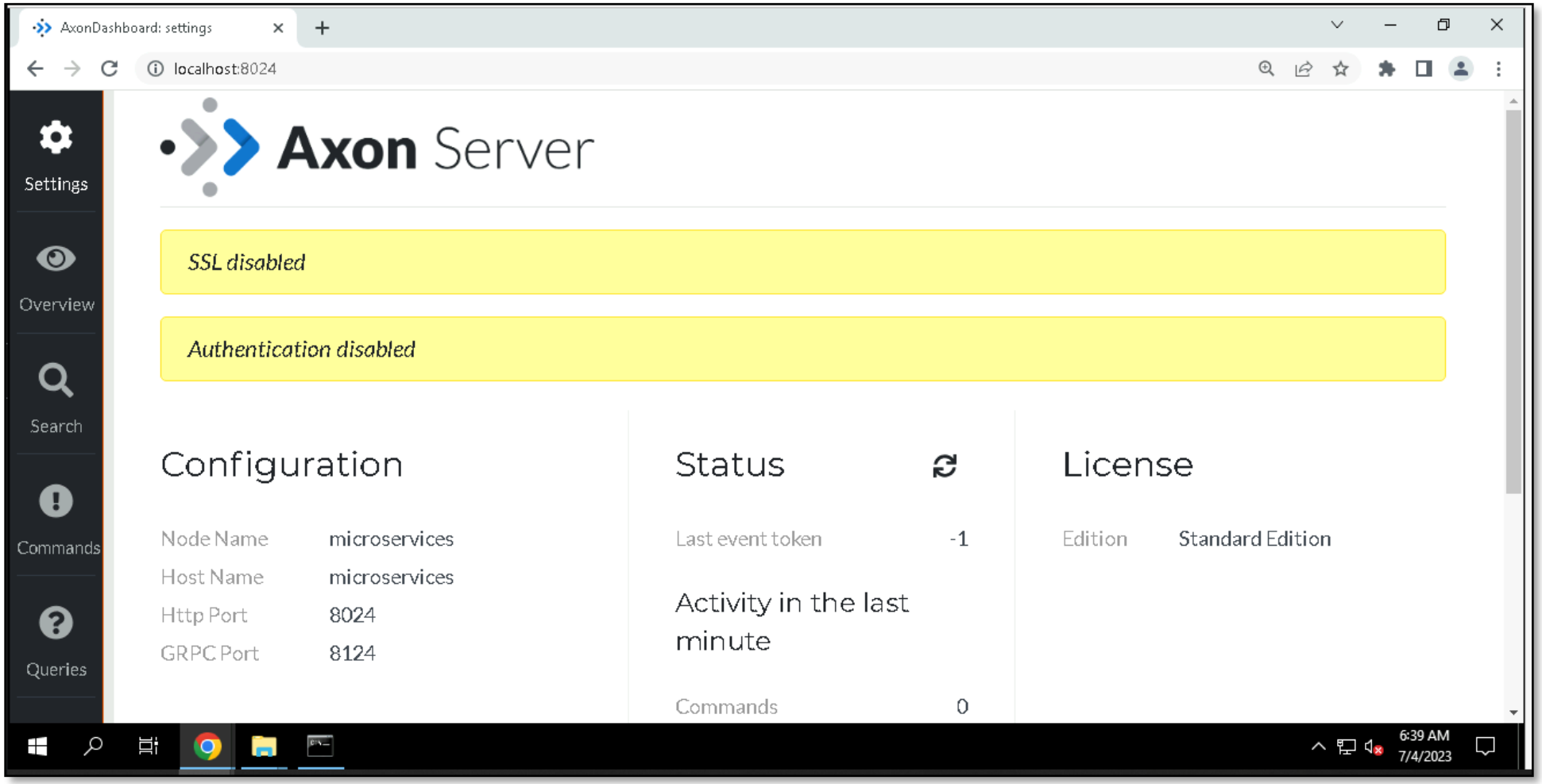


Download and run Axon Server as JAR application

- Run java -jar axonserver.jar

Download and run Axon Server as JAR application

- Access http://localhost:8024



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

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:

CreateProductRestModel.java

```
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:

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

Target Aggregate Identifier

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:

CreateProductCommand.java

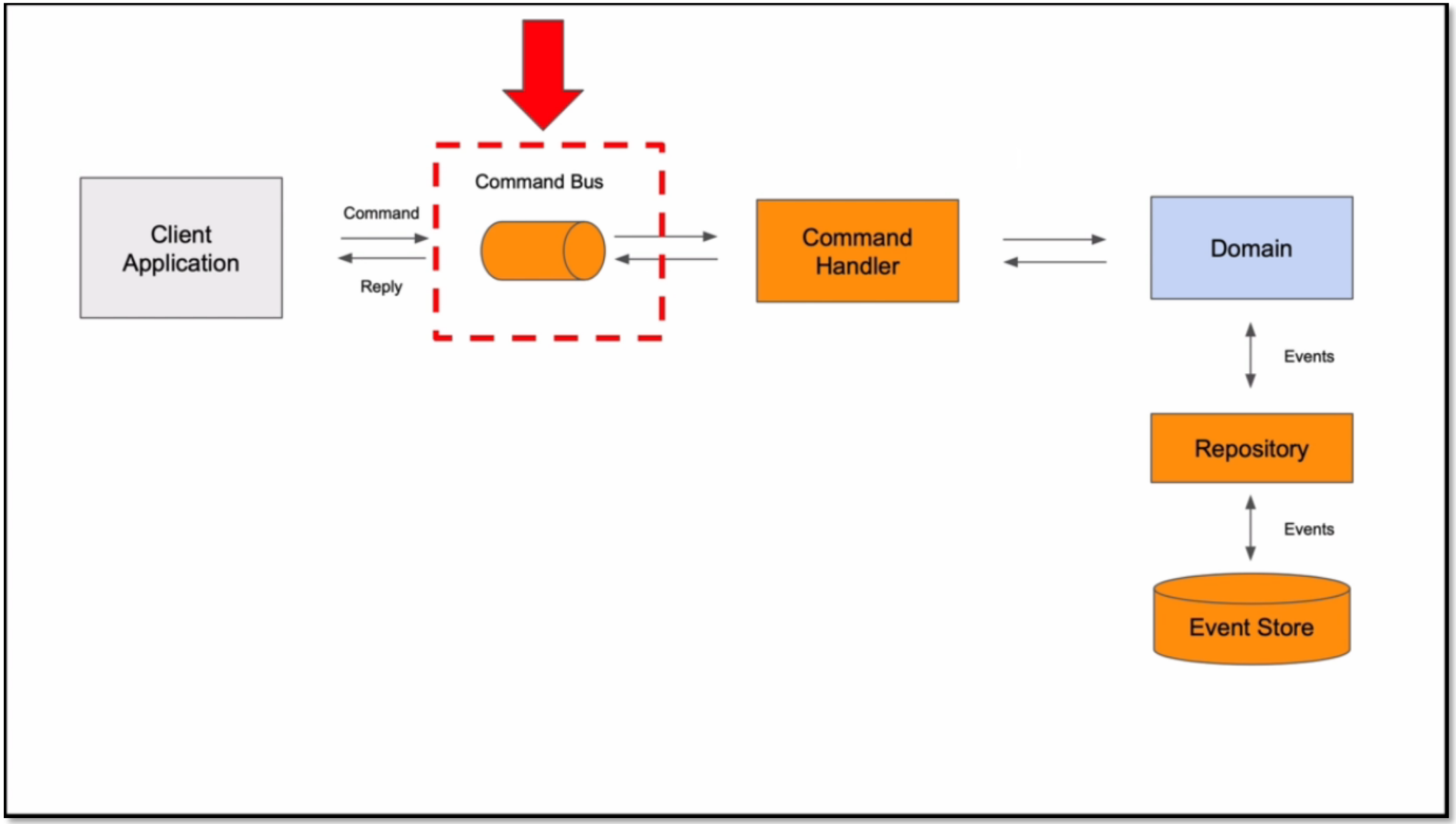
```java
package com.mphasis.command;

import java.math.BigDecimal;

import org.axonframework.modelling.command.TargetAggregateIdentifier;

import lombok.Builder;
import lombok.Data;

@Builder
@Data
public class CreateProductCommand {

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

}

```

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

ProductController.java

```
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

ProductCreatedEvent

ProductShippedEvent

ProductDeletedEvent

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:

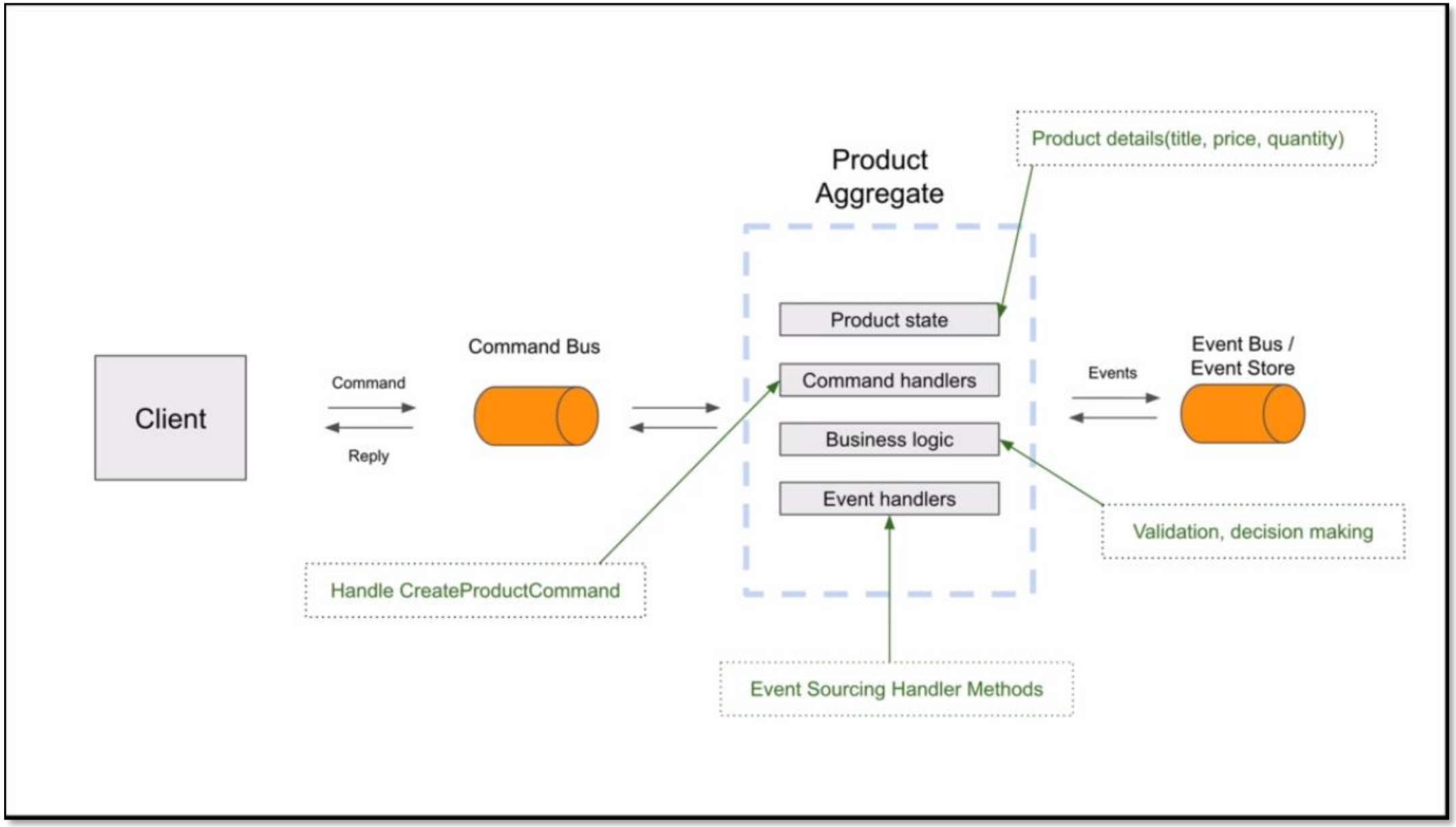
ProductCreatedEvent.java

```
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



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*.

© Mphasis 2020 Proprietary and confidential information

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

```java
package com.mphasis.command;

import org.axonframework.commandhandling.CommandHandler;
import org.axonframework.spring.stereotype.Aggregate;

@Aggregate
public class ProductAggregate {

    public ProductAggregate() {

    }

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

Validate the CreateProductCommand

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

ProductAggregate.java

```
@Aggregate
public class ProductAggregate {

    public ProductAggregate() {

    }

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

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

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

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

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

© Mphasis 2020 Proprietary and confidential information

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  }
```

@EventSourcingHandler

- **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

ProductAggregate.java

```java
package com.mphasis.command;

import java.math.BigDecimal;

@Aggregate
public class ProductAggregate {

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

    public ProductAggregate() {

    }

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

@EventSourcingHandler

ProductAggregate.java

```
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

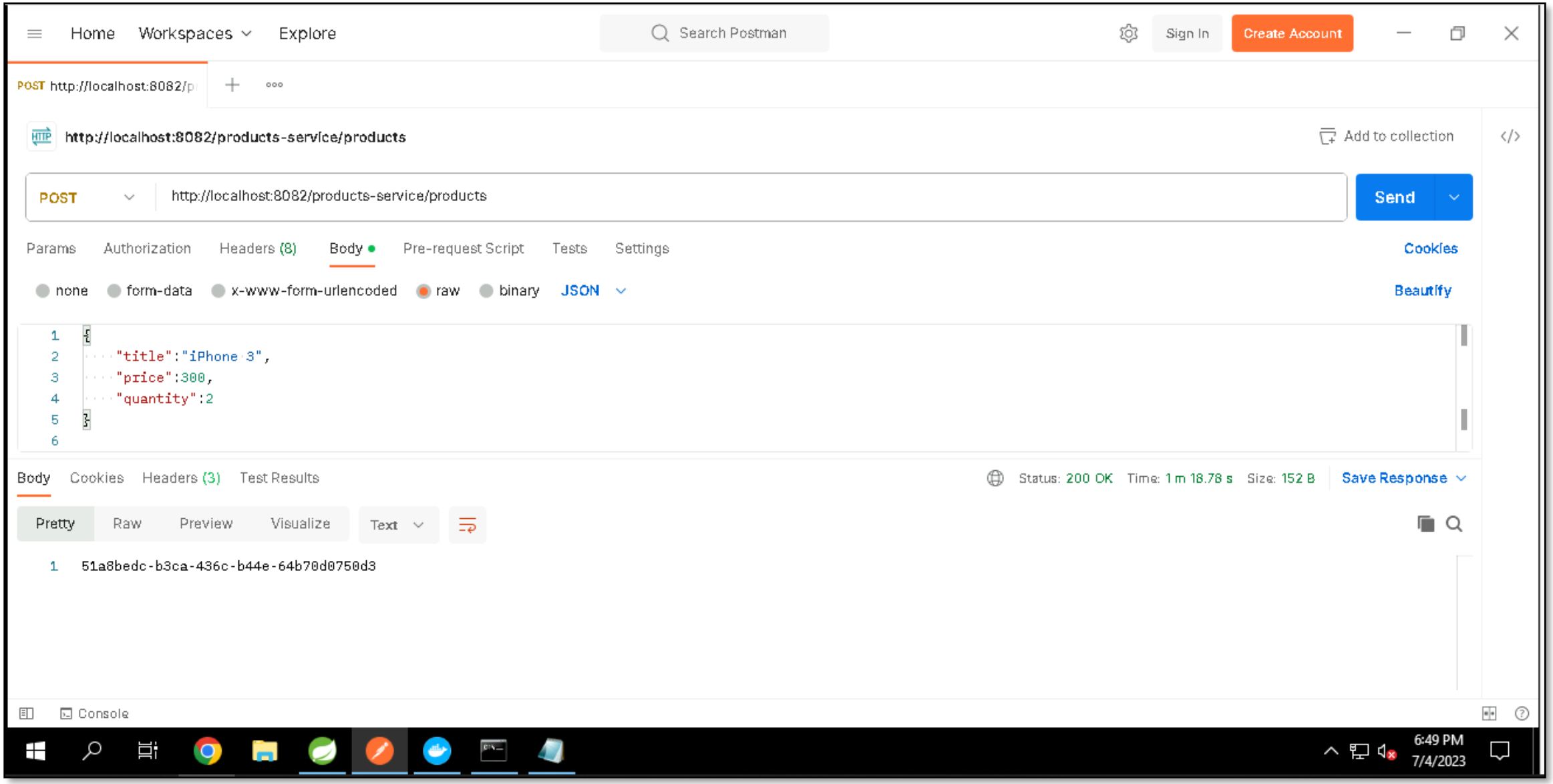
- When using Axon with Spring Cloud – Maven start demanding for the Google Guava dependency (due to transitive dependency).

```
<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

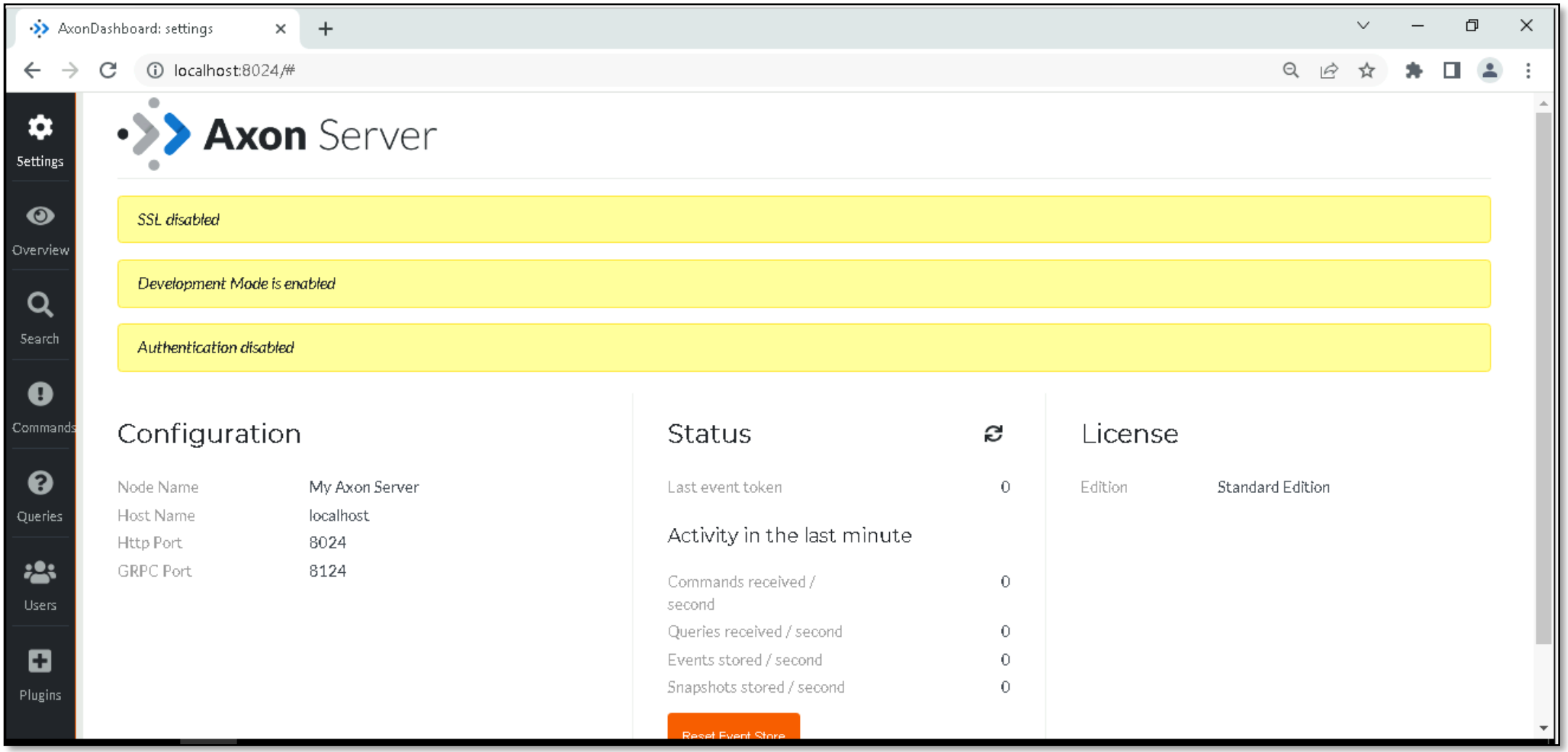
1. Run the AxonServer using Docker command.
2. Ensure the Discovery Server and Product Service is running.
3. Ensure the ApiGateway is running.

Send a POST request



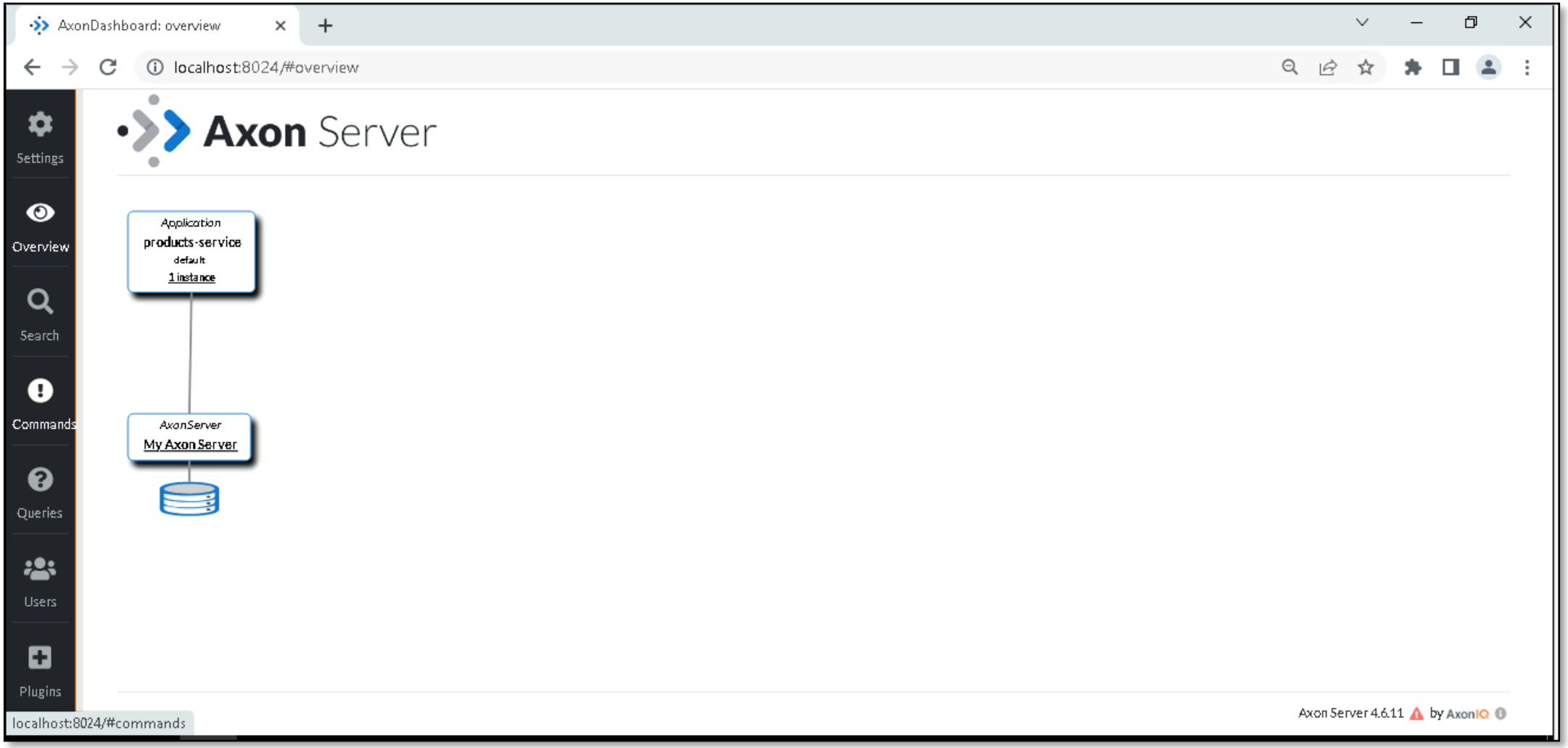
Previewing Event in the EventStore

- Go to Axon Server:



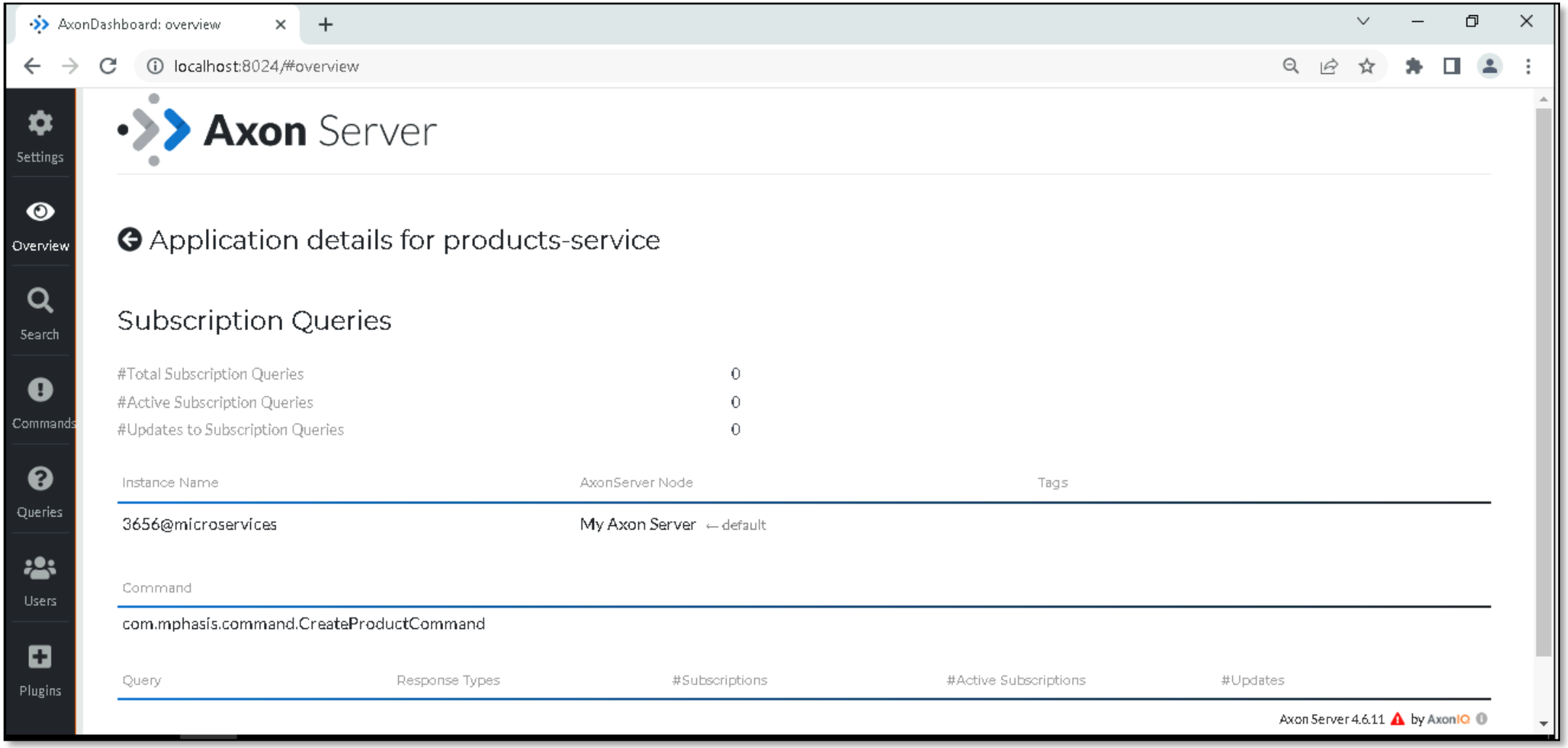
Previewing Event in the EventStore

- Click on **Overview** tab:



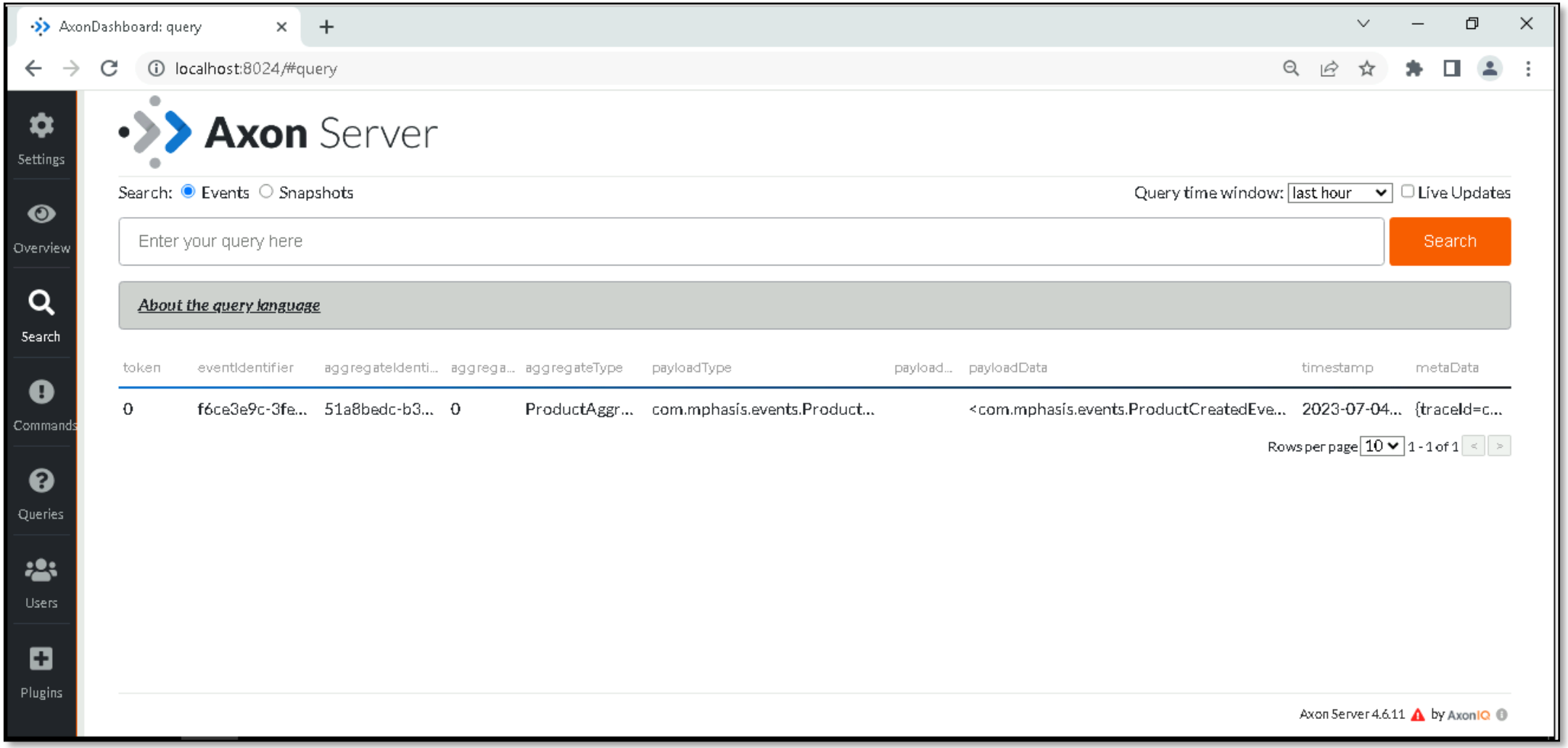
Previewing Event in the EventStore

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



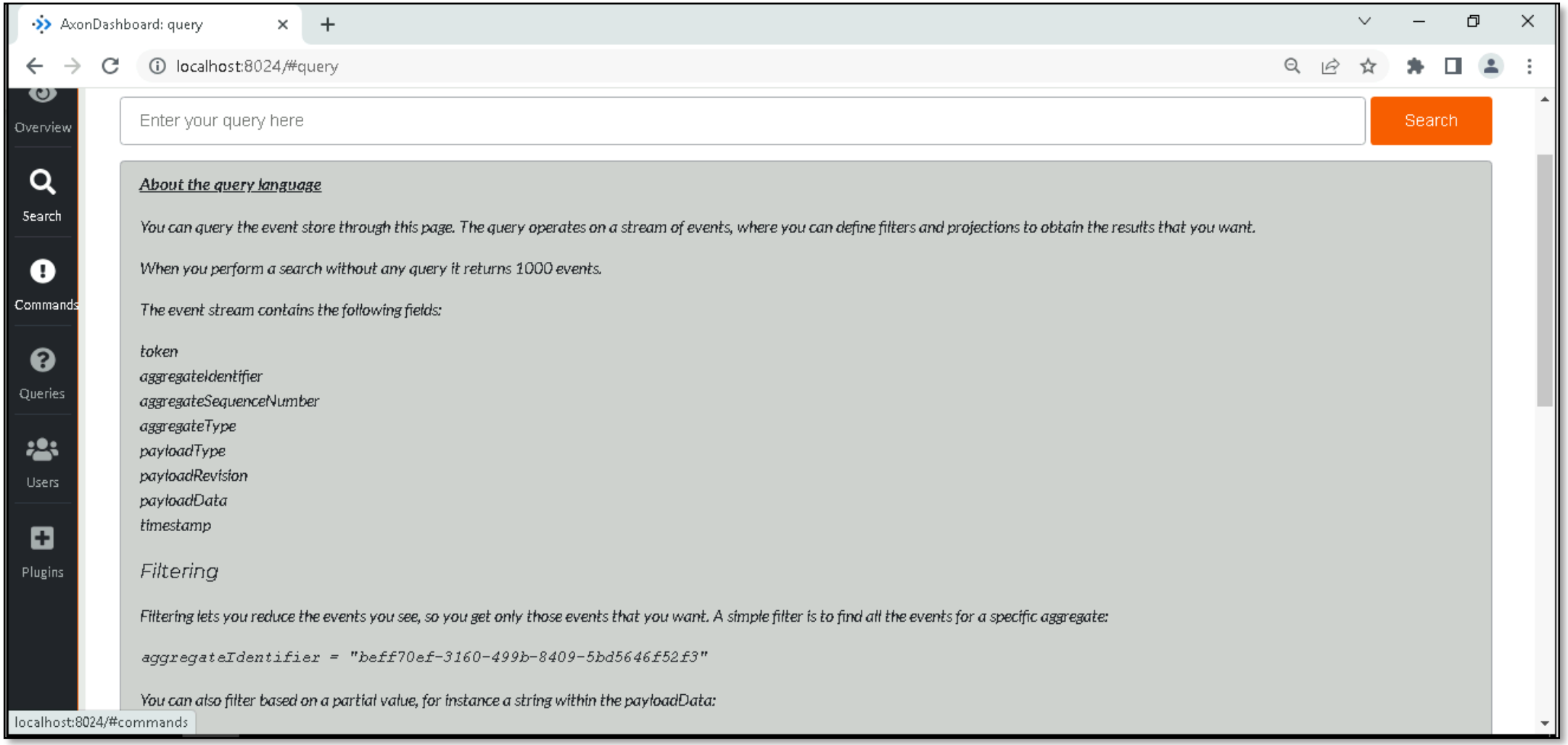
Previewing Event in the EventStore

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



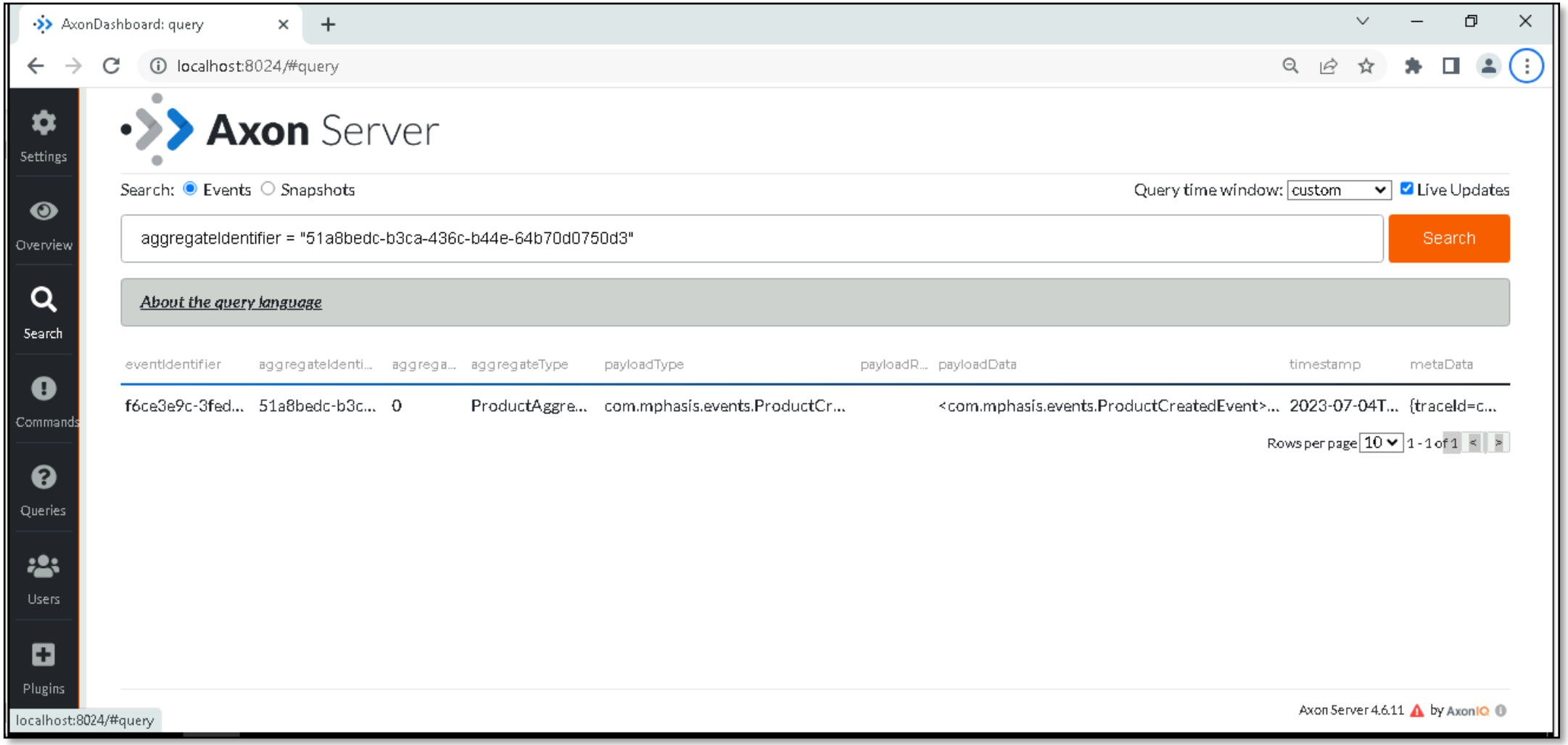
Previewing Event in the EventStore

- About the **query** language:



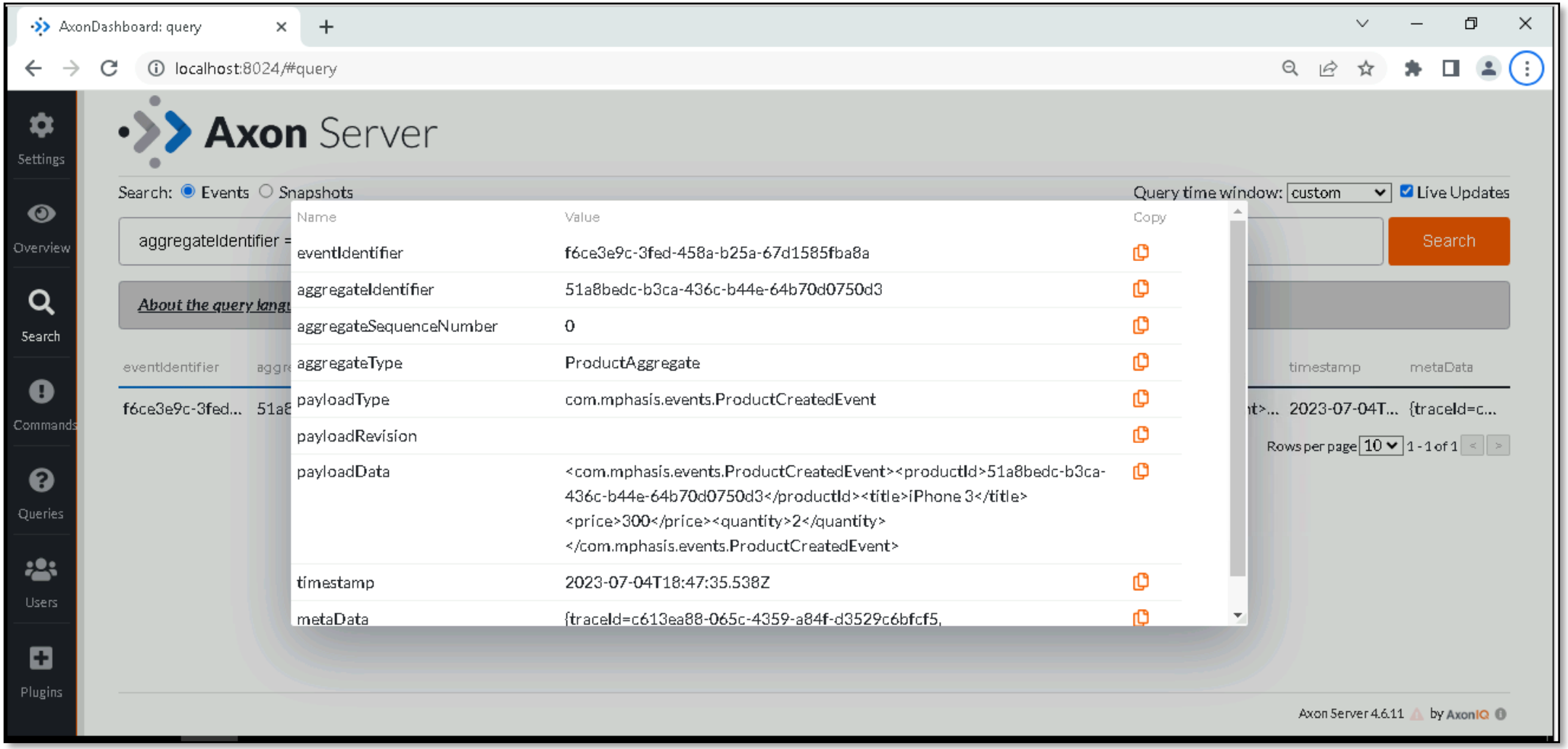
Previewing Event in the EventStore

- Search using the **aggregateIdentifier**:



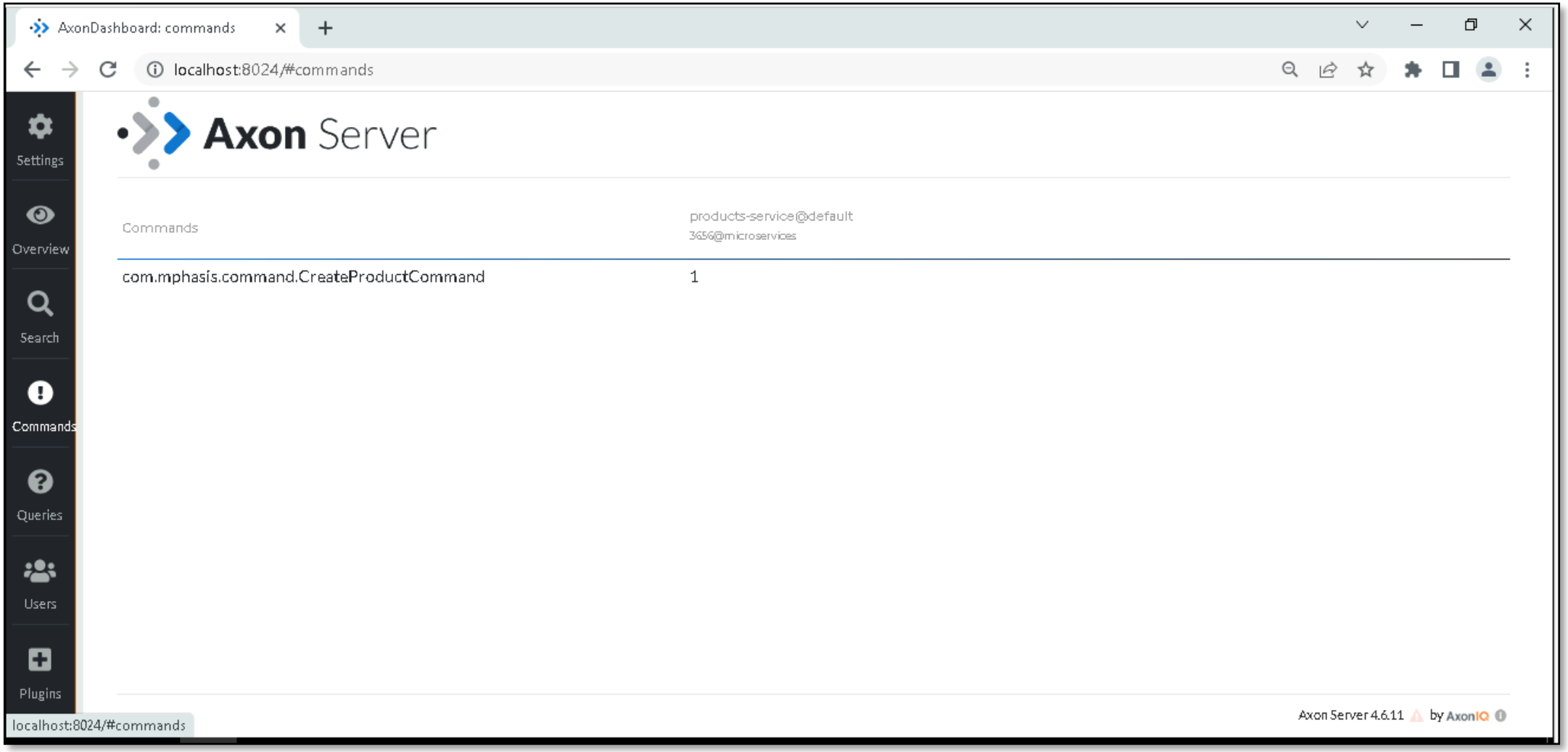
Previewing Event in the EventStore

- View the event details available in Event Store:



Previewing Event in the EventStore

- Go to Commands tab:



Recap of Day – 2

- 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

Day – 3 Agenda

- 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

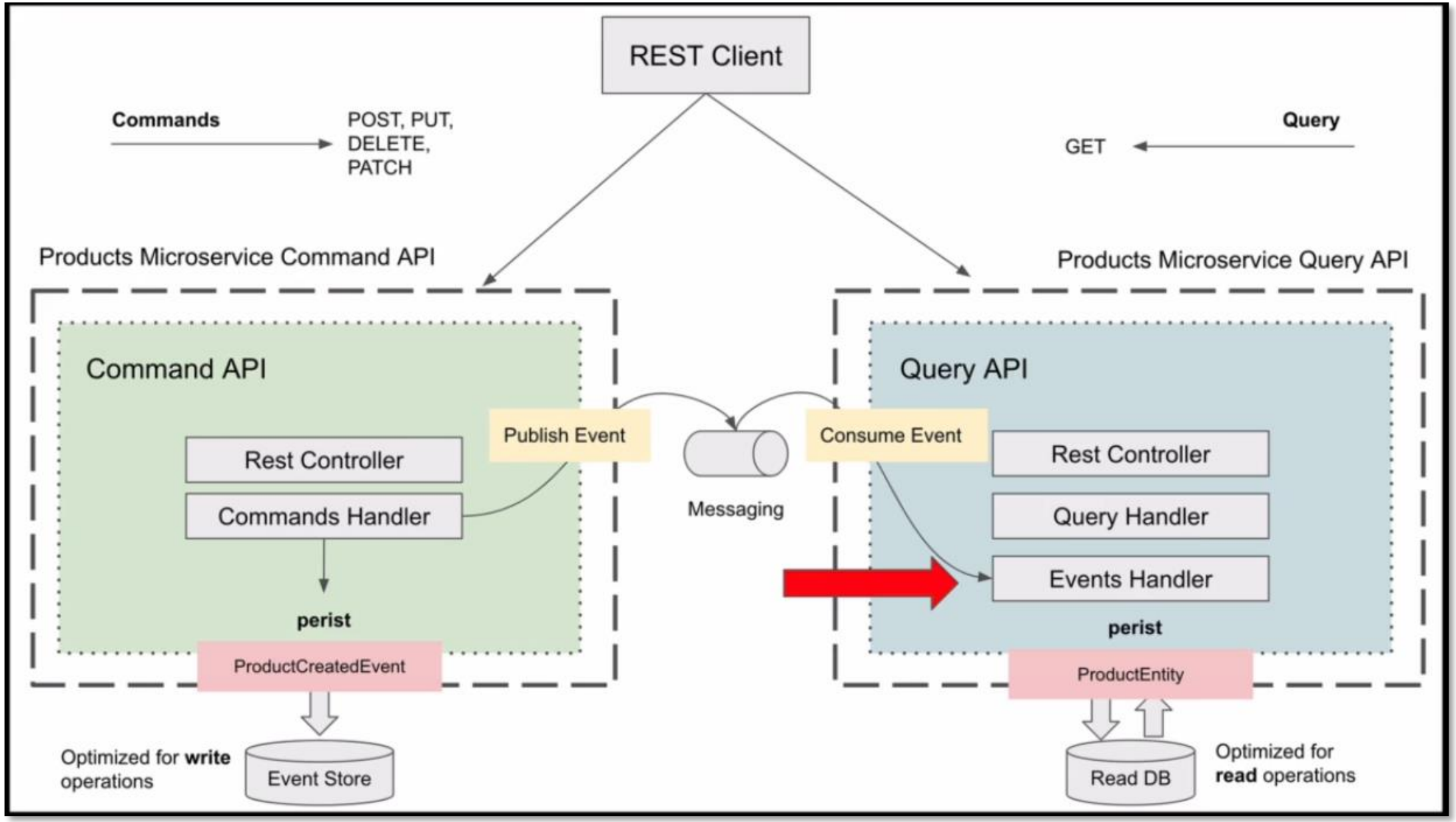
© Mphasis 2020 Proprietary and confidential information

Day - 3

CQRS Persisting Event in the Product Database

© Mphasis 2020 Proprietary and confidential information

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>
```

© Mphasis 2020 Proprietary and confidential information

Configuring database access in application.properties file

- Add the DB properties in application.properties.

application.properties

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

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

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

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

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

spring.jpa.properties.hibernate.show_sql=true
spring.jpa.properties.hibernate.format_sql=true
```

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 }
30
```

Create a Product Repository Interface

- Create a ProductRepository Interface:

ProductRepository.java

```
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

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

© Mphasis 2020 Proprietary and confidential information

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:

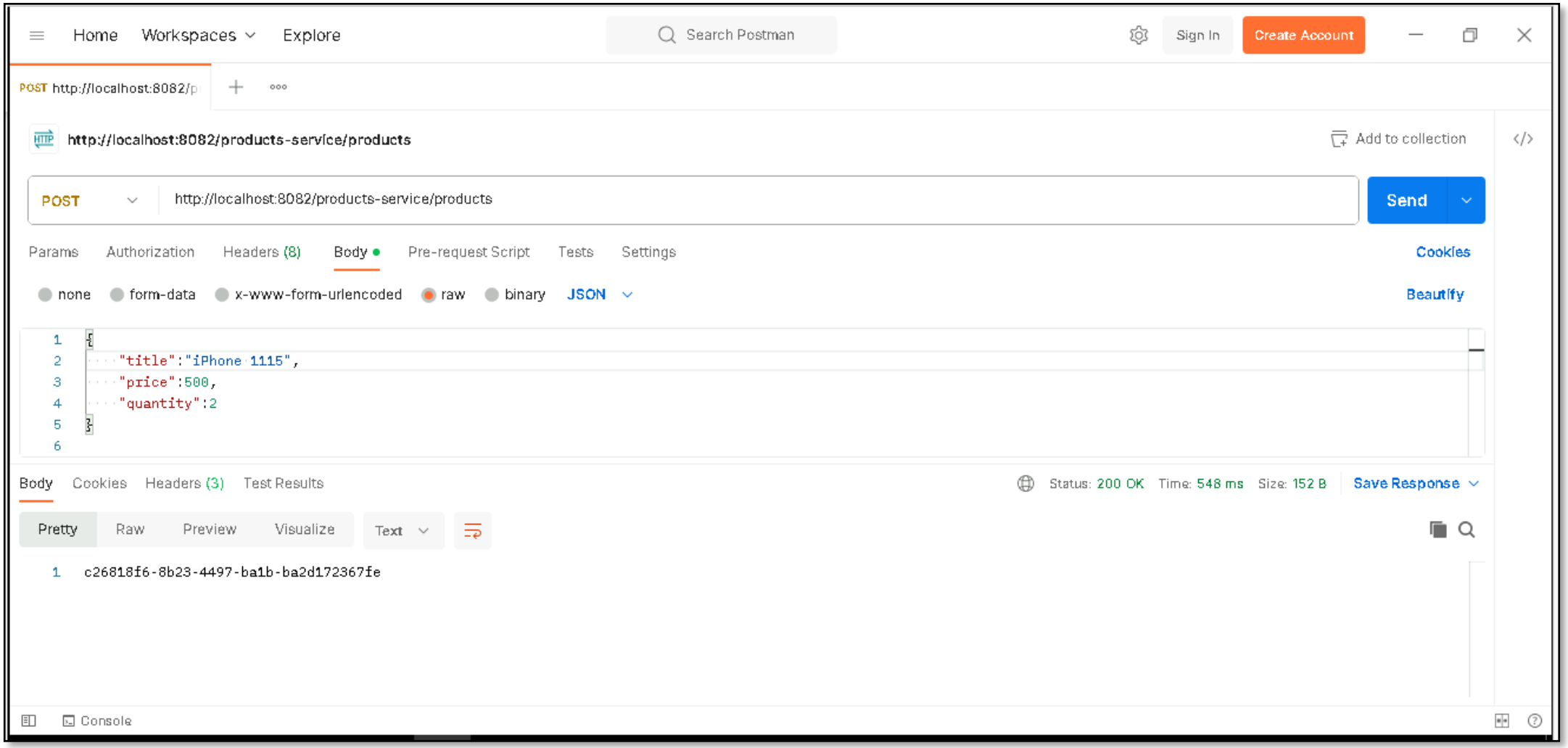
ProductEventsHandler.java

```
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
```

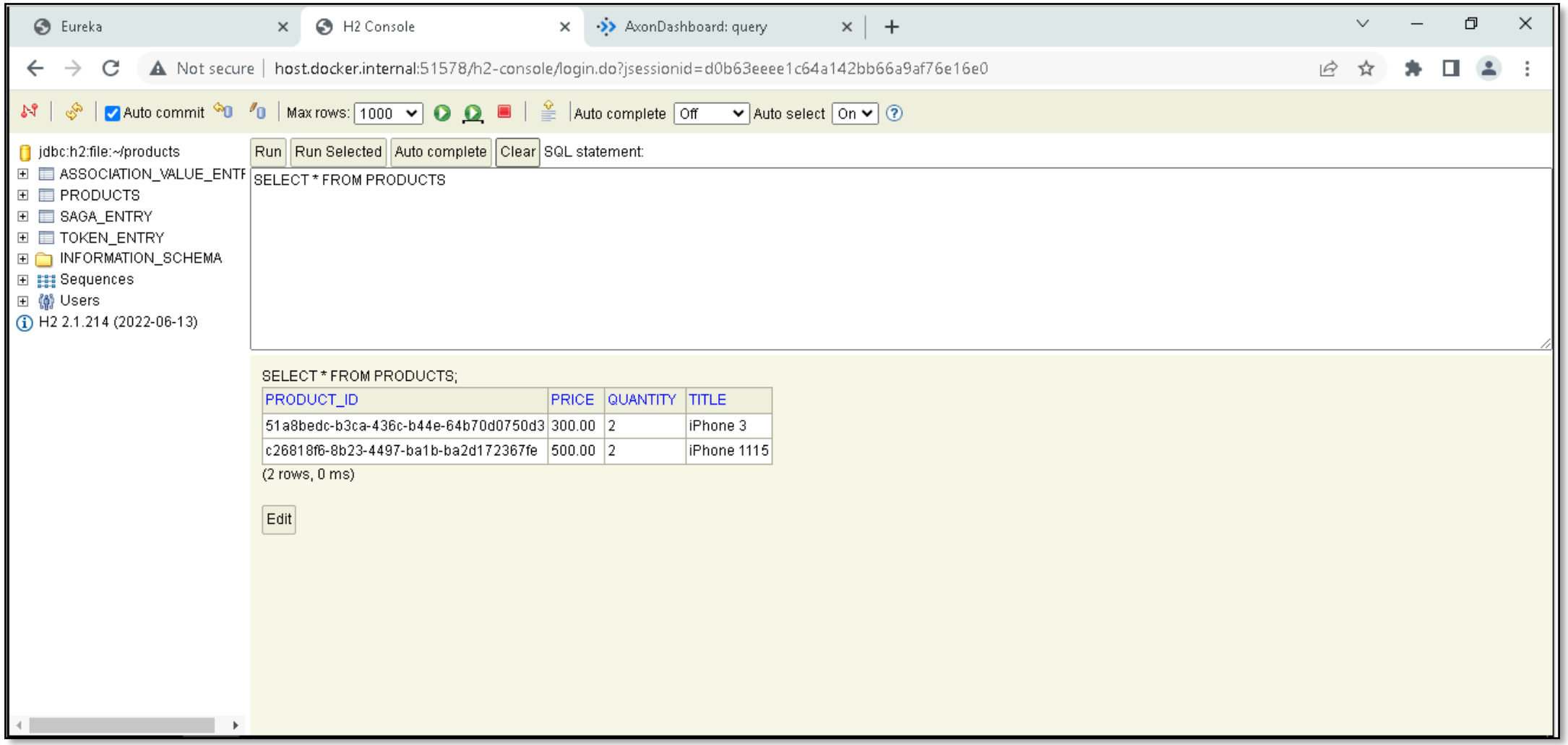
Trying how it works

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



Preview Product record in a Database



Day - 3

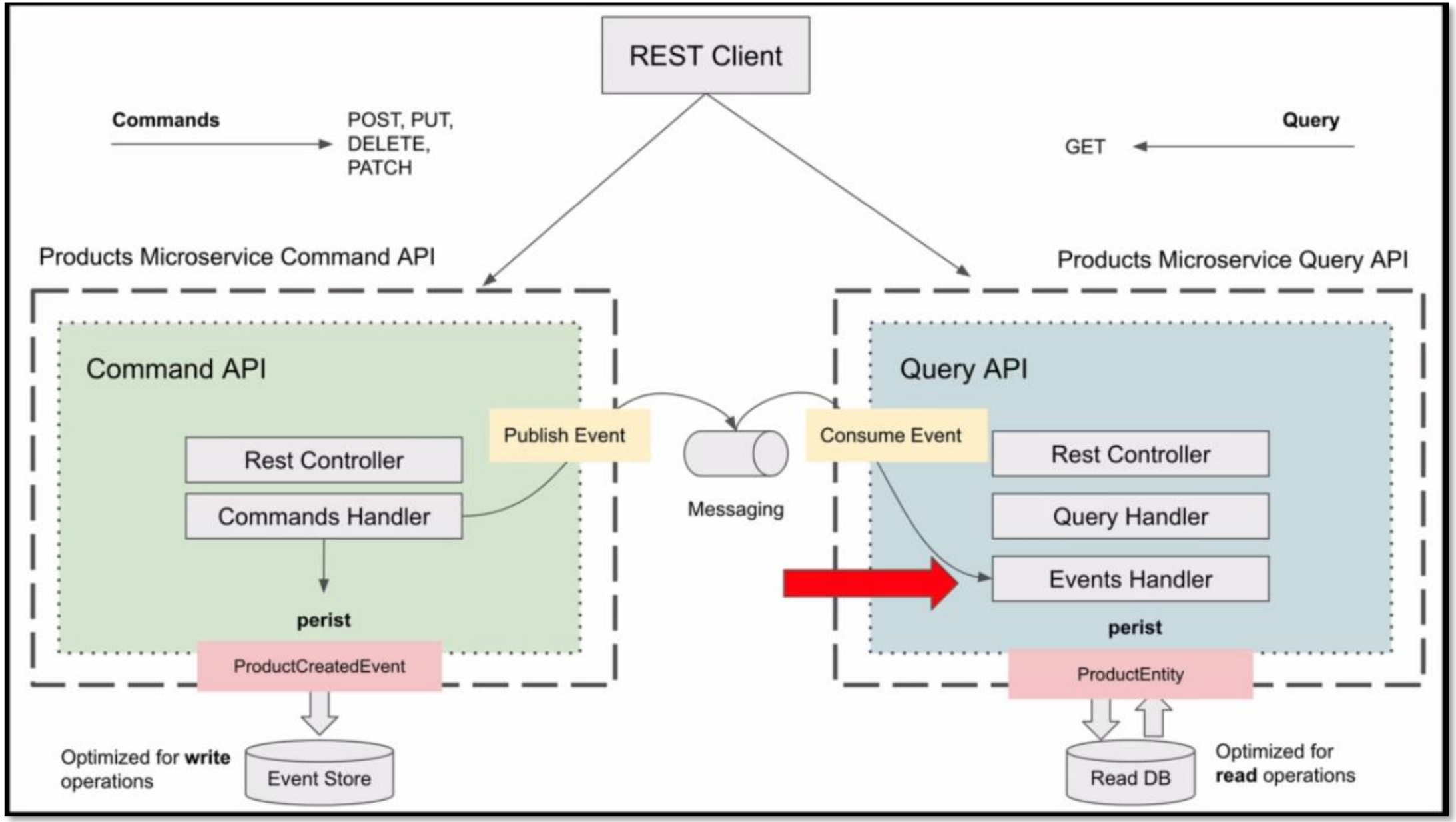
CQRS, Querying Data

© Mphasis 2020 Proprietary and confidential information

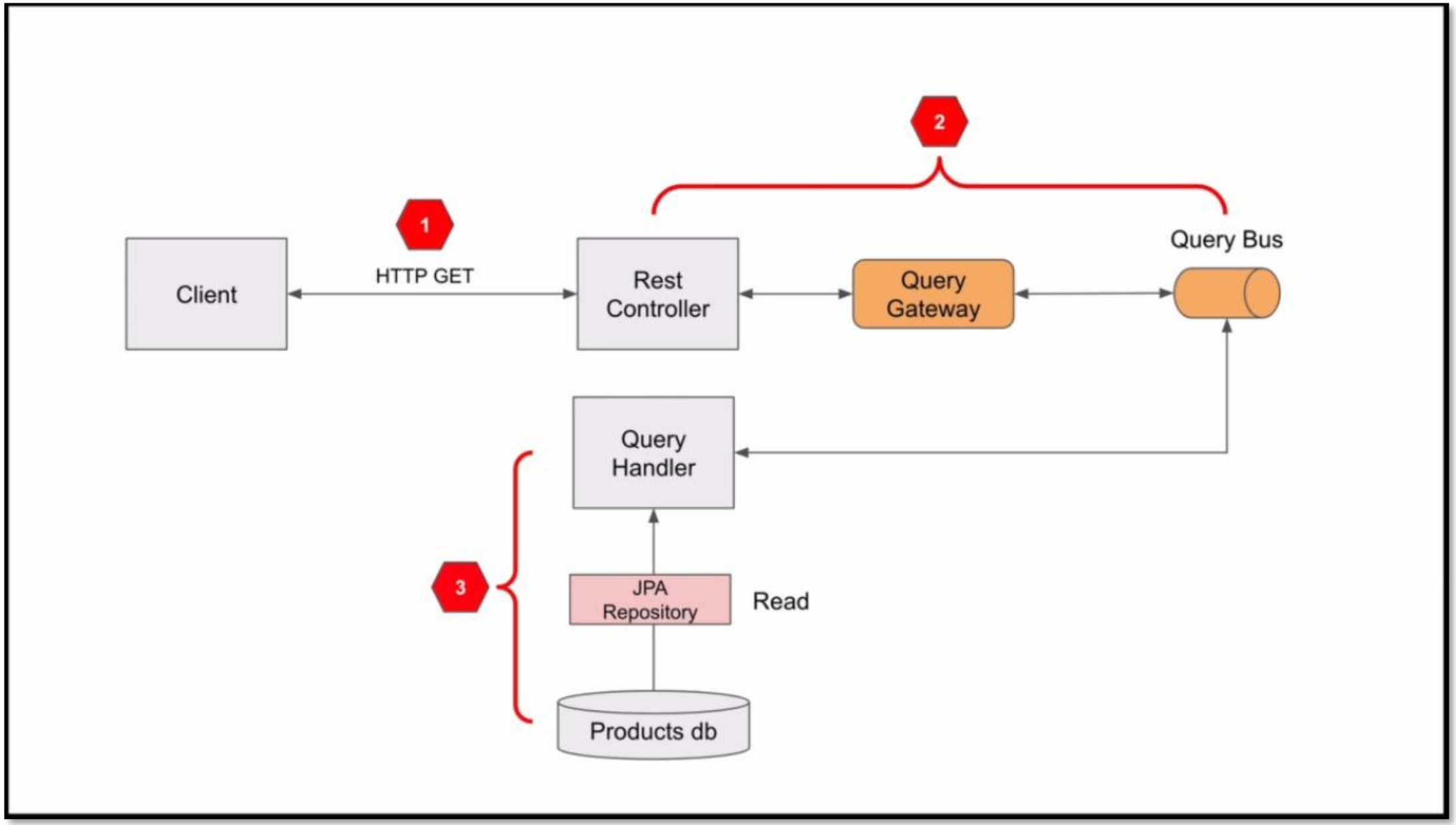
CQRS, Querying Data

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

CQRS, Querying Data



CQRS, Querying Data



Create a Controller class for Query API

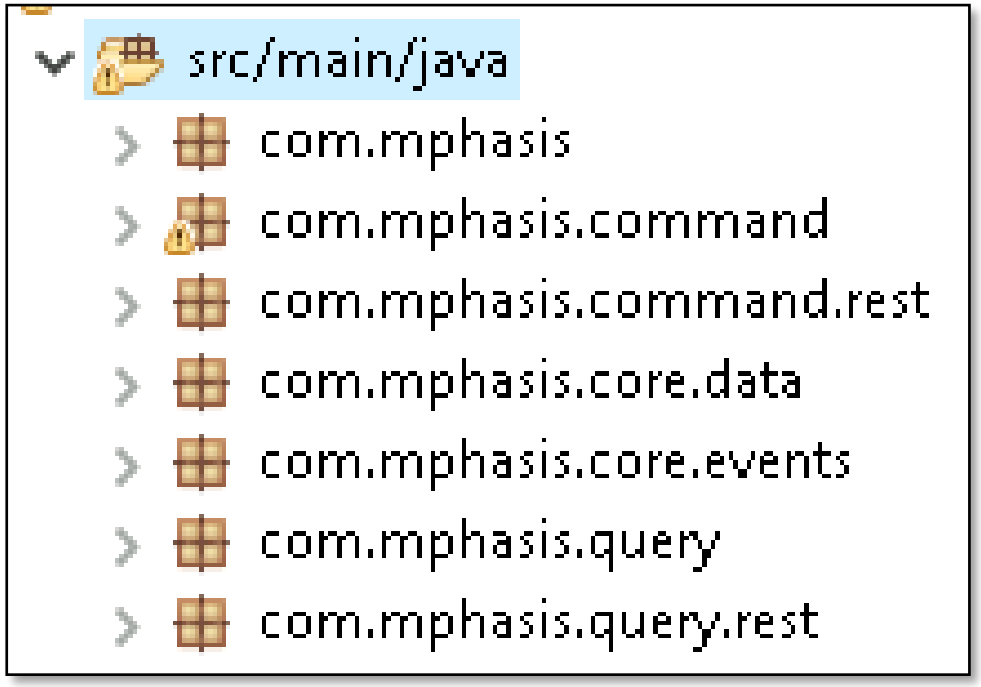
- Create a separate Controller class for Query API:

ProductsQueryController.java

```
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:

ProductRestModel.java

```
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
```

Get Products Web Service Endpoint

- Query the QueryGateway:

FindProductsQuery.java

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

QueryGateway

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

Get Products Web Service Endpoint

- Query the QueryGateway:

ProductsQueryController.java

```java
package com.mphasis.query.rest;

import java.util.List;

@RestController
@RequestMapping("/products")
public class ProductsQueryController {

    @Autowired
    private QueryGateway queryGateway;

    @GetMapping
    public List<ProductRestModel> getProducts() {

        FindProductsQuery findProductsQuery = new FindProductsQuery();
        List<ProductRestModel> products = queryGateway.query(findProductsQuery,
                ResponseTypes.multipleInstancesOf(ProductRestModel.class)).join();

        return products;
    }
}
```

QueryHandler

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

Create a ProductsQueryHandler class

- Create an ProductQueryHandler class:

ProductsQueryHandler.java

```java
package com.mphasis.query;

import org.springframework.stereotype.Component;

import com.mphasis.core.data.ProductRepository;

@Component
public class ProductsQueryHandler {

    private ProductRepository productRepository;

    public ProductsQueryHandler(ProductRepository productRepository) {

        this.productRepository = productRepository;
    }
}

```

Implementing the findProducts() method

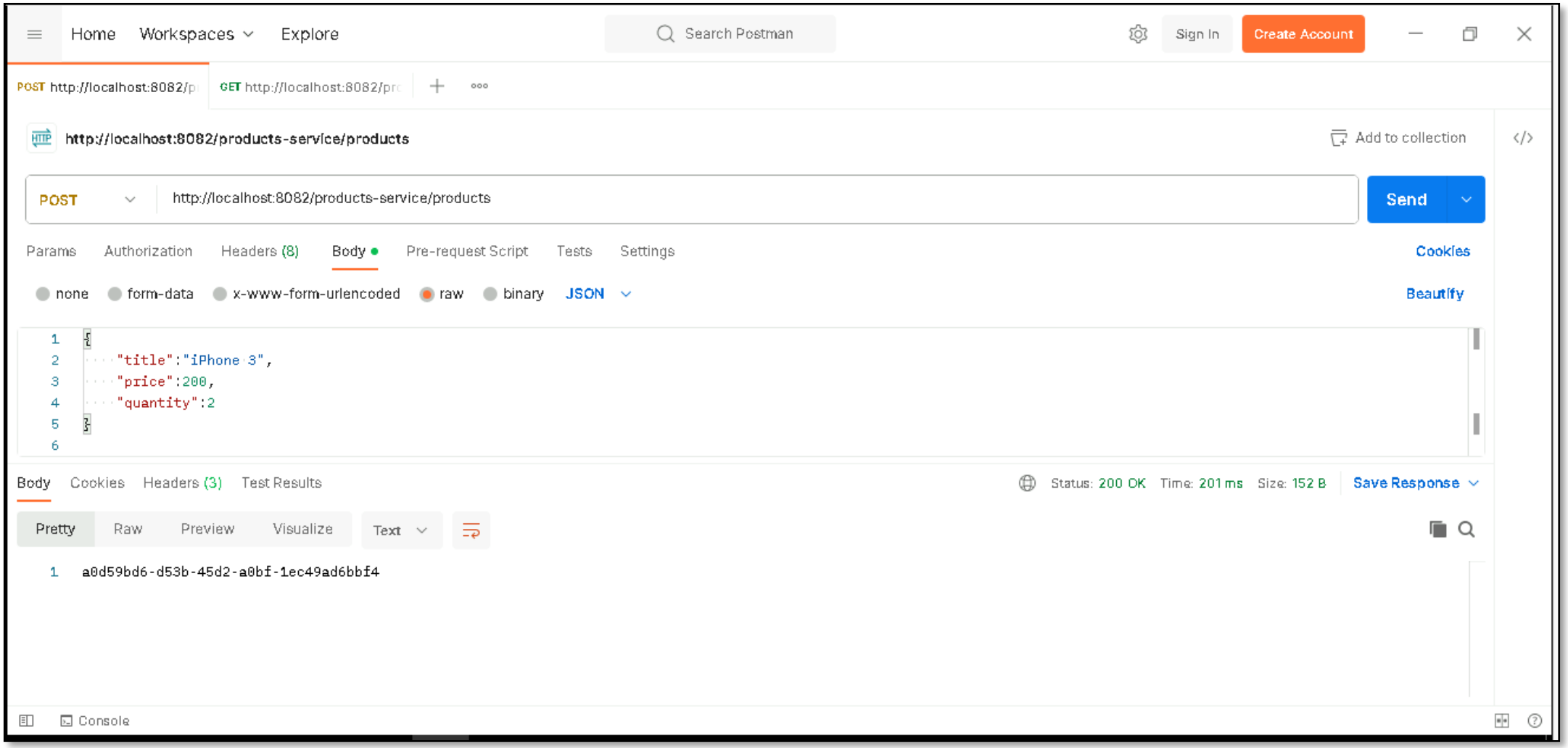
- Implementing the findProducts() method:

```
@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;
}
```

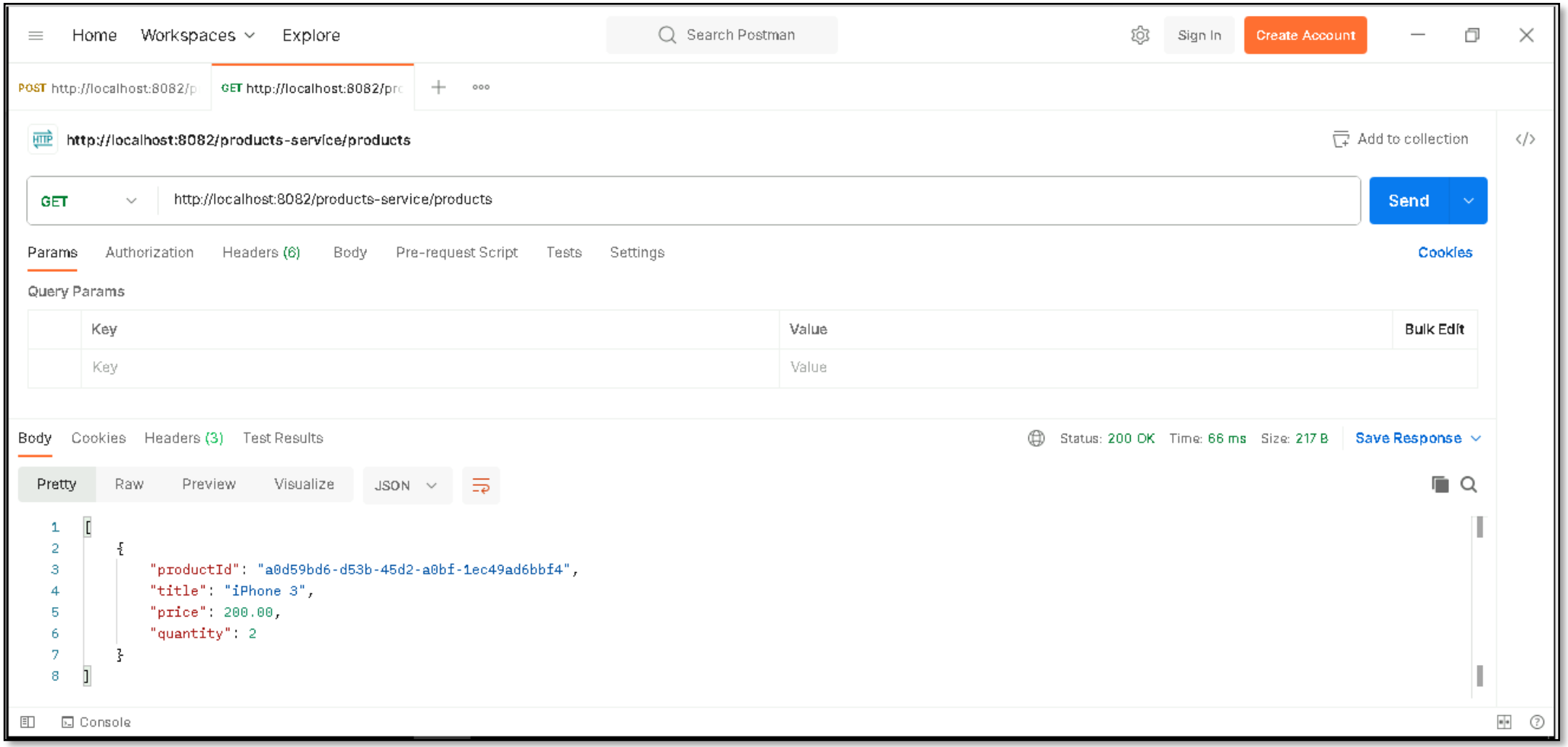
Trying how it works

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.

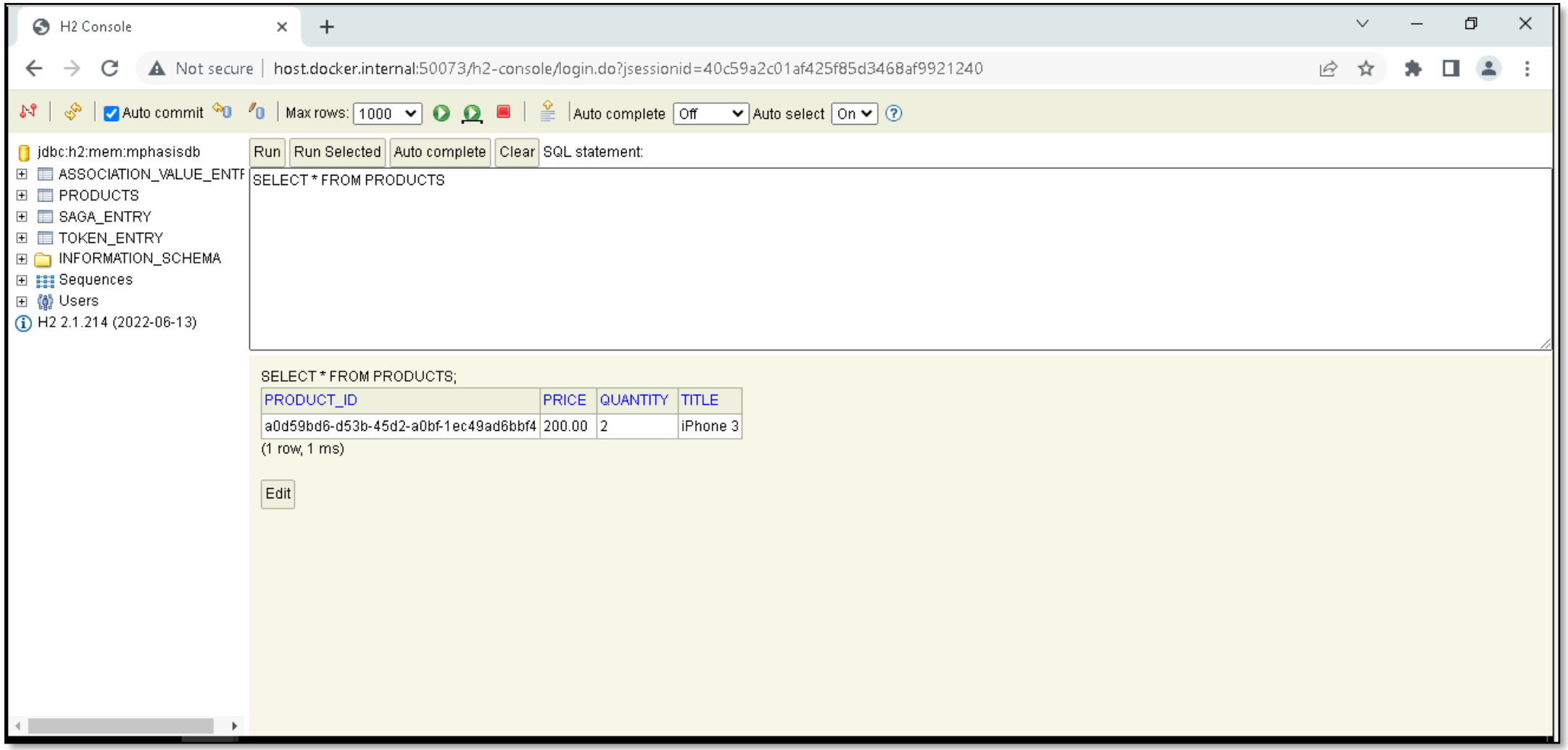
Send a POST request to Create Product



Send a GET request to Query the Products

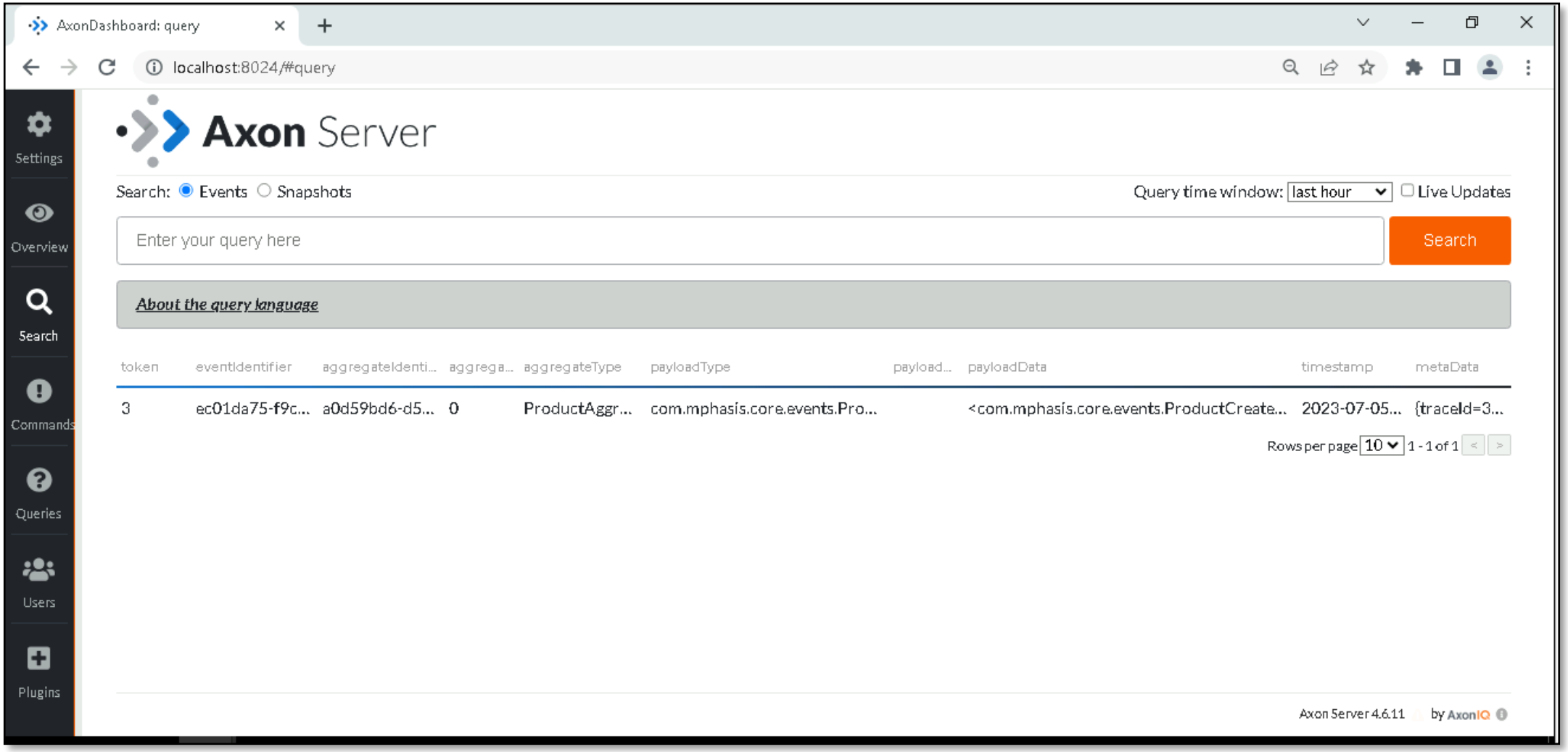


Preview Product record in a Database



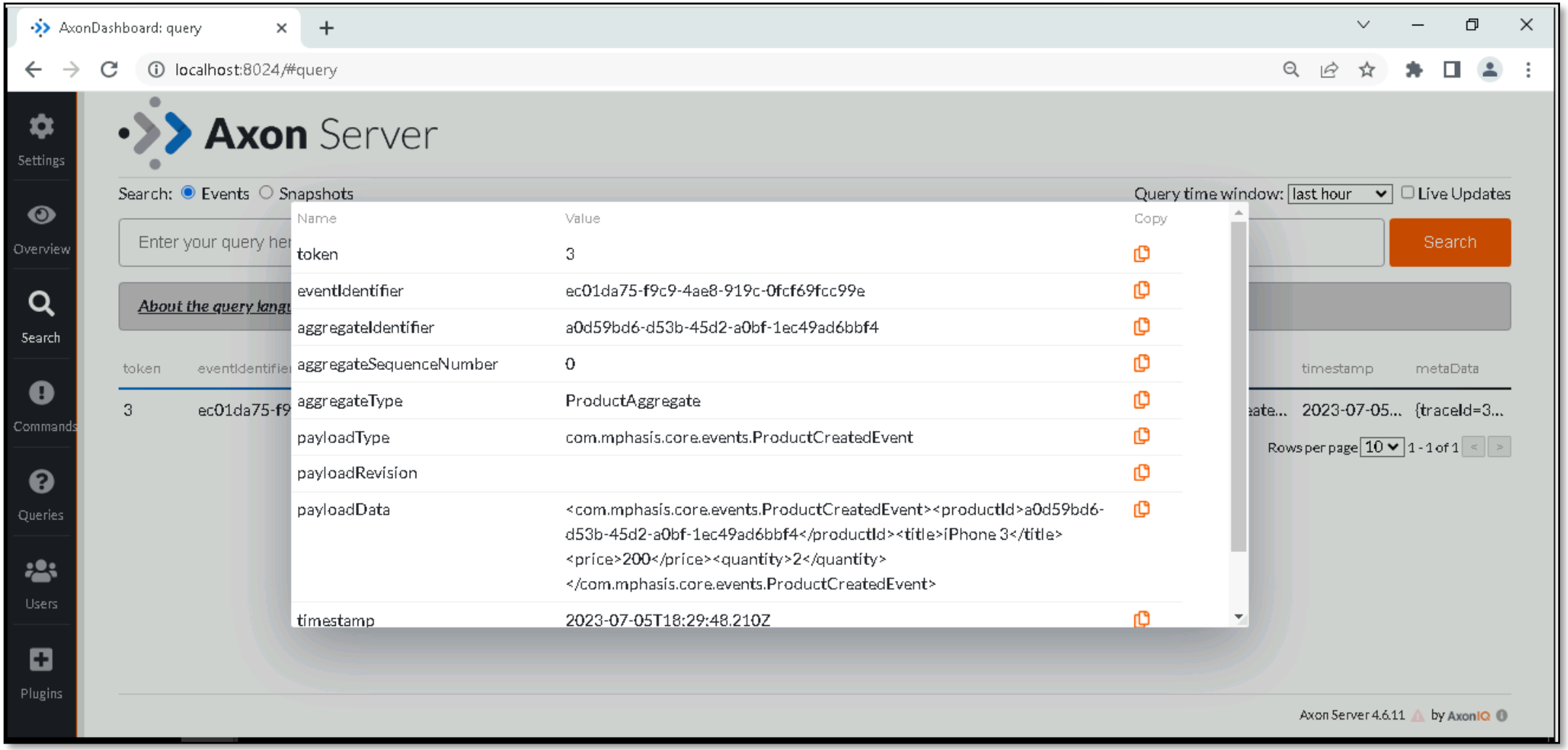
Previewing Event in the EventStore

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



Previewing Event in the EventStore

- View the event details available in Event Store:



Recap of 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 - 4

Day – 4 Agenda

- 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

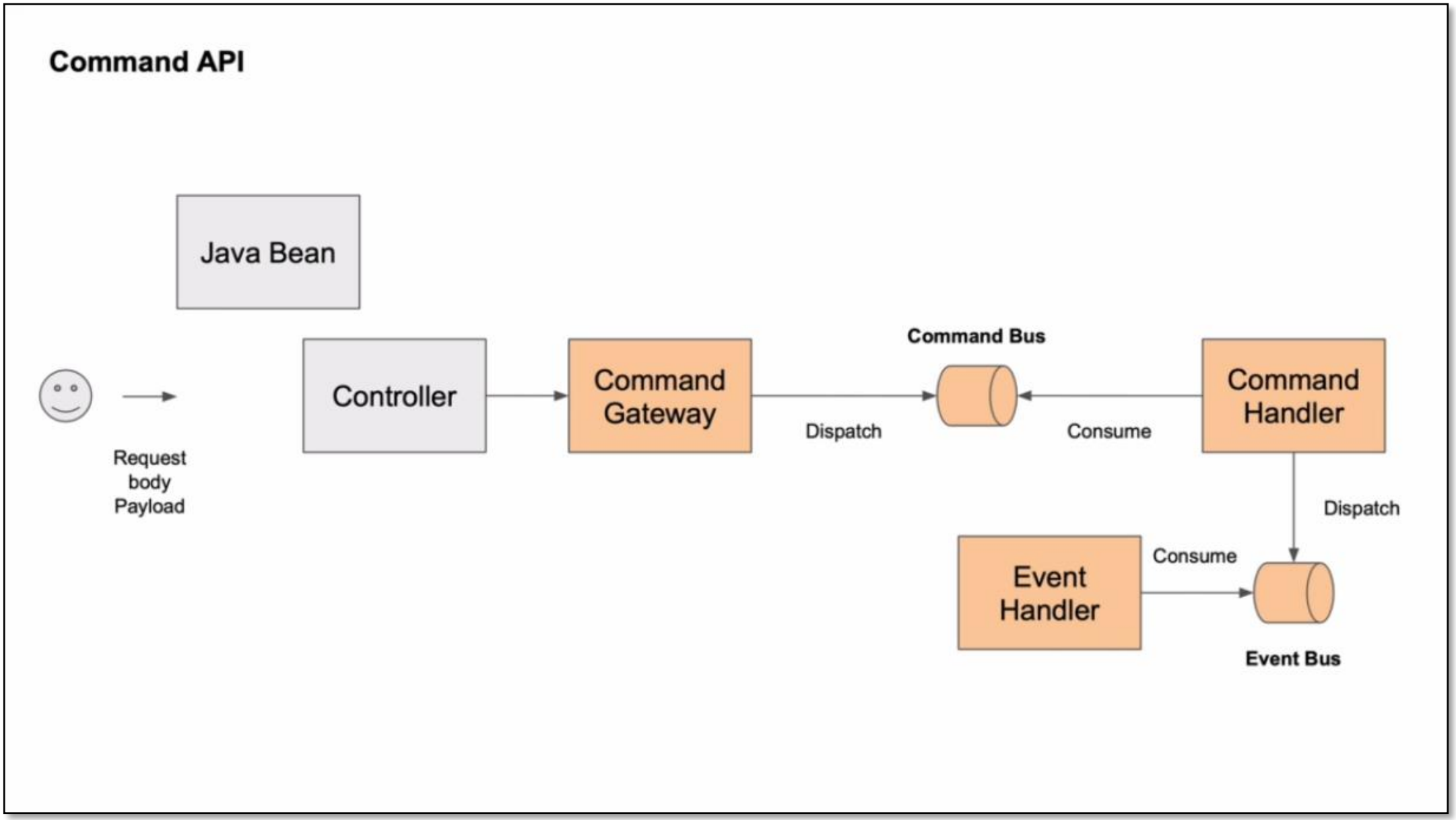
- Introduction to Set Based Consistency
- Create a Product Lookup Entity
- Create a Product Lookup Repository
- Create a Product Lookup Event Handler
- Persisting information into a ProductLookup table
- Update the MessageDispatchInterceptor class

Day - 4

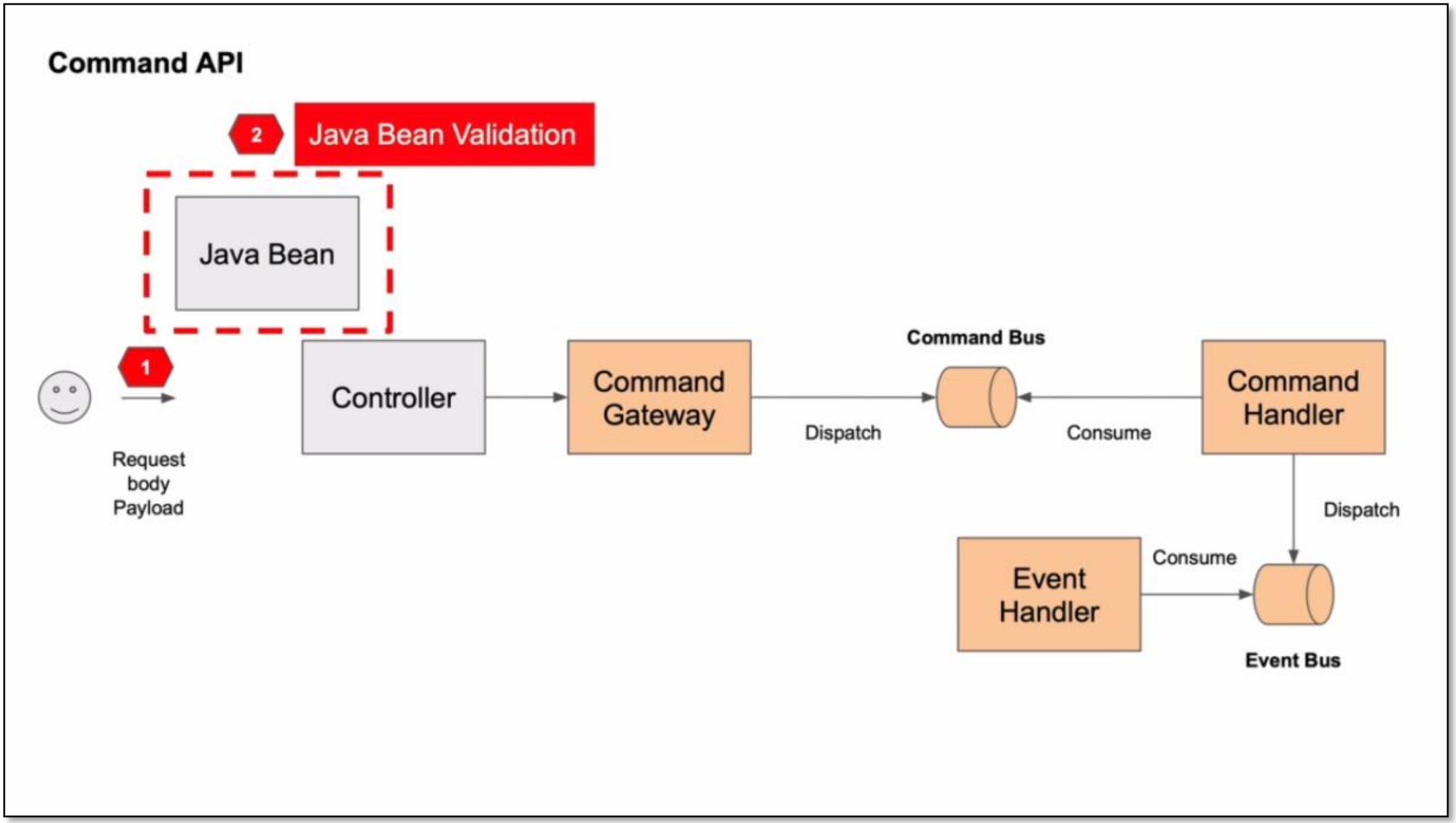
Validating Request Body, Bean Validation

© Mphasis 2020 Proprietary and confidential information

Validating Request Body, Bean Validation



Validating Request Body, Bean Validation



Bean Validation – with Hibernate Validation

```
@Data
public class CreateProductRestModel {

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

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

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

}
```

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

```
1
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
```

Bean Validation. Validating Request Body

- Add the validation annotation to the fields:

CreateProductRestModel.java

```
1  package com.mphasis.command.rest;
2
3  import java.math.BigDecimal;
10
11 @Data
12 public class CreateProductRestModel {
13
14     @NotBlank(message = "Product title is a required field")
15     private String title;
16
17     @Min(value=1, message = "Price cannot be lower than 1")
18     private BigDecimal price;
19
20     @Min(value=1, message = "Quantity cannot be lower than 1")
21     @Max(value=5, message = "Quantity cannot be larger than 1")
22     private Integer quantity;
23 }
24
```

Bean Validation. Validating Request Body

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

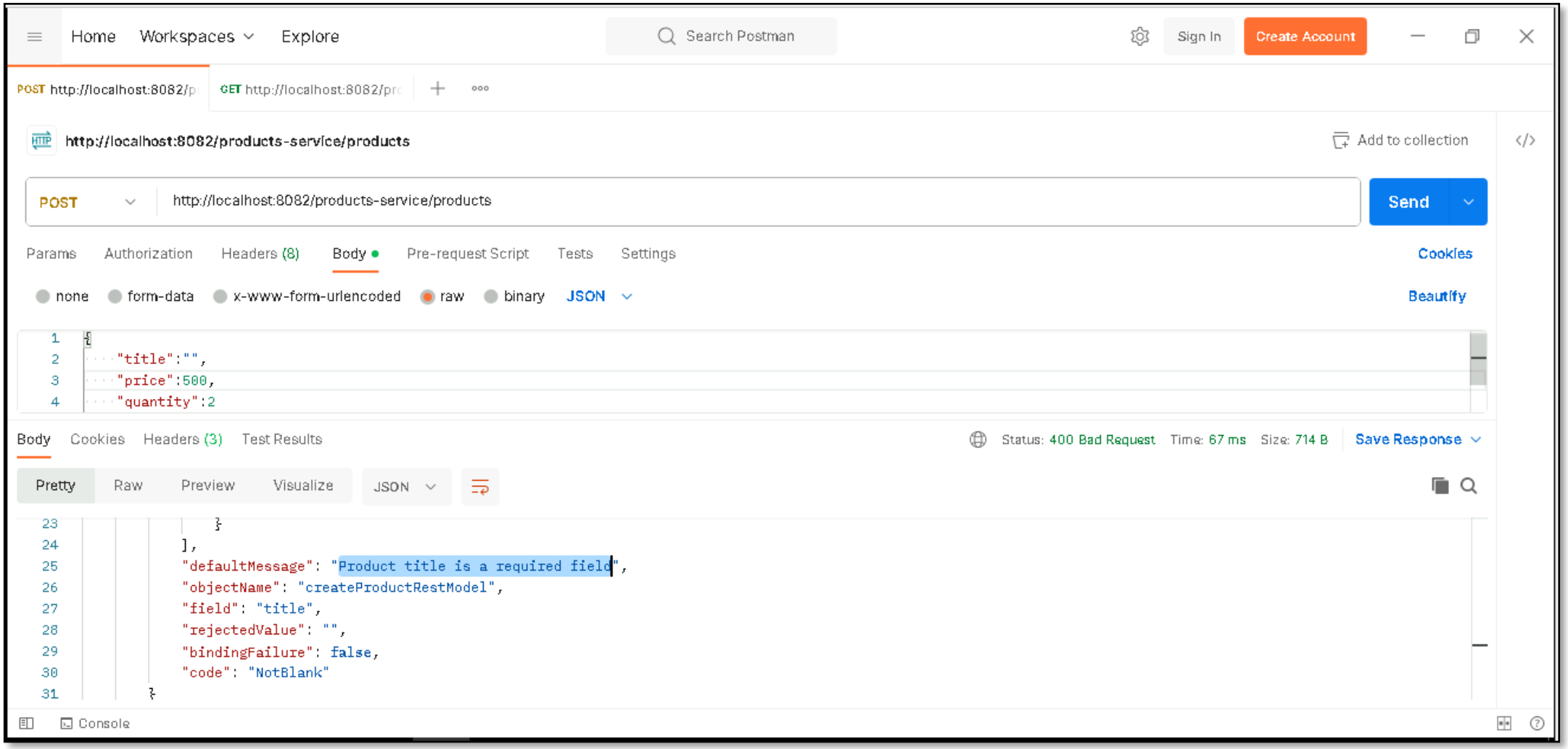
ProductCommandController.java

```
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
```

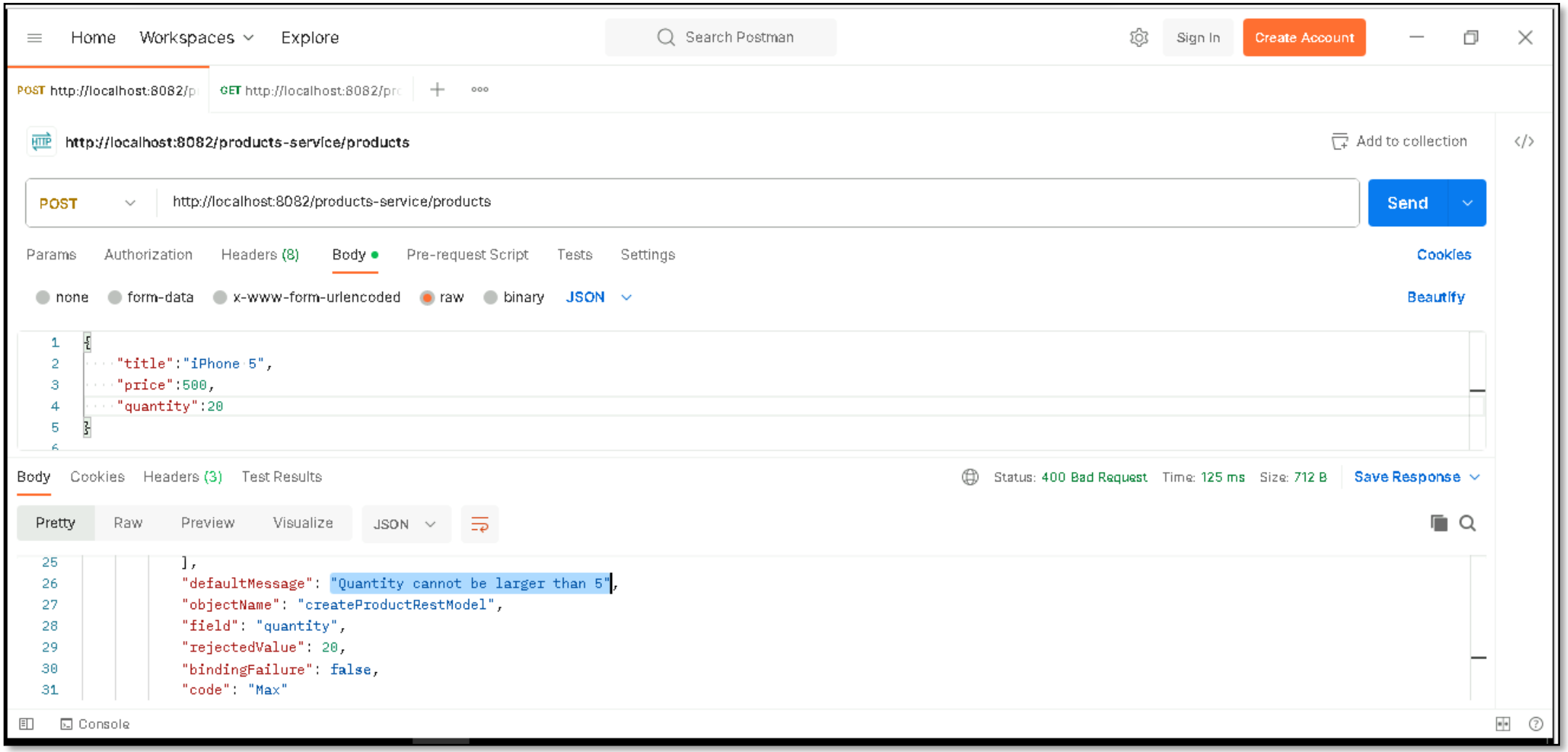
Trying how it works

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



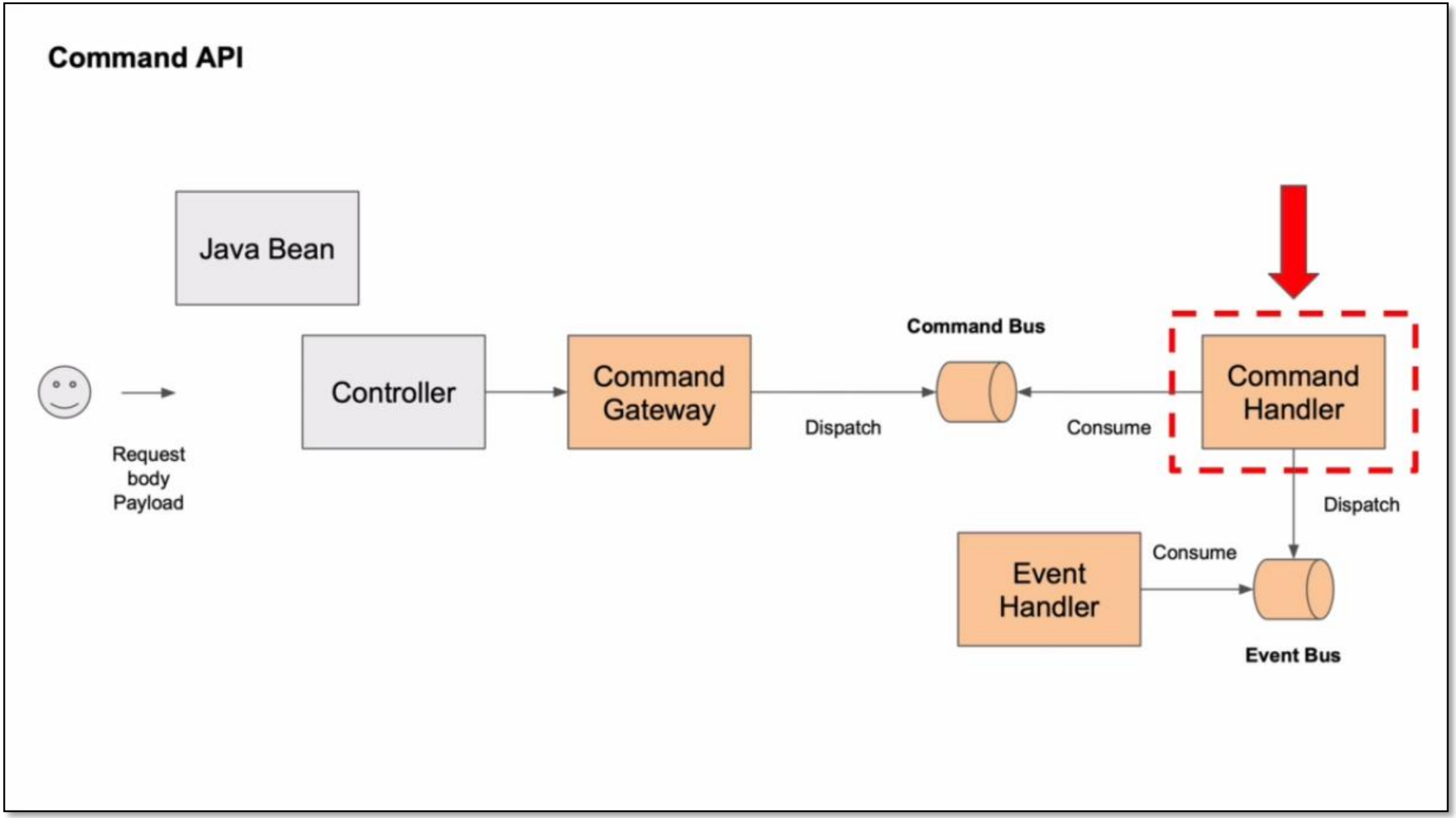
Send a POST request with Quantity greater than 20



Day - 4

Validation in the @CommandHandler method

Validation in the @CommandHandler method



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

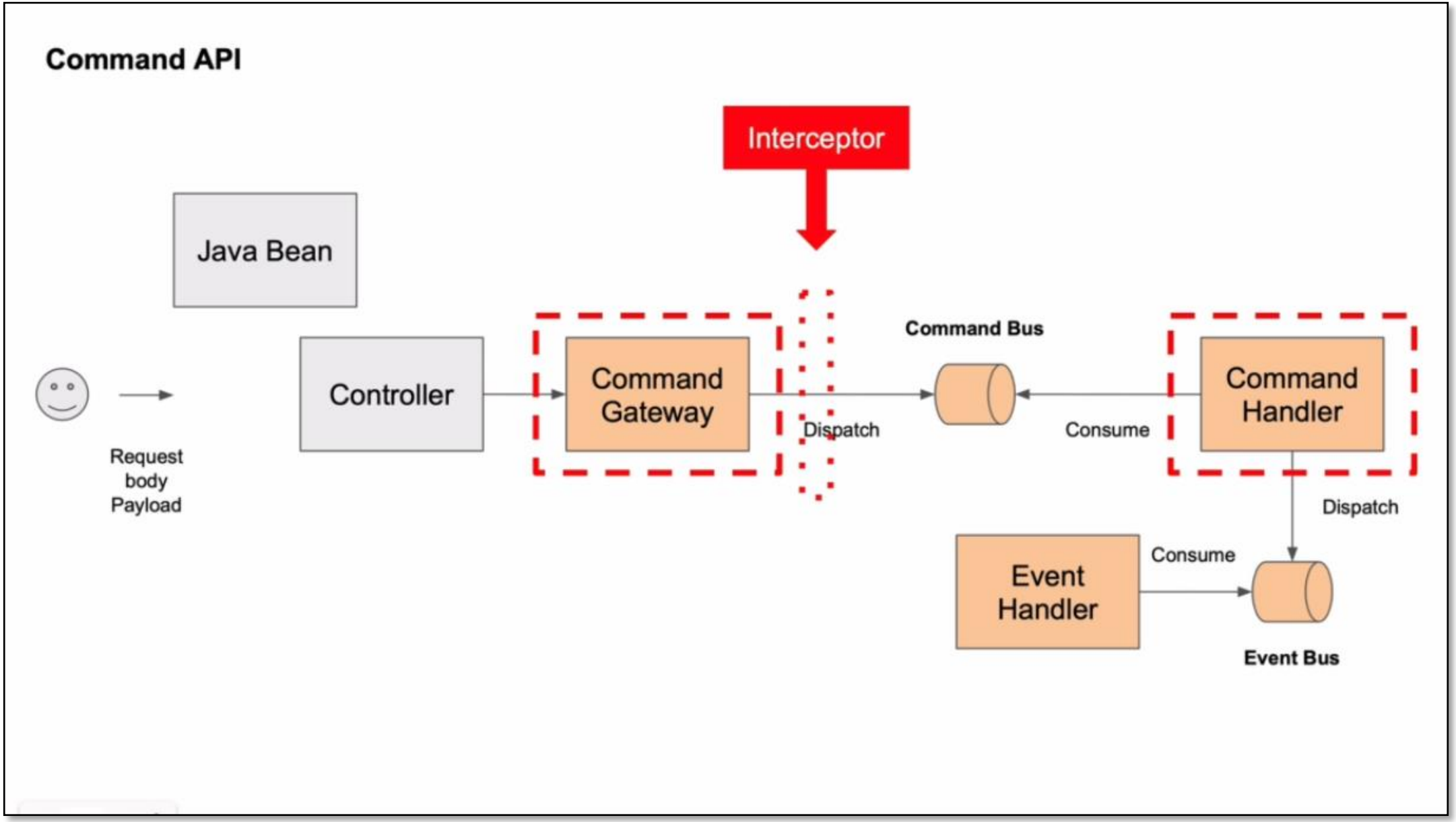
Message Dispatch Interceptor

© Mphasis 2020 Proprietary and confidential information

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



Creating a new Command Interceptor class

- Create a new CreateProductCommandInterceptor class:

```java
@Component
public class CreateProductCommandInterceptor implements MessageDispatchInterceptor<CommandMessage<?>>{

    private static final Logger LOGGER = LoggerFactory.getLogger(CreateProductCommandInterceptor.class);

    @Override
    public BiFunction<Integer, CommandMessage<?>, CommandMessage<?>> handle(
            List<? extends CommandMessage<?>> messages) {

        return (index, command) -> {

            LOGGER.info("Intercepted command: " + command.getPayloadType());

            if(CreateProductCommand.class.equals(command.getPayloadType())) {

                CreateProductCommand createProductCommand = (CreateProductCommand)command.getPayload();

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

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

Register Message Dispatch Interceptor

- Register the CreateProductCommandInterceptor class:

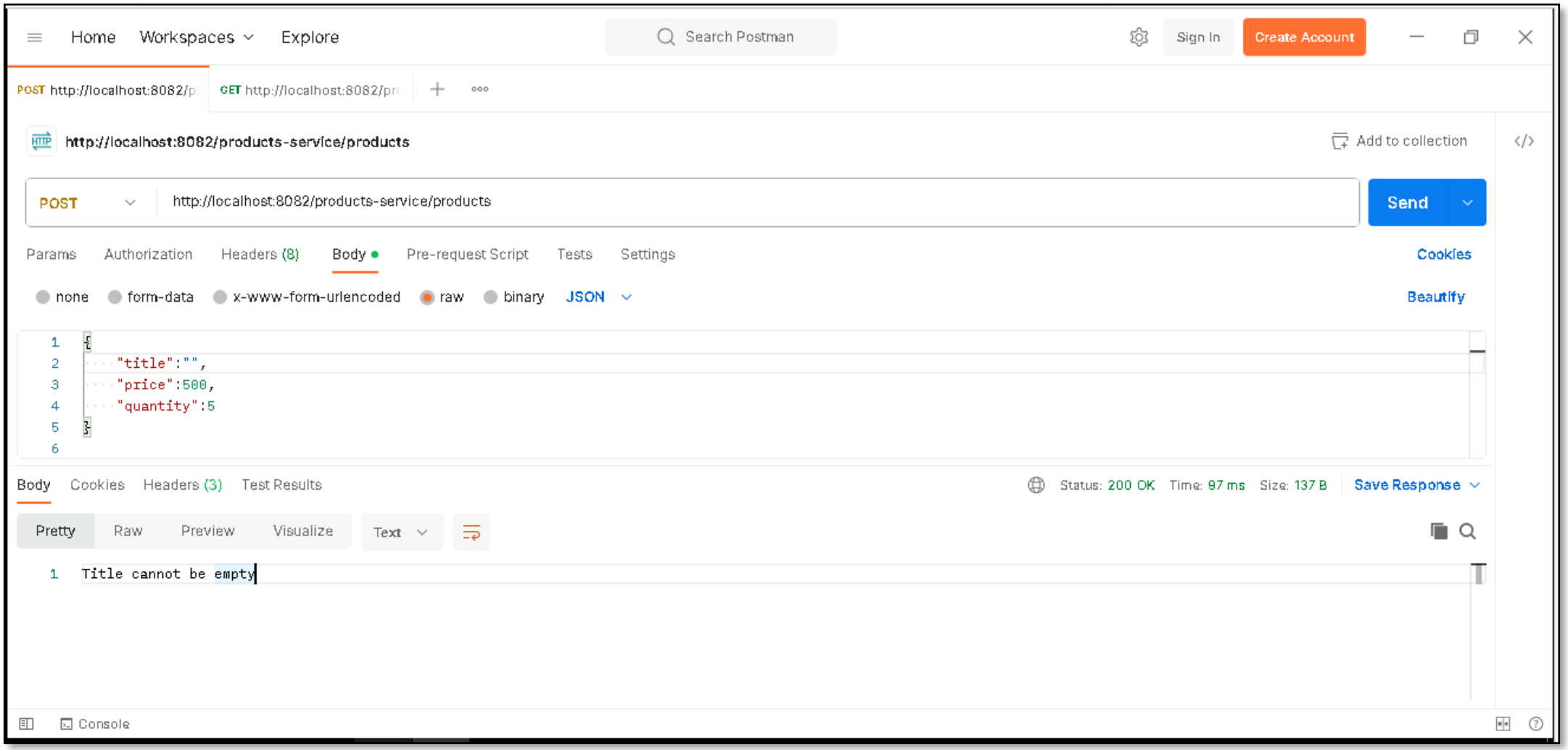
ProductServiceApplication.java

```
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
import org.springframework.context.ApplicationContext;

import com.mphasis.command.interceptor.CreateProductCommandInterceptor;

@EnableDiscoveryClient
@SpringBootApplication
public class ProductServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(ProductServiceApplication.class, args);
    }

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

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

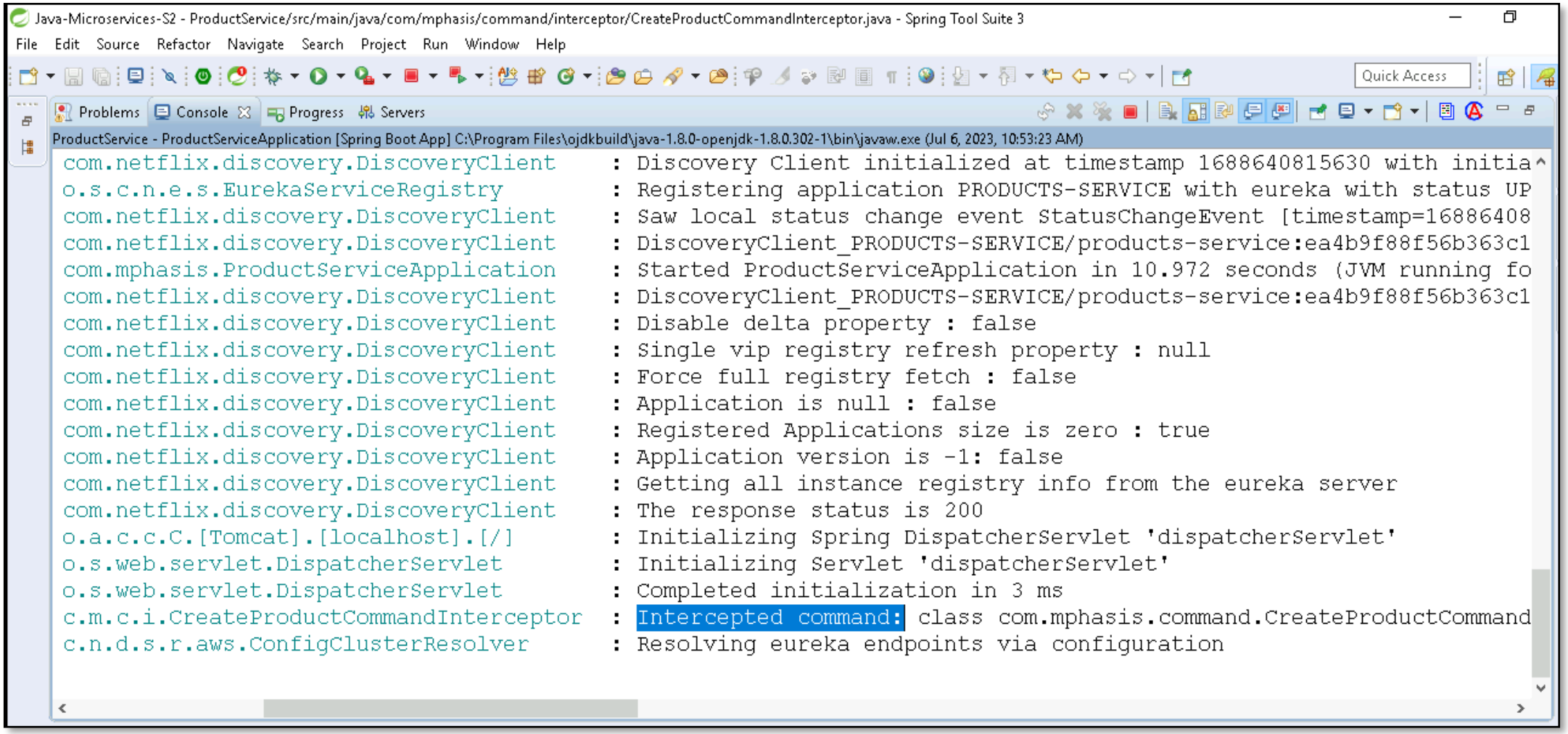
Trying how it works

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



Check the ProductServiceApplication Console

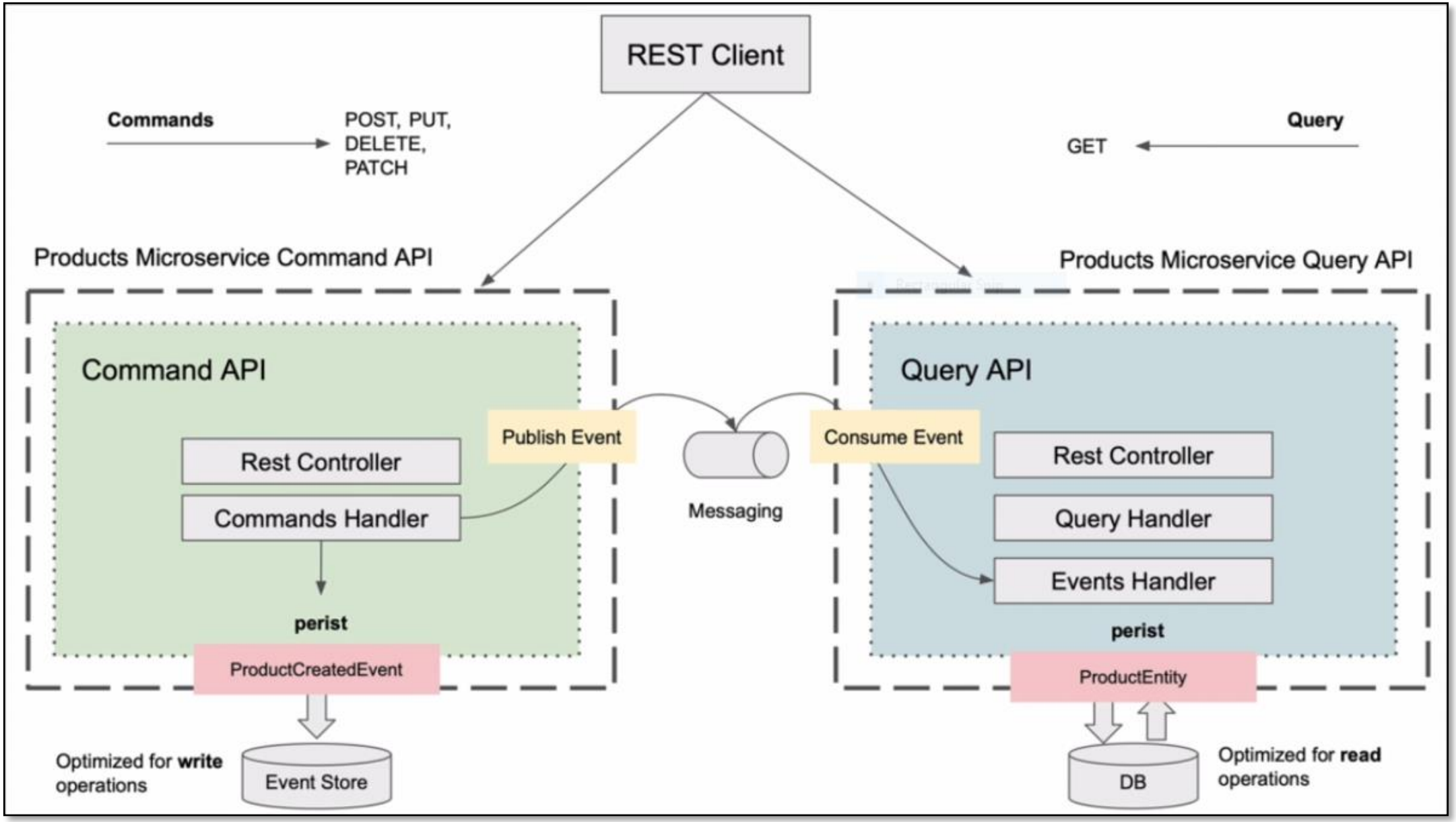


Day - 4

Set Based Consistency

© Mphasis 2020 Proprietary and confidential information

Introduction to Set Based Consistency



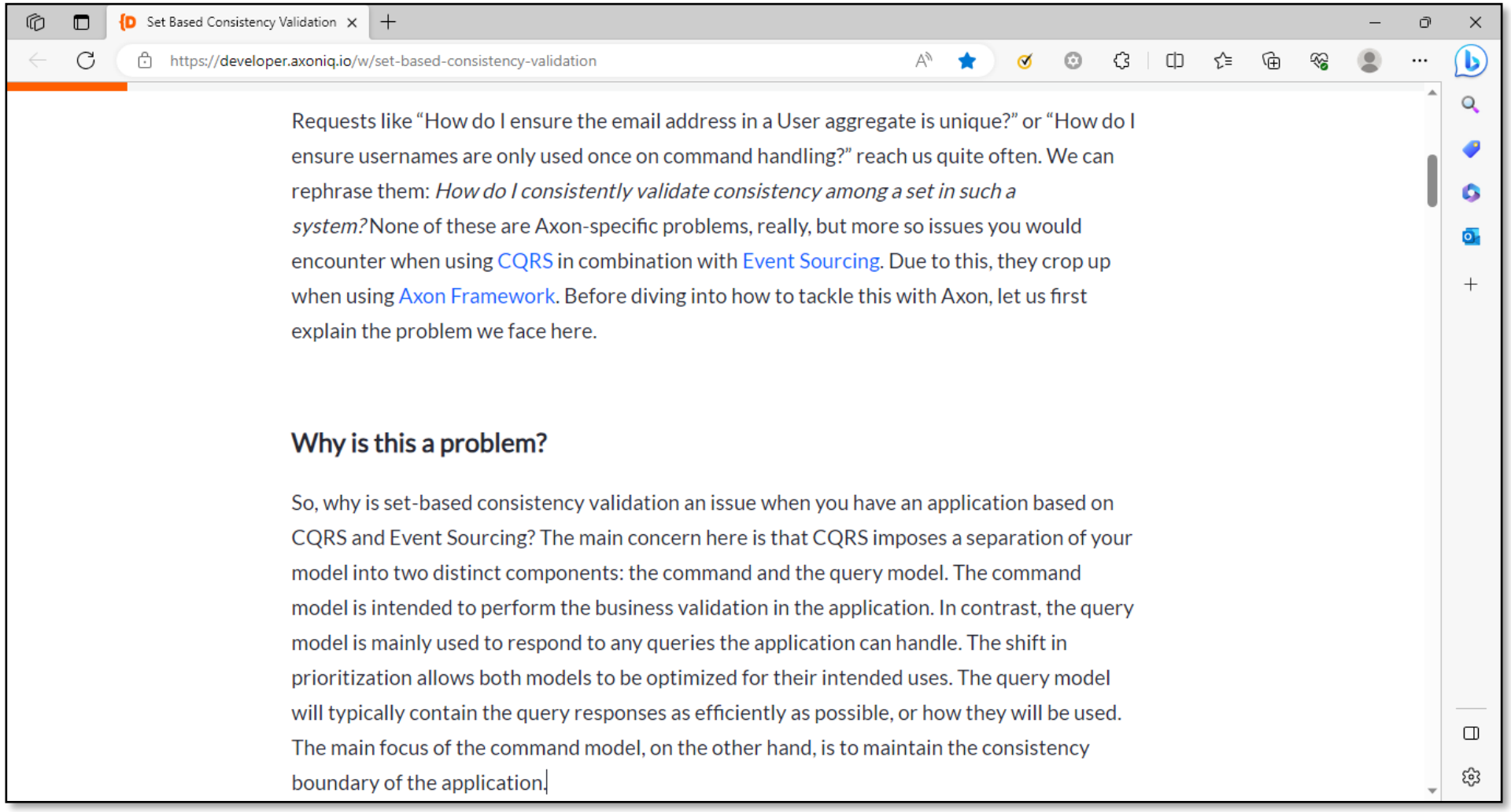
Introduction to Set Based Consistency

- 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

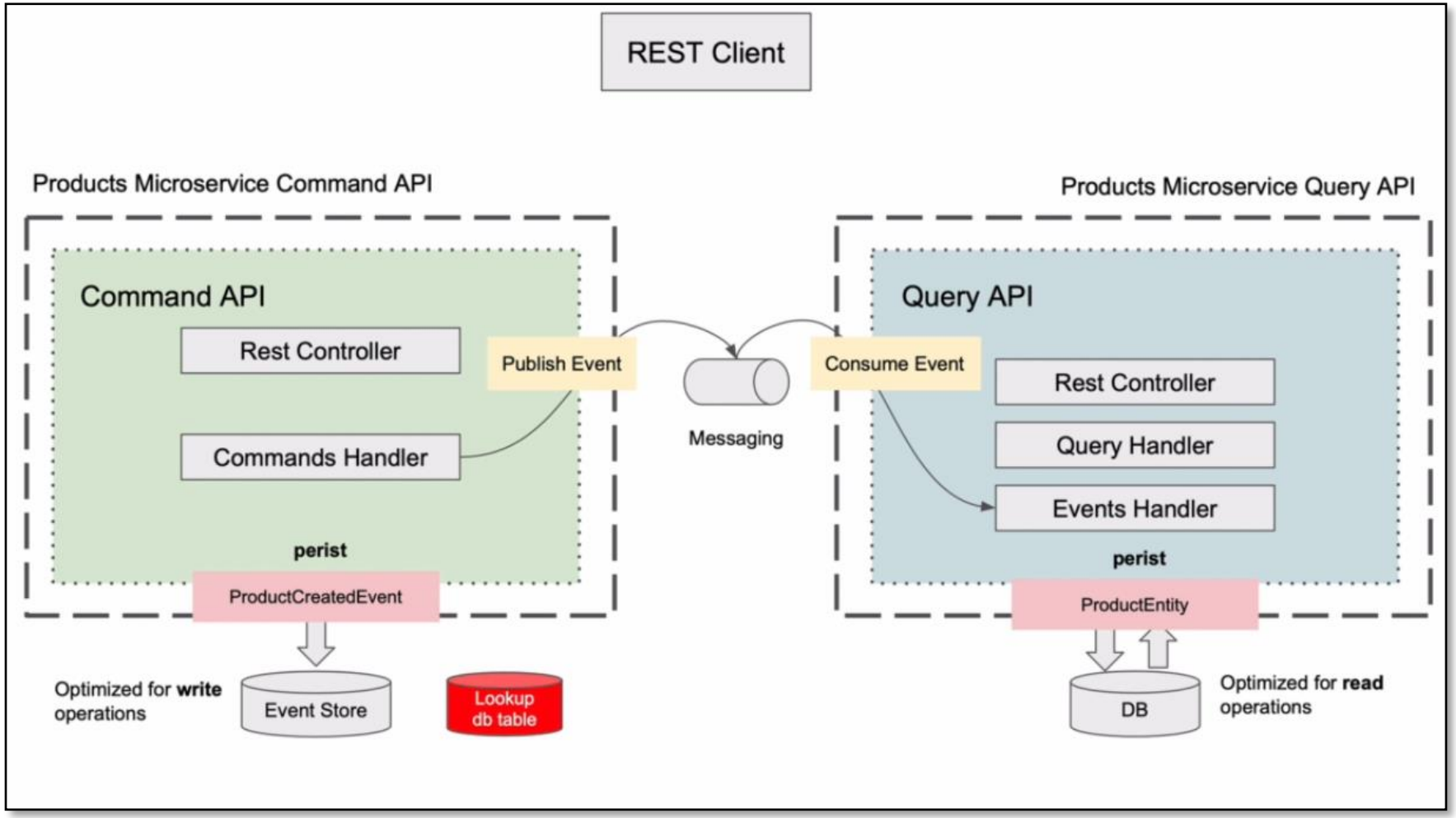
- Go to https://developer.axoniq.io/w/set-based-consistency-validation



© Mphasis 2020 Proprietary and confidential information

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.



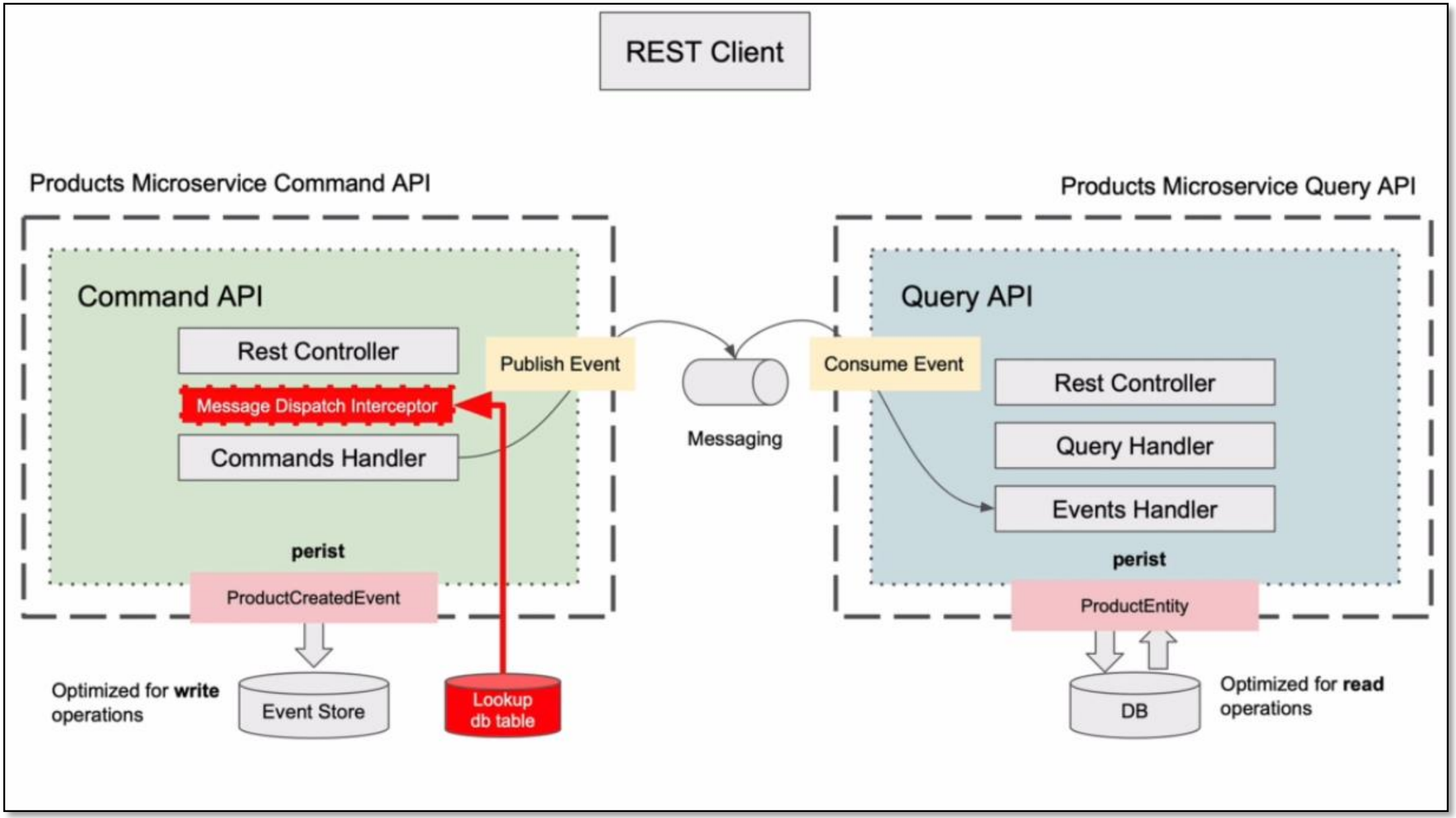
Lookup Table

- 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.

Set Based Consistency

- 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.

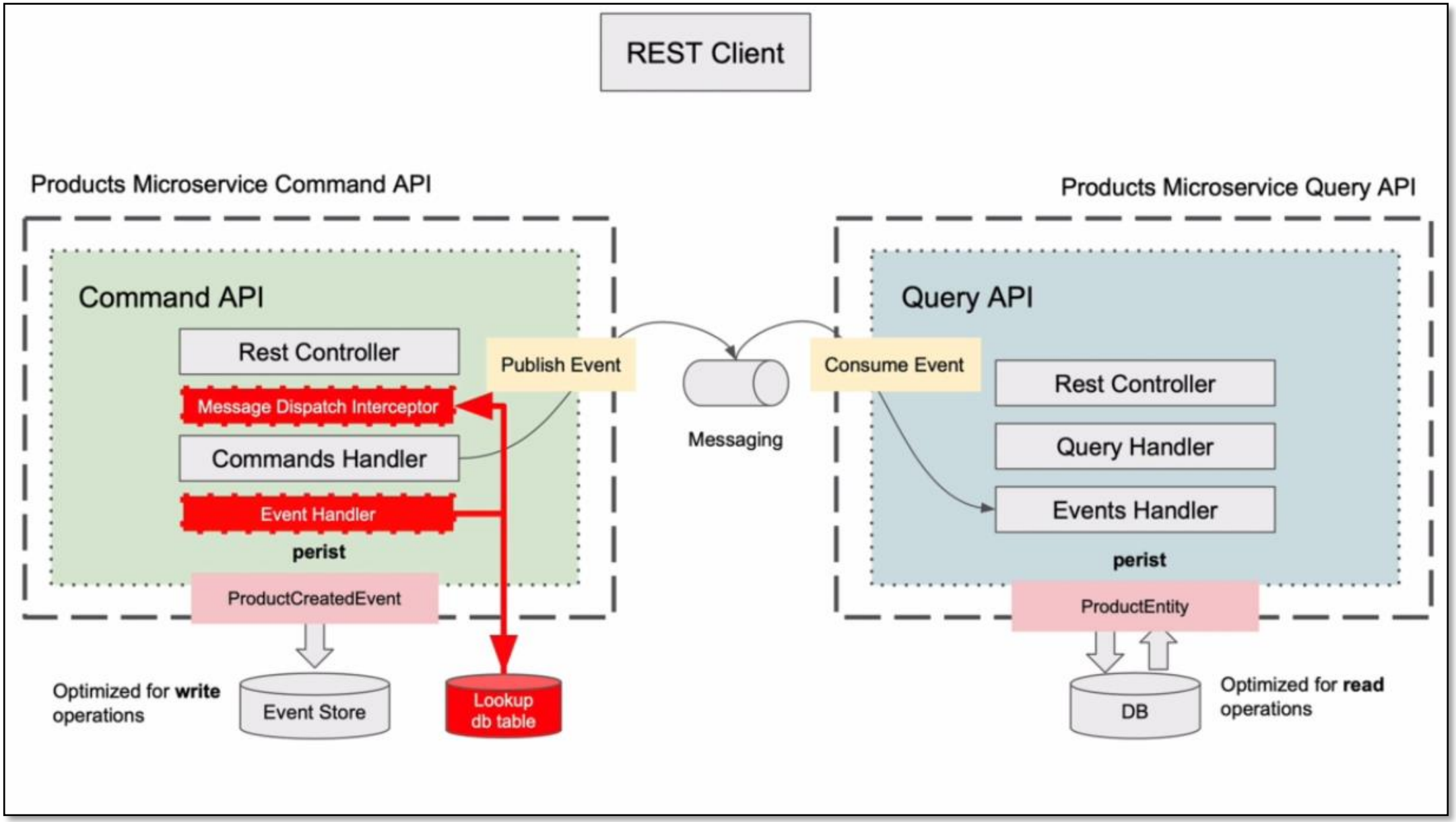
Set Based Consistency



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;
13
14 @Data
15 @NoArgsConstructor
16 @AllArgsConstructor
17 @Entity
18 @Table(name = "productlookup")
19 public class ProductLookupEntity implements Serializable {
20
21     private static final long serialVersionUID = -8509850126045725343L;
22
23     @Id
24     private String productId;
25
26     @Column(unique = true)
27     private String title;
28 }
29
30
```

Create a ProductLookupRepository class

- Create a ProductLookupRepository class:

ProductLookupRepository.java

```
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:

ProductLookupEventsHandler.java

```
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

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

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

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

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

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

```

© Mphasis 2020 Proprietary and confidential information

Persisting information into a ProductLookup table

- Persisting information into a ProductLookup table:

ProductLookupEventsHandler.java

```
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

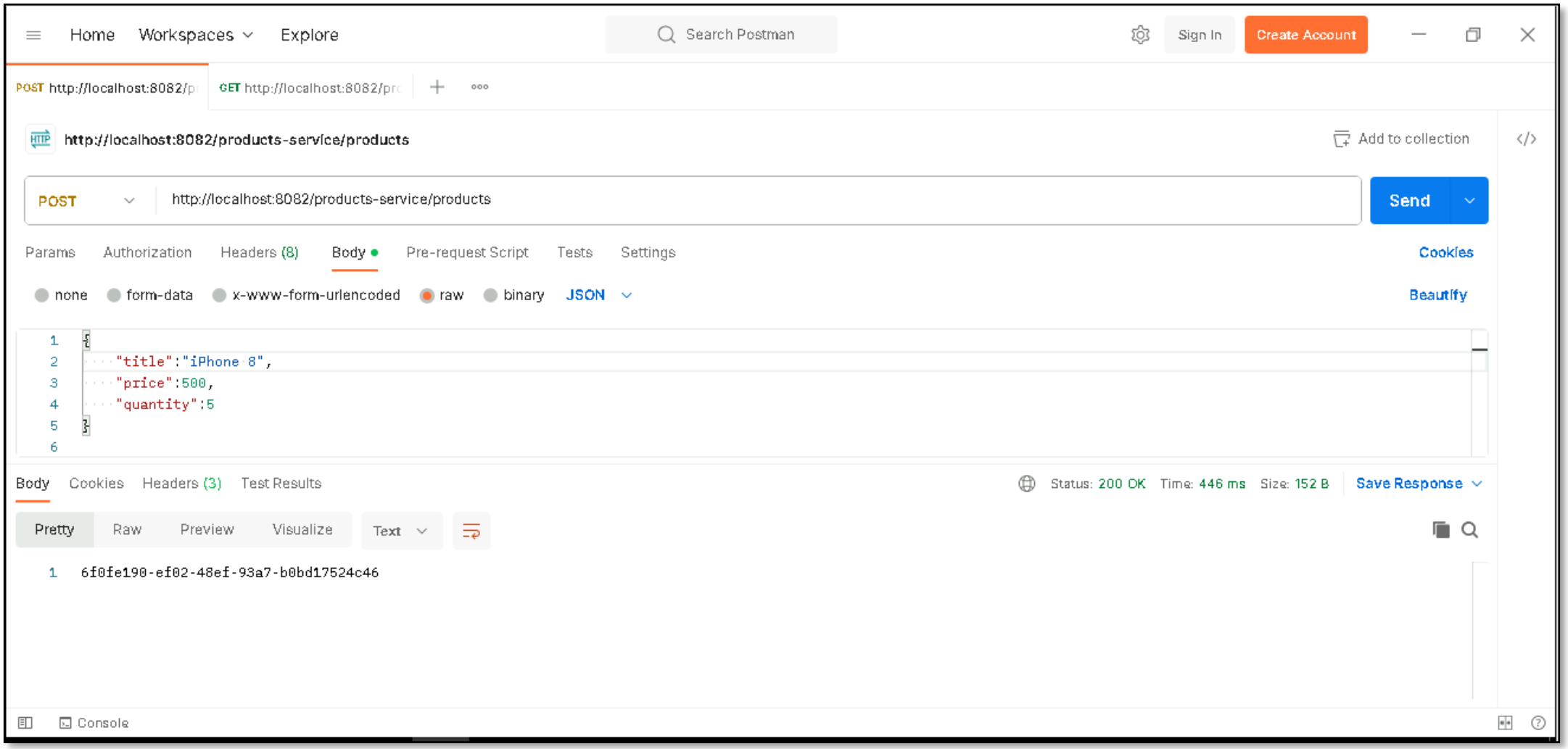
- Let’s remove the if conditions of Price & Title from the CreateProductCommandInterceptor class:

```
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.findByProductIdOrTitle(
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              return command;
51          };
```

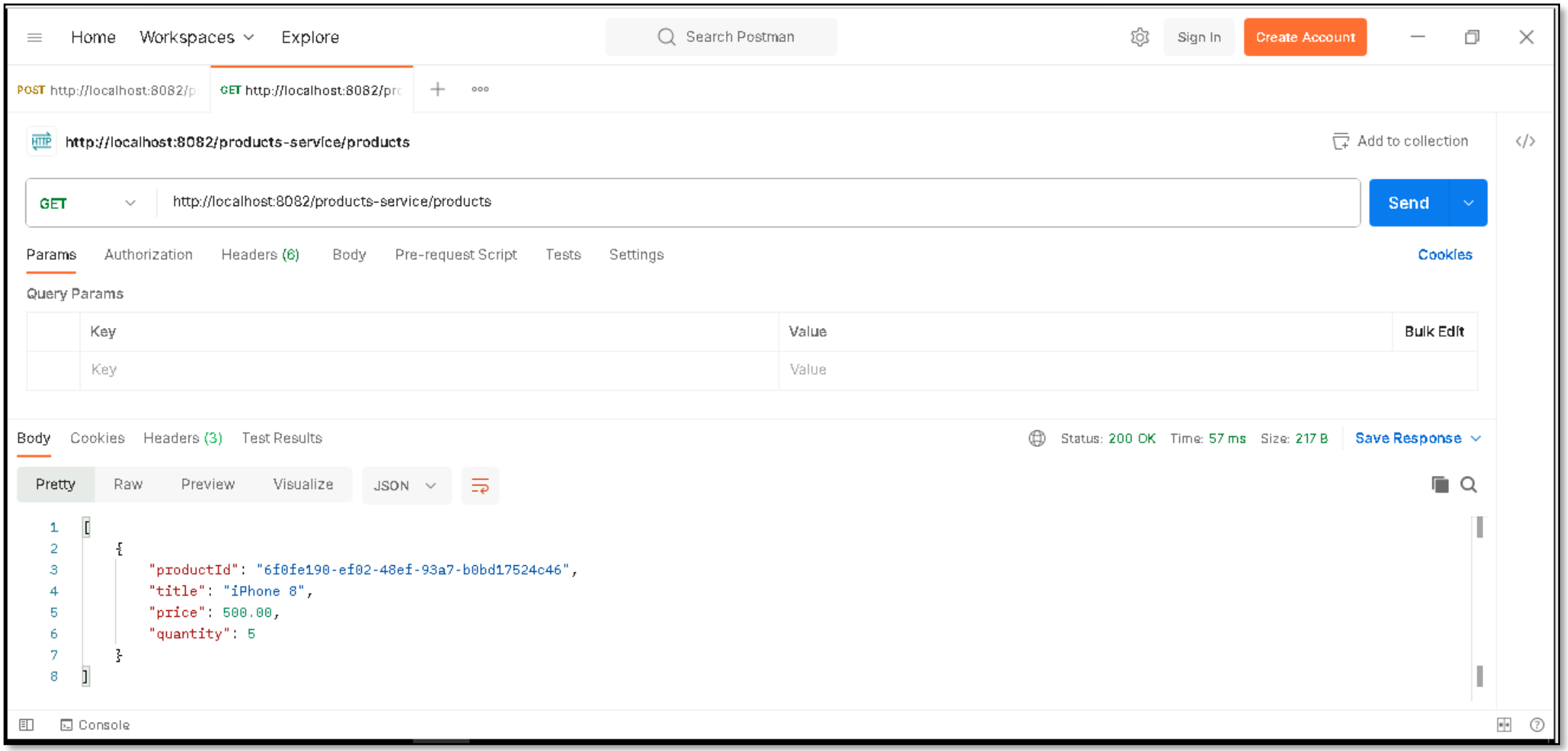
Trying how it works

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.

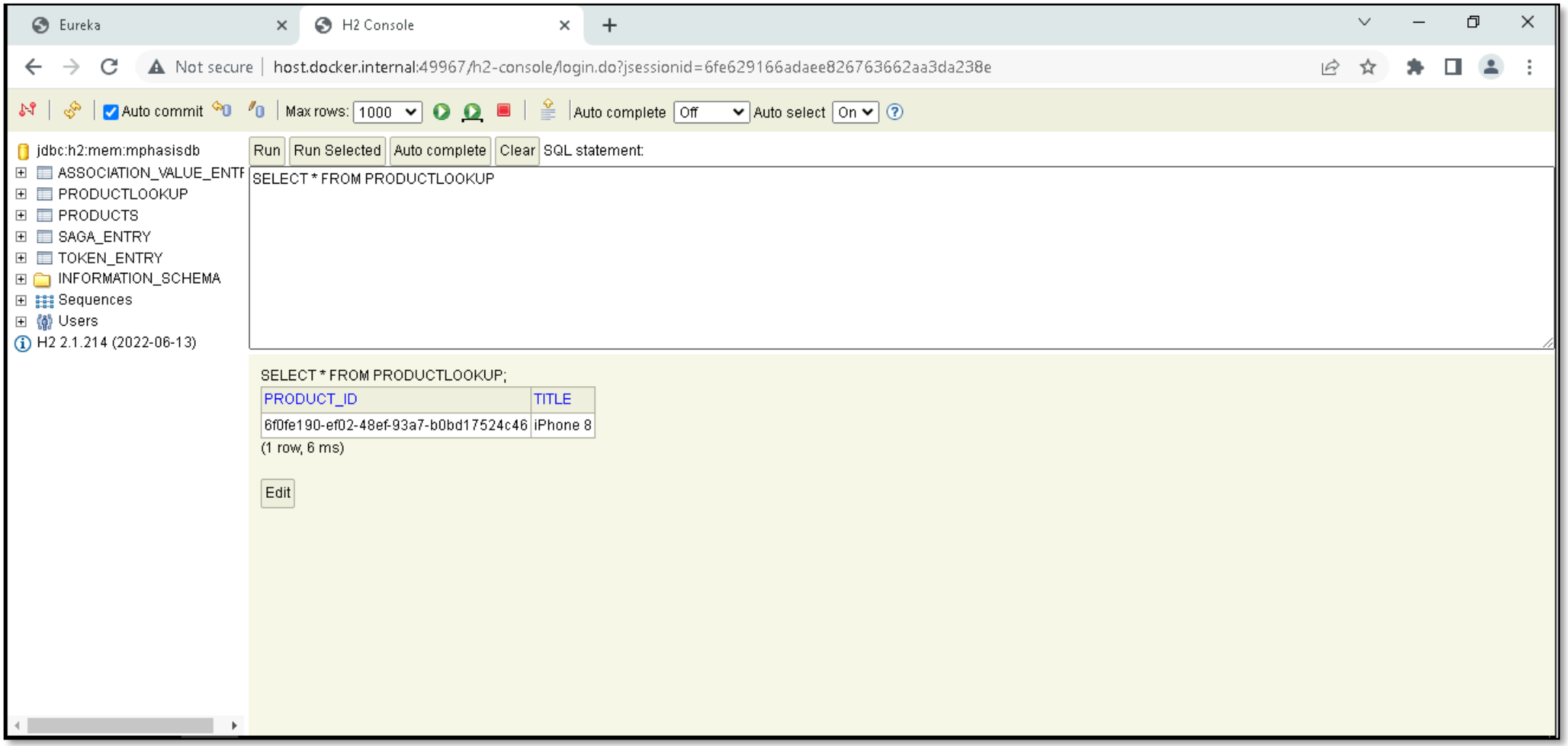
Send a POST request to Create Product



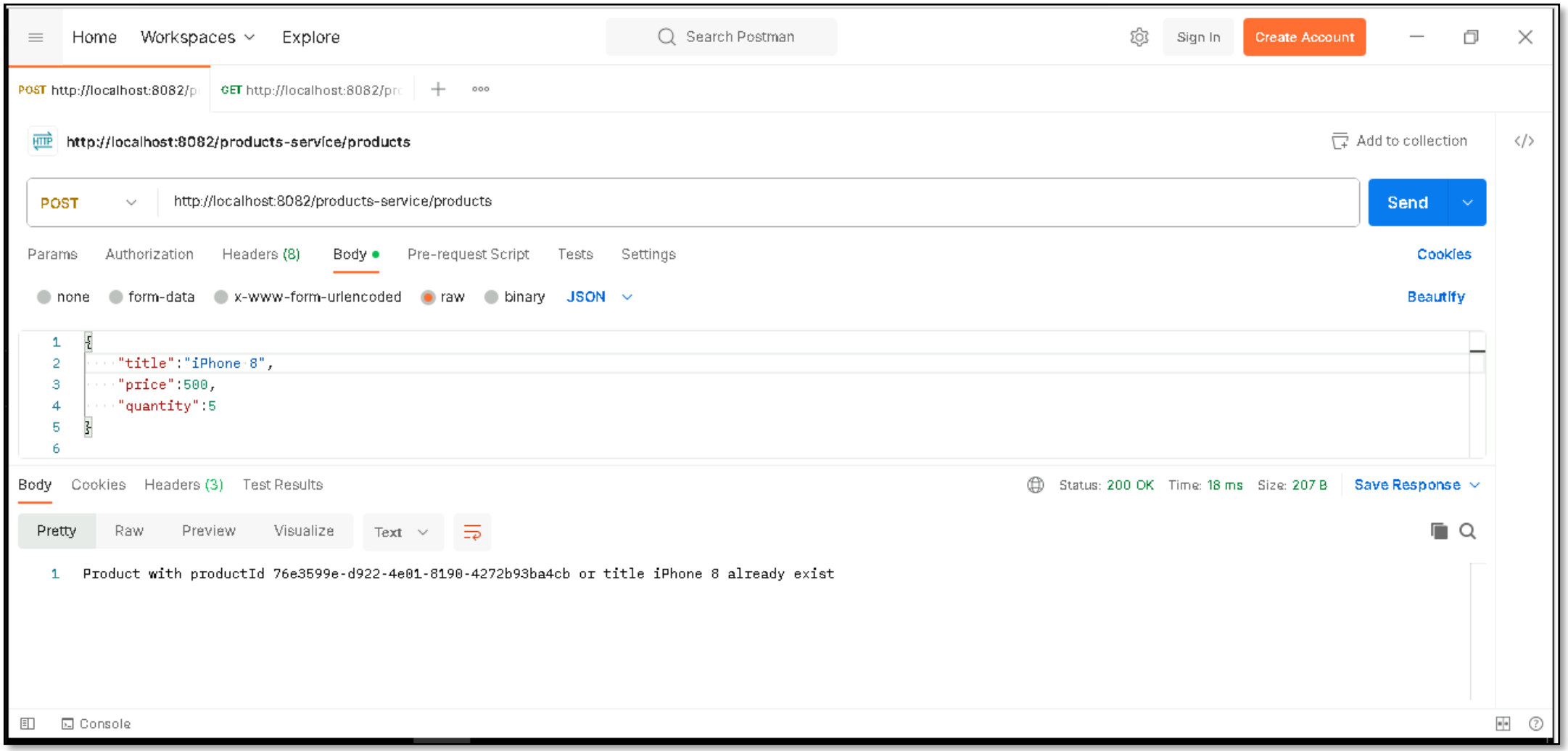
Send a GET request to Query the Products



Product Lookup Table



Send a POST request to Create Product with same JSON



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

© Mphasis 2020 Proprietary and confidential information

Mphasis
The Next Applied

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 ($C=X2C^2_{TM}=1$) 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