

Practice Exercise

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

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

Document Control

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

Contents

Practice Exercise	1
Document Control.....	2
Problem Statement 1: Basic Docker Commands.....	4
Problem Statement 2: Create Spring Boot Microservice for ProductService	5
Problem Statement 3: Configure Microservices with Eureka Service Registry Server	6
Problem Statement 4: Enable Dynamic Registration to Product Microservice	8
Problem Statement 5: Implementing Spring Cloud Gateway in Microservices	9
Problem Statement 6: Running Axon Server in Docker Container	13
Problem Statement 7: Implementing the CQRS & Event Sourcing Design Pattern in Product Microservice.....	15

Note: Every Problem Statement start on a new page

Problem Statement 1: Basic Docker Commands

Try out some docker basic commands:

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

Problem Statement 2: Create Spring Boot Microservice for ProductService

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

Technology stack:

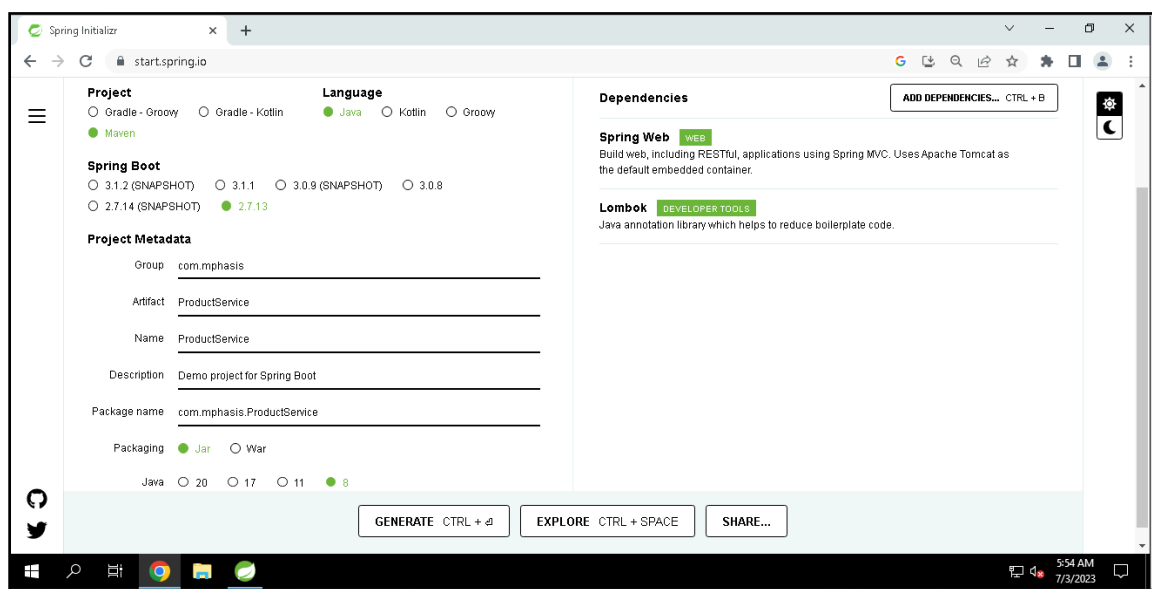
- Spring Web
- Lombok

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

Initially we will have only controller layers into the application:

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

Refer the below screenshots:



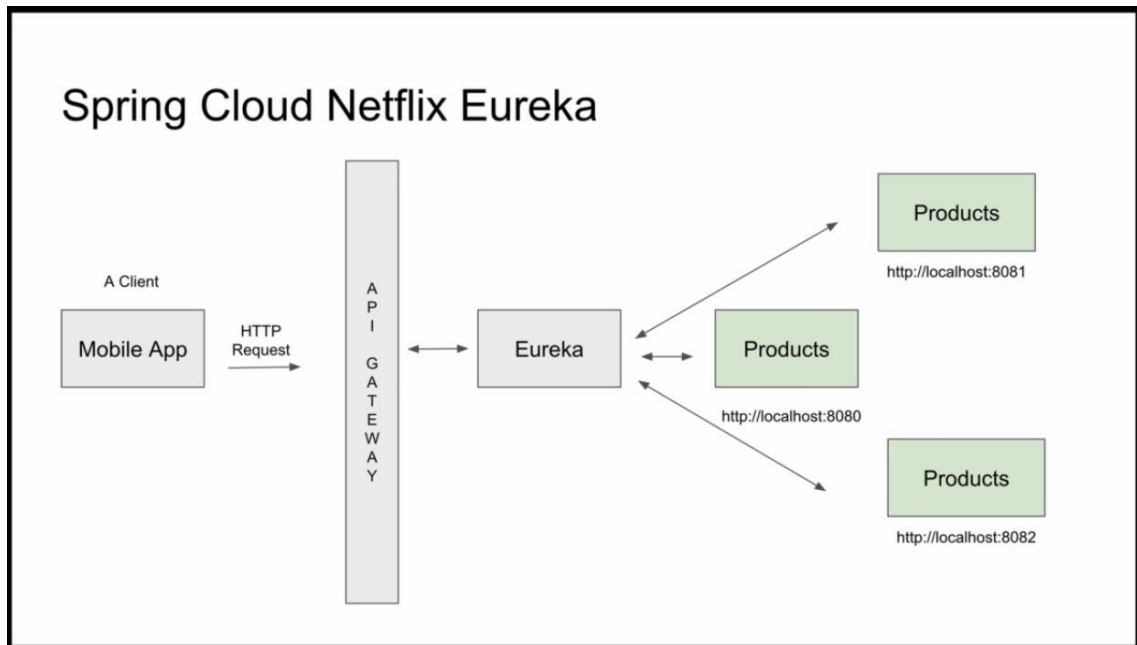
2. Select the **Spring Web**, **Lombok**, and **Eureka Discovery Client** dependencies.
3. Click on **GENERATE** button or **CTRL + Enter** to create the project structure.
4. Let's import the **ProductService** maven project in **STS**.
5. Will take some time for the project to be imported and the maven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

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

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

Problem Statement 3: Configure Microservices with Eureka Service Registry Server

The "eBookStore" application instance will 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.



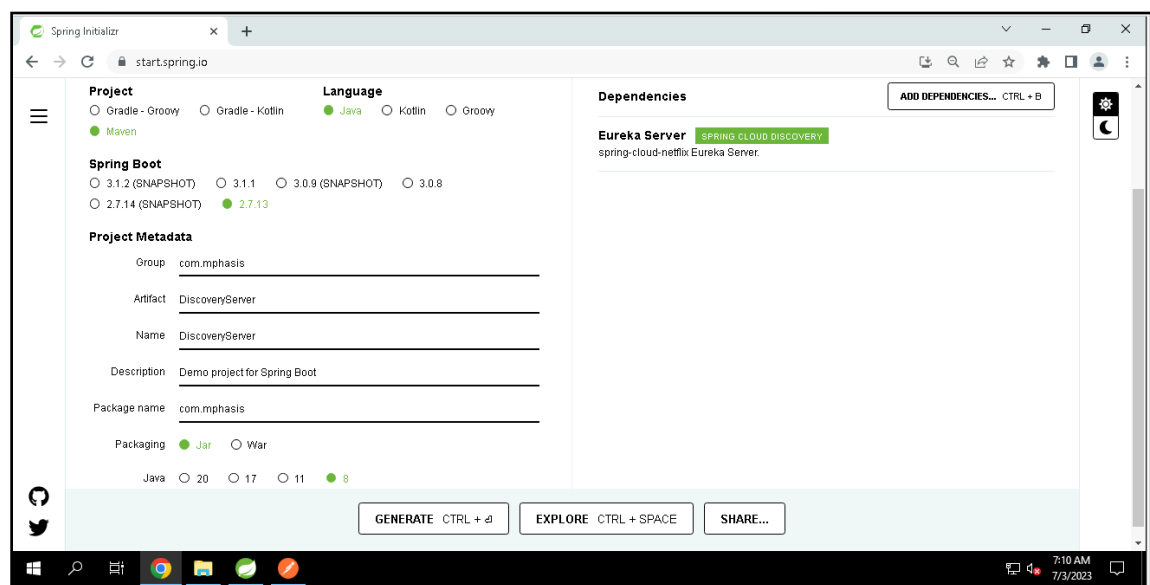
Technology stack:

- Spring Cloud Eureka Server

Steps for Spring Cloud Config Server:

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

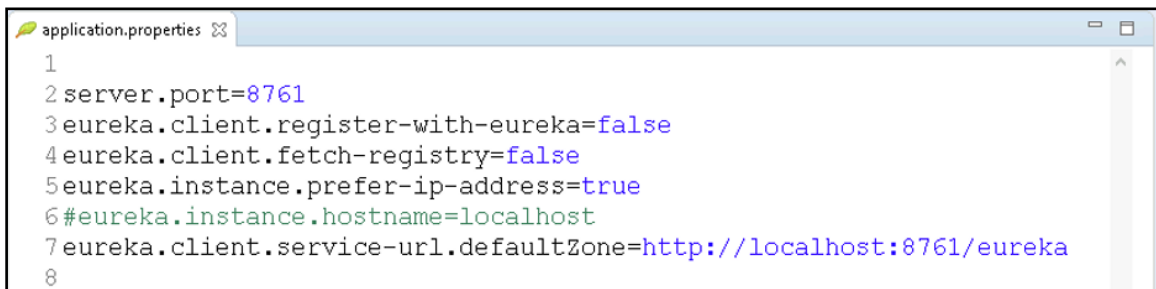
Refer the below screenshots:



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

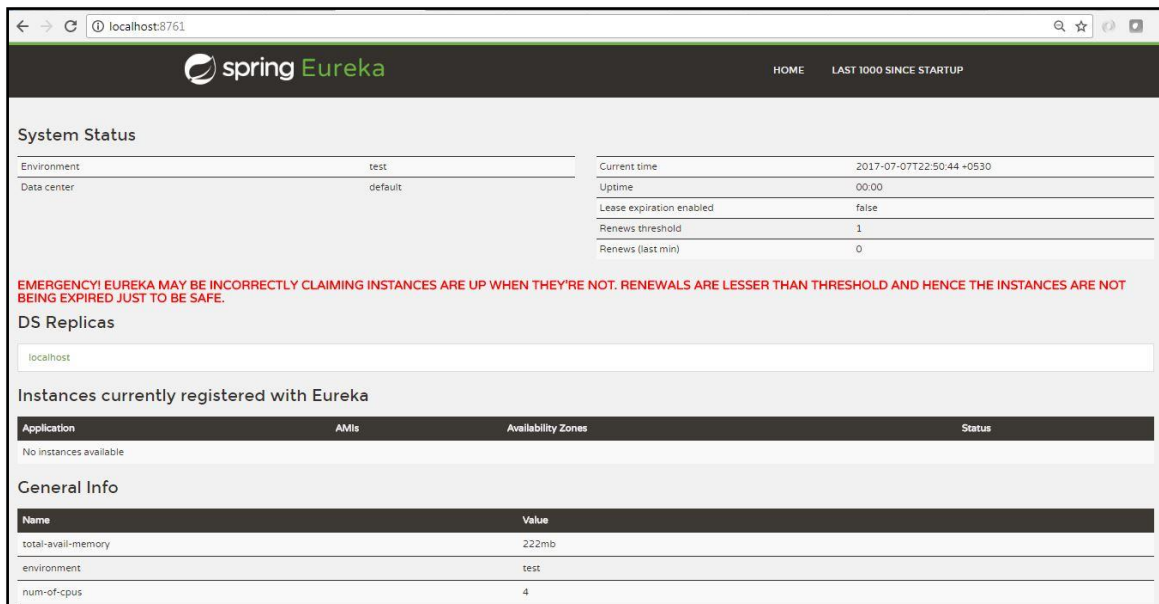
```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

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



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

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



The screenshot shows the Spring Eureka Server web interface in a browser. The address bar shows localhost:8761. The page has a dark header with the "spring Eureka" logo and navigation links for "HOME" and "LAST 1000 SINCE STARTUP".

System Status

Environment	test	Current time	2017-07-07T22:50:44 +0530
Data center	default	Uptime	00:00
		Lease expiration enabled	false
		Renews threshold	1
		Renews (last min)	0

EMERGENCY! EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP WHEN THEY'RE NOT. RENEWALS ARE LESSER THAN THRESHOLD AND HENCE THE INSTANCES ARE NOT BEING EXPIRED JUST TO BE SAFE.

DS Replicas

localhost

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
No instances available			

General Info

Name	Value
total-avail-memory	222mb
environment	test
num-of-cpus	4

Problem Statement 4: Enable Dynamic Registration to Product Microservice

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

Technology stack:

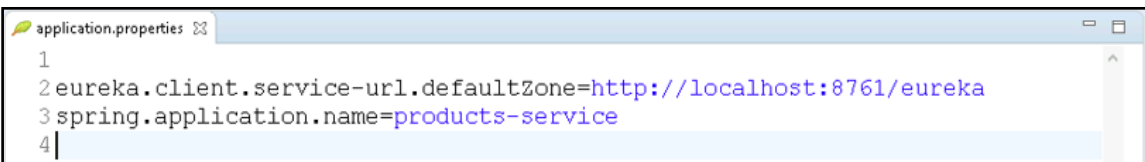
- Spring Cloud Eureka Client

Steps for Spring Cloud Config Client:

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

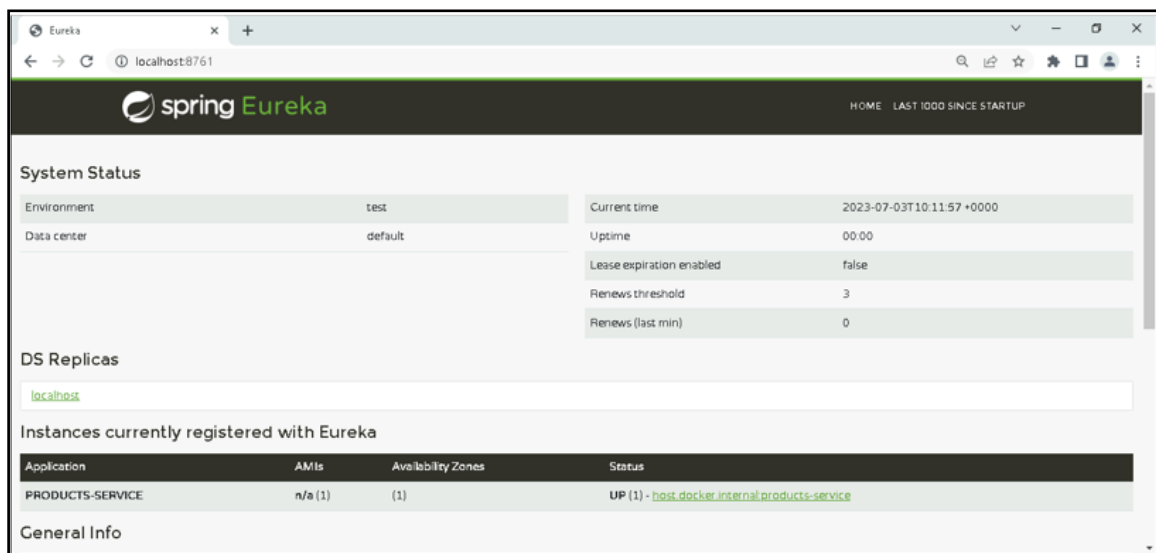
```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

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



```
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/>



The screenshot shows the Spring Eureka Server web interface at localhost:8761. The page displays system status, DS Replicas, and instances currently registered with Eureka.

System Status	
Environment	test
Data center	default
Current time	2023-07-03T10:11:57 +0000
Uptime	00:00
Lease expiration enabled	false
Renews threshold	3
Renews (last min)	0

DS Replicas: localhost

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
PRODUCTS-SERVICE	n/a (1)	(1)	UP (1) - host.docker.internal/products-service

General Info

Problem Statement 5: Implementing Spring Cloud Gateway in Microservices

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

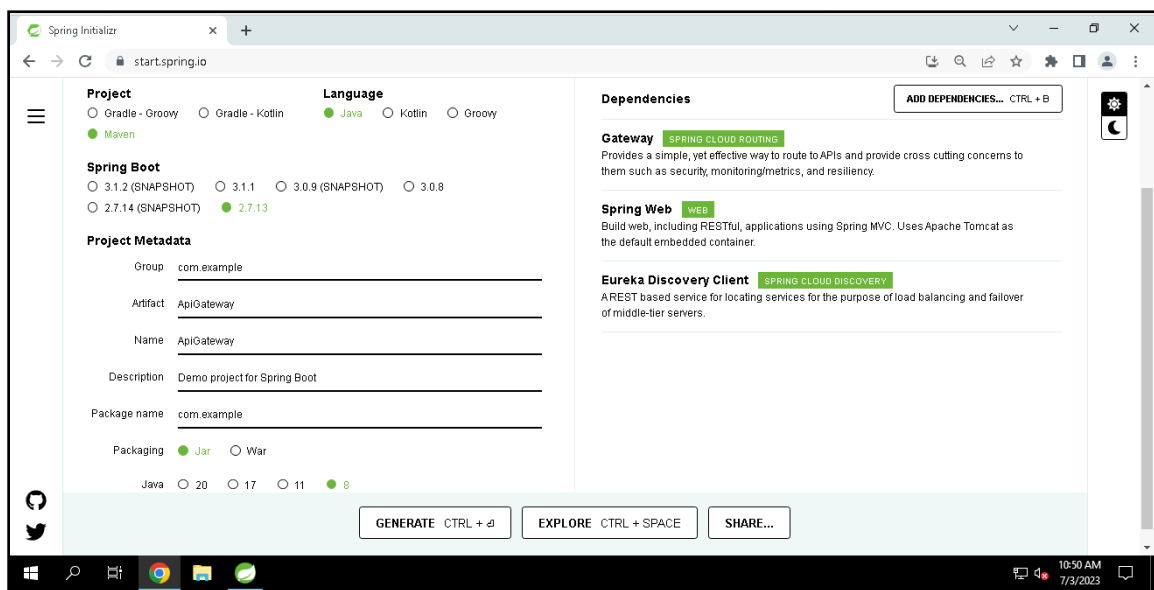
Technology stack:

- Spring Cloud Routing - Gateway

Steps for implementing API Gateway:

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

Refer the below screenshots:



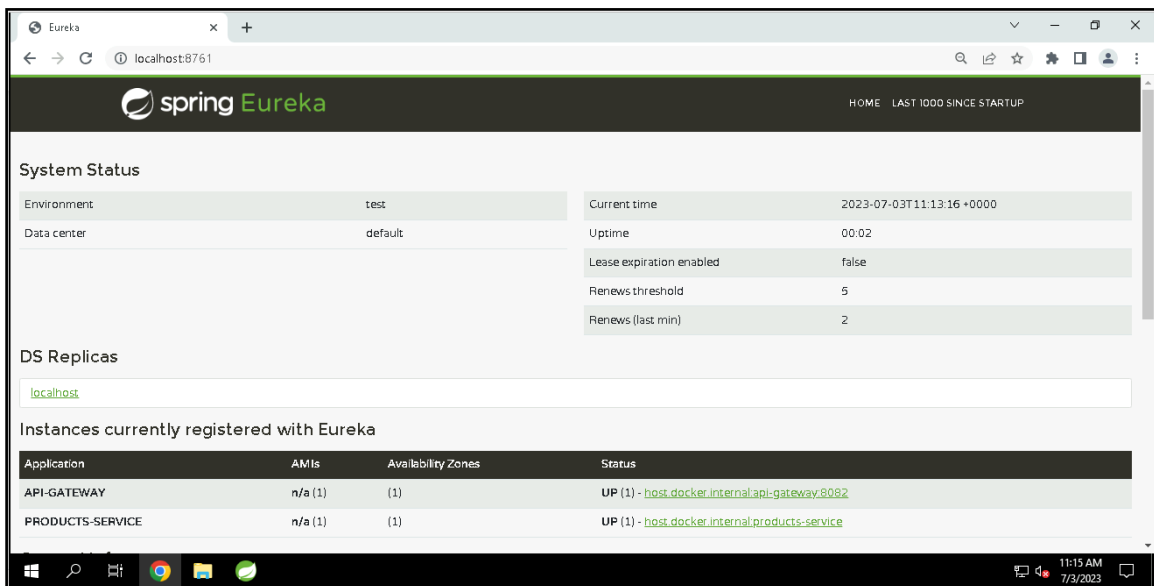
4. Select the **Gateway**, **Spring Web**, and **EurekaDiscoveryClient** dependencies.
5. Click on GENERATE button or CTRL + Enter to create the project structure.
6. Let's import the ApiGateway maven project in STS.
7. Will take some time for the project to be imported and the maven dependencies to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

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

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

10. Start the ApiGateway.


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



The screenshot shows the Spring Eureka Server web interface. The 'Instances currently registered with Eureka' table is as follows:

Application	AMIs	Availability Zones	Status
API-GATEWAY	n/a (1)	(1)	UP (1) - host.docker.internal:api-gateway:8082
PRODUCTS-SERVICE	n/a (1)	(1)	UP (1) - host.docker.internal:products-service

12. Test the Proxy: <http://localhost:8082/products-service/products>

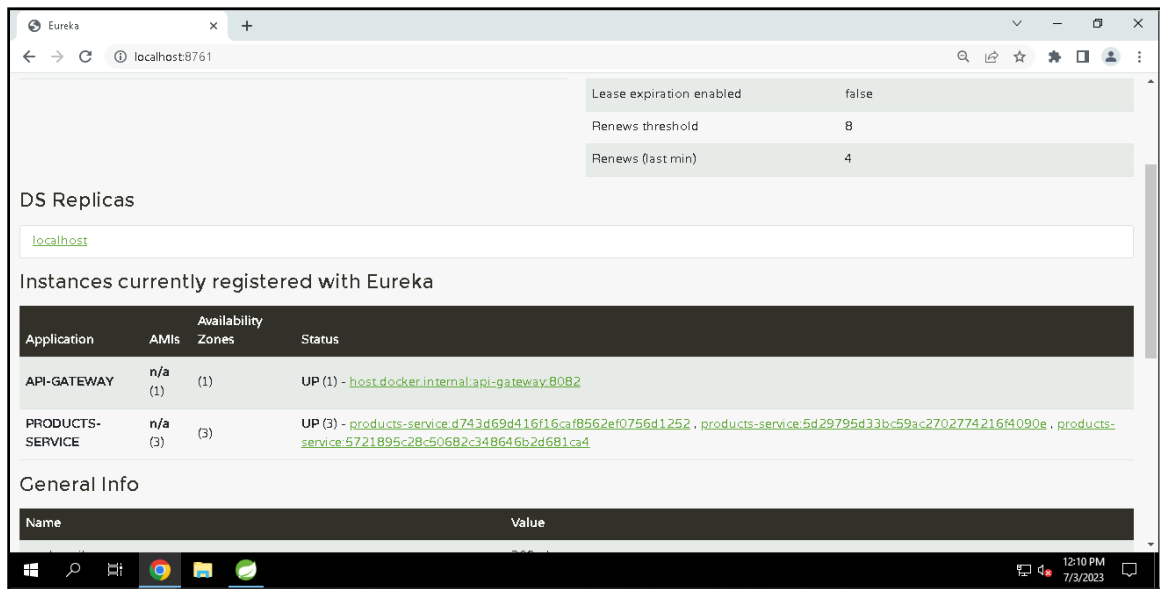


The screenshot shows a web browser window with the URL localhost:8082/products-service/products. The response is 'HTTP GET Handled'.

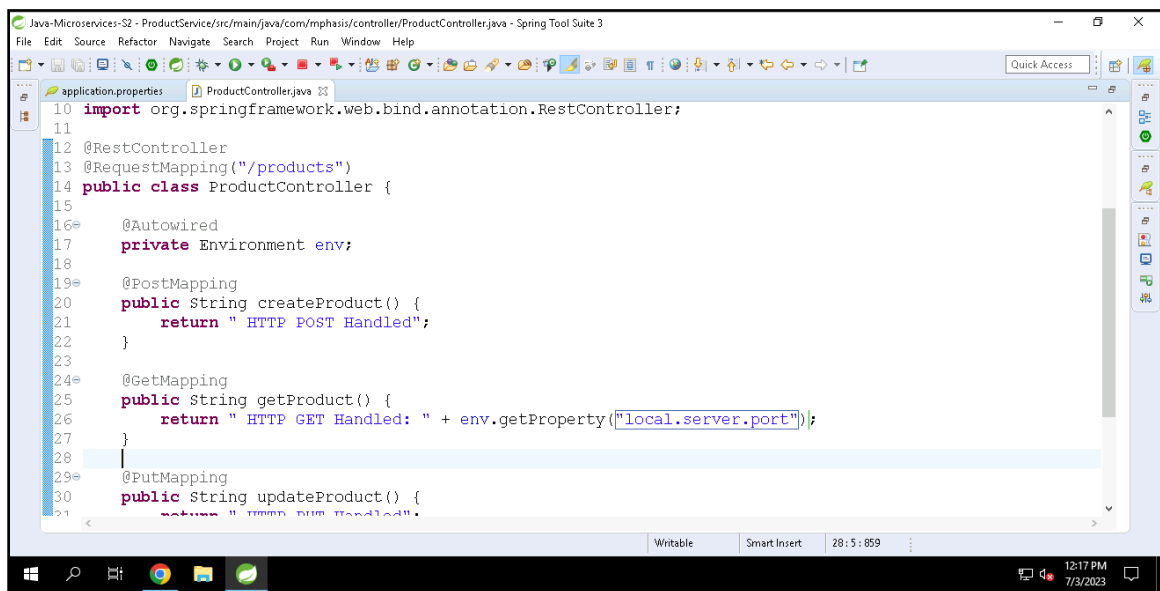
13. Let's add the random port and instance-id property to ProductService/application.properties file:

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

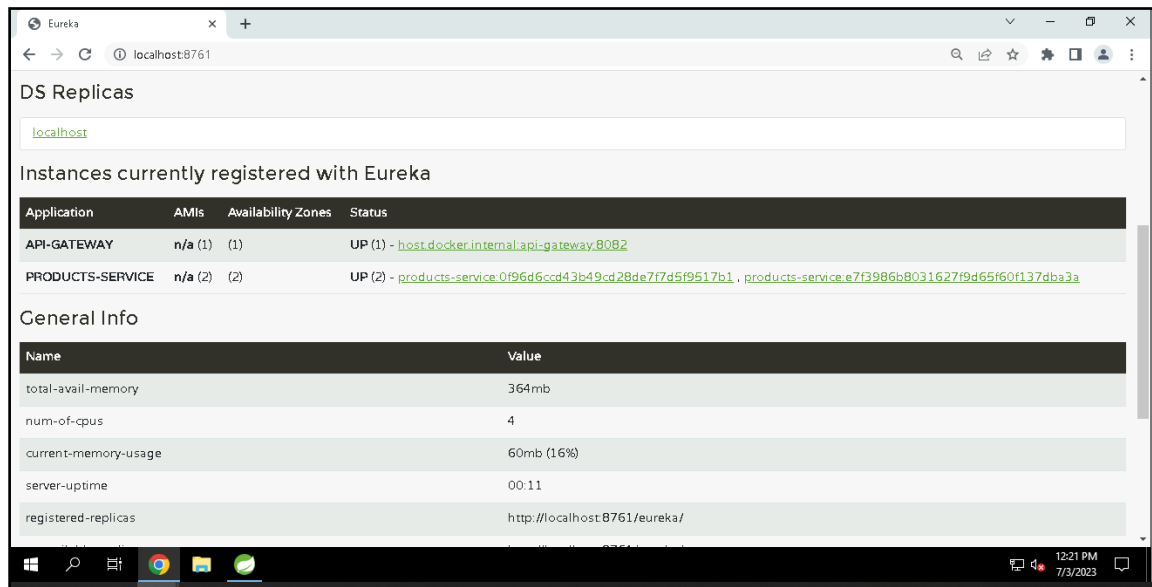
14. Restart the Discovery Server and execute Product Service three times, you will find 3 instances are running.



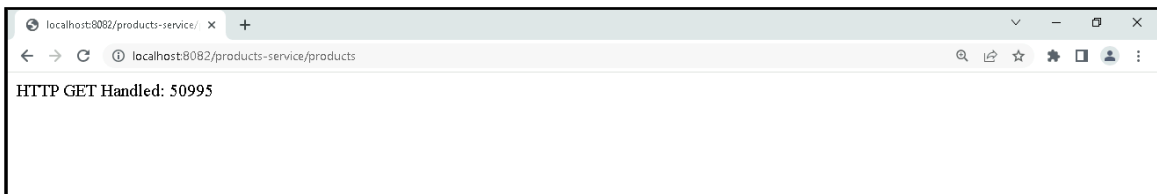
15. Test how the Load Balancing works.
16. Modify the code of GET handler method in ProductController class.



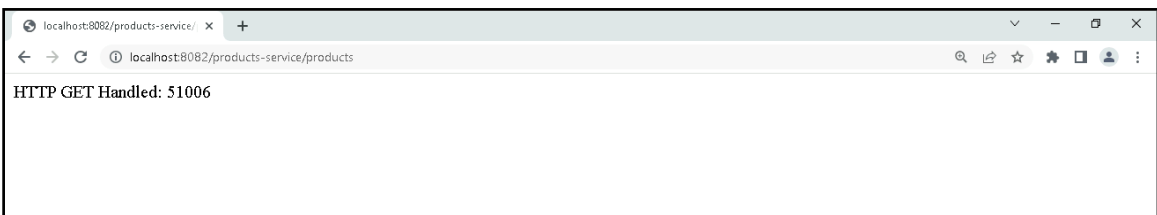
17. Restart the Discovery Server and execute Product Service two times, you will find 2 instances are running.
18. Restart the ApiGateway also.



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



20. Refresh again:



Problem Statement 6: Running Axon Server in Docker Container

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

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

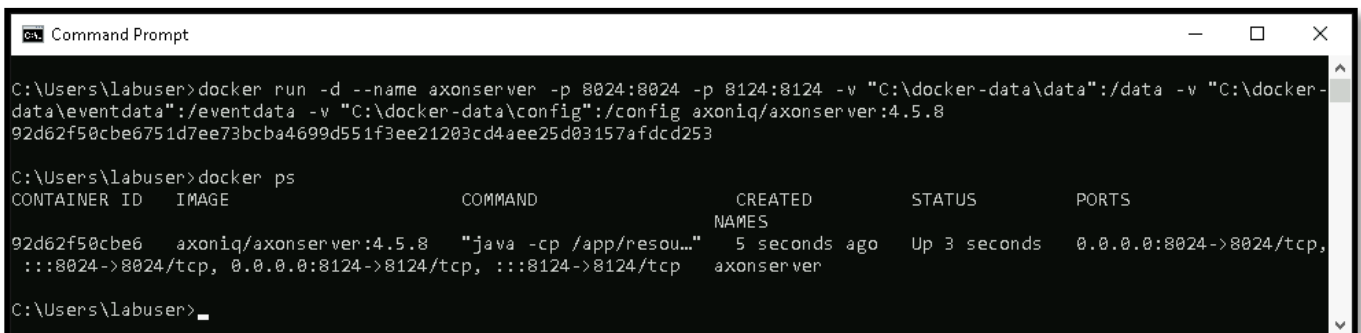
Steps for Running Axon Server in Docker Container:

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

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

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

```
docker run -d \
--name axonserver \
-p 8024:8024 \
-p 8124:8124 \
-v "C:\docker-data\data":/data \
-v "C:\docker-data/eventdata":/eventdata \
-v "C:\docker-data/config":/config \
axoniq/axonserver:4.5.8
```

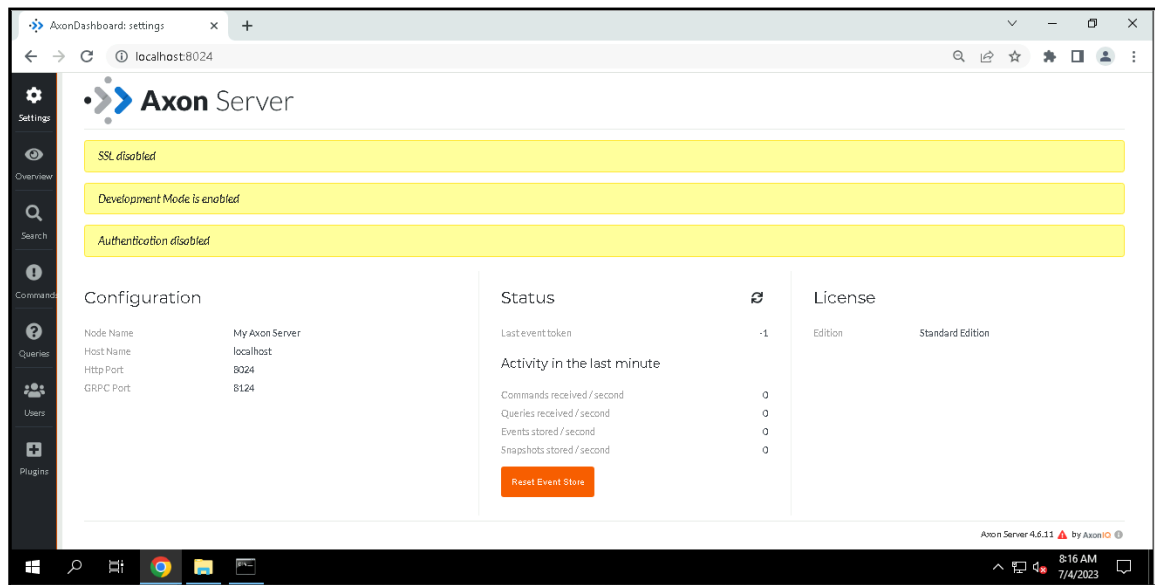


```
Command Prompt
C:\Users\labuser>docker run -d --name axonserver -p 8024:8024 -p 8124:8124 -v "C:\docker-data\data":/data -v "C:\docker-
data\eventdata":/eventdata -v "C:\docker-data\config":/config axoniq/axonserver:4.5.8
92d62f50cbe6751d7ee73bcba4699d551f3ee21203cd4aee25d03157afdc253

C:\Users\labuser>docker ps
CONTAINER ID   IMAGE               COMMAND                  CREATED        STATUS        PORTS
92d62f50cbe6   axoniq/axonserver:4.5.8   "java -cp /app/resou..."   5 seconds ago   Up 3 seconds   0.0.0.0:8024->8024/tcp,
:::8024->8024/tcp, 0.0.0.0:8124->8124/tcp, :::8124->8124/tcp
axonserver
```

Java Microservices – S2 – Practice Exercise

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

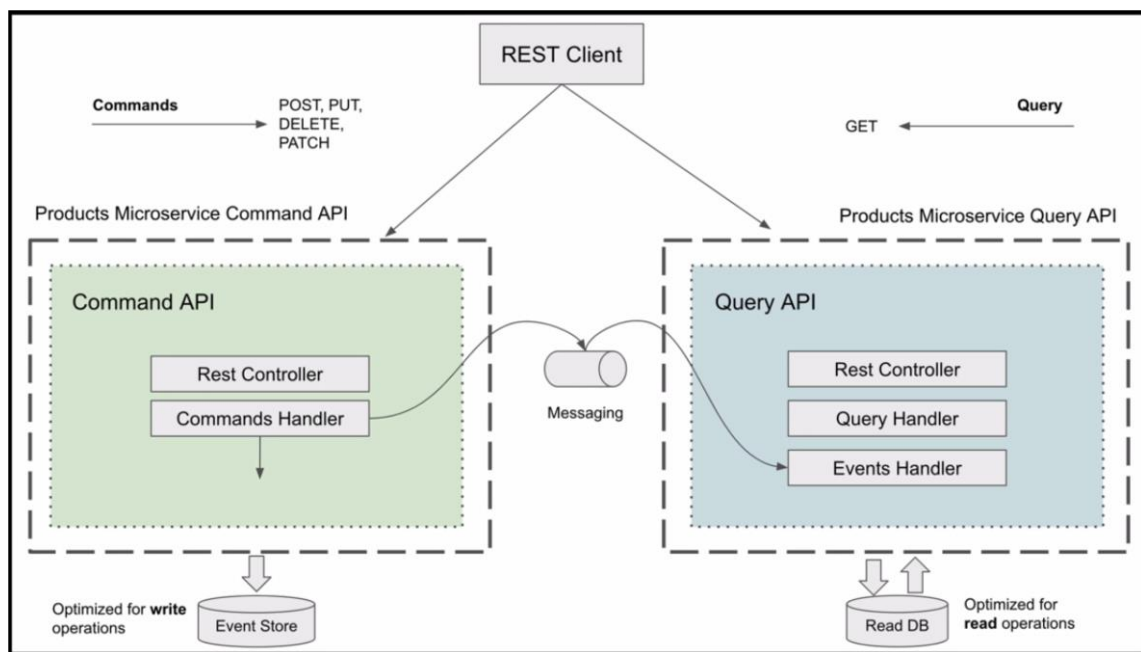


Problem Statement 7: Implementing the CQRS & Event Sourcing Design Pattern in Product Microservice

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

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

Here is a diagram explaining the same:

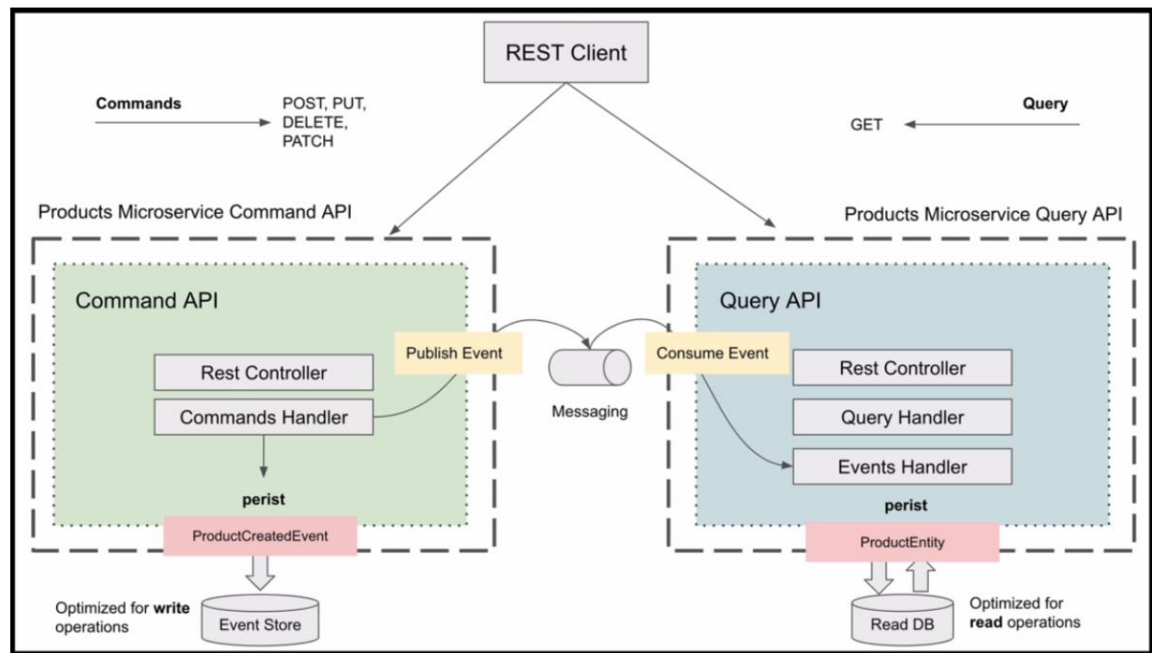


In some scenarios though you would want more than the current state, you might need all the states which the customer entry went through. For such cases, the design pattern **"Event Sourcing"** helps.

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

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

Here is a diagram explaining the same:



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

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

Technology stack:

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

Steps for implementing CQRS and Event Sourcing using Axon Server:

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

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

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

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

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

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

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

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

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

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

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

7. The CreateProductRestModel annotated with @Data and should have the following fields:
private String title;
private BigDecimal price;
private Integer quantity;
8. This controller class should use the **Axon's CommandGateway** and publish the CreateProductCommand.
9. The CreateProductCommand annotated with @Data, @Builder and should have the following fields:
private final String productId;
private final String title;
private final BigDecimal price;
private final Integer quantity;

Where:
productId - is a randomly generated value. For example, UUID.randomUUID().toString() and annotated with **@TargetAggregateIdentifier**.
Should return the productId as String. If the exception raised by published code, handle and return the Localized Message as String.
10. Create a new class called ProductAggregate and make it handle the CreateProductCommand using **@CommandHandler** and publish the ProductCreatedEvent.
11. The ProductCreatedEvent class annotated with @Data and should have the following fields:
private String productId;
private String title;
private BigDecimal price;
private Integer quantity;
12. Apply the below validation to price and title in the Command Handler.

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

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

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

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

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

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

CQRS Persisting Event in the Product database:

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

```
application.properties
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.jpa.database-platform=org.hibernate.dialect.H2Dialect
8
9 spring.h2.console.settings.web-allow-others=true
10
11 spring.datasource.url=jdbc:h2:mem:mphasisdb
12 spring.datasource.driver-class-name=org.h2.Driver
13 spring.datasource.username=sa
14 spring.datasource.password=password
15
16 #Accessing the H2 Console
17 spring.h2.console.enabled=true
18 spring.h2.console.path=/h2-console
```

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

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

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

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

and make the ProductEntity class have the following fields:

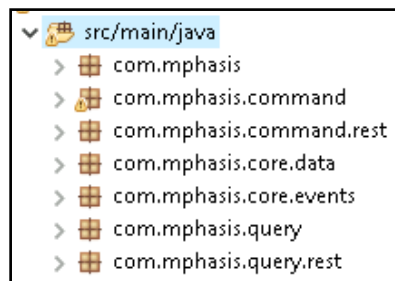
```
@Id
@Column(unique = true)
private String productId;
@Column(unique = true)
private String title;
private BigDecimal price;
private Integer quantity;
```

CQRS, Querying Data:

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

21. Refactor Command API REST Controller:

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



22. Create the ProductsQueryController class inside com.mphasis.query.rest package.

23. The method that accepts the HTTP Get request, should have List<ProductRestModel> as a response body.

24. The ProductRestModel annotated with @Data and should have the following fields:

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

25. This controller class should use the **Axon's QueryGateway** to dispatch an instance of FindProductQuery. As we use the gateway's query() method to issue a point-to-point query. Because we are specifying ResponseTypes.multipleInstancesOf(ProductRestModel.class), Axon knows we only want to talk to query handlers whose return type is a collection of ProductRestModel objects.

26. Create a new @Component class called ProductsQueryHandler inside com.mphasis.query package.

27. Refer the JPA Repository called ProductRepository and inject it into ProductsQueryHandler using constructor-based dependency injection.

28. The ProductsQueryHandler class should have one **@QueryHandler** method that handles the FindProductsQuery and fetch all the product details from the "read" database.

Run and make it work:

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

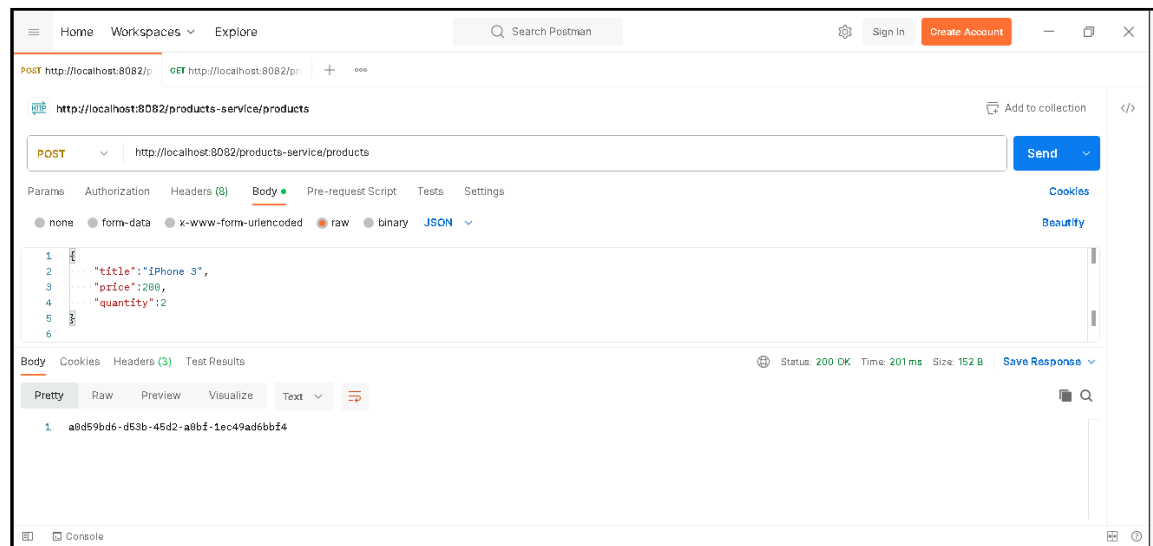


```
44     }
45     return returnValue;
46 }
47 /*
48 @GetMapping
49 public String getProduct() {
50     return " HTTP GET Handled: " + env.getProperty("local.server.port");
51 }
52
53 @PutMapping
54 public String updateProduct() {
55     return " HTTP PUT Handled";
56 }
57
58 @DeleteMapping
59 public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

30. Run the Axon Server using Docker command.

31. Start the Discovery Server (Eureka Server), Product Service, and ApiGateway.

32. Send a POST request to Create Product.



POST http://localhost:8082/products-service/products

Params Authorization Headers (8) Body Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary JSON

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

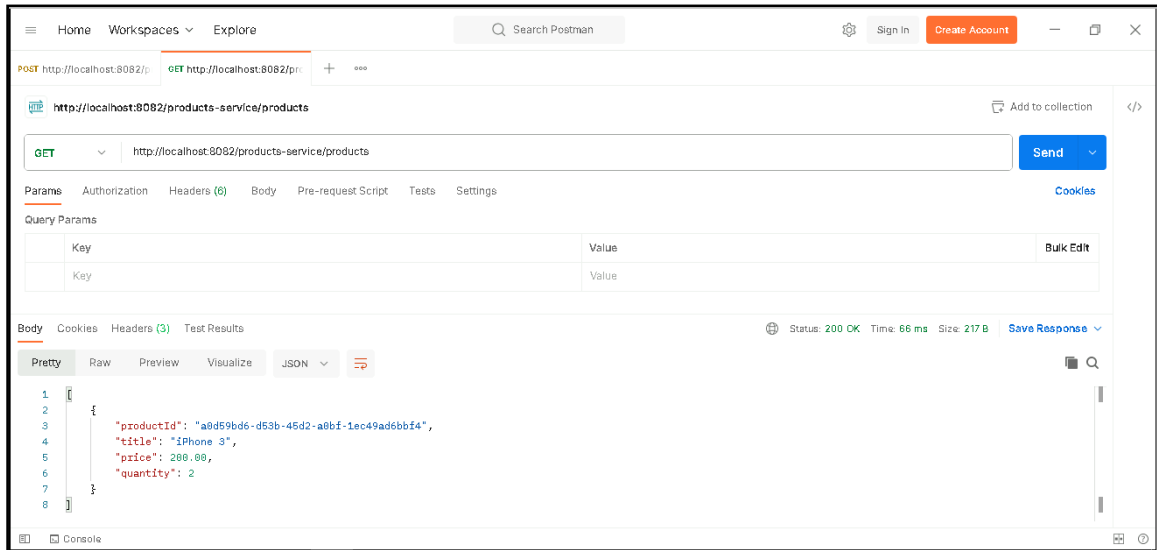
Body Cookies Headers (3) Test Results

Pretty Raw Preview Visualize Text

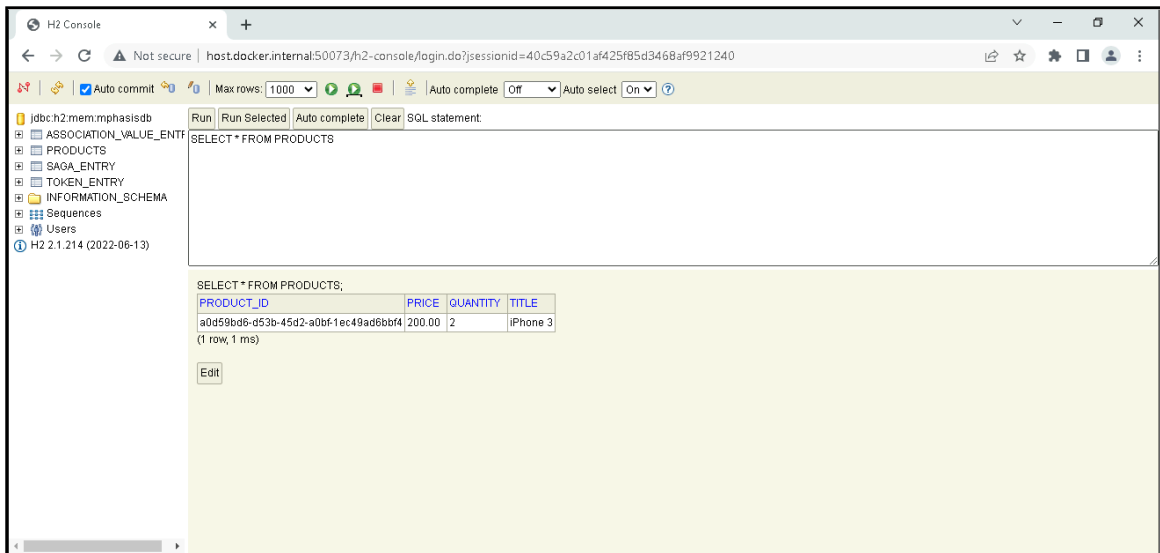
1 a8d59bd6-d53b-45d2-a0bf-1ec49ad6bbf4

Status: 200 OK Time: 201 ms Size: 152 B Save Response

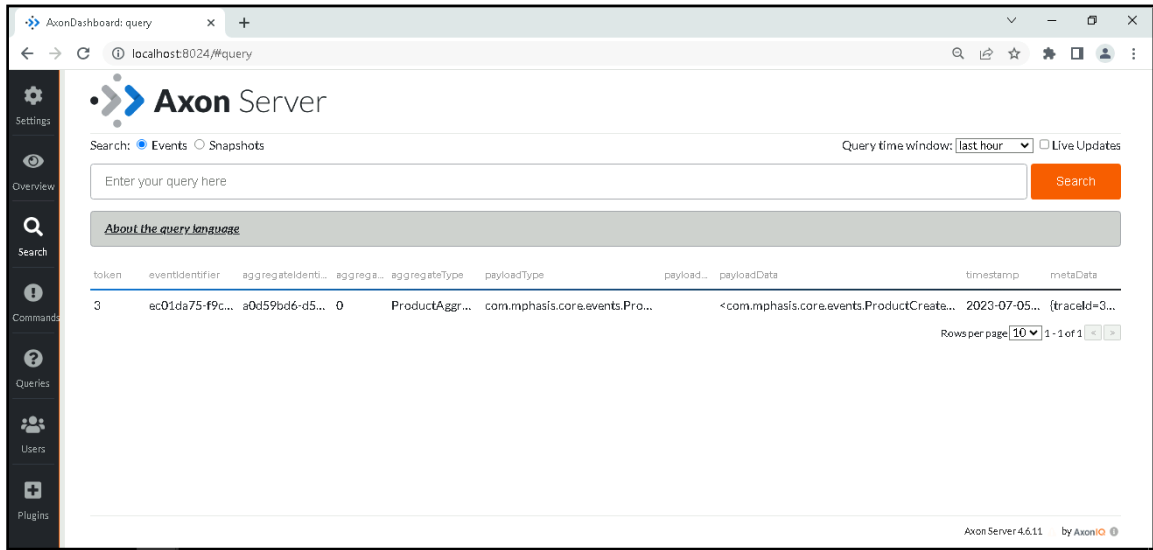
33. Send a GET request to Query the Products.



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



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



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

