

CQRS

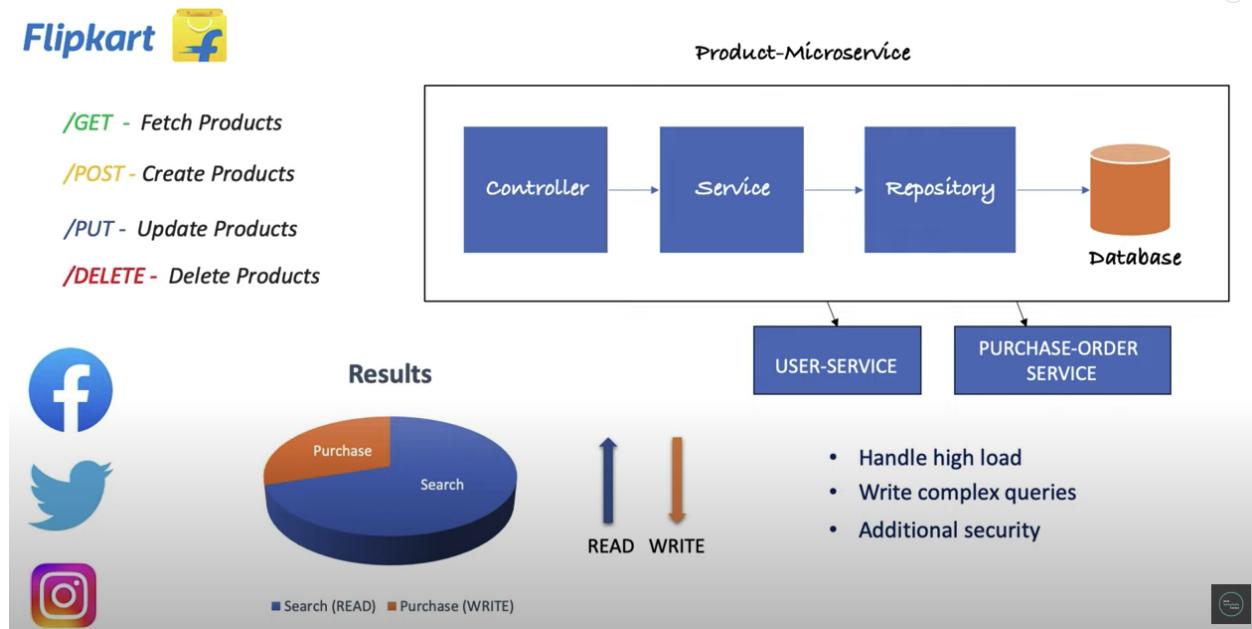
Design Pattern

CQRS

Command & Query Responsibility Segregation



Segregate read and write operations instead of single micro service



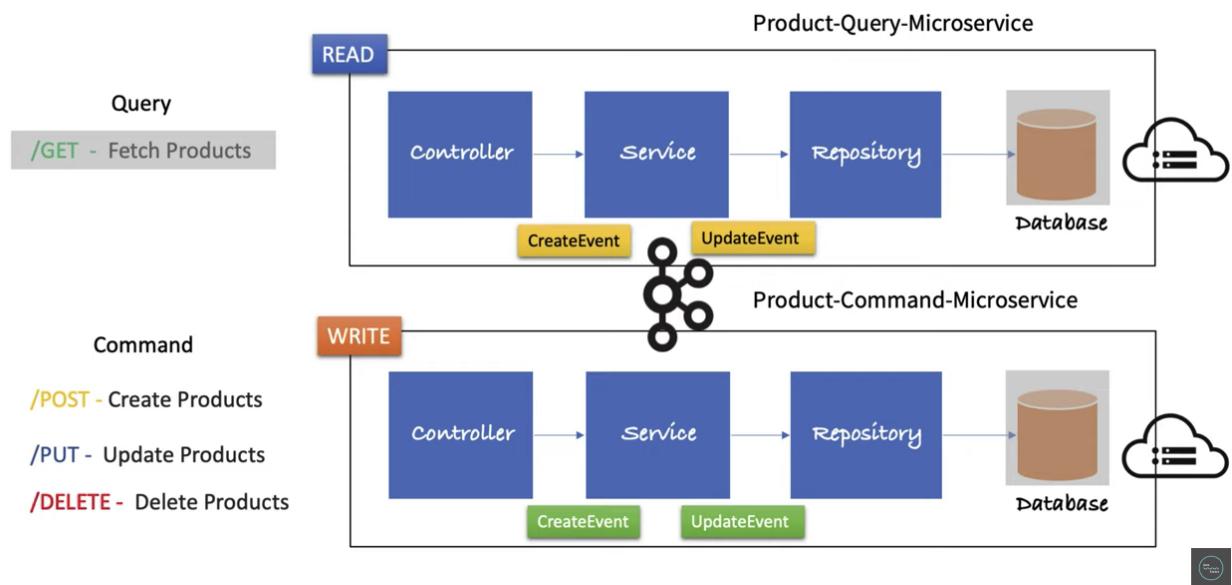
Regular approaches and its problems.

Problem Statement: Because we have read operations more - we can scale the application only for read operation during Peak.

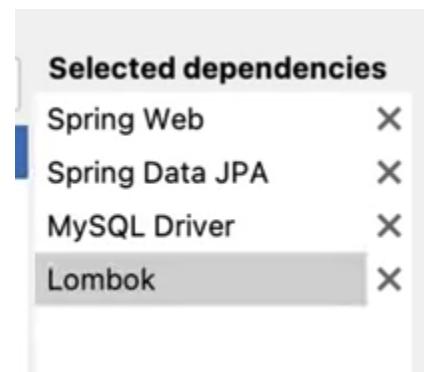
We have to maintain more complex queries if our Micro service integrate with other micro services to fetch orders created by a user and so on.

Need to add additional security system.

Solution : **CQRS** (Command Query Responsibility Segregation)



Because our DBs are different whatever create order or update order wont available in the Read application. So bring kafka as solution whatever events happen in the command application publish them to read application.



Create a spring boot micro service for command application and read application with above dependencies



The screenshot shows the IntelliJ IDEA interface with the code editor open to a file named `Product.java`. The code defines a simple entity class `Product` with attributes `id`, `name`, `description`, and `price`. The `@Entity` and `@Table` annotations are present. A code completion tooltip is displayed, listing several annotations: `@Entity`, `@Table(name = "PRODUCT_COMMAND")`, `@Data`, `@AllArgsConstructor`, `@NoArgsConstructor`, and `public class Product {`. The `@NoArgsConstructor` entry is currently highlighted.

```
import jakarta.persistence.Table;
@Entity
@Table(name = "PRODUCT_COMMAND")
public class Product {
    @Id
    @GeneratedValue
    private Long id;
    private String name;
    private String description;
    private double price;
}
```

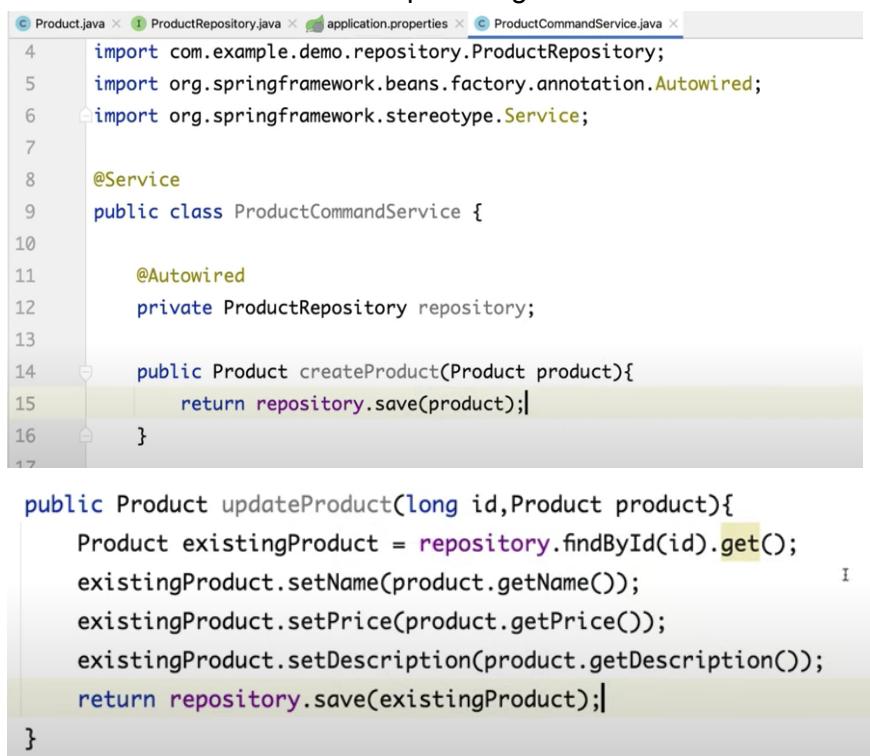
The screenshot shows the IntelliJ IDEA interface with the following details:

- Project View:** Shows the project structure under "src/main/java/com/example/demo".
- Code Editor:** Displays the `ProductRepository.java` file content.
- Toolbars:** Standard IntelliJ IDEA toolbars for file operations and configuration.
- Status Bar:** Shows a warning icon with the number 1.

```
package com.example.demo.repository;  
import com.example.demo.entity.Product;  
import org.springframework.data.jpa.repository.JpaRepository;  
  
public interface ProductRepository extends JpaRepository<Product, Long> {  
}
```

```
Product.java × ProductRepository.java × application.properties × Redis Helper  
1 spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver  
2 spring.datasource.url = jdbc:mysql://localhost:3306/javatechie  
3 spring.datasource.username = root  
4 spring.datasource.password = Password  
5 spring.jpa.show-sql = true  
6 spring.jpa.hibernate.ddl-auto = update  
7 spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.  
8 server.port=9191 Codota
```

Service class with create and update logic.



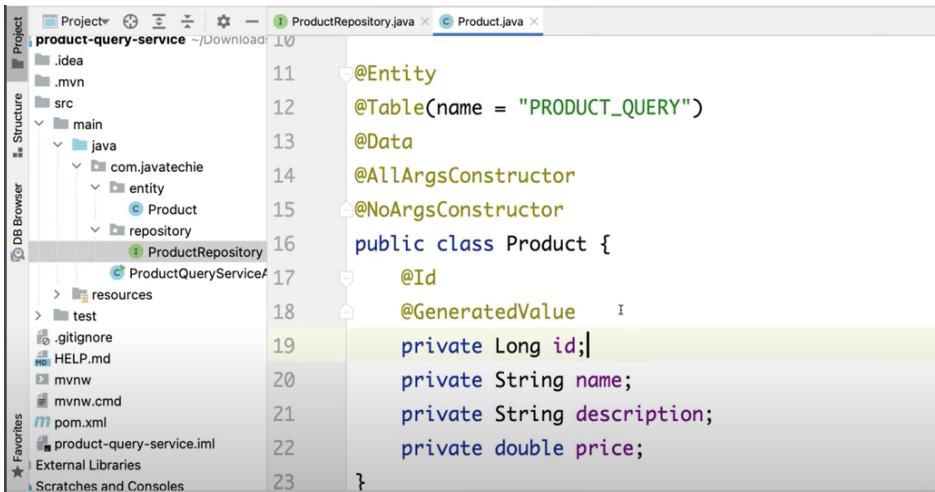
```
4 import com.example.demo.repository.ProductRepository;
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.stereotype.Service;
7
8 @Service
9 public class ProductCommandService {
10
11     @Autowired
12     private ProductRepository repository;
13
14     public Product createProduct(Product product){
15         return repository.save(product);
16     }
17
18
19     public Product updateProduct(long id,Product product){
20         Product existingProduct = repository.findById(id).get();
21         existingProduct.setName(product.getName());
22         existingProduct.setPrice(product.getPrice());
23         existingProduct.setDescription(product.getDescription());
24         return repository.save(existingProduct);
25     }
26 }
```

Controllers for create and update.



```
9 @RestController
10 @RequestMapping("/products")
11 public class ProductCommandController {
12
13     @Autowired
14     private ProductCommandService commandService;
15
16
17     @PostMapping
18     public Product createProduct(@RequestBody Product product){
19         return commandService.createProduct(product);
20     }
21
22     @PutMapping("/{id}")
23     public Product updateProduct(@PathVariable long id,@RequestBody Product product){
24         return commandService.updateProduct(id, product);
25     }
26 }
```

Copy entity and repository classes to query service



The screenshot shows the IntelliJ IDEA interface with the 'product-query-service' project open. The 'Product.java' file is selected in the top navigation bar. The code editor displays the following Java code:

```
11  @Entity
12  @Table(name = "PRODUCT_QUERY")
13  @Data
14  @AllArgsConstructor
15  @NoArgsConstructor
16  public class Product {
17      @Id
18      @GeneratedValue
19      private Long id;
20      private String name;
21      private String description;
22      private double price;
23  }
```

The code editor highlights the 'id' field with a yellow background.

One advantage is there is no need for an exact match for Entity. Only whatever fields we are interested in - we can define only them.

Query service class



The screenshot shows the IntelliJ IDEA interface with the 'ProductQueryService.java' file selected in the top navigation bar. The code editor displays the following Java code:

```
10  @Service
11  public class ProductQueryService {
12
13      @Autowired
14      private ProductRepository repository;
15
16      public List<Product> getProducts(){
17          return repository.findAll();
18      }
19  }
```

The code editor highlights the 'getProducts()' method with a blue background.

Query controller class

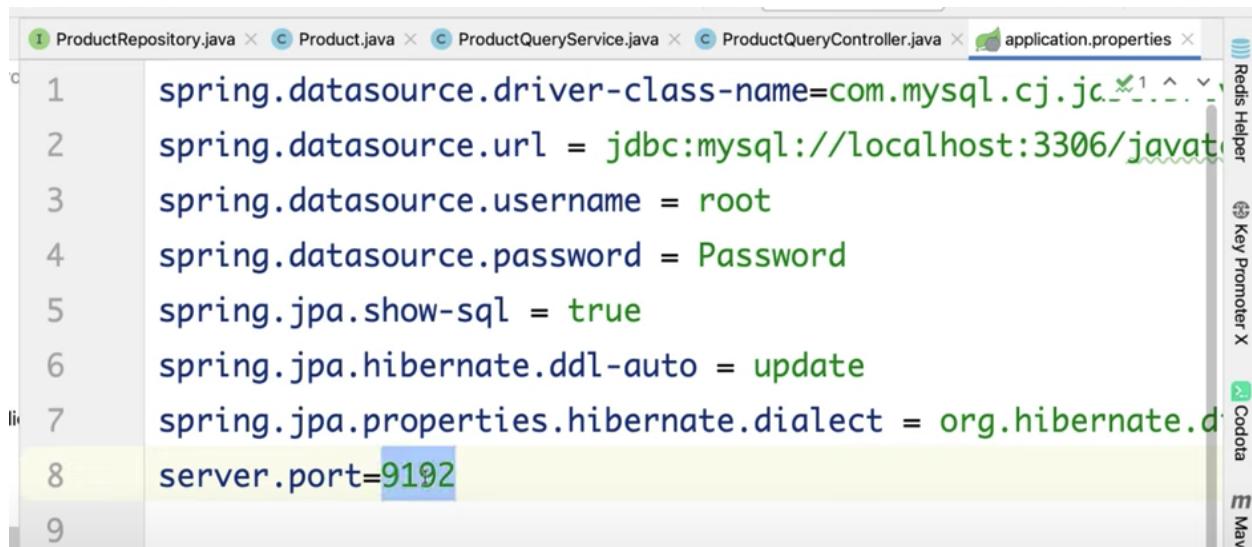


The screenshot shows the IntelliJ IDEA interface with the 'ProductQueryController.java' file selected in the top navigation bar. The code editor displays the following Java code:

```
12  @RequestMapping("/products")
13  @RestController
14  public class ProductQueryController {
15
16      @Autowired
17      private ProductQueryService queryService;
18
19      @GetMapping
20      public List<Product> fetchAllProducts(){
21          return queryService.getProducts();
22      }
23  }
```

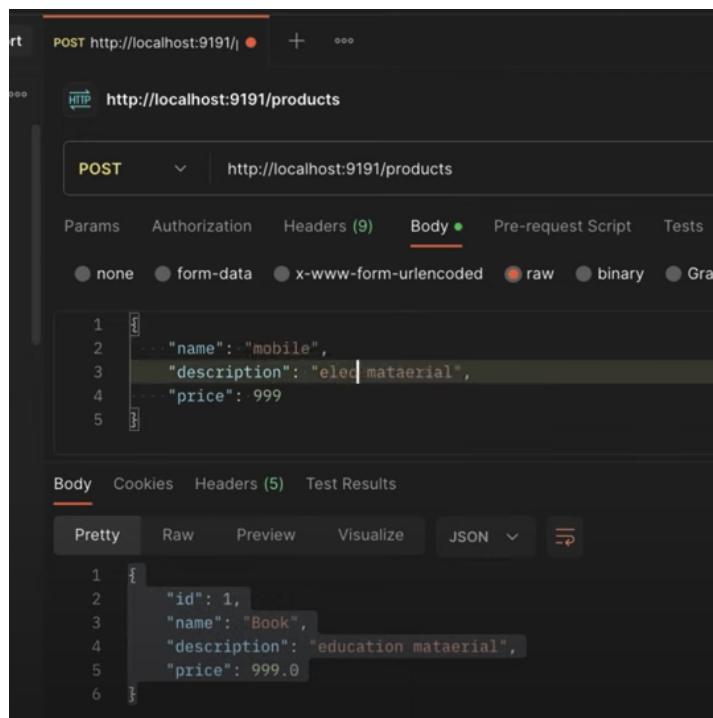
The code editor highlights the '@GetMapping' annotation with a yellow background.

Application properties of query service same as command service except port number



```
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url = jdbc:mysql://localhost:3306/javatpoint
spring.datasource.username = root
spring.datasource.password = Password
spring.jpa.show-sql = true
spring.jpa.hibernate.ddl-auto = update
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5InnoDBDialect
server.port=9192
```

Start the application and add some products



POST http://localhost:9191/

HTTP <http://localhost:9191/products>

POST http://localhost:9191/products

Params Authorization Headers (9) Body Pre-request Script Tests

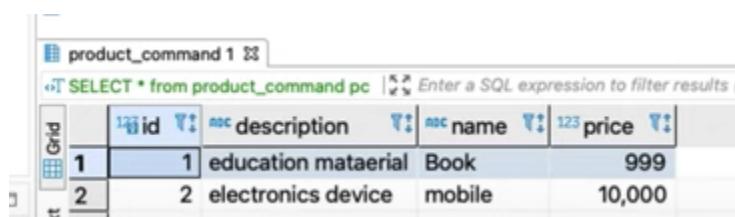
Body (raw JSON)

```
1 {
2   "name": "mobile",
3   "description": "electronics device",
4   "price": 999
5 }
```

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

```
1 {
2   "id": 1,
3   "name": "Book",
4   "description": "education mataerial",
5   "price": 999.0
6 }
```



product_command 1

SELECT * from product_command pc

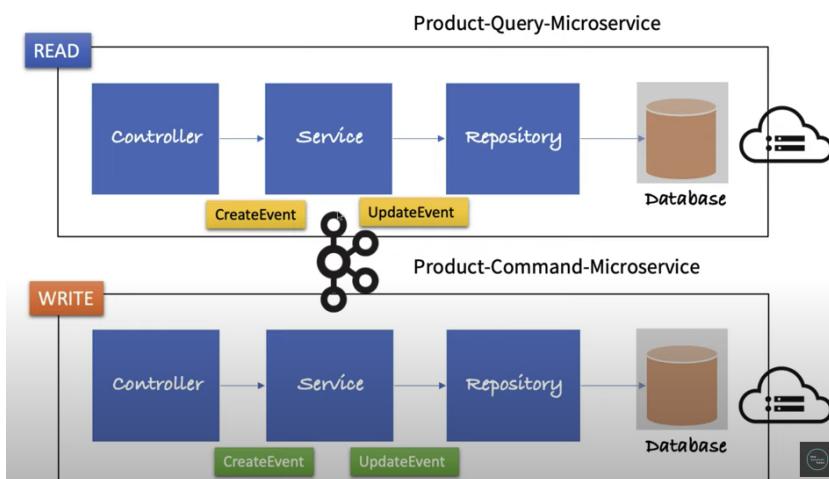
	id	description	name	price
1	1	education mataerial	Book	999
2	2	electronics device	mobile	10,000

When we query. We are getting empty response.

The screenshot shows the Postman interface. At the top, there are two tabs: 'PUT http://localhost:9191/p' and 'GET http://localhost:9292/c'. Below them is a main header for 'http://localhost:9292/products'. The 'Params' tab is selected, showing a table with one row: 'Key' (empty) and 'Value' (empty). The 'Body' tab is selected, showing a JSON response with a single object containing an empty array: { "products": [] }. The status bar at the bottom indicates '200 OK' with a 418 ms response time and 166 B size.

The screenshot shows the DBeaver interface with a SQL query window titled 'product_query 1'. The query is 'SELECT * FROM product_query pq'. The results grid is empty, showing four columns: 'id', 'description', 'name', and 'price'.

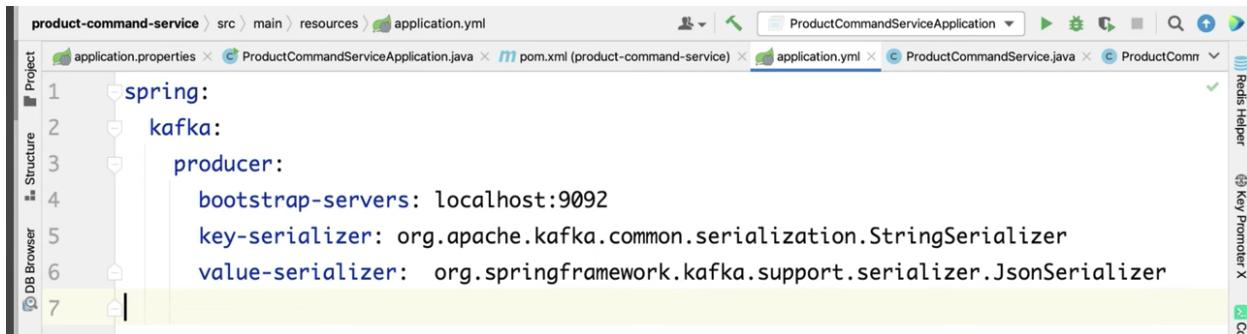
Now Kafka come into picture



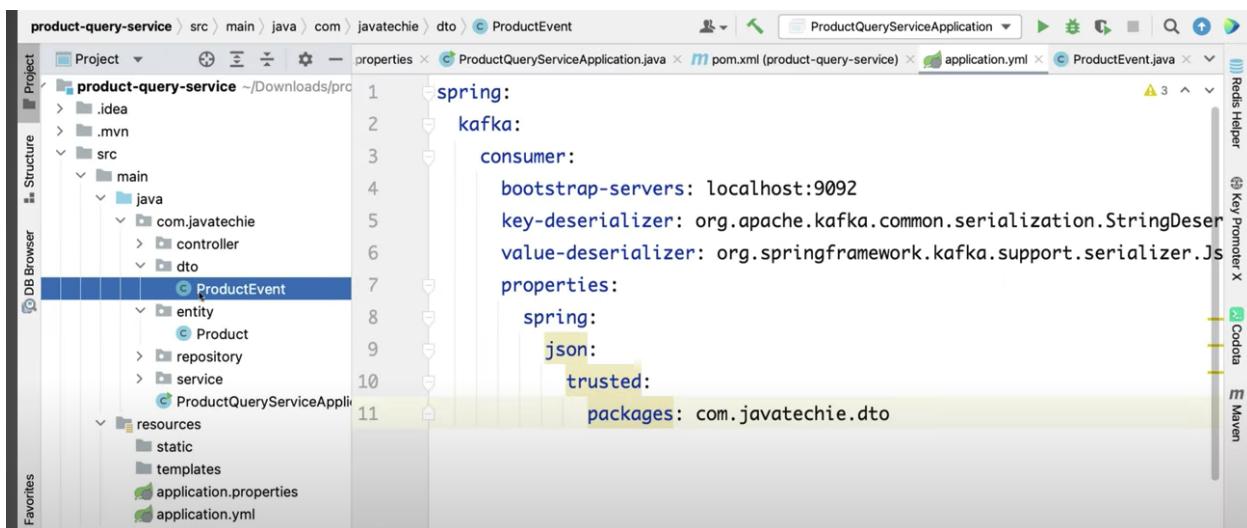
Command service should act as producer and query service should act as listener for each create and update events.

```
</dependency>
<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka</artifactId>
</dependency>
```

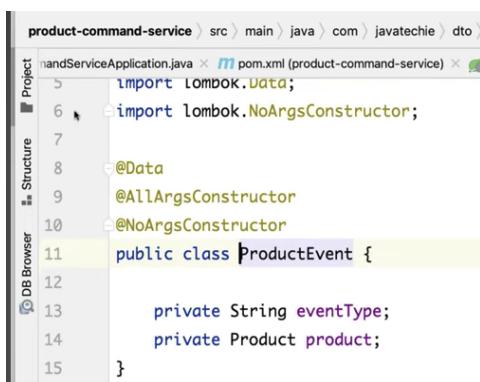
Add this dependency in the both services



```
spring:
  kafka:
    producer:
      bootstrap-servers: localhost:9092
      key-serializer: org.apache.kafka.common.serialization.StringSerializer
      value-serializer: org.springframework.kafka.support.serializer.JsonSerializer
```



```
spring:
  kafka:
    consumer:
      bootstrap-servers: localhost:9092
      key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
      value-deserializer: org.springframework.kafka.support.serializer.JsonDeserializer
  properties:
    spring:
      json:
        trusted:
          packages: com.javatechie.dto
```



```
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor
public class ProductEvent {
```

```
    private String eventType;
    private Product product;
}
```

Create a product event class which takes the event type and what is the product.

Update service class with kafka to send product event on creation and update

The screenshot shows an IntelliJ IDEA interface with several tabs at the top: main, java, com, javatechie, service, ProductCommandService, createProduct, ProductCommandServiceApplication, ProductApplication.java, pom.xml (product-command-service), application.yml, ProductEvent.java, ProductCommandService.java, and ProductCommandController.java. The ProductCommandService.java tab is active.

```
    @Autowired  
    private KafkaTemplate<String, Object> kafkaTemplate;  
  
    public Product createProduct(ChangeEvent productEvent){  
        Product productDO = repository.save(productEvent.getProduct());  
        ProductEvent event=new ProductEvent(eventType: "CreateProduct", productDO);  
        kafkaTemplate.send(topic: "product-event-topic", event);  
        return productDO;  
    }  
  
    public Product updateProduct(long id,ChangeEvent productEvent){  
        Product existingProduct = repository.findById(id).get();  
        Product新产品=productEvent.getProduct();  
        existingProduct.setName(newProduct.getName());  
        existingProduct.setPrice(newProduct.getPrice());  
        existingProduct.setDescription(newProduct.getDescription());  
        Product productDO = repository.save(existingProduct);  
        ProductEvent event=new ProductEvent(eventType: "UpdateProduct", productDO);  
        kafkaTemplate.send(topic: "product-event-topic", event);  
        return productDO;  
    }
```

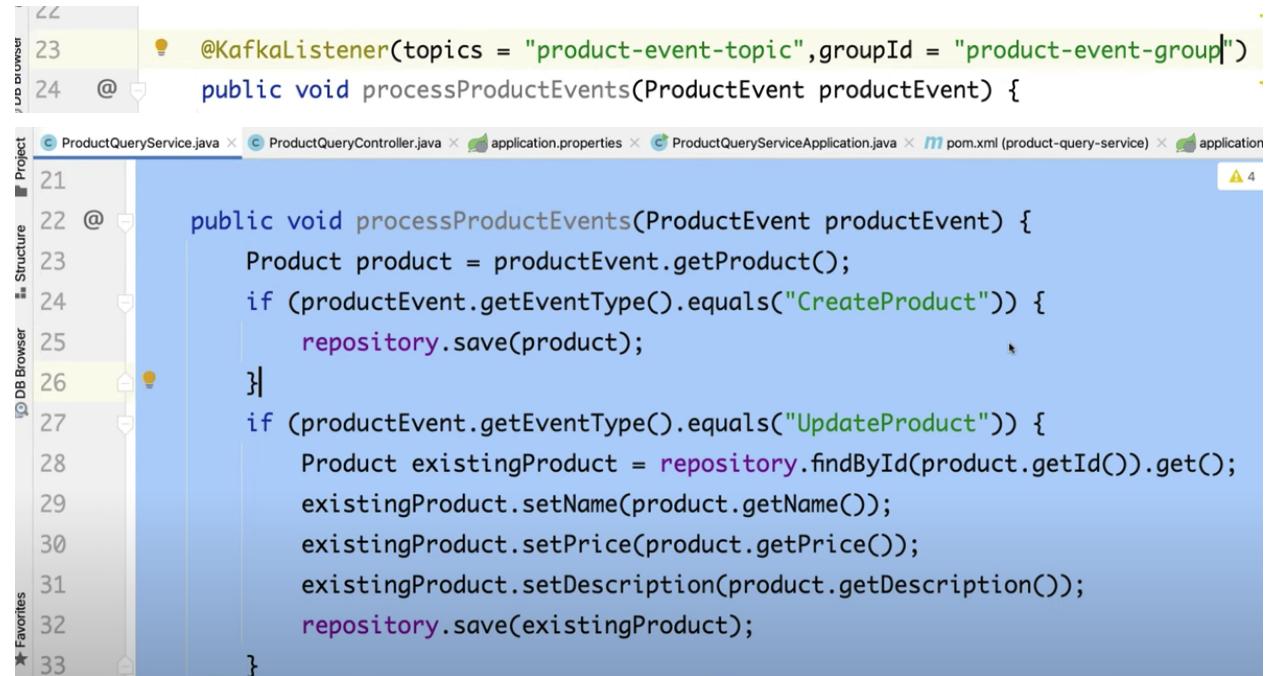
Update controller class accordingly.

```
15  
16     @PostMapping  
17     public Product createProduct(@RequestBody ProductEvent productEvent) {  
18         return commandService.createProduct(productEvent);  
19     }
```

```
@PutMapping("/{id}")
public Product updateProduct(@PathVariable long id, @RequestBody ProductEvent productEvent) {
    return commandService.updateProduct(id, productEvent);
}
```

Now consumer part

Update query service class with product events



The screenshot shows an IDE interface with a code editor containing Java code. The code is annotated with Kafka annotations: `@KafkaListener` and `@Transactional`. It defines a method `processProductEvents` that takes a `ProductEvent` parameter. Inside the method, it retrieves a `Product` from a repository based on the event type. If the event type is "CreateProduct", it saves the new product. If the event type is "UpdateProduct", it finds the existing product by ID and updates its name, price, and description before saving it again. The code editor has syntax highlighting and code completion features.

```
22
23     @KafkaListener(topics = "product-event-topic", groupId = "product-event-group")
24     public void processProductEvents(ProductEvent productEvent) {
25
26         public void processProductEvents(ProductEvent productEvent) {
27             Product product = productEvent.getProduct();
28             if (productEvent.getEventType().equals("CreateProduct")) {
29                 repository.save(product);
30             }
31             if (productEvent.getEventType().equals("UpdateProduct")) {
32                 Product existingProduct = repository.findById(product.getId()).get();
33                 existingProduct.setName(product.getName());
34                 existingProduct.setPrice(product.getPrice());
35                 existingProduct.setDescription(product.getDescription());
36                 repository.save(existingProduct);
37         }
```

Now add some products

HTTP <http://localhost:9191/products>

POST <http://localhost:9191/products>

Params Authorization Headers (9) Body **JSON** Pre-request Script Tests Settings

Body Cookies Headers (5) Test Results

200 OK 1207 ms 241 B

```

1 {
2   "type": "CreateProduct",
3   "product": [
4     {
5       "name": "Watch",
6       "description": "Samsung latest model",
7       "price": 23000
8     }
9   ]
10 }
```

Pretty Raw Preview Visualize JSON

product_command 1

SELECT * from product_command pc

	Grid	Text	SQL	Copy	Delete	Edit	Details	Logs
	Grid	Text	SQL	Copy	Delete	Edit	Details	Logs
1	1	education mataerial	Book	999				
2	2	electronics device	mobile	20,000				
3	52	Samsung latest mod	Watch	23,000				
4	53	JBL	earphone	2,500				

product_query 1

SELECT * FROM product_query pq

	Grid	Text	SQL	Copy	Delete	Edit	Details	Logs
	Grid	Text	SQL	Copy	Delete	Edit	Details	Logs
1	1	Samsung latest model	Watch	23,000				
2	2	JBL	earphone	2,500				

Because id is not in sync just delete the records and start test the producer with update and create events.