Building Event-Driven Architecture

With

Spring Event And Spring Modulith

Stone Huang

JCConf Taiwan 2024

About Stone Huang

- XREX Inc.
- 金融業、電信業、系統整合商、新創軟體公司等
- 系統架構設計、分散式系統開發、區塊鏈應用等
- 證照 : AWS、CKA、CKAD、SCJP、SCWCD、Microsoft SQL Server、CCNA、Neo4j及相關金融證照
- 興趣 : 籃球，觀察生態(甲蟲類)

Agenda

- What is EDA
- Spring Event Lifecycle
- Spring Modulith Event
- Event Error Handling

What Is EDA

Event-driven architecture (EDA) is a microservice architectural pattern that utilizes asynchronous communication triggered by events rather than conventional request-response patterns.

EDA enhances scalability, responsiveness, and real-time processing, making it ideal for modern, dynamic applications across diverse domains.

What Is Event

An event in EDA refers to a significant occurrence or action within a system, typically triggering responses or processes.

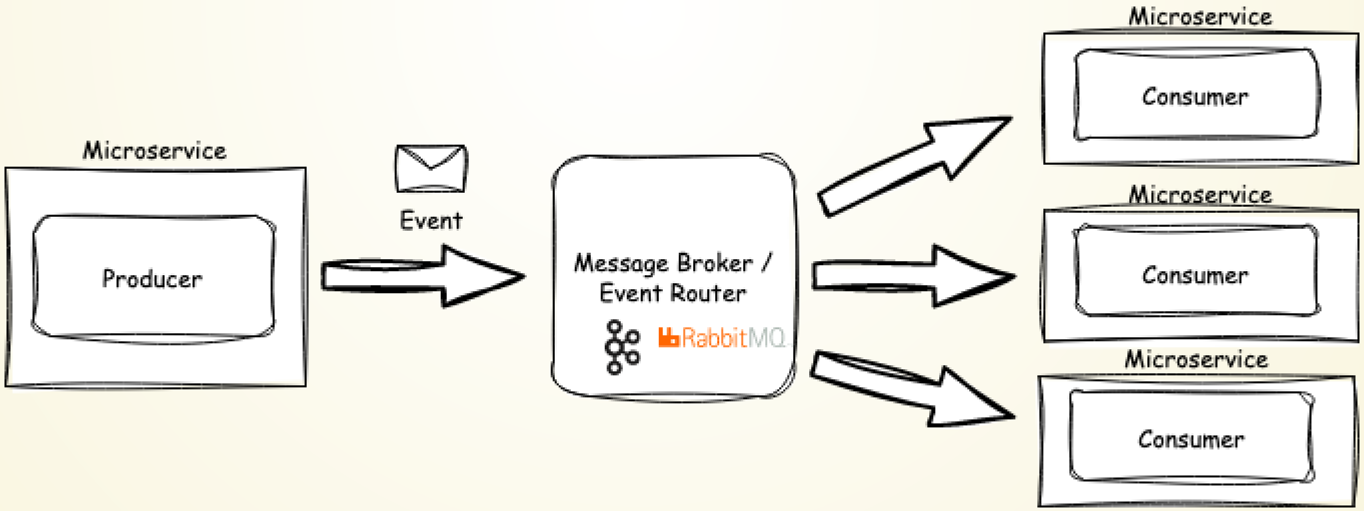
What Is Event

Examples :

- Business/Domain Event (e.g. order created, successfully sign up)
- System Event (e.g. application launches or shutdown)
- Alert Event (e.g. price alert in exchange)
- Period Event (Cron job triggers)
- etc

Three Parts Of Typical EDA

- *Producer*
- *Message Broker / Event Router*
- *Consumer*



Benefits Of EDA

- Loosely-coupled architecture
- Scalibility of Consumers
- Async processing
- Single Responsibility Principle(SRP) and Open-Closed Principle (OCP).

EDA Within Applicaiton

Typical EDA is a distributed system achieved by EXTERNAL events.

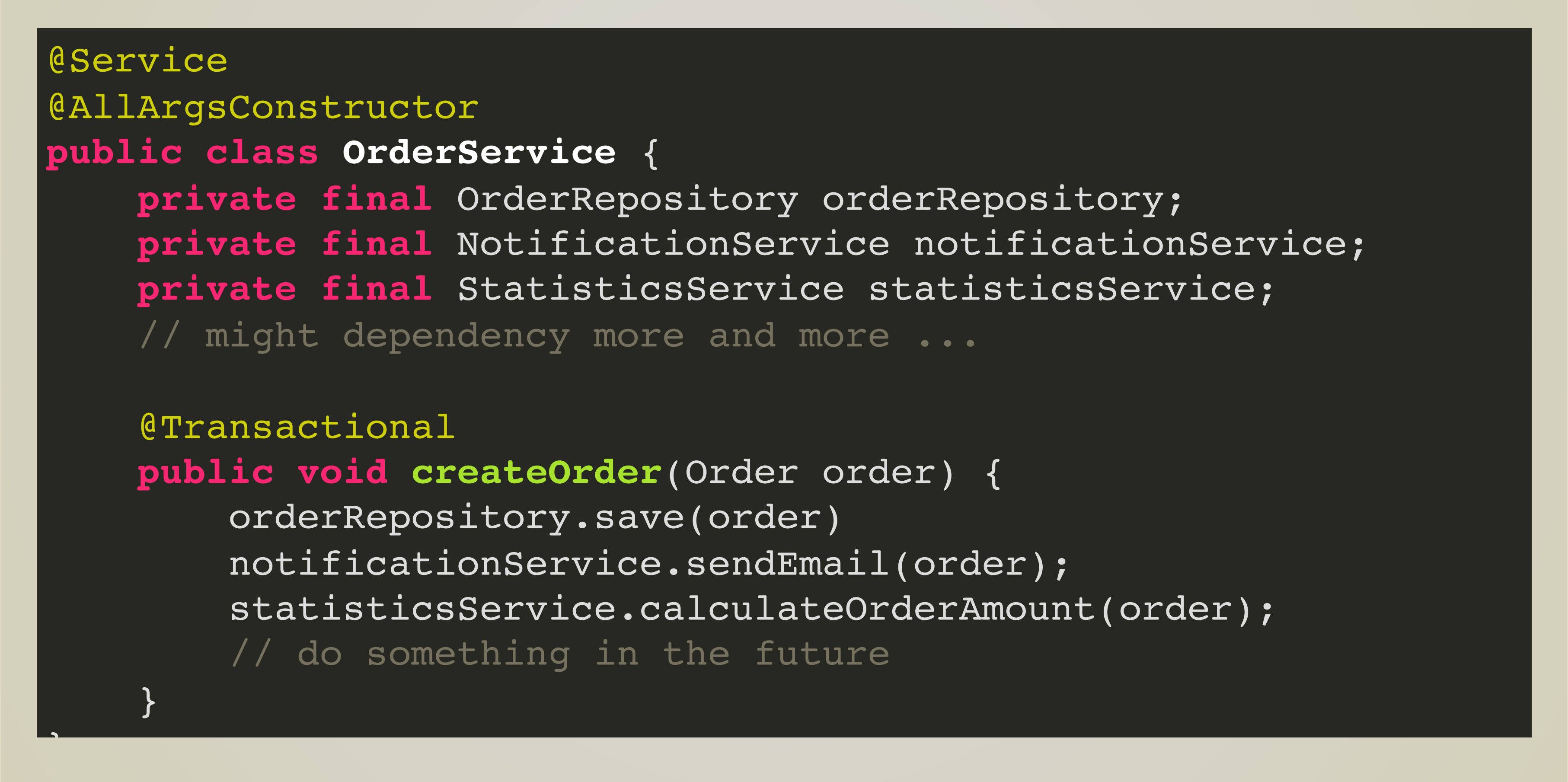
But, how about INTERNAL events communication within an application ?

Scenario

在一個沒有其它微服務的系統架構下，一個單體式的應用服務"訂單系統"，在完成訂單後

- 需要發送信件給客戶
- 並且需要累計銷量供後續管理人員查詢
- And more in the future

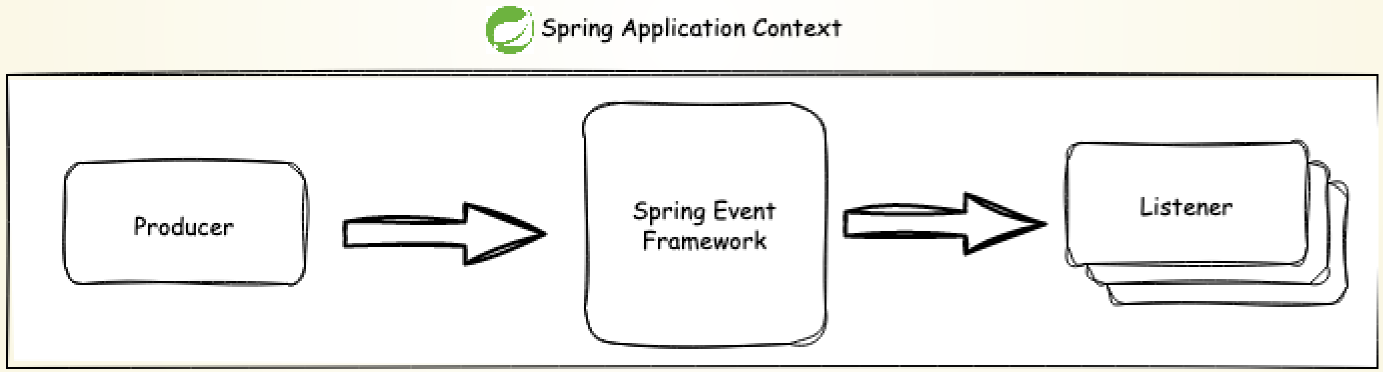
Sample Code



```java
@Service
@AllArgsConstructor
public class OrderService {
    private final OrderRepository orderRepository;
    private final NotificationService notificationService;
    private final StatisticsService statisticsService;
    // might dependency more and more ...

    @Transactional
    public void createOrder(Order order) {
        orderRepository.save(order)
        notificationService.sendEmail(order);
        statisticsService.calculateOrderAmount(order);
        // do something in the future
    }
}
```

EDA With Spring Event

use build-in Spring Event Framework



Spring Event Introduction

Spring Event is a standard observer design pattern that implements Pub-Sub mechanism in Spring Application. A fundamental part of Spring Framework.

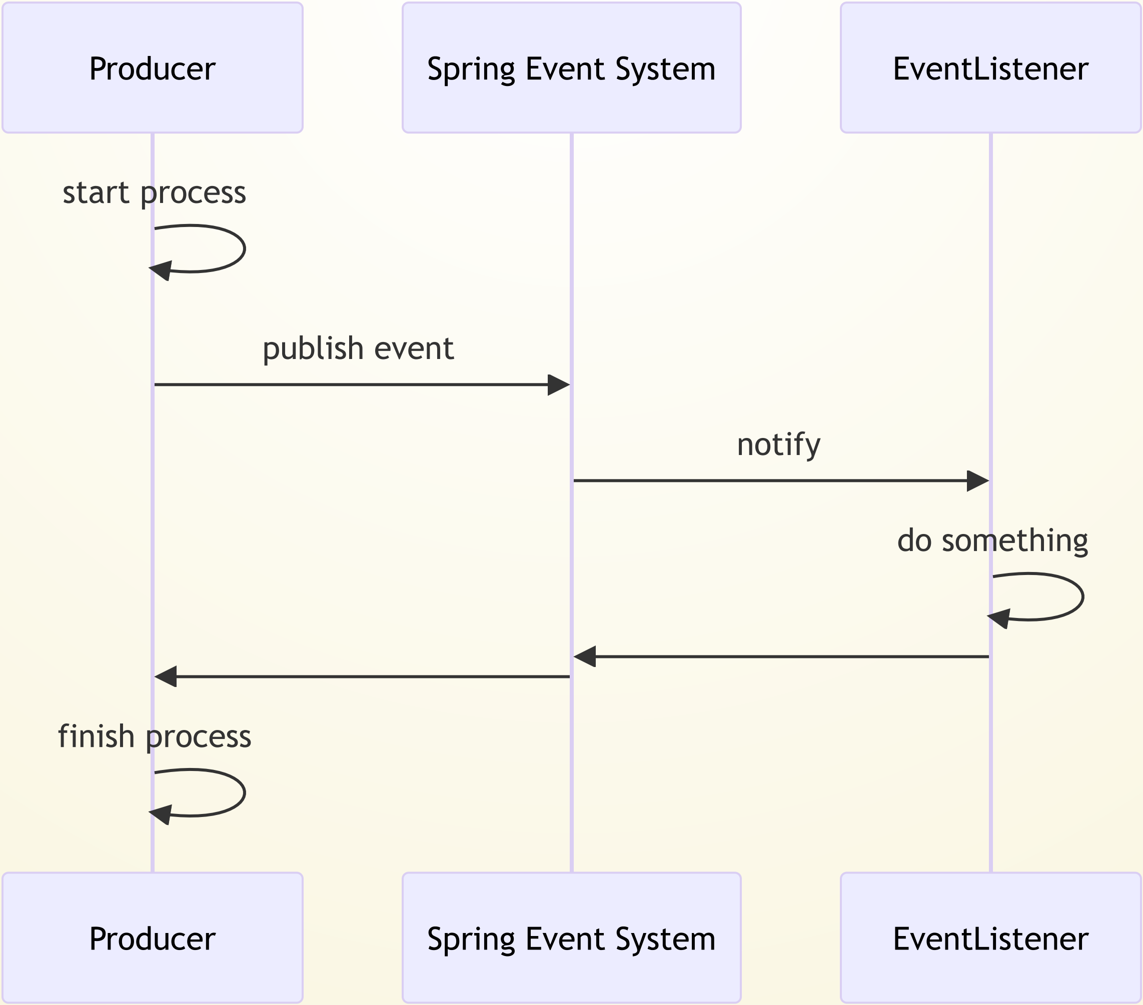
By leveraging Spring *ApplicationEventPublisher* and *@EventListener* and other components to achieve this goal.

Spring Version

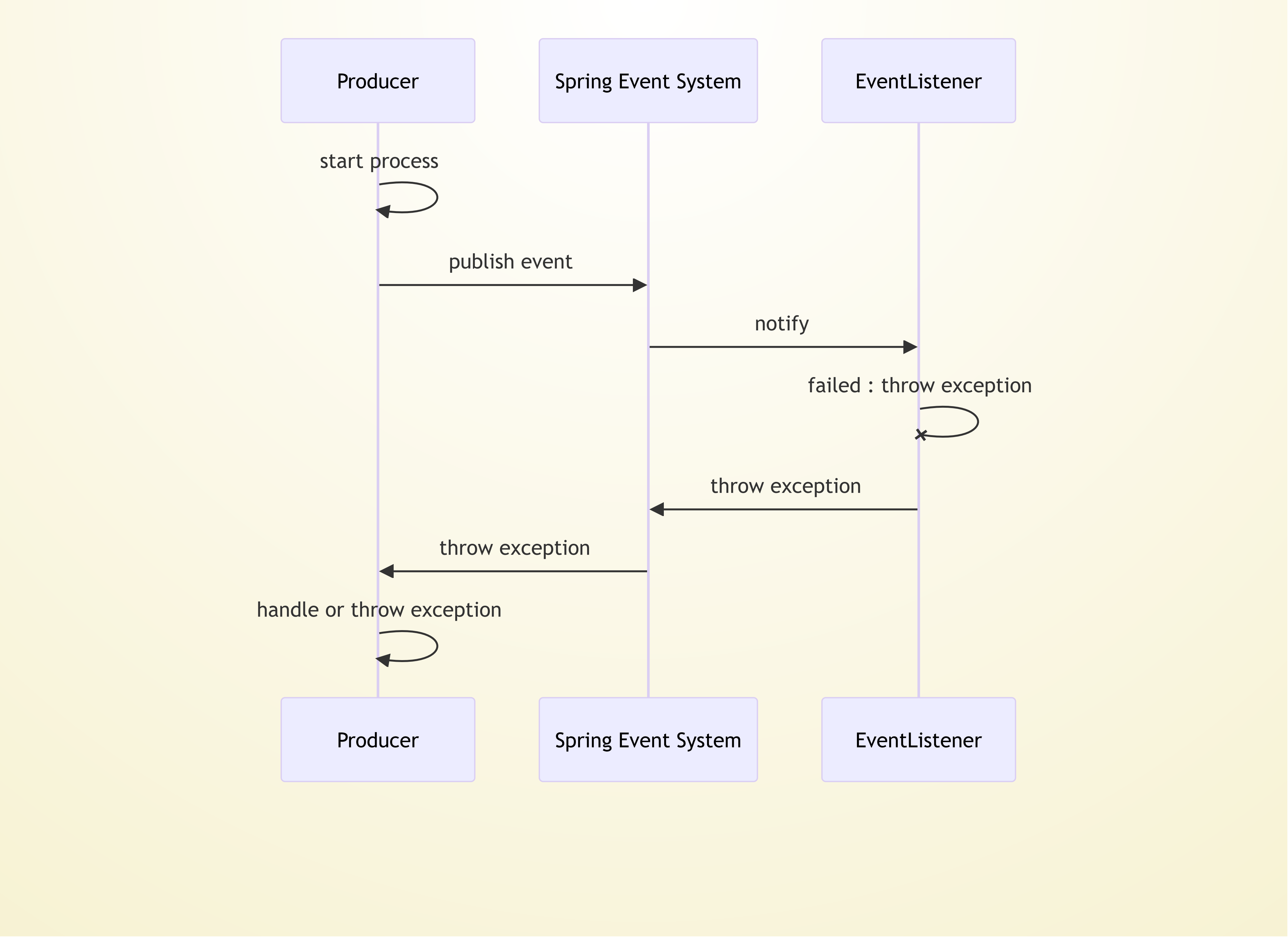
- JDK 17
- Springboot 3.2
- Spring Framework 6.1

Use Spring Event Listener (1)

Regular event listener (assume listener has subscribed)


Producer
Spring Event System
EventListener
start process
publish event
notify
do something
finish process
Producer
Spring Event System
EventListener

Use Spring Event Listener (2)


Producer
Spring Event System
EventListener
start process
publish event
notify
failed : throw exception
throw exception
throw exception
handle or throw exception
Producer
Spring Event System
EventListener

Define Your Event Object

```java
public record OrderCreatedEvent(Order order) {}
```

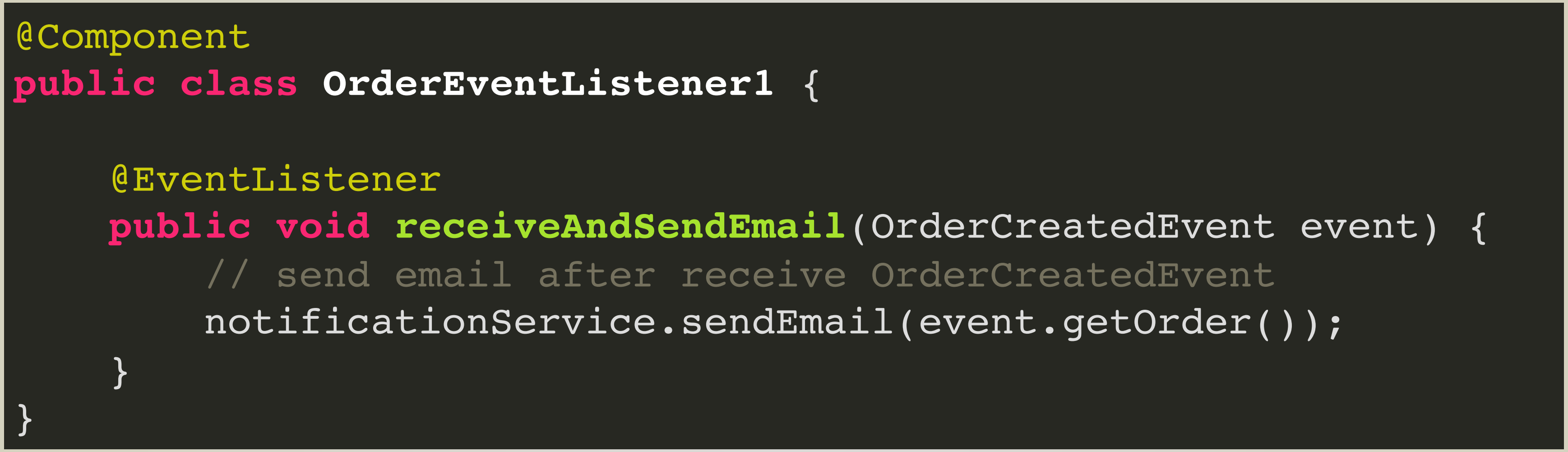
Publish Your Event

use *ApplicationEventPublisher*

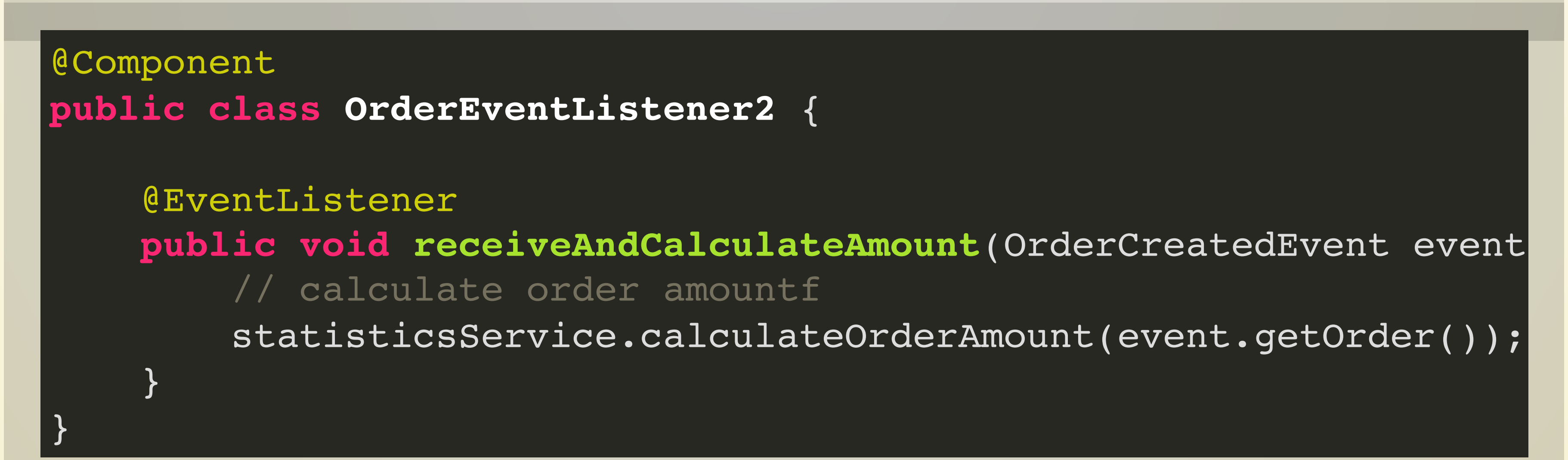
```
@Service
@AllArgsConstructor
public class OrderService {
    private final OrderRepository orderRepository;
    private final ApplicationEventPublisher publisher;

    @Transactional
    public void createOrder(Order order) {
        orderRepository.save(order)

        publisher.publish(new OrderCreatedEvent(order));

        // no need any more
        // notificationService.sendEmail(order);
        // statisticsService.calculateOrderAmount(order);
    }
```

Create Event Listener

use ***@EventListener***



```
@Component
public class OrderEventListener1 {

    @EventListener
    public void receiveAndSendEmail(OrderCreatedEvent event) {
        // send email after receive OrderCreatedEvent
        notificationService.sendEmail(event.getOrder());
    }
}
```

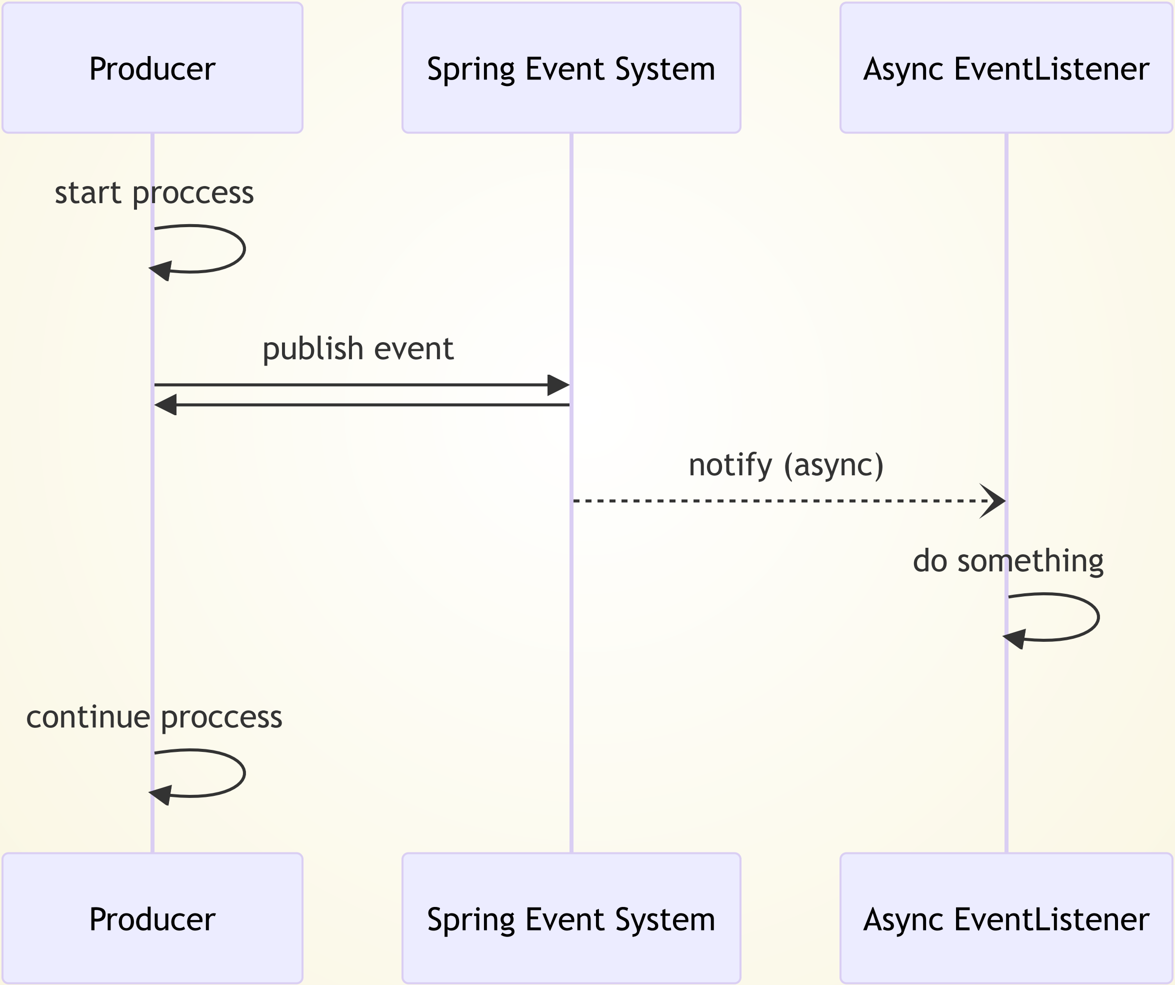


```
@Component
public class OrderEventListener2 {

    @EventListener
    public void receiveAndCalculateAmount(OrderCreatedEvent event
        // calculate order amountf
        statisticsService.calculateOrderAmount(event.getOrder());
    }
}
```

Async Event

By default, regular event listener is synchronous runs on the same thread with producer. If your operations of a listener take long time, It is better to to use asynchronous event listeners.

Async Event


Producer
Spring Event System
Async EventListener
start proccess
publish event
notify (async)
do something
continue proccess
Producer
Spring Event System
Async EventListener

Async Event

use *@Async*

```
@Component
public class OrderEventListener {

    @Async
    @EventListener
    public void receiveAndSendEmail(OrderCreatedEvent event) {...

}
```

Event Ordering

How to control the execution order of listeners

Event Ordering

use ***@Order***

```
@Component
public class OrderEventListener1 {

    @EventListener
    @Order(1)
    public void receiveAndSendEmail(OrderCreatedEvent event) {...

}
```

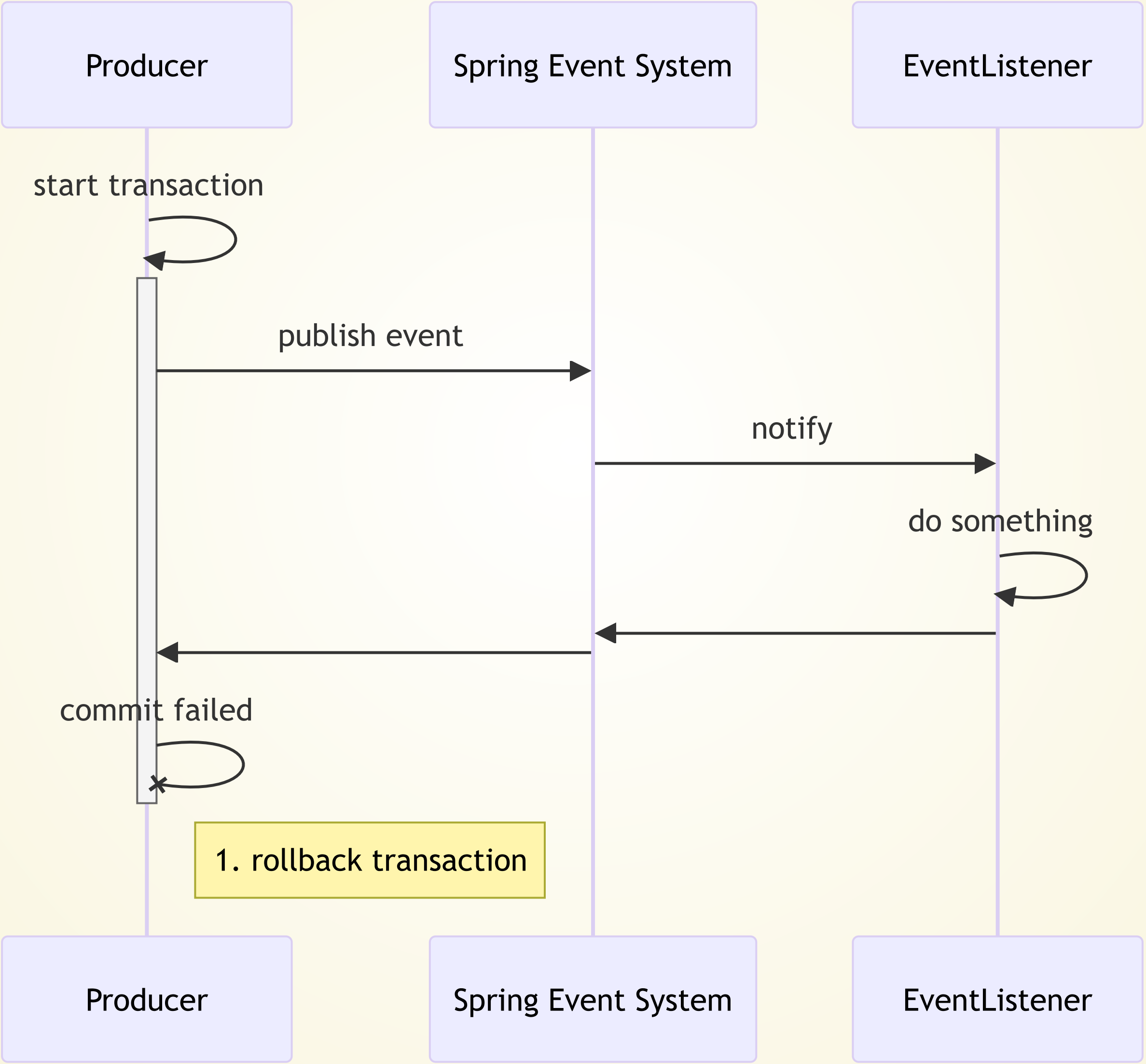
```
@Component
public class OrderEventListener2 {

    @EventListener
    @Order(2)
    public void receiveAndCalculateAmount(OrderCreatedEvent event

}
```

DEMO

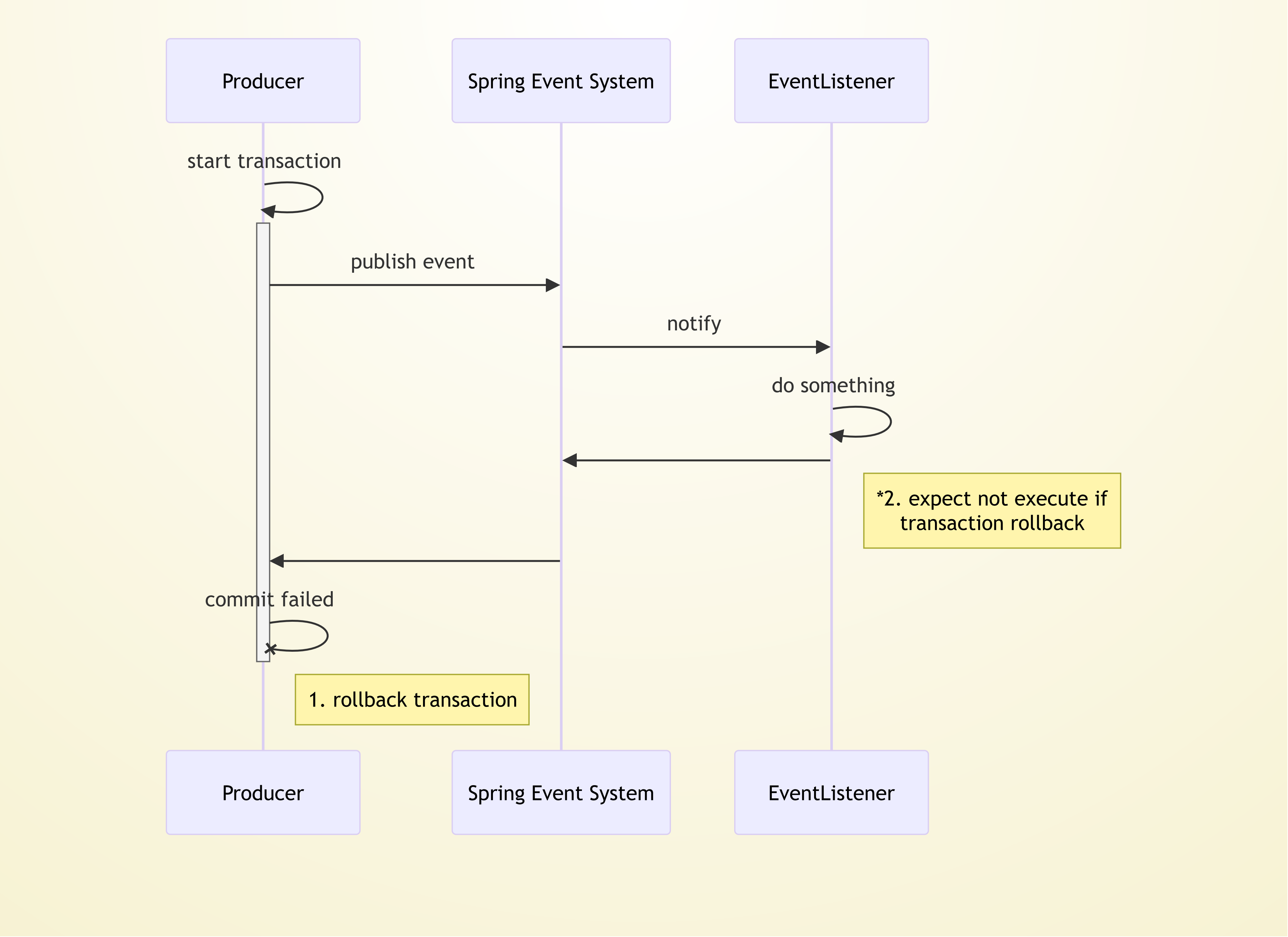
Question 1

What if the producer side failed (transaction rollback) but the listener already took action?

Transaction Rollback (1)



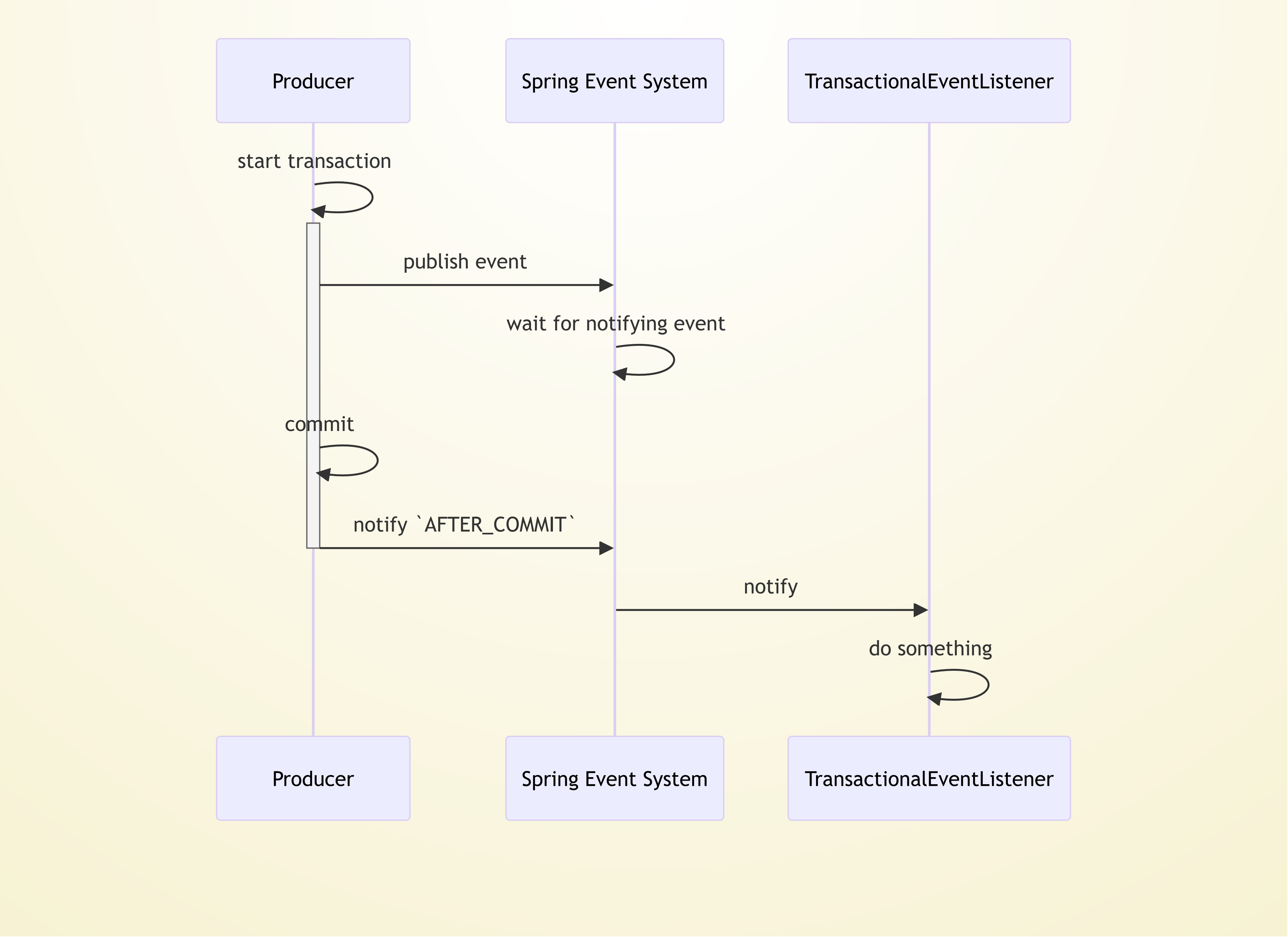
Transaction Rollback (2)


Producer
Spring Event System
EventListener
start transaction
publish event
notify
do something
*2. expect not execute if transaction rollback
commit failed
1. rollback transaction
Producer
Spring Event System
EventListener

Use Transactional Event Listener

> *“ Since Spring 4.2, the listener of an event can be bound to a phase of the transaction. ”*

Use Transactional Event Listener


Producer
Spring Event System
TransactionalEventListener
start transaction
publish event
wait for notifying event
commit
notify `AFTER_COMMIT`
notify
do something
Producer
Spring Event System
TransactionalEventListener

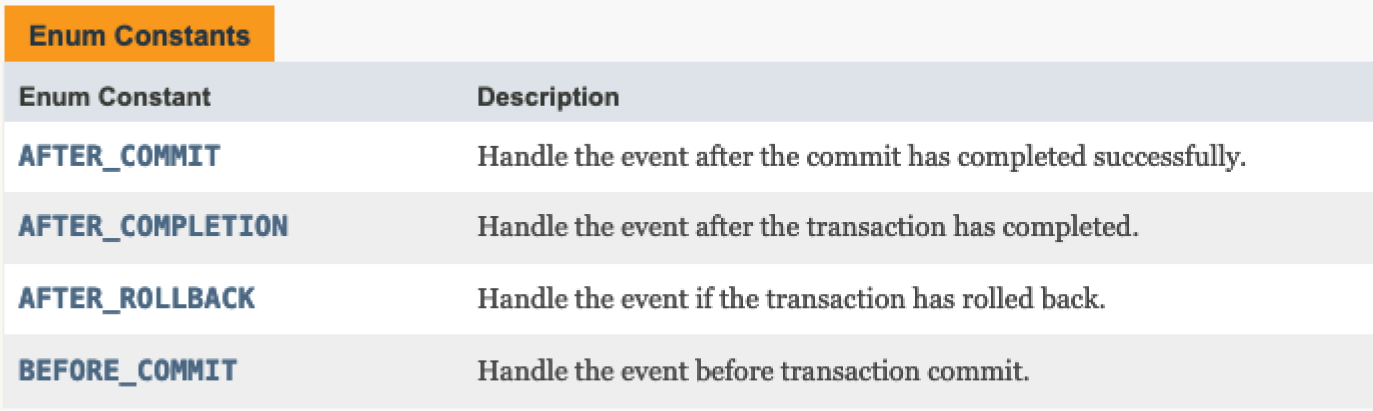
Use Transactional Event Listener

use ***@TransactionalEventListener*** on listener

```
@Component
public class OrderEventListener {

    @Async
    // @EventListener
    @TransactionalEventListener
    public void receiveAndSendEmail(OrderCreatedEvent event) {...

}
```

Transactional Event Phase

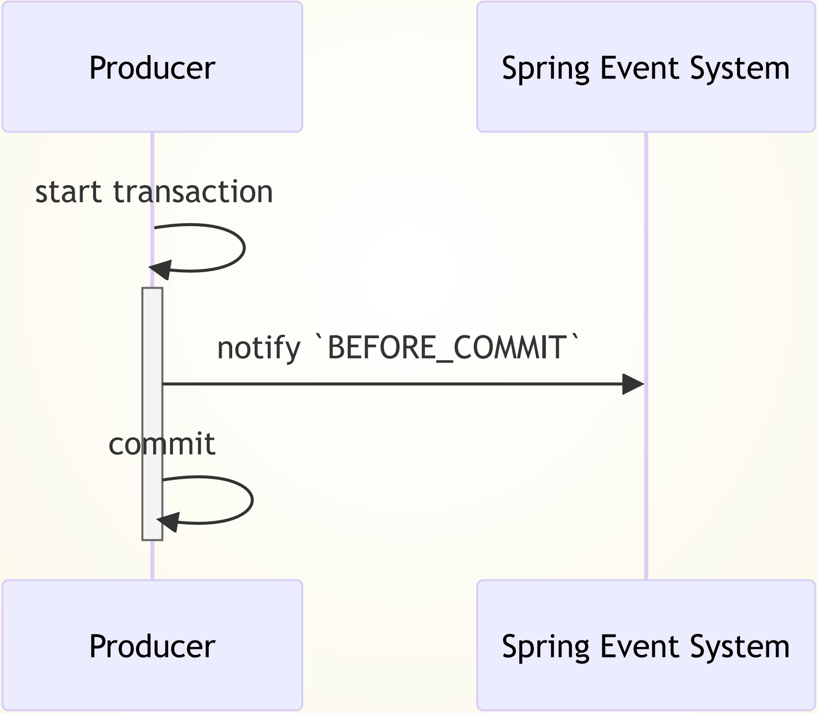
TransactionalEventListener.TransactionPhase



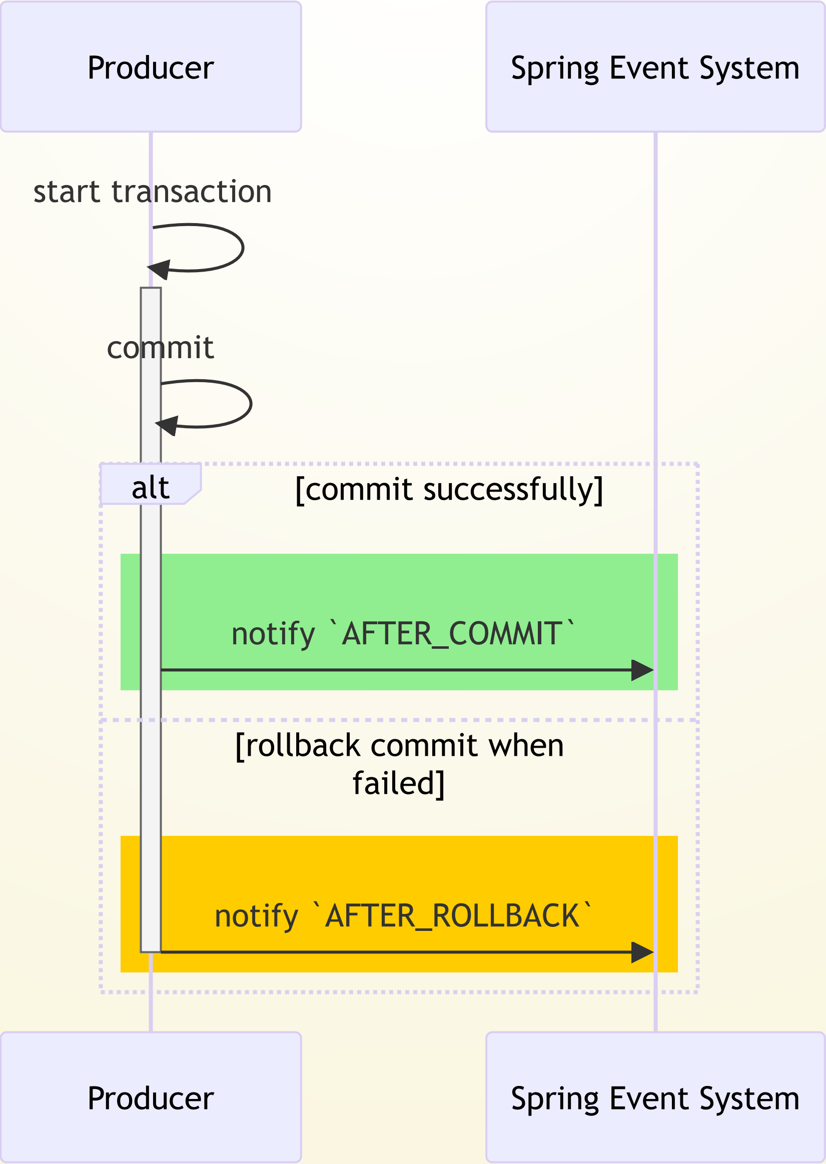
Enum Constants

Enum Constant	Description
AFTER_COMMIT	Handle the event after the commit has completed successfully.
AFTER_COMPLETION	Handle the event after the transaction has completed.
AFTER_ROLLBACK	Handle the event if the transaction has rolled back.
BEFORE_COMMIT	Handle the event before transaction commit.

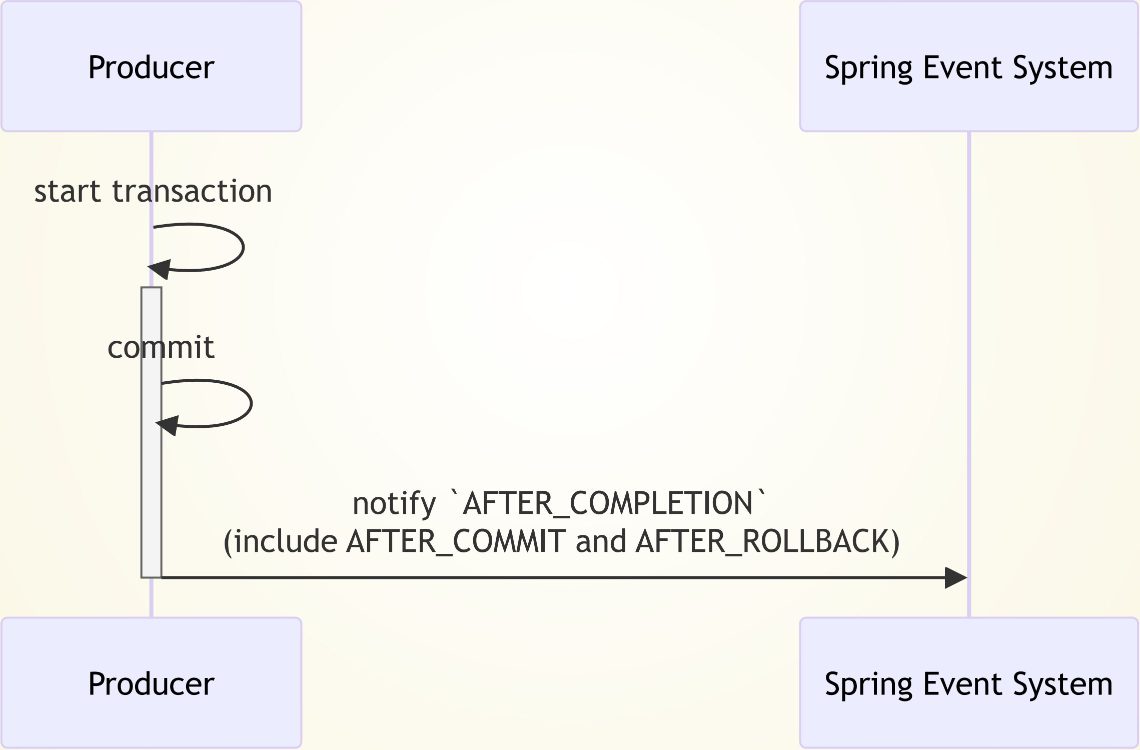
Transactional Event Phase



Transactional Event Phase


Producer
Spring Event System
start transaction
commit
alt
[commit successfully]
notify `AFTER_COMMIT`
[rollback commit when failed]
notify `AFTER_ROLLBACK`
Producer
Spring Event System

Transactional Event Phase



DEMO

Question 2

What if event listener occurs failure ?

The business logic inside the listener is incompleted, it might cause data inconsistency.

Use Spring Modulith Event

Make use of **Spring Modulith Event**.

Spring Modulith Introduction (1)

Spring Modulith is an opinionated toolkit to build domain-driven, modular applications with Spring Boot.

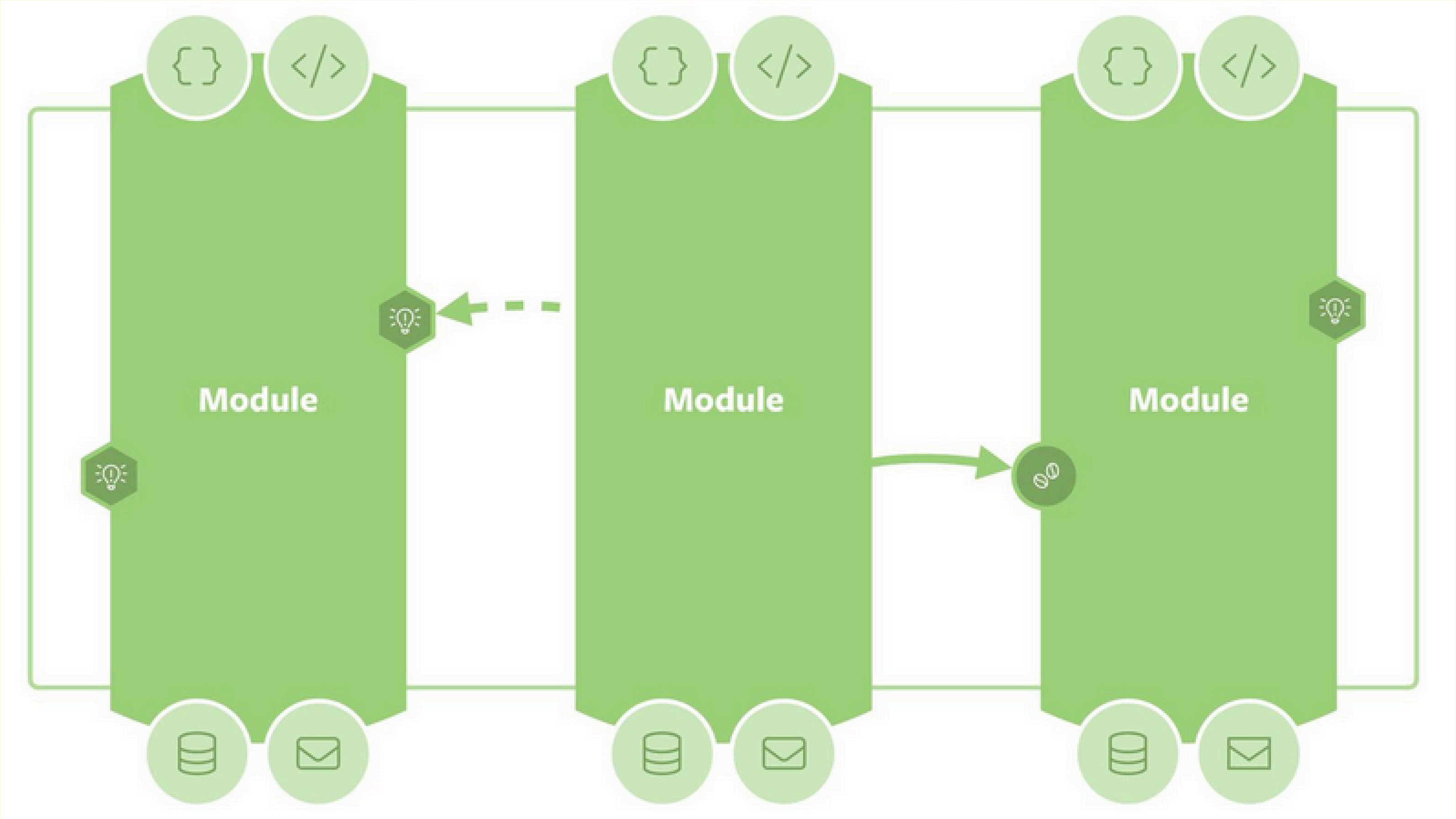
Modulith = Monolithic + Modular Application

Spring Modulith Introduction (2)

Modulize domains within Spring application.

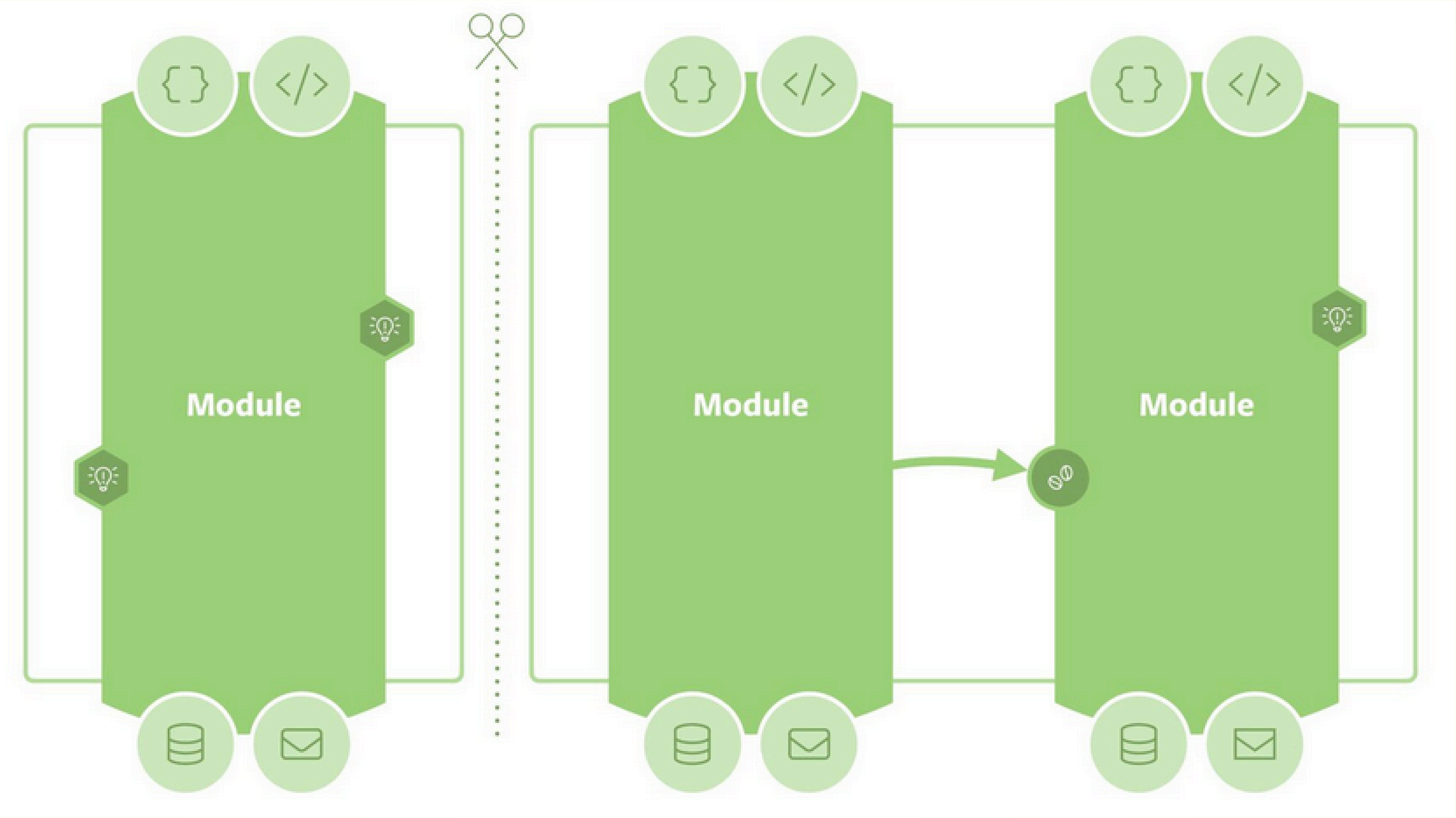
```
Example
└─ src/main/java
   ├─ example
   │  └─ Application.java
   │
   ├─ example.inventory      <-- Inventory Domain
   │  ├─ InventoryManagement.java
   │  └─ SomethingInventoryInternal.java
   │
   ├─ example.order          <-- Order Domain
   │  └─ OrderManagement.java
   └─ example.order.internal
      └─ SomethingOrderInternal.java
```

Spring Modulith Introduction (3)



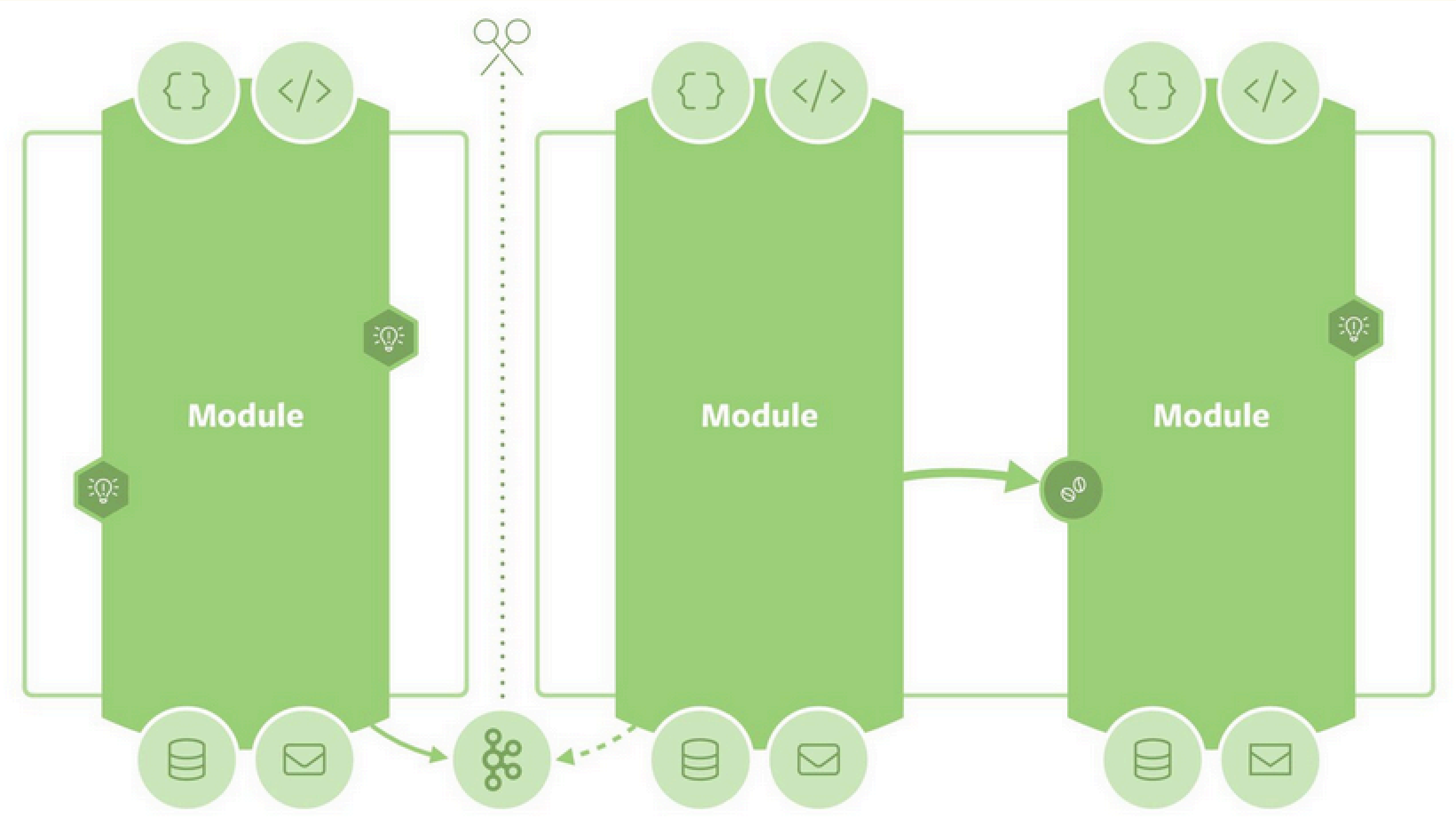
source [https://speakerdeck.com/olivergierke/spring-modulith-a-deep-dive]

Spring Modulith Introduction (4)



source [https://speakerdeck.com/olivergierke/spring-modulith-a-deep-dive]

Spring Modulith Introduction (5)

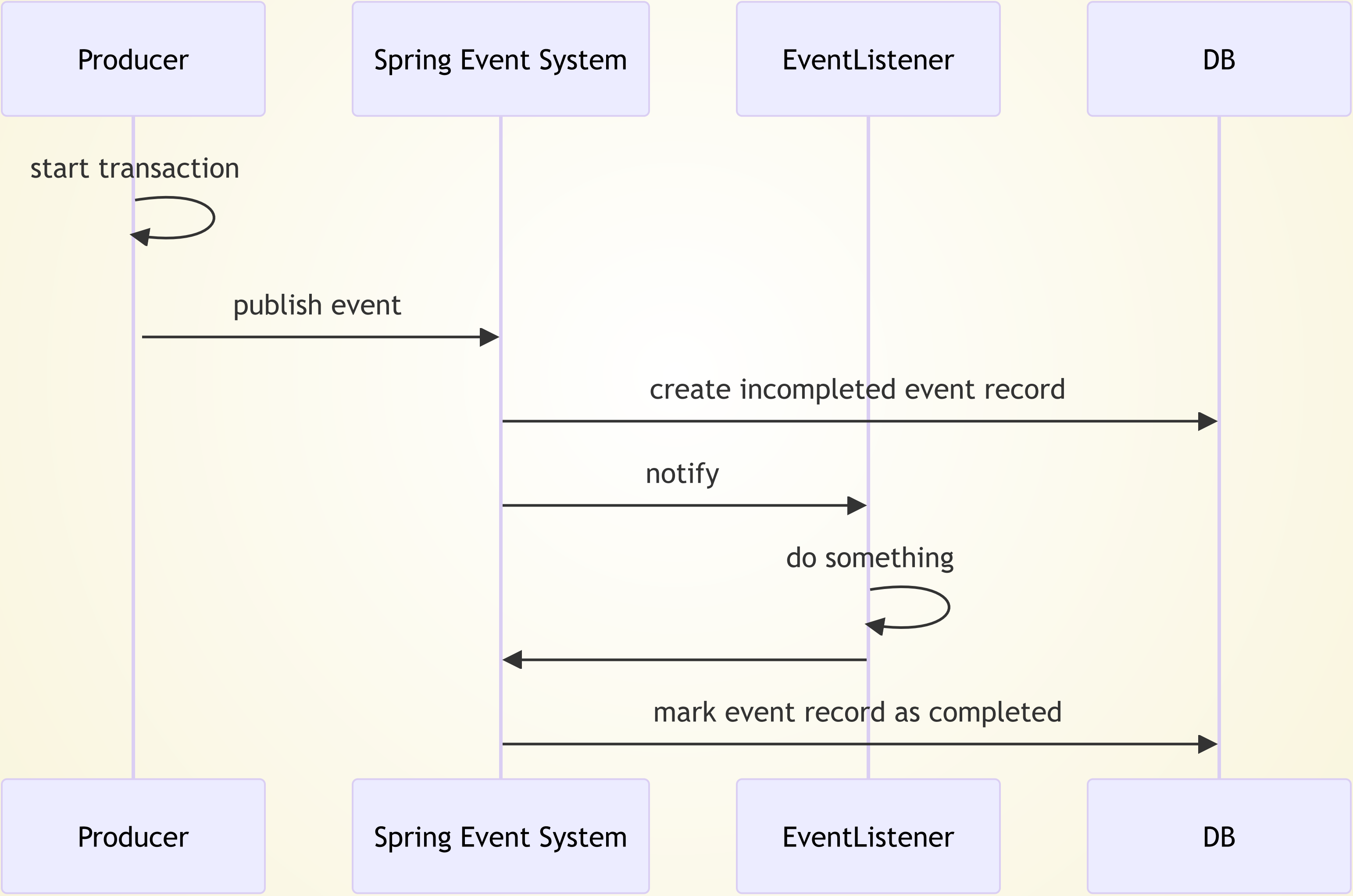


source [https://speakerdeck.com/olivergierke/spring-modulith-a-deep-dive]

Spring Modulith Version

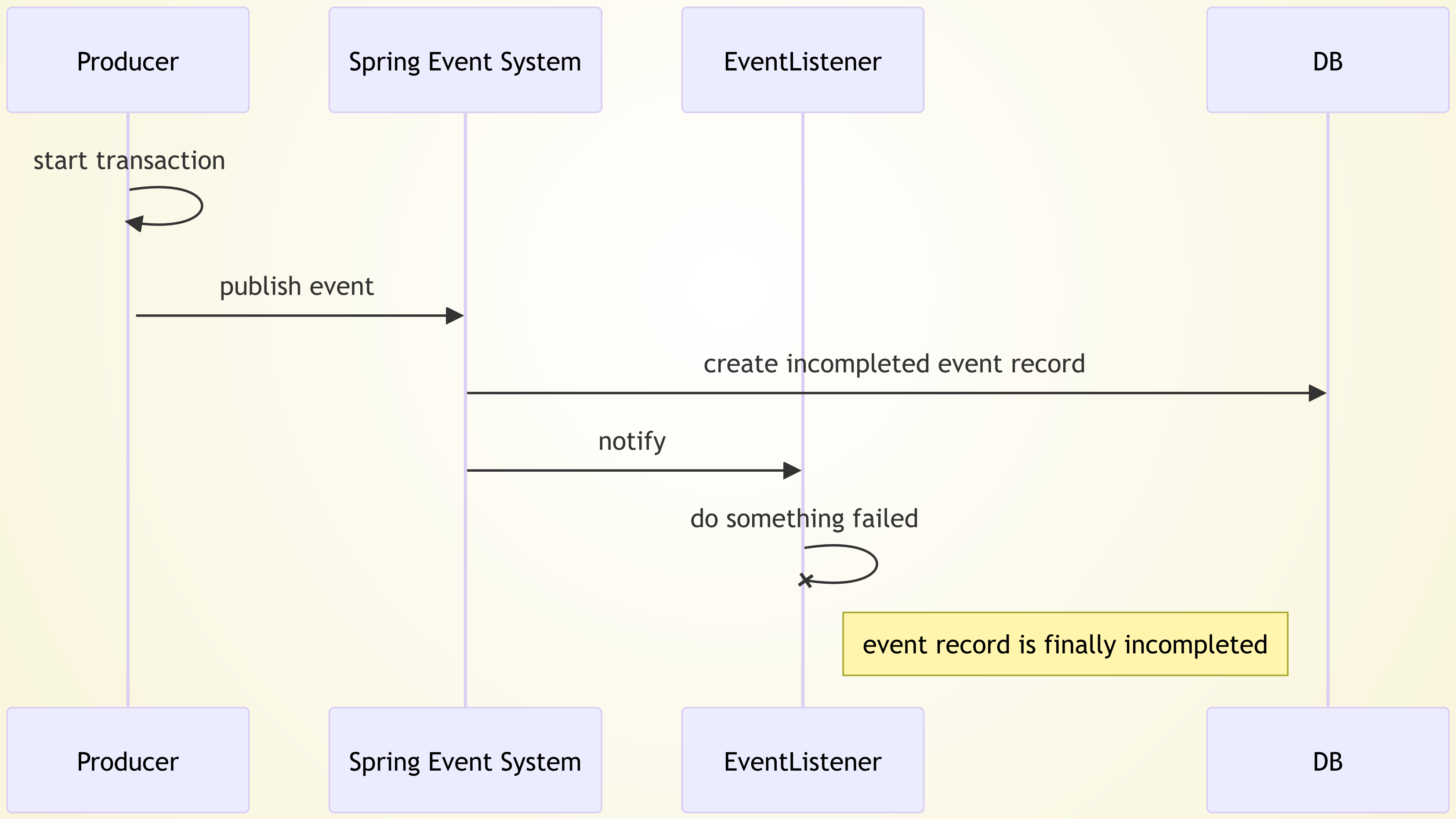
SpringModulith : 1.1.2

Spring Modulith Event (1)



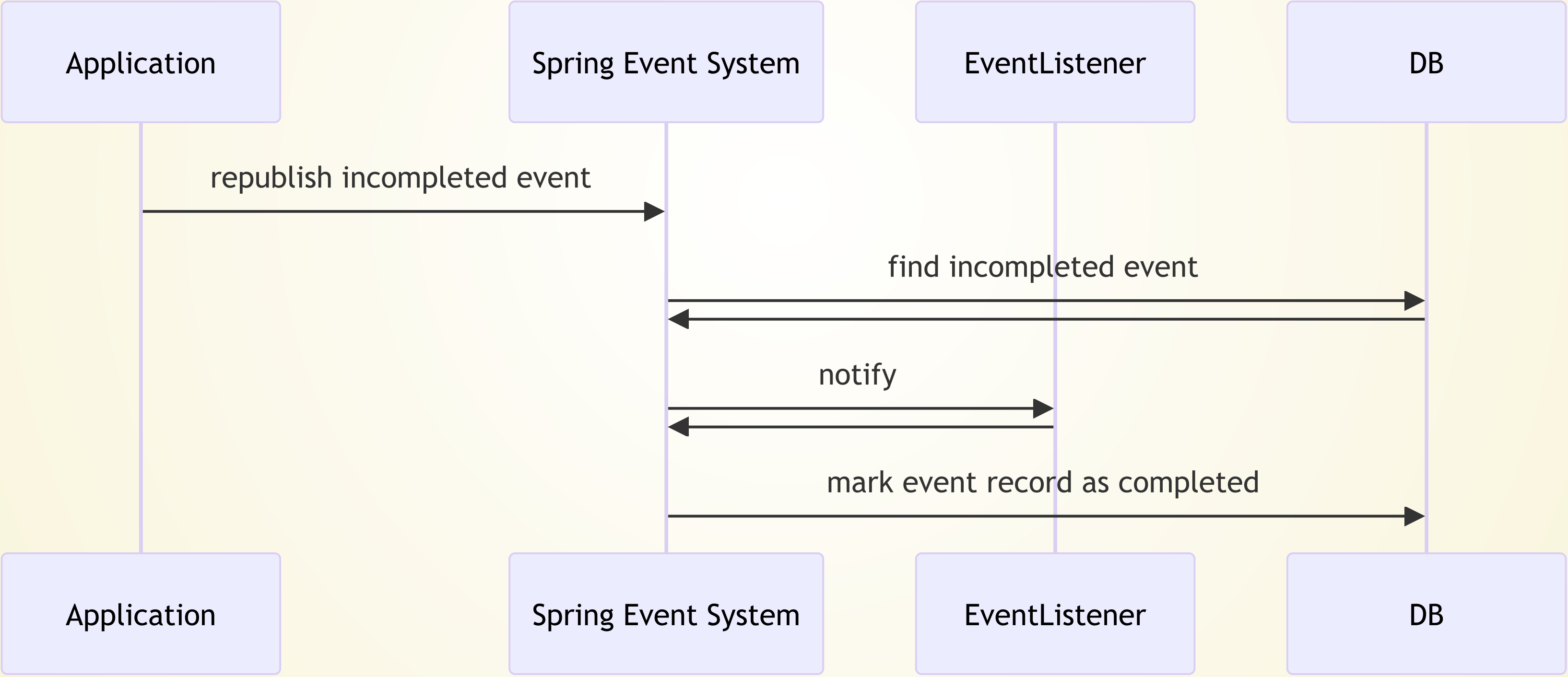
Spring Modulith Event (2)

Event listener occurs failure.



Spring Modulith Event (3)

Republish incompleted event.



How To Use Spring Modulith Event (1)

build.gradle

```
dependencyManagement {
  imports {
    mavenBom "org.springframework.modulith:spring-modulith-bom:1.1.2"
  }
}

dependencies {
  ...
  implementation 'org.springframework.modulith:spring-modulith-starter-core'
  implementation 'org.springframework.modulith:spring-modulith-starter-jdbc'
}
```

How To Use Spring Modulith Event (2)

application.yml

```
spring:
  modulith:
    // republish incompleted event when application restart
    republish-outstanding-events-on-restart: true
    events:
      jdbc:
        schema-initialization:
         enabled: true  // initialize event table
```

How To Use Spring Modulith Event (3)

Use ***@ApplicationModuleListener*** instead to simplify the annotation.

```
@TransactionalEventListener
@Transactional(propagation = Propagation.REQUIRES_NEW)
@Async
public void receiveAndSendEmail(OrderCreatedEvent event) {...}

}
```

Change to

```
@ApplicationModuleListener
public void receiveAndSendEmail(OrderCreatedEvent event) {...}
```

DEMO

Question 3

How to integrate with external brokers such as KafKa, RabbitMQ from internal Spring events.

Scenario (Original)

在一個沒有其它微服務的系統架構下，一個單體式的應用服務"訂單系統"，在完成訂單後

- 需要發送信件給客戶
- 並且需要累計銷量供後續管理人員查詢

Scenario (New)

組織內開發了一個以基於RabbitMQ為架構的發送信件微服務， 原本的"訂單系統"，在完成訂單後

- ~~需要發送信件給客戶~~ 發送訂單事件到RabbitMQ，由信件微服務消費發送信件給客戶
- 並且需要累計銷量供後續管理人員查詢

Integrate With External Brokers (1)

Approach 1 : Implement sending event in event listener

```
@Component
@RequiredArgsConstructor
public class OrderEventListener1 {

    private final RabbitTemplate rt;

    @ApplicationModuleListener
    public void receiveAndSendEmail(OrderCreatedEvent event) {

        rt.convertAndSend("exchange.key", "routing.key", event);

    }
```

Integrate With External Brokers (2)

Approach 2 : use Spring Modulith Externalizing Events. Now it supports external borkers ***Kafka*, *AMQP, JMS, SQS, SNS (Redis is possible in the future)*.**

Externalizing Events (1)

Scenario : use RabbitMQ as message borker.

bulid.gradle

```
dependencies {
    ...
    implementation 'org.springframework.modulith:spring-modulith-events-api'
    // here we use RabbitMQ as message borker
    implementation 'org.springframework.modulith:spring-modulith-events-amqp'
}
```

Externalizing Events (2)

Enable externalizing events

application.yml

```
spring:
  modulith:
    republish-outstanding-events-on-restart: true
    events:
      jdbc:
        schema-initialization:
         enabled: true
        externalization:
         enabled: true    // Enable externalizing events
```

Externalizing Events (3)

use ***@Externalized*** on your event object

```
// expression in RabbitMQ
@Externalized("{exchange name}::{routing key}")

// expression in Kafka
@Externalized("{topic name}::{partition key}")
```

Externalizing Events (3)

For Example

RabbitMQ exchange name : `order.created`

```
@Externalized("order.created")
public record OrderCreatedEvent(Order order) {}
```

DEMO

Consumer Of Microservices

We can still leverage Spring Modulith Event features such as persistent event table and resubmit(replay) events if failed in consumers of microservices.

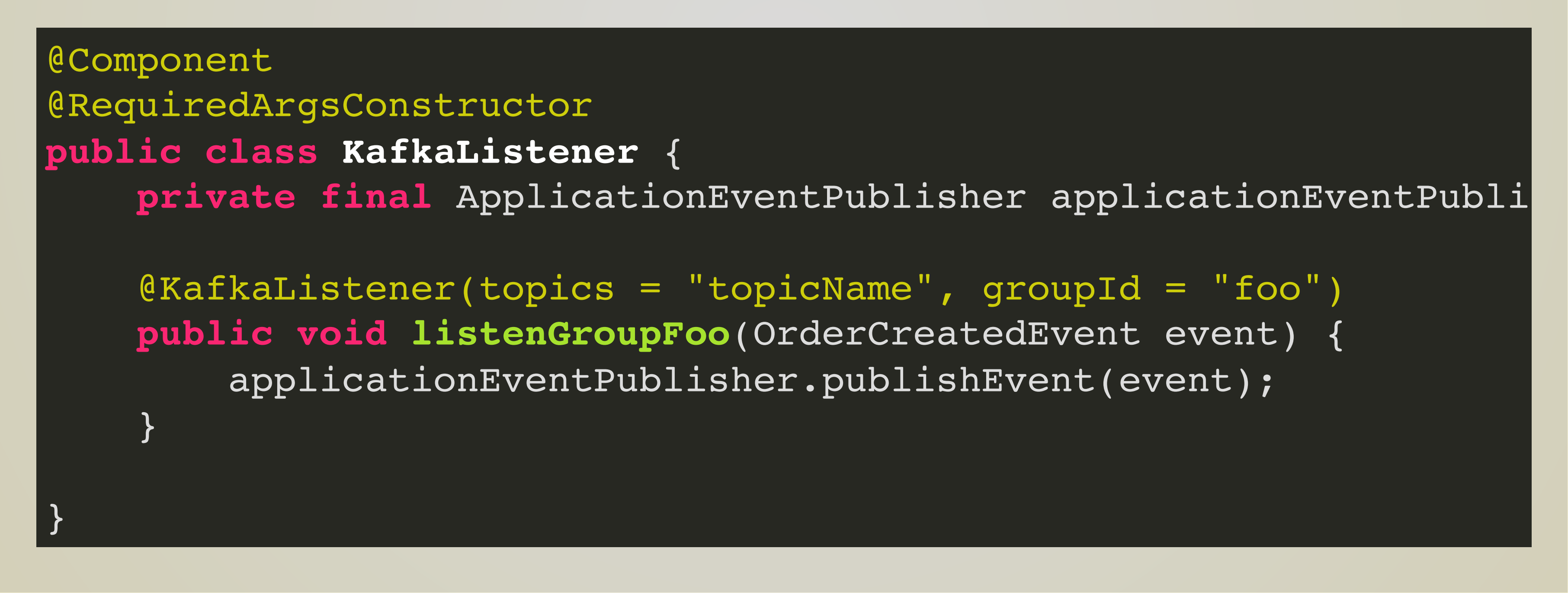
Consumer Of Microservices

Receive event from RabbitMQ then send to Spring Event, then handle event in Spring Event Listener.

```
@Component
@RequiredArgsConstructor
public class RabbitmqListener {
    private final ApplicationEventPublisher applicationEventPubli

    @RabbitListener(queues = "order.queue")
    public void receiveEvent(OrderCreatedEvent event) {
        applicationEventPublisher.publishEvent(event);
    }

}
```

Consumer Of Microservices

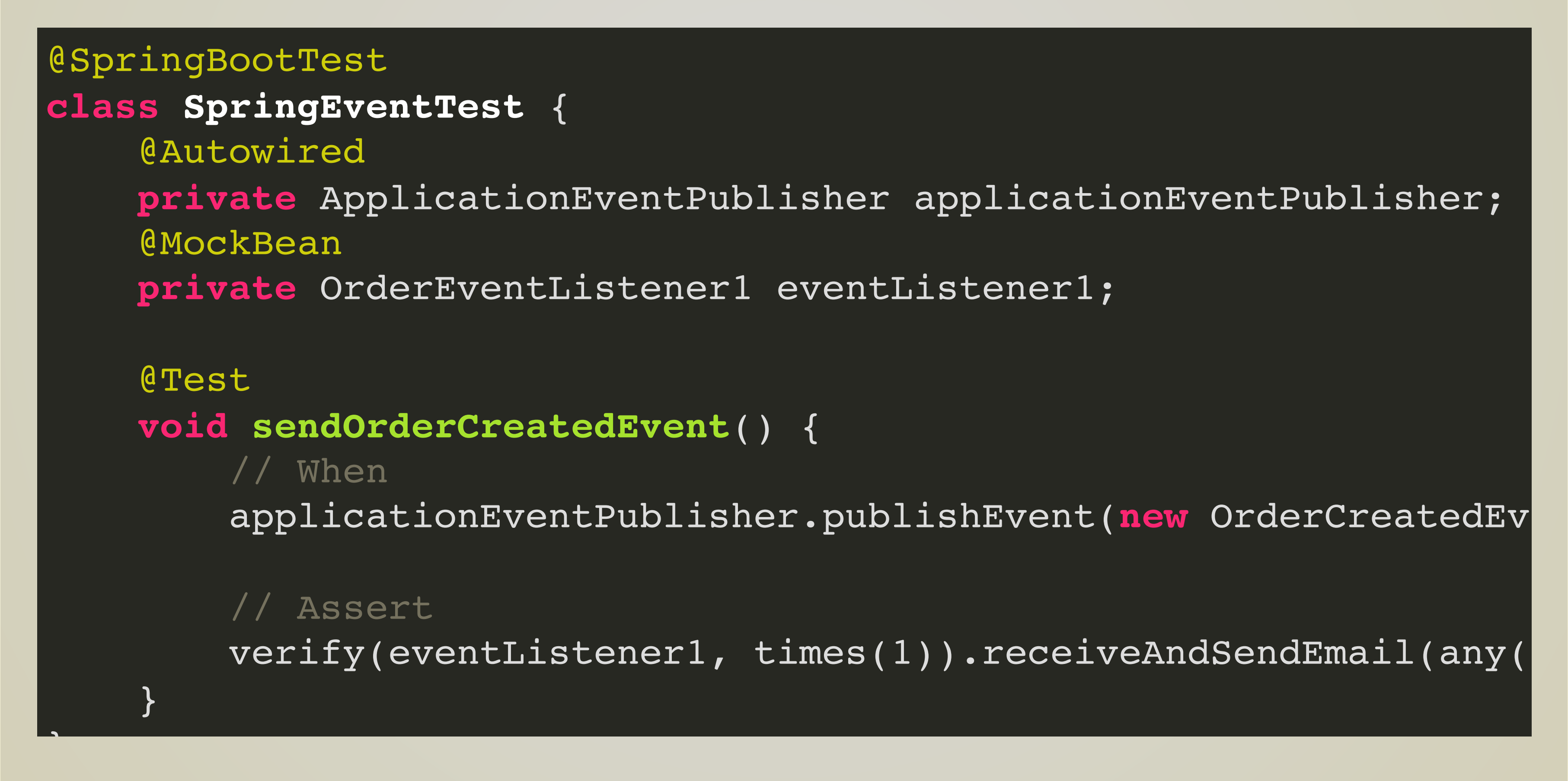
If we have to use or change another external broker like Kafka, just adjust the consumer class, the Spring Event Listener do not change anything.



```
@Component
@RequiredArgsConstructor
public class KafkaListener {
    private final ApplicationEventPublisher applicationEventPubli

    @KafkaListener(topics = "topicName", groupId = "foo")
    public void listenGroupFoo(OrderCreatedEvent event) {
        applicationEventPublisher.publishEvent(event);
    }

}
```

Consumer Of Microservices

Also easily to do unit test.



```
@SpringBootTest
class SpringEventTest {
    @Autowired
    private ApplicationEventPublisher applicationEventPublisher;
    @MockBean
    private OrderEventListener1 eventListener1;

    @Test
    void sendOrderCreatedEvent() {
        // When
        applicationEventPublisher.publishEvent(new OrderCreatedEv

        // Assert
        verify(eventListener1, times(1)).receiveAndSendEmail(any(
    }
```

Other Topics

- Idempotent Processing (Idempotent Key)
- Spring Event Performance (see listeners’ workload)
- Documentation (e.g. Springwolf)
- How to find/define event (e.g. DDD Event Storming)

Recap

- ***@EventListener*** : regular event listener. run within a single thread of producer’s process.
- ***@TransactionalEventListener*** : transaction-bound events listener. run according to transactional phase.
- ***@Async*** : run parallel.
- ***@Order*** : control execution order of listeners.

Recap

- ***@ApplicationModuleListener*** : to simplify listener annotation. (*Recommend).
- ***@Externalized*** : enable externalize events.

References

- A Deep Dive Into Spring Application Events
- Mastering Events In Spring Boot: A Comprehensive Guide
- Spring Modulith / Working With Application Events
- Transaction-Bound Events

Thank You