Spring Event

Spring Event

- Sprint Event
  - ✓ 스프링 프레임워크는 다양한 타입의 이벤트 메시지를 게시하고 구독할 수 있는 일련의 메커니즘을 제공하는데 이를 스프링 이벤트라고 함
  - ✓ 스프링 이벤트는 스프링 프레임워크에서 제공하는 클래스와 어노테이션으로 사용
  - ✓ 스프링 이벤트와 가장 많이 혼동하는 것은 RabbitMQ. ActiveMQ 같은 외부 시스템 메시지 브로커(message broker)를 사용하는 이벤트 처리 아키텍처인데 스프링 이벤트처럼 메시지를 게시하고 구독할 수 있지만 메시지 브로커 아키텍처는 프로세스와 프로세스 또는 인스턴스와 인스턴스 사이에 이벤트 메시지를 게시하고 구독하는 목적으로 사용하는데 이때 메시지 브로커가 중간에서 이벤트 메시지를 전달하는 큐(queue) 역할을 수행하므로 프로세스 사이에 서로 데이터를 주고 받을 수 있고 메시지를 게시 및 구독하는 과정에서 신뢰성을 보장하므로 데이터 누락 없이 이벤트 메시지를 전파할 수 있기 때문에 MSA 환경에서는 한 컴포넌트에서 다른 여러 컴포넌트에 데이터를 전파하는 목적으로 사용하거나 메시지 큐에 메시지를 쌓아 두고 주기적으로 배치 프로세싱하는데 주로 사용
  - ✓ 스프링 이벤트는 스프링 애플리케이션 내부에서 이벤트를 게시하고 구독하는 목적으로 사용하는데 ApplicationContext는 메시지 큐처럼 이벤트 메시지를 전달하는 역할을 수행하며 애플리케이션 내부에서 객체와 객체 사이에 이벤트 메시지를 전달할 수 있으며 싱글 스레드 뿐 만 아니라 멀티 스레드로 이벤트를 구독하고 처리할 수 있어서 메시지 큐 아키텍처와 스프링 이벤트는 사용하는 목적이 다름

Spring Event

- Sprint Event 장점
  - ✓ 이벤트를 게시하는 클래스와 이벤트를 구독하는 클래스의 의존 관계를 분리
  - ✓ 이벤트를 게시 구독하는 두 클래스를 비동기로 처리
  - ✓ 게시된 하나의 이벤트 메시지를 여러 개의 구독 클래스가 수신 가능
  - ✓ 스프링 이벤트를 이용하여 트랜잭션을 효율적으로 사용
  - ✓ 시나리오
    - 스프링 투어는 가입자 숫자를 늘리는 이벤트를 개발하려고 한다.
    - 신규 고객이 서비스에 가입하면 3만 원 가량의 포인트가 포함된 웰컴 쿠폰을 이메일로 발송하기로 결정했다.
    - 기획자는 스프링 투어에 가입하는 즉시 이메일을 전송하는 방식을 제안했다. 또한 이 이벤트는 오픈 후 1개월 동안 유지하고, 가입자 증가 추세에 따라 다른 이벤트로 변경할 수 있다고 했다.
    - 개발자는 현재 가입 프로세스 코드가 복잡해서 이벤트 관련 기능을 추가하는 것이 달갑지 않았다. 가입 이벤트가 언제 어떻게 변경되고 종료될지 정확히 결정되지 않아 가입 클래스에 다른 기능을 추가하는 것이 부담되었다. 이미 가입 클래스에는 수많은 코드가 스파게티처럼 꼬여 있었기 때문이다.
    - 스프링 이벤트를 사용하면 가입 기능과 쿠폰 발송 기능을 분리할 수 있음

Spring Event

- ❖ Sprint Event 장점
  - ✓ 일반적인 경우
    - 사용자 가입의 URI는 POST/users 라고 가정하면 REST-API는 사용자 정보를 생성하는 UserService 의 createUser 메서드를 호출하고 createUser 메서드 내부에서는 웰컴 쿠폰을 이메일로 발송하는 EventService 클래스의 sendEmail 메서드를 호출
    - 기존 UserService 클래스

```
@Slf4j
@Service
public class UserService {
    public Boolean createUser(String userName, String emailAddress) {
        // 사용자 생성 로직

        return Boolean.TRUE;
    }
}
```

Spring Event

- ❖ Sprint Event 장점
  - ✓ 일반적인 경우
    - ● EventService 클래스 생성

```
@Slf4j
@Service
public class EventService {

    public void sendEventMail(String emailAddress) {
        // 이메일 발송 로직
        log.info("Send Email attached welcome coupons. {}", emailAddress);
    }
}
```

Spring Event

❖ Sprint Event 장점

✓ 일반적인 경우

● UserService 클래스 수정

```
@Slf4j
@Service
@RequiredArgsConstructor
public class UserService {
    private EventService eventService;

    public Boolean createUser(String userName, String emailAddress) {
        // 사용자 생성 로직

        eventService.sendEventMail(emailAddress);
        return Boolean.TRUE;
    }
}
```

Spring Event

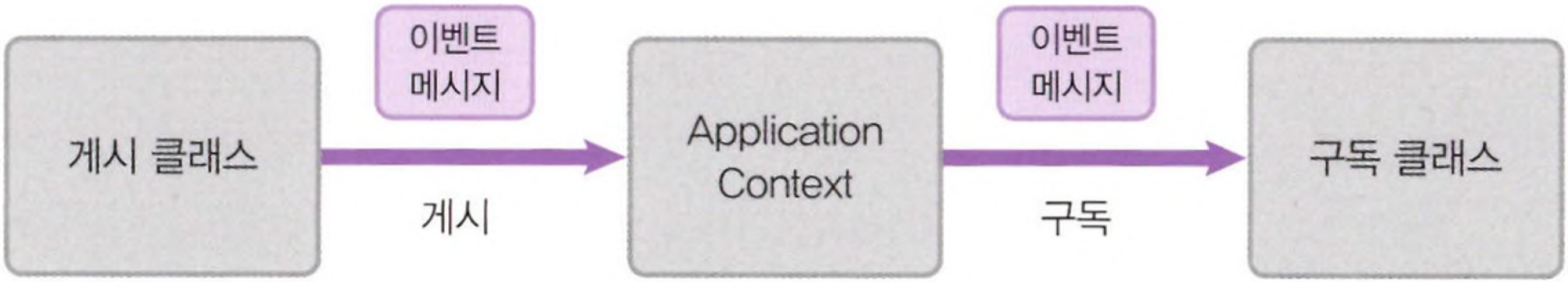
- Sprint Event 장점
  - ✓ 일반적인 경우
    - UserService 클래스의 코드를 보면 EventService 스프링 빈을 주입받아서 사용하는데 두 클래스는 느슨하게 결합되어 있지만 기능적 측면에서 복잡도는 증가
    - 사용자 가입 API에서 사용자를 생성하는 UserService 클래스는 주요 기능이며 웰컴 쿠폰을 보내는 EventService는 부가 기능
    - 상황에 따라 웰컴 쿠폰 대신 포인트를 적립하는 이벤트로 대체 하거나 심지어는 이벤트를 종료할 수 있지만 사용자를 생성하는 기능은 이벤트 기능에 따라 변경되지 않기 때문에 두 기능은 서로 관련이 없으므로 서로 분리하여 결합도를 낮추는 것이 유지 보수에 유리
    - 스레드 풀을 스프링 이벤트에 적용하면 이벤트 처리 부분을 비동기로 동작하도록 작성할 수 있는데 사용자를 가입시키는 UserService의 createUser()를 실행하는 스레드와 쿠폰 이메일을 발송하는 EventService의 sendEventMail()을 실행하는 스레드를 분리할 수 있음
    - UserService의 createUser() 메서드가 종료되면 사용자에게 REST-API 응답을 하고 애플리케이션 내부에서는 EventService의 sendEventMail() 메서드를 동시에 실행할 수 있기 때문에 별도의 스레드에서 동작하는 시간만큼 사용자에게 빠르게 응답할 수 있음

Spring Event

- ❖ Sprint Event 장점
  - ✓ 일반적인 경우
    - 스프링 이벤트로 시스템 리소스를 효과적으로 사용하는데 데이터베이스 트랜잭션을 효율적으로 사용할 수 있음
    - UserService의 createUser() 메서드는 사용자 정보와 관련된 A, B, C 테이블에 트랜잭션과 함께 CRUD 쿼리를 사용한다고 하고 EventService의 sendEventMail() 메서드도 사용자와 이벤트에 관련된 C, D, E, F 테이블에 CRUD 쿼리를 사용한다고 가정하면 이 두 기능을 하나의 트랜잭션에서 실행한다면 트랜잭션 시간이 길어짐
    - 시간이 길어지고 쿼리를 하는 데이터가 많아진다면 해당 리소스에 락을 유지하는 시간도 길어지게 되고 이런 쿼리가 동시에 실행된다면 그만큼 데드락이 발생할 확률이 높아짐
    - 비지니스 로직 상 두 트랜잭션을 분리할 수 있다면 데이터베이스 리소스를 효율적으로 사용할 수 있음
    - EventService의 sendEventMail() 메서드에 적용된 트랜잭션도 더욱 정교하게 사용할 수 있게 되는데 ApplicationContext는 트랜잭션 종료 시점을 여러 단계로 구분하고 정의된 단계에서 이벤트를 처리하는 기능을 제공
    - sendEventMail() 메서드 내부에 쿼리 관련 기능과 이메일을 전송하는 SMTP 연동 코드가 포함되어 있다고 생각하면 SMTP의 API를 호출하는 시간까지 트랜잭션 처리 시간에 포함되므로 그 시간만큼 트랜잭션이 유지되는데 트랜잭션이 정상적으로 커밋된 후 SMTP의 API를 호출할 수 있다면 트랜잭션을 보다 효율적으로 관리할 수 있음

Spring Event

- ❖ 사용자 정의 이벤트 처리
  - ✓ 직접 정의한 이벤트 메시지를 게시하고 구독하기 위해 필요한 클래스
    - Event Message: 게시 클래스 와 구독 클래스 사이에 공유할 데이터를 포함하는 클래스로 이벤트 객체, 메시지 객체 등 여러 가지 이름으로 부름
    - Publisher: 구독 클래스에 전달할 이벤트 메시지를 생산하고 게시하는 클래스로 ApplicationContext에 이벤트 메시지를 전달하는 과정을 게시(publish)라고 함
    - Listener: ApplicationContext에서 전달받은 이벤트 메시지를 구독하고 부가 기능을 실행하는 클래스로 ApplicationContext에서 이벤트 메시지를 전달받는 과정을 구독(listen)이라고 함



Spring Event

- 사용자 정의 이벤트 처리
  - ✓ 이벤트 메시지를 게시하고 구독하기 위해 필요한 스프링의 기능
    - org.springframework.context.ApplicationEvent: 애플리케이션에서 사용할 이벤트 메시지 클래스는 ApplicationEvent 클래스를 상속받아야 함
    - org.springframework.context.ApplicationPublisher: 스프링 프레임워크에서 기본으로 제공하는 스프링 빈 이며 이벤트 메시지를 게시할 수 있는 publishEvent() 메서드를 제공하는데 이벤트 메시지를 게시하려면 ApplicationPublisher 스프링 빈을 주입받아 publishEvent() 메서드를 실행해야 하므로 게시 클래스에서 사용
    - org.springframework.context.ApplicationListener: 이벤트 메시지를 구독할 수 있는 onApplicationEvent() 메서드를 제공하는 인터페이스로 구독 클래스가 구현해야 하는 인터페이스
    - 게시 클래스에서는 ApplicationPublisher를 사용하여 이벤트 메시지 객체를 게시하고 구독 클래스에서는 ApplicationListener를 구현하여 onApplicationEvent() 메서드를 개발해야 하는데 이때 ApplicationContext는 스프링 이벤트에서 이벤트 메시지 객체를 중계하는 역할을 하기 때문에 ApplicationContext가 이벤트 메시지의 클래스 타입을 확인하고 적절한 구독 클래스의 onApplicationEvent() 메서드를 찾고 이벤트 메시지 객체를 인자로 넣어 실
    - ApplicationContext가 이 과정을 처리하려면 게시 클래스와 구독 클래스 모두 스프링 빈으로 로딩되어야 함

Spring Event

- ❖ 사용자 정의 이벤트 처리
  - ✓ 이벤트 메시지를 게시하고 구독
    - ● EventMessage 클래스 생성 – UserEvent

```
@Getter
public class UserEvent extends ApplicationEvent {
    private Type type;
    private Long userId;
    private String emailAddress;
```

Spring Event

- 사용자 정의 이벤트 처리
  - ✓ 이벤트 메시지를 게시하고 구독
    - EventMessage 클래스 생성 – UserEvent

```
private UserEvent(Object source, Type type, Long userId, String emailAddress)
{
    super(source);
    this.type = type;
    this.userId = userId;
    this.emailAddress = emailAddress;
}

public static UserEvent created(Object source, Long userId, String
 emailAddress) {
    return new UserEvent(source, Type.CREATE, userId, emailAddress);
}

public enum Type {
    CREATE, DELETE
}
}
```

Spring Event

- ❖ 사용자 정의 이벤트 처리
  - ✓ 이벤트 메시지를 게시하고 구독
    - EventMessage 클래스 생성 – UserEvent
      - ◆ UserEvent는 사용자를 생성하거나 삭제할 때 게시하는 이벤트 메시지 클래스이므로 ApplicationEvent를 상속
      - ◆ 부모 클래스 ApplicationEvent 생성자의 Object source 인자는 이벤트를 게시하는 클래스의 객체를 의미하는데 UserService 클래스에서 UserEvent 객체를 생성하고 게시한다면 UserService 객체를 Object source 인자로 넘기면 되며 ApplicationEvent 객체에 할당된 Object source는 Object getSource() 메서드를 사용하면 참조할 수 있음
      - ◆ UserEvent의 타입을 의미하는 열거형이며 상수 값은 CREATE, DELETE가 있는데 사용자를 생성할 때 UserEvent 객체는 UserEvent.Type.CREATE 타입 속성이며 사용자를 삭제할 때는 UserEvent.Type.DELETE 속성이 할당
      - ◆ UserEvent 클래스의 속성은 이벤트 타입인 UserEvent.Type 사용자의 고유 아이디 값인 Long userid, 이메일 주소인 String emailAddress가 있으며 클래스의 정적 팩토리 메서드는 인자들을 받아 각각 적합한 속성에 할당하는 내부 생성자를 호출하고 created() 정적 팩토리 메서드는 사용자가 생성될 때 호출되며 내부에서는 Type.CREATE 열거형 상수를 UserEvent 생성자의 인자로 전달

Spring Event

- ❖ 사용자 정의 이벤트 처리
  - ✓ 이벤트 메시지를 게시하고 구독
    - ● 게시자 클래스 생성 – UserEventPublisher

```
@Slf4j
@Component
public class UserEventPublisher {

    private final ApplicationEventPublisher applicationEventPublisher;

    public UserEventPublisher(ApplicationEventPublisher applicationEventPublisher)
    {
        this.applicationEventPublisher = applicationEventPublisher;
    }

    public void publishUserCreated(Long userId, String emailAddress) {
        UserEvent userEvent = UserEvent.created(this, userId, emailAddress);
        log.info("Publish user created event.");
        applicationEventPublisher.publishEvent(userEvent);
    }
}
```

Spring Event

- ❖ 사용자 정의 이벤트 처리
  - ✓ 이벤트 메시지를 게시하고 구독
    - ● 게시자 클래스 생성 – UserEventPublisher
      - ◆ 구조적으로 게시 클래스와 구독 클래스는 ApplicationContext로 서로 분리되어 있으며 오직 이벤트 메시지 객체만 서로 주고받을 수 있으므로 이벤트 메시지 객체는 구독 클래스에서 필요한 정보를 포함하는 것이 좋음
      - ◆ UserEvent 이벤트 메시지 객체에 userid가 없다면 구독 클래스에서 필요한 사용자 정보를 데이터베이스에서도 조회할 수 없음
      - ◆ 사용자 생성 이벤트이므로 사용자의 고유 값(userid)과 구독 클래스에서 필요한 사용자 이메일 주소(emailAddress)를 속성으로 포함하고 이 정보를 사용하여 구독 클래스에서 필요한 기능을 실행
      - ◆ UserEvent 이벤트 메시지를 게시하는 UserEventPublisher 클래스를 따로 설계해서 이벤트가 발생하는 어떤 클래스라도 UserEventPublisher의 publishUserCreated() 메서드를 호출
      - ◆ UserService의 createUser() 메서드에서 publishIlserCreated() 메서드를 호출
      - ◆ 사용자를 삭제하는 이벤트를 처리해야 한다면 publishUserDeleted() 같은 메서드를 개발하면 되므로 유연하게 대처할 수 있어서 UserEvent의 Type 속성은 UserEvent.Type.CREATE 와 DELETED 상수가 될 수 있음

Spring Event

- ❖ 사용자 정의 이벤트 처리
  - ✓ 이벤트 메시지를 게시하고 구독
    - ● 구독자 클래스 생성 – UserEventListener

```
@Slf4j
@Component
@RequiredArgsConstructor
public class UserEventListener implements ApplicationListener<UserEvent> {

    private final EventService eventService;

    @Override
    public void onApplicationEvent(UserEvent event) {
        if (UserEvent.Type.CREATE == event.getType()) {
            log.info("Listen CREATE event. {}, {}", event.getUserId(),
     event.getEmailAddress());
            eventService.sendEventMail(event.getEmailAddress());
        } else if (UserEvent.Type.DELETE == event.getType()) {
            log.info("Listen DELETE event");
        } else {
            log.error("Unsupported event type. {}", event.getType());
        }
    }
}
```

Spring Event

- ❖ 사용자 정의 이벤트 처리
  - ✓ 이벤트 메시지를 게시하고 구독
    - ● 구독자 클래스 생성 – UserEventListener
      - ◆ 이벤트를 구독하는 클래스는 ApplicationListener 인터페이스를 구현하여 개발하는데 ApplicatonListener의 제네릭 클래스 타입은 구독할 이벤트의 클래스 타입을 의미하므로 ApplicationListener<UserEvent>
      - ◆ UserEvent 이벤트 객체의 타입에 따라 기능을 실행할 EventService 스프링 빈을 주입받는데 UserEventListener는 UserEvent를 구독하는 기능만 담당하고 실제 이벤트를 처리하는 기능은 별도의 클래스에 위임하는 구조
      - ◆ UserEvent의 type이 UserEvent.Type.CREATE일 때 EventService의 sendEventMail() 메서드를 호출

Spring Event

- ❖ 사용자 정의 이벤트 처리
  - ✓ 이벤트 메시지를 게시하고 구독
    - ● UserService 클래스 수정

```
@Slf4j
@Service
@RequiredArgsConstructor
public class UserService {
    private final UserEventPublisher userEventPublisher;

    public Boolean createUser(String userName, String emailAddress) {
        // 사용자 생성 로직
        log.info("created user. {}, {}", userName, emailAddress);
        userEventPublisher.publishUserCreated(1239876L, emailAddress);
        log.info("done create user");
        return Boolean.TRUE;
    }
}
```

- ◆ UserService와 EventService는 서로 의존성이 없으므로 이벤트가 변경되더라도 UserService의 코드에는 변경 사항이 발생하지 않음
- ◆ UserService의 코드가 매우 복잡하다면 스프링 이벤트를 사용하여 복잡성을 줄일 수 있음

Spring Event

- ❖ 사용자 정의 이벤트 처리
  - ✓ 이벤트 메시지를 게시하고 구독
    - Application 코드를 수정해서 실행

```
@SpringBootApplication
public class EventApplication {

    public static void main(String[] args) {
        SpringApplicationBuilder appBuilder = new
     SpringApplicationBuilder(EventApplication.class);
        SpringApplication application = appBuilder.build();

        ConfigurableApplicationContext ctxt = application.run(args);

        UserService userService = ctxt.getBean(UserService.class);
        userService.createUser("ItStudy", "itstudy@adamsoft.com");
    }
}
```

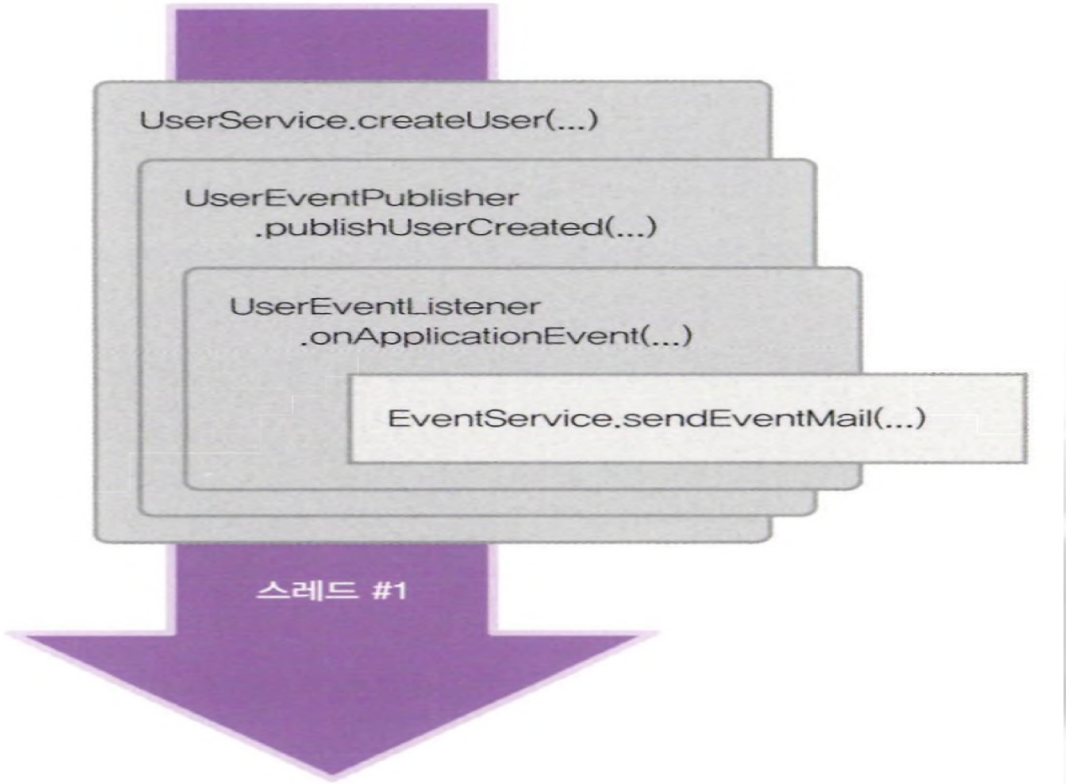
Spring Event

- ❖ 사용자 정의 이벤트 처리
  - ✓ 이벤트 메시지를 게시하고 구독
    - Application 코드를 수정해서 실행

```
2023-03-19T09:38:42.613+09:00  INFO 9876 --- [  restartedMain] com.example.event.service.UserService    : created user. ItStudy, itstudy@adamsoft.com
2023-03-19T09:38:42.613+09:00  INFO 9876 --- [  restartedMain] c.e.event.service.UserEventPublisher     : Publish user created event.
2023-03-19T09:38:42.614+09:00  INFO 9876 --- [  restartedMain] c.e.event.service.UserEventListener      : Listen CREATE event. 1239876, itstudy@adamsoft.com
2023-03-19T09:38:42.614+09:00  INFO 9876 --- [  restartedMain] com.example.event.service.EventService   : Send Email attached welcome coupons. itstudy@adamsoft.com
2023-03-19T09:38:42.614+09:00  INFO 9876 --- [  restartedMain] com.example.event.service.UserService    : done create user
```

Spring Event

- ❖ 사용자 정의 이벤트 처리
  - ✓ 이벤트 메시지를 게시하고 구독
    - ● 스레드 하나에서 실행된 클래스



Spring Event

- ❖ 사용자 정의 이벤트 처리
  - ✓ 이벤트 메시지를 게시하고 구독
    - ● 스레드 하나에서 실행된 클래스
      - ◆ 클래스 네 개의 메서드들이 스택 구조로 실행된 것을 확인할 수 있음
      - ◆ UserService -> UserEventPublisher -> UserEventListener -> Eventservice 순서대로 메서드가 실행되고 메서드의 종료는 역순
      - ◆ UserEvent, UserEventPublisher, UserEventListener는 UserService와 Eventservice의 관계를 느슨하게 만들지만 모든 클래스는 하나의 공통 스레드에서 실행
      - ◆ UserService.createUser() 메서드에 @Transactional 어노테이션이 적용되어 트랜잭션이 실행된다고 생각해 보면 모든 메서드가 종료되고 createUser() 메서드도 종료되어야 해당 트랜잭션이 종료되기 때문에 리소스에 낭비가 발생할 수 있는 상황

Spring Event

- 비동기 사용자 정의 이벤트 처리
  - ✓ ApplicationContext 내부에서는 이벤트를 게시하기 위해 ApplicationEventMultiCaster를 사용
  - ✓ 사용자가 ApplicationEventPublisher의 publishEvent()를 사용하여 이벤트 메시지를 게시하면 ApplicationContext 내부에서는 ApplicationEventMultiCaster로 이벤트 메시지를 게시하는데 별도의 설정이 없다면 ApplicationEventMultiCaster는 싱글 스레드로 동작
  - ✓ AbstractApplicationContext는 ApplicationEventMulticaster를 클래스 속성으로 포함하는데 이 ApplicationEventMulticaster는 인터페이스이며 스프링 프레임워크에서는 SimpleApplicationEventMulticaster 구현체를 제공하는데 기본값으로 설정된 ApplicationContext의 내부에서는 SimpleApplicationEventMulticaster 객체를 생성하여 사용하고 이때 SimpleApplicationEvent Multicaster에 별도의 스레드 풀이 없다면 ApplicationEventPublisher의 publishEvent() 메서드가 실행된 스레드에서 이벤트를 게시하고 그 이벤트를 구독하기 때문에 실행 결과처럼 하나의 스레드에서 이벤트를 게시하고 구독한
  - ✓ AbstractApplicationContext의 내부 코드에서는 APPLICATION_EVENT_MULTICASTER_BEAN_NAME 상수에 정의된 이름 applicationEventMulticaster를 사용하여 스프링 빈을 찾는데 찾지 못하면 새로운 SimpleApplicationEventMulticaster를 생성하지만 스프링 빈이 정의되어 있다면 해당 스프링 빈을 사용

Spring Event

- 비동기 사용자 정의 이벤트 처리
  - ✓ 개발자가 직접 스레드 풀을 설정한 SimpleApplicationEventMulticaster 객체를 생성하고 스프링 빈 이름을 applicationEventMulticaster로 설정한다면 비동기로 이벤트를 구독할 수 있음
  - ✓ 기존 게시 클래스와 구독 클래스의 수정 없이 비동기로 이벤트를 구독할 수 있음

Spring Event

- 비동기 사용자 정의 이벤트 처리
  - ✓ ApplicationEventMulticaster 빈을 생성하기 위한 config 클래스 생성 – config.AsyncEventConfig

```
@Configuration
public class AsyncEventConfig {
    @Bean(name = "applicationEventMulticaster")
    public ApplicationEventMulticaster applicationEventMulticaster(TaskExecutor
     asyncEventTaskExecutor) {
        SimpleApplicationEventMulticaster eventMulticaster = new
     SimpleApplicationEventMulticaster();
        eventMulticaster.setTaskExecutor(asyncEventTaskExecutor);
        return eventMulticaster;
    }

    @Bean
    public TaskExecutor asyncEventTaskExecutor() {
        ThreadPoolTaskExecutor asyncEventTaskExecutor = new
     ThreadPoolTaskExecutor();
        asyncEventTaskExecutor.setMaxPoolSize(10);
        asyncEventTaskExecutor.setThreadNamePrefix("eventExecutor-");
        asyncEventTaskExecutor.afterPropertiesSet();
        return asyncEventTaskExecutor;
    }
}
```

Spring Event

- 비동기 사용자 정의 이벤트 처리
  - ✓ ApplicationEventMulticaster 빈을 생성하기 위한 config 클래스 생성 – config.AsyncEventConfig
    - 반드시 AbstractApplicationContext에서 지정한 스프링 빈 이름으로 설정
    - TaskExecutor 스프링 빈을 정의하는데 ThreadPoolTaskExecutor 구현체를 사용하여 객체를 생성하고 풀에서 관리할 수 있는 최대 스레드의 개수는 열 개로 설정하는데 이때 스레드 풀에 포함된 스레드들의 이름은 머리말이 eventExecutor로 시작하고 마지막으로 afterPropertiesSet() 메서드를 실행하여 최종 설정
    - 주입받은 asyncEventTaskExecutor 스프링 빈을 eventMulticaster의 setTaskExecutor() 메서드를 사용하여 스레드 풀 설정에 사용

Spring Event

❖ 비동기 사용자 정의 이벤트 처리

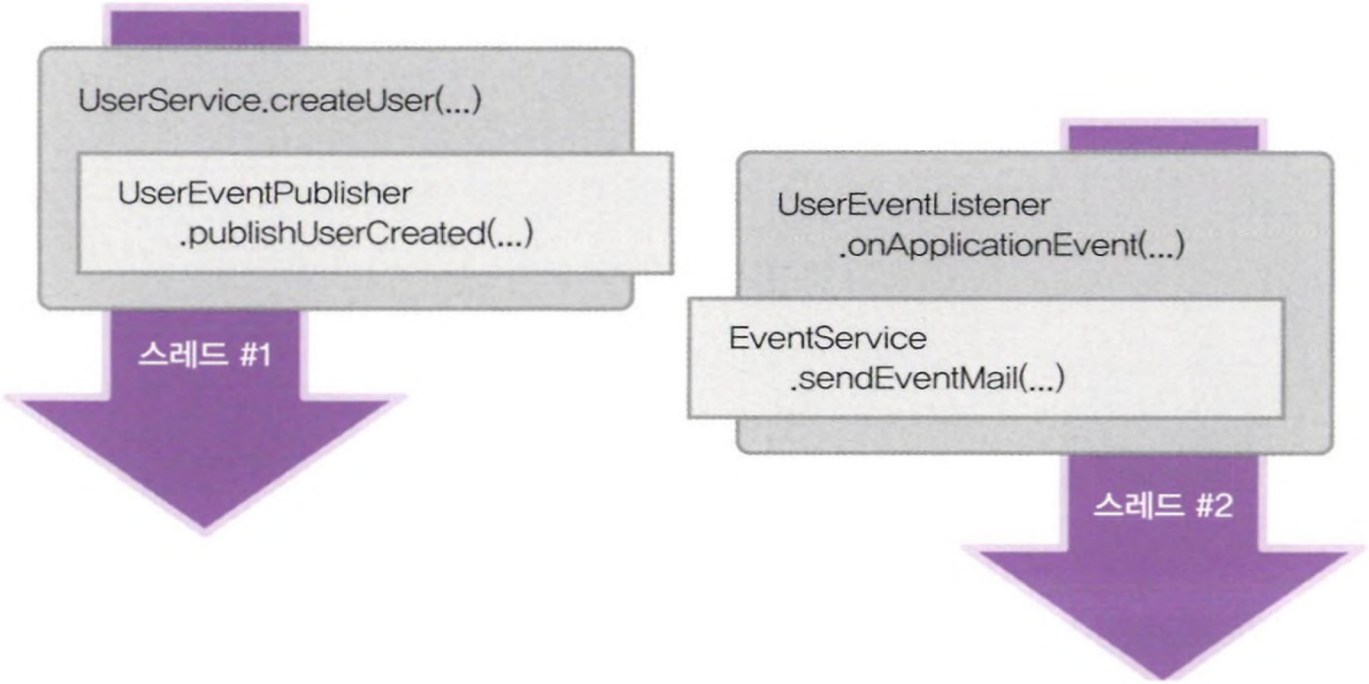
✓ 실행

```
[  restartedMain] com.example.event.EventApplication      : Started EventApplication in 1.463 seconds (process running
[  restartedMain] com.example.event.service.UserService   : created user. ItStudy, itstudy@adamsoft.com
[  restartedMain] c.e.event.service.UserEventPublisher    : Publish user created event.
[  restartedMain] com.example.event.service.UserService   : done create user
[eventExecutor-1] c.e.event.service.UserEventListener     : Listen CREATE event. 1239876, itstudy@adamsoft.com
[eventExecutor-1] com.example.event.service.EventService  : Send Email attached welcome coupons. itstudy@adamsoft.com
```

- 코드를 변경하지 않고 ApplicationEventMulticaster를 설정한 것만으로도 비동기로 이벤트 메시지를 처리할 수 있음
- 실행 결과 로그를 보면 UserService를 실행하는 스레드 restartedMain은 이벤트를 게시하는 UserEventPublisher의 publishUserCreatecd() 메서드까지 실행하고 이벤트 메시지를 구독하는 UserEventListener와 이벤트를 처리하는 EventService의 sendEmail() 메서드는 eventExecutor-1 스레드에서 실행되는데 앞서 설정한 스레드 풀에서 실행된 것을 알 수 있음

Spring Event

- 비동기 사용자 정의 이벤트 처리
  - ✓ 실행



Spring Event

- @Async 어노테이션을 사용한 비동기 이벤트 처리
  - ✓ 스프링 프레임워크는 비동기 프로그래밍을 위한 @Async와 @EnableAsync 어노테이션을 제공하는데 이 기능은 스프링 AOP로 구현된 메커니즘이므로 사용자는 간단히 어노테이션을 선언하여 기능을 사용할 수 있음
  - ✓ 비동기로 실행하고 싶은 스프링 빈의 메서드에 @Async 어노테이션을 정의하면 해당 메서드는 비동기로 실행되고 @EnableAsync 어노테이션을 자바 설정 클래스에 선언하면 비동기 기능을 활성화 할 수 있는데 @Async 어노테이션을 사용하려면 반드시 @EnableAsync 어노테이션을 정의해야 함

Spring Event

❖ @Async 어노테이션을 사용한 비동기 이벤트 처리

✓ @Async 사용 방법

```
@Service
public class EventService {
    @Async
    public void sendEventMail(String emailAddress) {
            // 생략
    }
}
```

- @Async 어노테이션은 sendEventMail() 메서드에 선언되어 있으므로 sendEventMail() 메서드를 호출하면 sendEventMail()은 별도의 스레드에서 비동기로 실행
- @Async 어노테이션은 메서드 선언부와 클래스 선언부에 선언할 수 있음
- 클래스에 @Async를 선언하면 해당 클래스의 모든 public 메서드는 비동기로 동작
- 메서드에 정의하면 해당 메서드만 비동기로 동작

Spring Event

- @Async 어노테이션을 사용한 비동기 이벤트 처리
  - ✓ @Async 가 정상적으로 동작하기 위한 조건
    - @Async 대상은 스프링 빈으로 정의되어야 함
    - public 메서드만 비동기로 동작
    - this 키워드가 아닌 자기 주입을 사용하여 자신의 메서드를 호출
  - ✓ @EnableAsync 은 애플리케이션에 비동기 실행 환경을 활성화하는 설정 어노테이션으로 자바 설정 클래스에 선언하면 됨
  - ✓ 비동기 실행 환경이 활성화된 상태에서 @Async가 정의된 메서드를 실행하면 매번 새로운 스레드를 생성하여 실행하므로 서버의 리소스를 효율적으로 사용하려면 스레드 풀을 생성하여 함께 사용하는 것이 좋음
  - ✓ 스레드 풀은 java.util. concurrent의 Executor 인터페이스 구현체를 사용하면 되는데 ApplicationEventMulticaster 에서 사용한 ThreadPoolTaskExecutor도 Executor의 구현체 중 하나
  - ✓ 스프링 프레임워크는 비동기 실행 환경을 설정할 수 있는 org.springframework.scheduling.annotation 패키지의 AsyncConfigurer 인터페이스를 제공
  - ✓ AsyncConfigurer 인터페이스의 getAsyncExecutor() 메서드는 스프링 애플리케이션이 시작하면서 스레드 풀을 설정할 때 사용하는 콜백 메서드로 getAsyncExecutor() 메서드를 구현하면 @Async가 사용하는 스레드 풀을 설정할 수 있음

Spring Event

- @Async 어노테이션을 사용한 비동기 이벤트 처리
  - ✓ 스레드 풀을 설정하기 위한 클래스 – config.AsyncExecutionConfig

```
@EnableAsync
@Configuration
public class AsyncExecutionConfig implements AsyncConfigurer {

    @Override
    public Executor getAsyncExecutor() {
        return getExecutor();
    }

    private Executor getExecutor() {
        ThreadPoolTaskExecutor threadPoolTaskExecutor = new
     ThreadPoolTaskExecutor();
        threadPoolTaskExecutor.setCorePoolSize(10);
        threadPoolTaskExecutor.setMaxPoolSize(10);
        threadPoolTaskExecutor.setThreadNamePrefix("asyncExecutor-");
        threadPoolTaskExecutor.afterPropertiesSet();
        return threadPoolTaskExecutor;
    }

}
```

Spring Event

- ❖ @Async 어노테이션을 사용한 비동기 이벤트 처리
  - ✓ 스레드 풀을 설정하기 위한 클래스 – config.AsyncExecutionConfig
    - @EnableAsync: 애플리케이션에서 @Async 어노테이션을 사용하려고 @EnableAsync 어노테이션을 설정
    - implements AsyncConfigurer: @Async 애너테이션과 스레드 풀의 스레드를 사용하려면 AsyncExecutionConfig 자바 설정 클래스는 AsyncConfigurer 인터페이스를 구현
    - AsyncConfigurer 인터페이스의 getAsyncExecutor() 메서드는 프레임워크가 스레드 풀을 설정 할 때 사용하는 콜백 메서드
    - 기본 스레드 개수는 열 개, 최대 스레드 개수는 열 개, 스레드 이름은 asyncExecutor-로 시작하는 스레드 풀을 생성

Spring Event

- @Async 어노테이션을 사용한 비동기 이벤트 처리
  - ✓ @Async 어노테이션 기능을 사용하면 ApplicationEventMulticaster에 스레드 풀을 설정하지 않아도 비동기 이벤트 프로그래밍을 할 수 있고 원하는 로직만 비동기로 프로그래밍 할 수 있음
  - ✓ ApplicationEventMulticaster에 스레드 풀을 설정하면 이벤트를 구독하는 모든 구독 클래스는 비동기로 실행되지만 @Async 어노테이션 기능을 사용하면 원하는 이벤트 구독 메서드만 선택적으로 비동기로 실행할 수 있음
  - ✓ @Async 애너테이션이 정의된 메서드는 별도의 스레드에서 비동기로 실행되므로 이벤트를 구독하는 메서드인 UserEventListener의 onApplicationEvent() 메서드에 @Async 어노테이션을 적용하면 onApplicationEvent()와 그 메서드가 호출하는 EventService의 sendEventMail() 까지 다른 스레드에서 실행됨

Spring Event

❖ @Async 어노테이션을 사용한 비동기 이벤트 처리

✓ 구현

- Event 클래스 – HotelCreateEvent

```
@Getter
@ToString
public class HotelCreateEvent {

    private Long hotelId;
    private String hotelAddress;

    private HotelCreateEvent(Long hotelId, String hotelAddress) {
        this.hotelId = hotelId;
        this.hotelAddress = hotelAddress;
    }

    public static HotelCreateEvent of(Long hotelId, String hotelAddress) {
        return new HotelCreateEvent(hotelId, hotelAddress);
    }
}
```

Spring Event

- ❖ @Async 어노테이션을 사용한 비동기 이벤트 처리
  - ✓ 구현
    - Event 클래스 – HotelCreateEvent
      - ◆ 이벤트 메시지 객체는 불변(immutable) 클래스로 설계해야 하는데 한번 생성된 이벤트 메시지 객체의 속성은 변경할 수 없도록 해야 함
      - ◆ 게시 클래스가 게시한 이벤트 메시지 객체는 ApplicationContext로 구독 클래스들에 전달되는데 게시하고 구독하는 이벤트 메시지 객체는 서로 같아야 하는데 첫 번째 이벤트 구독 클래스와 두 번째 이벤트 구독 클래스도 서로 공유하는 데이터
      - ◆ 첫 번째 이벤트 구독 클래스가 이벤트 객체의 데이터를 수정한다면 두 번째 이벤트 구독자에도 영향이 있으므로 예상하지 못한 버그가 발생할 수 있음
      - ◆ 불변 클래스로 설계하는 방법은 매우 간단한데 클래스의 속성을 private 접근 제어자로 선언하여 객체 외부에 공개하지 않고 이 속성들을 수정할 수 있는 메서드도 제공하지 않도록 작성하면 되므로 hotelId 속성을 설정하는 setHotelId() 같은 메서드는 볼 수 없음
      - ◆ 정적 팩토리 메서드 외에 클래스의 속성에 값을 할당하는 코드는 찾아볼 수 없음
      - ◆ 속성 값을 참조만 하는 메서드들. 뿐

Spring Event

- ❖ @Async 어노테이션을 사용한 비동기 이벤트 처리
  - ✓ 구현
    - ● 구독자 클래스 – HotelEventPublisher

```
@Slf4j
@Component
public class HotelEventPublisher {

    private ApplicationEventPublisher applicationEventPublisher;

    public HotelEventPublisher(ApplicationEventPublisher
     applicationEventPublisher) {
        this.applicationEventPublisher = applicationEventPublisher;
    }

    public void publishHotelCreated(Long hotelId, String address) {
        HotelCreateEvent event = HotelCreateEvent.of(hotelId, address);
        log.info("Publish hotel created event.");
        applicationEventPublisher.publishEvent(event);
    }

}
```

Spring Event

- ❖ @Async 어노테이션을 사용한 비동기 이벤트 처리
  - ✓ 구현
    - ● 서비스 클래스 – PropagationService

```
@Slf4j
@Service
public class PropagationService {

    public void propagateHotelEvent() {
        log.info("propagation of hotel event");
    }

    public void propagateResourceEvent() {
        log.info("propagation of resource event");
    }
}
```

Spring Event

- ❖ @Async 어노테이션을 사용한 비동기 이벤트 처리
  - ✓ 게시자 클래스
    - ● @EvnetListener
      - ◆ 구독 클래스인 UserEventListener는 이벤트를 구독하기 위해 ApplicationEventListener 인터페이스를 구현했는데 스프링 4.2 버전부터는 @EventListener 애너테이션을 사용할 수 있음
      - ◆ 이벤트를 구독하여 이벤트 메시지를 처리할 메서드에 @EventListener 어노테이션을 정의
      - ◆ ApplicationContext는 해당 메서드에 이벤트 메시지를 인자로 전달하고 메서드를 실행

Spring Event

- ❖ @Async 어노테이션을 사용한 비동기 이벤트 처리
  - ✓ 구현
    - ● 게시자 클래스 – HotelEventListener

```
@Slf4j
@Component
public class HotelEventListener {

    private PropagationService propagationService;

    public HotelEventListener(PropagationService propagationService) {
        this.propagationService = propagationService;
    }

    @Async
    @Order(1)
    @EventListener(value = HotelCreateEvent.class)
    public void handleHotelCreateEvent(HotelCreateEvent hotelCreateEvent) {
        log.info("handle HotelCreatedEvent : {}", hotelCreateEvent);
        propagationService.propagateHotelEvent();

    }
```

Spring Event

- ❖ @Async 어노테이션을 사용한 비동기 이벤트 처리
  - ✓ 구현
    - ● 게시자 클래스 – HotelEventListener

```
    @Async
    @Order(2)
    @EventListener(value = HotelCreateEvent.class)
    public void handleResourceCreateEvent(HotelCreateEvent hotelCreateEvent) {
        log.info("handle resourceCreatedEvent : {}", hotelCreateEvent);
        propagationService.propagateResourceEvent();
    }
}
```

Spring Event

- ❖ @Async 어노테이션을 사용한 비동기 이벤트 처리
  - ✓ 구현
    - ● 게시자 클래스 – HotelEventListener
      - ◆ @EventListener 어노테이션의 value 속성은 하나 이상의 클래스 타입을 설정할 수 있으며 구독할 이벤트 메시지의 클래스 타입을 정의할 수 있음
      - ◆ 정의된 이벤트 메시지가 게시되면 @EventListener 애너테이션이 정의된 메서드가 이벤트를 구독하고 실행
      - ◆ HotelCreateEvent. class 이벤트 메시지를 구독
      - ◆ 구독한 이벤트 객체를 메서드의 인자로 받을 수 있어서 인자로 이벤트 객체를 정의하면 ApplicationContext가 발행한 이벤트 객체를 주입하므로 메서드 내부에서 참조할 수 있음

Spring Event

- ❖ @Async 어노테이션을 사용한 비동기 이벤트 처리
  - ✓ 구현
    - ● 게시자 클래스 – HotelEventListener
      - ◆ 이벤트 메시지가 게시되면 하나 이상의 구독 메서드를 실행할 수 있는데 이때 @EventListener 어노테이션을 사용하면 간편하게 여러 구독 메서드를 실행할 수 있음
      - ◆ HotelEventListener를 보면 두 개의 @EventListener 어노테이션이 정의되어 있는데 이벤트 메시지는 한 번 발행되지만 두 번 이상 수신할 수 있는데 이런 상황에서는 멀티 스레드 환경에서 이벤트를 구독 하는 것을 고려할 수 있음
      - ◆ 싱글 스레드에서 여러 구독 클래스를 실행해야 한다면 @Order 어노테이션을 같이 사용하여 실행 순서를 설정하는 것도 고려
      - ◆ @Order의 값이 낮은 순서대로 실행되므로 handleHotelCreateEvent() 메서드가 먼저 실행되고 그 뒤에 handleResourceCreateEvent() 메서드가 실행됨

Spring Event

❖ @Async 어노테이션을 사용한 비동기 이벤트 처리

✓ 구현

● 서비스 클래스 – HotelService

```
@Slf4j
@Service
public class HotelService {

    private final HotelEventPublisher hotelEventPublisher;

    public HotelService(HotelEventPublisher hotelEventPublisher) {
        this.hotelEventPublisher = hotelEventPublisher;
    }

    public Boolean createHotel(String hotelName, String hotelAddress) {
        // 호텔 생성 로직 생략
        log.info("created hotel. {}, {}", hotelName, hotelAddress);
        hotelEventPublisher.publishHotelCreated(999111222L, hotelAddress);
        log.info("done create hotel");
        return Boolean.TRUE;
    }
}
```

Spring Event

❖ @Async 어노테이션을 사용한 비동기 이벤트 처리

✓ 구현

● Application 클래스 수정

```
@SpringBootApplication
public class EventApplication {

    public static void main(String[] args) {
        ConfigurableApplicationContext ctxt =
                SpringApplication.run(EventApplication.class, args);

        HotelService hotelService = ctxt.getBean(HotelService.class);
        hotelService.createHotel("The Ritz-Carlton, Marina del Rey", "4375 Admiralty
    Way, Marina Del Rey, CA 90292");
    }

}
```

Spring Event

❖ @Async 어노테이션을 사용한 비동기 이벤트 처리

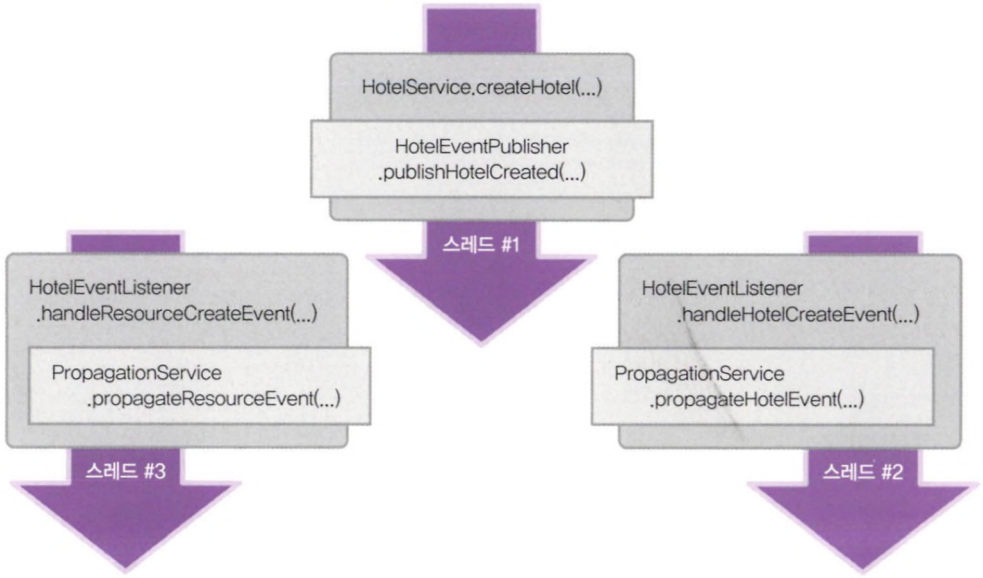
✓ 구현

● 실행 결과

```
[  restartedMain] com.example.event.service.HotelService   : created hotel. The Ritz-Carlton, Marina del Rey, 4375 Admiralty Way, Marina Del Rey, CA 90292
[  restartedMain] c.e.event.service.HotelEventPublisher    : Publish hotel created event.
[  restartedMain] com.example.event.service.HotelService   : done create hotel
[asyncExecutor-1] c.e.event.service.HotelEventListener     : handle HotelCreatedEvent : HotelCreateEvent(hotelId=999111222, hotelAddress=4375 Admiralty Wa
[asyncExecutor-2] c.e.event.service.HotelEventListener     : handle resourceCreatedEvent : HotelCreateEvent(hotelId=999111222, hotelAddress=4375 Admiralty
[asyncExecutor-1] c.e.event.service.PropagationService     : propagation of hotel event
[asyncExecutor-2] c.e.event.service.PropagationService     : propagation of resource event
```

Spring Event

- ❖ @Async 어노테이션을 사용한 비동기 이벤트 처리
  - ✓ 구현
    - ● 실행 결과
      - ◆ 실행 결과를 확인하면 HotelEventListener와 Propagationservice의 메서드들은 각각 asyncExecutor-1과 asyncExecutor-2 스레드에서 실행되었는데 스레드들은 AsyncExecutionConfig에서 설정한 스레드 풀에서 할당받은 것



Spring Event

- ❖ @Async 어노테이션을 사용한 비동기 이벤트 처리
  - ✓ 구현
    - ● 실행 결과
      - ◆ Hotelservice의 createHotel() 메서드가 게시한 이벤트는 두 번 구독하는데 HotelEventListener의 handleHotelCreateEvent()와 handleResource CreateEvent() 메서드
      - ◆ 이전 예제는 게시 클래스가 게시한 이벤트 메시지를 하나의 구독 클래스가 처리하는 형태였지만 여러 이벤트 리스너가 동작할 수 있음
      - ◆ HotelEventListener는 어노테이션을 사용하여 이벤트 메시지를 구독할 수 있음
      - ◆ 각 메서드에는 @Async, @Order 어노테이션이 설정되어 있는데 @Async 어노테이션 때문에 별도의 스레드에서 비동기로 구독 클래스의 메서드가 실행되고 두 개 이상의 구독 메서드가 동작할 때 순서가 필요하다면 @Order 어노테이션을 사용
      - ◆ HotelEventListener 는 비동기로 동작하므로 순서가 크게 중요하지 않음

Spring Event

- Spring Application Event
  - ✓ 스프링 프레임워크는 스프링 애플리케이션이 시작하여 실행 가능한 상태까지 발생할 수 있는 여러 이벤트를 미리 정의하고 이를 게시하는 기능을 제공
  - ✓ 프레임워크에서 발생한 이벤트를 구독하는 구독 클래스를 작성할 때는 @EventListener 대신 ApplicationListener 인터페이스를 구현하는 방법을 권장
  - ✓ 스프링 프레임워크에서 제공하는 이벤트
    - ApplicationStartingEvent: 애플리케이션이 시작하는 시점에 발생하는 이벤트로 애플리케이션이 시작하는 이벤트이므로 스프링 빈 스캔 및 로딩 같은 작업도 시작하지 않은 상태인데 ApplicationContext의 Listener 나 Initalizer를 등록하는 과정은 실행된 상태
    - ApplicationEnvironmentPreparedEvent: ApplicationContext에서 사용하는 환경 설정 객체 인 Environment가 사용 준비된 시점에 발생하는 이벤트지만 ApplicationContext 객체가 생성되기 전에 발생
    - ApplicationContextInitializedEvent: ApplicationContext가 준비되고 Application ContextInitializers를 호출한 후 발생하는 이벤트로 이때도 스프링 빈이 로딩되기 전
    - ApplicationPreparedEvent: 정의된 스프링 빈들이 이미 로딩되고 refresh 되기 전에 발생하는 이벤트
    - ApplicationStartedEvent: refresh된 후 발생하는 이벤트지만 애플리케이션과 커맨드 라인 러너가 호출되기 전에 발생
    - AvailabilityChangeEvent: 애플리케이션이 실행 준비됨을 의미하는 LivenessState. CORRECT 다음에 발생하는 이벤트

Spring Event

- Spring Application Event
  - ✓ 스프링 프레임워크에서 제공하는 이벤트
    - ApplicationReadyEvent: 모든 애플리케이션과 커맨드 라인 러너가 실행된 후 발생하는 이벤트
    - AvailabilityChangeEvent: 애플리케이션이 서비스 요청을 처리할 수 있음을 의미하는 ReadinessState.ACCEPTING_TRAFFIC 다음에 발생하는 이벤트
    - ApplicationFailedEvent: 예외가 발생하여 애플리케이션이 정상으로 실행되지 못할 때 발생하는 이벤트

Spring Event

- ❖ Spring Application Event
  - ✓ 스프링 프레임워크에서 제공하는 이벤트
    - 스프링 애플리케이션이 기동하는 과정을 여러 단계로 구분하고 각 단계에서 발생할 수 있는 이벤트들을 스프링 프레임워크는 정의하고 게시
    - 최종 단계에서는 ApplicationContext가 스프링 빈을 스캔하고 로딩하고 사용할 수 있지만 이전 단계에서는 스프링 빈을 스캔조차 하지 않는 단계도 있음
    - ApplicationStartingEvent는 애플리케이션이 시작할 때 발생하는 이벤트로 스프링 빈을 스캔하지도 않은 상태인데 이벤트 구독 클래스가 스프링 빈으로 정의되어 있고 @EventListener 어노테이션을 사용하여 이벤트를 구독한다면 정상적으로 이벤트를 구독할 수 없으므로 스프링 프레임워크에서 게시하는 이벤트를 구독할 때는 다음 방식으로 개발하는 것을 권장
      - ◆ 구독 클래스는 ApplicationListener 인터페이스를 구현하고 onApplicationEvent() 추상 메서드를 구현
      - ◆ 구독 클래스 객체를 생성하고 SpringApplication의 addListener() 메서드를 사용하여 생성한 객체를 직접 추가
    - 단계에 따라 스프링 빈을 스캔한 이벤트도 있는데 ApplicationReadyEvent는 스프링 빈과 @EventListener 어노테이션으로 정의된 구독 클래스가 이벤트를 구독할 수 있지만 각각의 이벤트가 어떤 단계에서 발생하는지 확인해야 하는데 이런 작업이 힘들거나 이벤트 게시 시점에 확신이 없다면 ApplicationListener 인터페이스를 구현하는 방법을 추천

Spring Event

- Spring Application Event
  - ✓ 스프링 프레임워크에서 제공하는 이벤트
    - ApplicationEventListener 클래스 생성

```
@Slf4j
public class ApplicationEventListener implements
ApplicationListener<ApplicationReadyEvent> {

    @Override
    public void onApplicationEvent(ApplicationReadyEvent event) {

        long timestamp = event.getTimestamp();
        LocalDateTime eventTime =
LocalDateTime.ofInstant(Instant.ofEpochMilli(timestamp), ZoneId.systemDefault());

        log.info("Application is Ready. {}", eventTime);
    }
}
```

Spring Event

- ❖ Spring Application Event
  - ✓ 스프링 프레임워크에서 제공하는 이벤트
    - ● ApplicationEventListener 클래스 등록 - EventApplication

```
@SpringBootApplication
public class EventApplication {

    public static void main(String[] args) {
        SpringApplicationBuilder appBuilder = new
SpringApplicationBuilder(EventApplication.class);
        SpringApplication application = appBuilder.build();
        application.addListeners(new ApplicationEventListener());

        ConfigurableApplicationContext ctxt = application.run(args);

        UserService userService = ctxt.getBean(UserService.class);
        userService.createUser("adam", "itstudy@kakao.com");
    }

}
```

Spring Event

- ❖ Spring Application Event
  - ✓ 스프링 프레임워크에서 제공하는 이벤트
    - ● ApplicationEventListener 클래스 등록 - EventApplication
      - ◆ SpringApplication 객체와 Applicationcontext를 설정할 수 있는 메서드를 제공하는 SpringApplicationBuilder 클래스를 사용하여 appBuilder 객체를 생성
      - ◆ SpringApplicationBuilder build() 메서드를 호출하여 SpringApplication 객체를 생성
      - ◆ SpringApplication의 addListener() 메서드 인자는 하나 이상의 ApplicationListener 객체들을 받는데 ApplicationReadyEvent 이벤트 메시지를 구독하는 ApplicationEventListener 객체를 생성하고 addListener() 메서드 인자로 전달해서 Applicatio EventListener 객체는 ApplicationReadyEvent 이벤트를 구독할 수 있도록 설정
      - ◆ SpringApplication의 run() 메서드를 사용하여 스프링 부트 애플리케이션을 실행

Spring Event

- Spring Application Event
  - ✓ 스프링 프레임워크에서 제공하는 이벤트
    - 실행

```
[  restartedMain] com.example.event.EventApplication       : Started EventApplication in 1.391 seconds (process running
[  restartedMain] com.example.event.service.UserService    : created user. adam, itstudy@kakao.com
[eventExecutor-1] c.e.e.config.ApplicationEventListener    : Application is Ready. 2023-03-19T11:37:09.786
[  restartedMain] c.e.event.service.UserEventPublisher     : Publish user created event.
[  restartedMain] com.example.event.service.UserService    : done create user
[eventExecutor-1] c.e.event.service.UserEventListener      : Listen CREATE event. 1239876, itstudy@kakao.com
[eventExecutor-1] com.example.event.service.EventService   : Send Email attached welcome coupons. itstudy@kakao.com
```

Spring Event

- 트랜잭션 시점에 구독한 이벤트 처리
  - ✓ 스프링 프레임워크에서는 트랜잭션 종료 단계(phase)와 연계하여 이벤트를 구독하는 기능을 제공
  - ✓ 프레임워크에서 제공하는 @TransactionalEventListener 어노테이션을 구독 메서드에 정의하면 됨
  - ✓ @EventListener와 @TransactionalEventListener 애너테이션을 비교해 보면 공통점은 두 어노테이션 모두 구독 메서드에 정의한다는 것이고 다른 점은 @EventListener 어노테이션을 사용한 구독 메서드는 게시하는 즉시 바로 실행되고 @TransactionalEventListener 어노테이션을 사용한 구독 메서드는 게시한 시점이 아니라 트랜잭션이 종료되는 시점에 실행된다는 것

Spring Event

- 트랜잭션 시점에 구독한 이벤트 처리
  - ✓ 트랜잭션 종료 단계는 열거형 클래스에 정의되어 있고 @TransactionalEventListener 어노테이션의 속성에 정의하는데 org.springframework.transaction.event .Transactionphase 열거형

```
public enum Transactionphase {
    BEFORE_COMMIT,
    AFTER_COMMIT,
    AFTER_ROLLBACK,
    AFTER_COMPLETION
}
```

  - 트랜잭션을 커밋하기 직전에 이벤트를 처리
  - 트랜잭션을 커밋한 후 이벤트를 처리
  - 트랜잭션을 롤백한 후 이벤트를 처리
  - 트랜잭션을 롤백하거나 커밋한 후 이벤트를 처리

Spring Event

- 트랜잭션 시점에 구독한 이벤트 처리
  - ✓ TransactionalEventListener

```
@Target( {ElementType. METHOD, ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@EventListener
public @interface TransactionalEventListener {
    Transactionphase phase() default Transactionphase.AFTER_COMMIT;
    boolean fallbackExecution() default false;
    @AliasFor(annotation=EventListener.class, attribute="classes") Class<?>[] classes()
    default {};
    // 생략
}
```

- 어떤 트랜잭션 종료 단계에서 이벤트 구독 메서드를 실행할지 정의하는데 기본값은 AFTER_COMMIT이므로 커밋이 정상적으로 실행되면 구독 메서드가 실행되는데 롤백이나 다른 단계에서는 실행되지 않음
- 구독 메서드를 실행하는 단계에서 트랜잭션이 없다면 구독 메서드 실행 여부를 설정하는데 기본값이 false이므로 실행 중인 트랜잭션이 없으면 구독 메서드를 실행하지 않음
- 구독할 특정 이벤트 메시지 클래스의 클래스 타입을 설정

Spring Event

❖ 트랜잭션 시점에 구독한 이벤트 처리

✓ TransactionalEventListener 사용하는 방법

```
@Slf4j
@Component
public class HotelEventListener {
    @Order(1)
    @TransactionalEventListener(classes=HotelCreateEvent.class, fallbackExecution=true)
    public void handleHotelCreateEvent(HotelCreateEvent hotelCreateEvent) {
            //생략
```

- classes 속성에 정의된 HotelCreateEvent 이벤트 메시지가 게시되면 HotelEventListener의 handleHotelCreateEvent() 메서드가 실행
- fallbackExeciition=true로 설정했으므로 handleHotelCreateEvent() 메서드를 실행할 때 트랜잭션이 없어도 실행
- ApplicationEventMulticaster에 스레드 풀을 설정하지 않는다면 게시 클래스와 구독 클래스 모두 같은 스레드에서 동작하므로 게시 클래스의 트랜잭션에 포함하여 동작
- ApplicationEventMulticaster에 스레드 풀을 설정한다면 게시 클래스와 구독 클래스는 각각 다른 스레드에서 동작하므로 구독 클래스는 게시 클래스에서 사용된 트랜잭션에는 포함될 수 없는데 이 경우 @TransactionalEventListener의 fallbackExecution 설정을 반드시 true로 설정해야 하는데 그렇지 않으면 게시 클래스는 동작하지 않음