Spring Boot Event Listener Documentation

July 20, 2025

Contents

1 Overview

This documentation provides a comprehensive guide to implementing a Spring Boot application that uses `ApplicationEventPublisher` with both `@EventListener` and `@TransactionalEventListener` to handle domain events in a transaction-safe and decoupled manner. The example focuses on a user registration use case, where a notification (e.g., an email) is triggered after a user is successfully saved to the database. It compares `@EventListener` (synchronous, non-transactional) and `@TransactionalEventListener` (transaction-bound) to highlight their differences and optimal use cases.

Key features:

- **Event-Driven Design**: Uses Springs `ApplicationEventPublisher` to decouple user registration from notification logic.
- **Flexible Event Handling**: Supports both `@EventListener` and `@TransactionalEventListener` for different scenarios.
- **Clean Architecture**: Organizes code in a modular, reusable way.
- **Asynchronous Support**: Optionally uses `@Async` for non-blocking notification processing.

2 Project Structure

The project follows a clean, modular structure to ensure maintainability and scalability:

```
com.example.demo
 controller
    UserController.java
 model
    User.java
 repository
    UserRepository.java
 service
    UserService.java
 event
    UserRegisteredEvent.java
    NotificationListener.java
 DemoApplication.java
```

2.1 Explanation of Folders

- **controller/**: REST endpoints for user interaction.
- **model/**: JPA entities (e.g., `User`).
- **repository/**: Spring Data JPA repositories for database operations.
- **service/**: Business logic for user operations.
- **event/**: Event-related classes, including event models and listeners.

3 Key Components

3.1 User Entity

The `User` class represents the entity stored in the database.

```java
package com.example.demo.model;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;

@Entity
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String email;
}
```

Listing 1: User.java

- **Annotations**: Uses Lombok (`@Getter`, `@Setter`, etc.) to reduce boilerplate and JPA annotations for database mapping.
- **Fields**: Basic fields like `id`, `name`, and `email`.

3.2 UserRegisteredEvent

This is the event class that carries the `User` data when published.

```java
package com.example.demo.event;

import com.example.demo.model.User;
import lombok.AllArgsConstructor;
import lombok.Getter;

@Getter
@AllArgsConstructor
public class UserRegisteredEvent {
    private final User user;
}
```

Listing 2: UserRegisteredEvent.java

- **Purpose**: Acts as a payload for the event, carrying the `User` object.
- **Design**: Simple POJO (no need to extend `ApplicationEvent` in modern Spring versions).

3.3 UserRepository

The repository interface for database operations on `User`.

```
package com.example.demo.repository;

import com.example.demo.model.User;
import org.springframework.data.jpa.repository.JpaRepository;

public interface UserRepository extends JpaRepository<User, Long> {}
```

Listing 3: UserRepository.java

- **Purpose**: Provides CRUD operations for the `User` entity.
- **Extends**: `JpaRepository` for out-of-the-box database functionality.

3.4 UserService

The service layer handles business logic and publishes the event.

```
package com.example.demo.service;

import com.example.demo.model.User;
import com.example.demo.event.UserRegisteredEvent;
import com.example.demo.repository.UserRepository;
import lombok.RequiredArgsConstructor;
import org.springframework.context.ApplicationEventPublisher;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Service
@RequiredArgsConstructor
public class UserService {

    private final UserRepository userRepository;
    private final ApplicationEventPublisher eventPublisher;

    @Transactional
    public User registerUser(User user) {
        User savedUser = userRepository.save(user);
        eventPublisher.publishEvent(new
            UserRegisteredEvent(savedUser)); // Publish event
        return savedUser;
    }
}
```

Listing 4: UserService.java

- **@Transactional**: Ensures the database operation and event publishing are part of the same transaction.
- **Event Publishing**: Publishes `UserRegisteredEvent` after saving the user.

3.5 NotificationListener

The listener class that processes the event using both `@EventListener` and `@TransactionalEventListener`.

```java
package com.example.demo.event;

import com.example.demo.model.User;
import com.example.demo.event.UserRegisteredEvent;
import lombok.RequiredArgsConstructor;
import org.springframework.context.event.EventListener;
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;
import org.springframework.transaction.event.TransactionPhase;
import
    org.springframework.transaction.event.TransactionalEventListener;

@Component
@RequiredArgsConstructor
public class NotificationListener {

    @EventListener
    @Order(1)
    public void handleUserRegisteredSync(UserRegisteredEvent event) {
        User user = event.getUser();
        System.out.println(" [Sync] Logging user registration: " +
            user.getName());
    }

    @TransactionalEventListener(phase = TransactionPhase.AFTER_COMMIT)
    @Order(2)
    public void handleUserRegisteredTransactional(UserRegisteredEvent
        event) {
        User user = event.getUser();
        System.out.println(" [Transactional] Sending welcome email to:
            " + user.getEmail());
    }
}
```

Listing 5: NotificationListener.java

- **@EventListener**: Handles the event synchronously, immediately when published, within the transaction.
- **@TransactionalEventListener**: Triggers only after the transaction commits successfully, ideal for external actions like notifications.
- **@Order**: Ensures the synchronous listener runs before the transactional one for controlled execution.

3.6 UserController

Exposes a REST endpoint to trigger user registration.

```java
package com.example.demo.controller;

import com.example.demo.model.User;
import com.example.demo.service.UserService;
import lombok.RequiredArgsConstructor;
import org.springframework.http.ResponseEntity;
```

```java
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequiredArgsConstructor
@RequestMapping("/api/users")
public class UserController {

    private final UserService userService;

    @PostMapping
    public ResponseEntity<User> register(@RequestBody User user) {
        User saved = userService.registerUser(user);
        return ResponseEntity.ok(saved);
    }
}
```

Listing 6: UserController.java

- **Purpose**: Provides a REST API to trigger the user registration process.
- **Endpoint**: `POST /api/users` accepts a JSON `User` object.

4 Implementation Details

4.1 Comparing @EventListener and @TransactionalEventListener

Heres a detailed comparison of the two event handling approaches:

- **@EventListener**:
  - *Execution Timing*: Immediately when the event is published, within the current transaction.
  - *Transaction Dependency*: Works with or without a transaction.
  - *Safety for Side Effects*: Risky for external actions (e.g., email, API calls) if the transaction rolls back.
  - *Use Case*: Internal logging, UI updates, or non-critical tasks.
  - *Performance*: Synchronous, may block the main thread.
- **@TransactionalEventListener**:
  - *Execution Timing*: After the transaction commits (default: `AFTER_COMMIT`).
  - *Transaction Dependency*: Requires an active Spring-managed transaction.
  - *Safety for Side Effects*: Safe for external actions, as it runs only after successful commit.
  - *Supported Phases*: `AFTER_COMMIT`, `AFTER_ROLLBACK`, `AFTER_COMPLETION`, `BEFORE_COMMIT`.
  - *Use Case*: Notifications, external API calls, audit logging.
  - *Performance*: Can be combined with `@Async` for non-blocking execution.

Example Scenario:

- `@EventListener`: Logs the user registration immediately (useful for debugging or non-critical tasks).
- `@TransactionalEventListener`: Sends a welcome email only after the user is saved to the database, ensuring no email is sent if the transaction fails.

4.2 When to Use Each

- **@EventListener**:
  - For lightweight, internal tasks that dont depend on transaction outcomes (e.g., logging, updating in-memory state).
  - When you dont need transaction safety or when theres no transaction involved.
  - Example: Logging user activity for debugging.
- **@TransactionalEventListener**:
  - For actions that must occur only after a successful database commit (e.g., sending emails, updating external systems).
  - When you need to ensure consistency between database state and side effects.
  - Example: Sending a welcome email after user registration.

4.3 Transaction Safety

- `@Transactional` on `UserService.registerUser` ensures that the user save and event publishing are part of the same transaction.
- `@EventListener`: Executes within the transaction, so if the transaction rolls back, the listeners effects (e.g., logs) may persist, causing inconsistencies.
- `@TransactionalEventListener(phase = TransactionPhase.AFTER_COMMIT)`: Guarantees that the notification is triggered only if the transaction commits, avoiding side effects for failed operations.

4.4 Asynchronous Processing

To make listeners non-blocking, add `@Async`:

```
1 @Async
2 @TransactionalEventListener(phase = TransactionPhase.AFTER_COMMIT)
3 public void handleUserRegisteredTransactional(UserRegisteredEvent
      event) {
4     // ...
5 }
```

- Requires `@EnableAsync` in a configuration class.
- Improves API response time by offloading tasks to a background thread.

5 Setup and Dependencies

To run this project, you need:

- **Spring Boot**: Version 3.x (or compatible).
- **Dependencies** (add to `pom.xml` or `build.gradle`):

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

- **Database**: A configured database (e.g., H2 for testing, MySQL, or PostgreSQL for production).

5.1 Configuration

Add to `application.properties`:

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.jpa.hibernate.ddl-auto=update
```

For async support, add:

```java
package com.example.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.scheduling.annotation.EnableAsync;

@SpringBootApplication
@EnableAsync
public class DemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

Listing 7: DemoApplication.java

6 How It Works

1. A client sends a `POST /api/users` request with a JSON payload (e.g., `"name": "Piyush", "email": "piyush@example.com"`).
2. `UserController` delegates to `UserService.registerUser`.
3. `UserService`:
   - Saves the user to the database within a transaction.
   - Publishes a `UserRegisteredEvent`.
4. `NotificationListener`:

- The `@EventListener` method logs the event immediately (within the transaction).
- The `@TransactionalEventListener` method triggers the notification (e.g., email simulation) only after the transaction commits.

5. Notifications are printed to the console (replaceable with actual email logic).

7 Benefits of This Approach

- **Decoupled Logic**: User registration and notification logic are separated, improving maintainability.
- **Transaction Safety**: `@TransactionalEventListener` ensures notifications are triggered only after successful database commits, preventing inconsistencies.
- **Flexibility**: `@EventListener` allows quick, synchronous handling for non-critical tasks, while `@TransactionalEventListener` ensures safety for critical side effects.
- **Scalability**: Easily add more listeners (e.g., logging, analytics) without modifying `UserService`.
- **Reusability**: The event-driven pattern can be reused for other entities or events.
- **Performance**: Async listeners improve API response times by offloading tasks.

8 Potential Extensions

1. **Asynchronous Notifications**:

```
@Async
@EventListener
public void handleUserRegisteredSync(UserRegisteredEvent event) {
    // ...
}

@Async
@TransactionalEventListener(phase = TransactionPhase.AFTER_COMMIT)
public void handleUserRegisteredTransactional(UserRegisteredEvent
    event) {
    // ...
}
```

2. **Multiple Listeners**: Add more listeners with `@Order` for controlled execution:

```
@Component
public class AnalyticsListener {
    @TransactionalEventListener(phase =
        TransactionPhase.AFTER_COMMIT)
    @Order(3)
    public void updateAnalytics(UserRegisteredEvent event) {
        System.out.println(" Updating analytics for: " +
            event.getUser().getName());
    }
}
```

3. **Error Handling**: Add fallback logic for notification failures:

```
@TransactionalEventListener(phase = TransactionPhase.AFTER_COMMIT)
```

```
public void handleUserRegisteredTransactional(UserRegisteredEvent
    event) {
    try {
        // Email sending logic
    } catch (Exception e) {
        System.out.println(" Failed to send email: " +
            e.getMessage());
    }
}
```

4. **Rollback Handling**: Add a listener for rollback scenarios:

```
@TransactionalEventListener(phase =
    TransactionPhase.AFTER_ROLLBACK)
public void handleRollback(UserRegisteredEvent event) {
    System.out.println(" Transaction rolled back for user: " +
        event.getUser().getName());
}
```

9 Sample Output

Request:

```
curl -X POST http://localhost:8080/api/users -H "Content-Type: application/json" -d '{"name"
```

Console Output:

```
[Sync] Logging user registration: Piyush
[Transactional] Sending welcome email to: piyush@example.com
```

Response:

```
{
  "id": 1,
  "name": "Piyush",
  "email": "piyush@example.com"
}
```

Rollback Scenario (if `UserService` throws an exception):

- `@EventListener`: May still execute (logs appear).
- `@TransactionalEventListener`: Does not execute (no email sent).

10 Conclusion

This implementation provides a robust, scalable, and transaction-safe way to handle domain events in Spring Boot. By combining `@EventListener` for synchronous, non-critical tasks and `@TransactionalEventListener` for transaction-bound side effects, the application achieves flexibility and reliability. The clean folder structure and event-driven design make the codebase reusable and maintainable, ideal for both monolithic and microservices architectures.

Next Steps:

- Add `@Async` for non-blocking notifications.
- Implement additional listeners for logging or analytics.

- Extend to handle more events (e.g., `UserDeletedEvent`, `UserUpdatedEvent`).
- Integrate with a real email service (e.g., Spring Mail).