1. Why Spring Events

Sometimes in your spring application, you may want to **add capability for listening specific application events** so that you can process these events as per application logic.

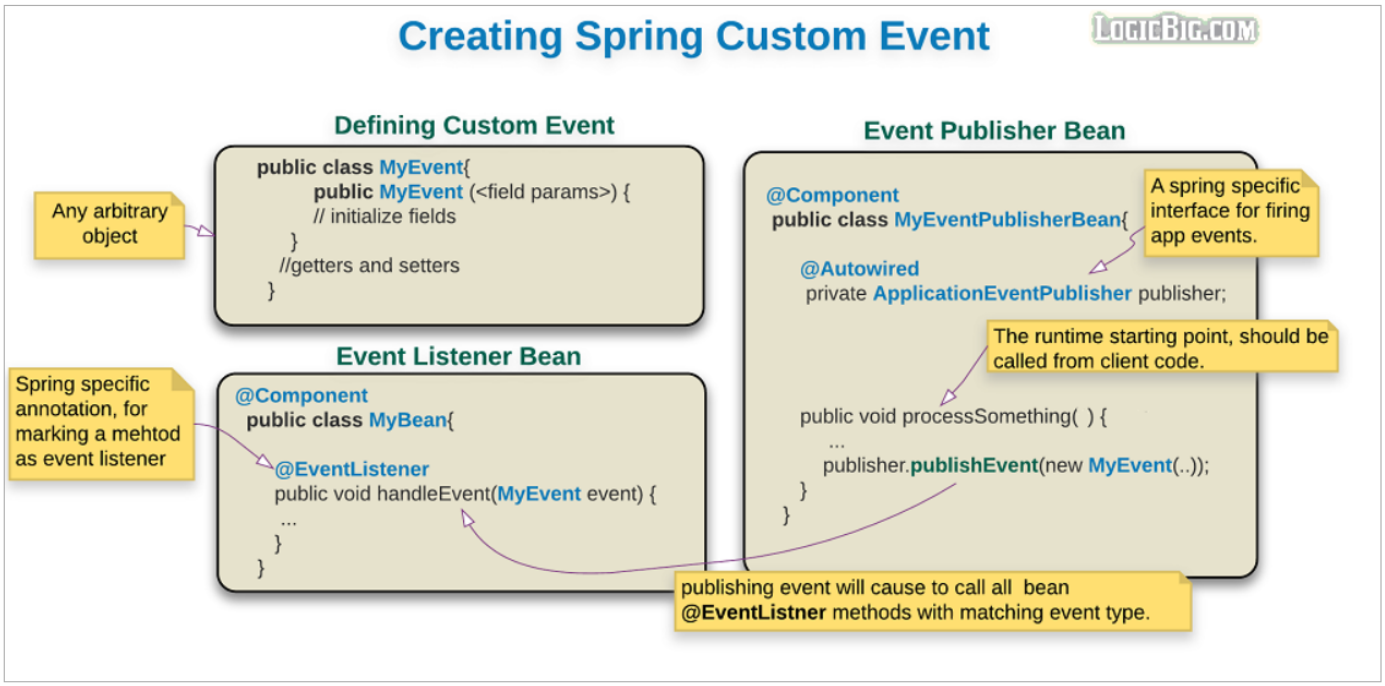
Examples of these events can be when a employee is added/ deleted; or a certain kind of transaction completed or rolled back. You can always write the event processing code as another method within existing application code, BUT then it will be tightly coupled with your existing code and you will not have much handle to change it later (suppose you don't want to process those events for certain duration).

If you can **configure event processors** through your application context file, then you need not to change your application code as well as event processor code in most circumstances. Any time you need to switch off event processing; OR want to add another event processor for that event the all you need to do it change your context file configuration and that's it. Sounds good !!

In spring event-based communication model, the **sender component just publishes an event without knowing who the receiver will be**. Actually, **there may be more than one receiver component**. Also, the receiver needn't know who is publishing the event. **Listener can listen to multiple events from different senders at the same time**. In this way, the sender and receiver components are loosely coupled.

Spring Event Handling, The major part of Spring is the **ApplicationContext**, Application context gives a method for determining instant messages and give a non specific approach to load file assets and it can distribute events to beans that are registered as listeners. The events of Spring as follows.

- **ContextRefreshedEvent:** It conveyed when the ApplicationContext is presented or revived. It can be functioned utilizing the Refresh() procedure on the ConfigurableApplicationContext interface. Here all the beans will be loaded and activated then Applicationcontext is ready to use.
- **ContextStartedEvent:** It passed on when the ApplicationContext is started using the start() method on the ConfigurableApplicationContext interface. Here all the beans in the life cycle will be restarted .
- **ContextStoppedEvent:** It passed on when the ApplicationContext is terminated utilizing the stop() method on the ConfigurableApplicationContext interface. Here life cycle of bean will get stop signal and wait for the start() call.
- **ContextClosedEvent:**It passed on when the ApplicationContext is closed utilizing the close() method on the ConfigurableApplicationContext interface. Here singleton beans will be vanished and start and refresh can not occur.
- **RequestHandledEvent:**It is a web-particular occasion telling all beans that a HTTP assignment has been adjusted. Here it uses DispatcherServlet.



Spring build-in events

Build-in Event	Description
ContextRefreshedEvent	Event fired when an ApplicationContext gets initialized or refreshed (refreshed via `context.refresh()` call).
ContextStartedEvent	Event fired when `context.start()` method is called.
ContextStoppedEvent	Event fired when `context.stop()` method is called
ContextClosedEvent.	Event fired when `context.close()` method is called.
RequestHandledEvent	This event can only be used in spring MVC environment. It is called just after an HTTP request is completed.

How to listen to the events?

There are two ways to listen to the events.

1. Using annotation EventListener on any bean method and injecting the specific event parameter (typically a subtype of ApplicationEvent) to the method.

```
@Component
public class MyBean{

    @EventListener
    public void handleContextRefresh(ContextRefreshedEvent event) {
     ...
    }
 }
```

This method is called when spring context is refreshed.

When event is fired, a proper instance of ContextRefreshedEvent is passed by the framework.

We can name the bean method whatever we want, that doesn't matter here.

Note: The annotation support for event listener was added in Spring 4.2.

2. Or We can implement ApplicationListener<E extends ApplicationEvent> to our bean and implement the method `onApplicationEvent(E event)`

```
@Component
public class MyBean implements ApplicationListener <ContextRefreshedEvent> {

    @Override
    public void onApplicationEvent(ContextRefreshedEvent event) {
      ...
    }
 }
```

Defining custom events

It's not very difficult to create new custom events type, we just have to extend ApplicationEvent class.

```
public class MyEvent extends ApplicationEvent{
      //custom fields

      public MyEvent (Object source, <custom params>) {
          super(source);
          //initialize custom fields
      }

    //getters and setters
  }
```

Publishing custom events

To publish custom events, we will need an instance of ApplicationEventPublisher and then call the method `ApplicationEventPublisher#publishEvent(..)`.

There are two ways to get the instance of ApplicationEventPublisher, inject it as a bean in any spring managed bean or extends ApplicationEventPublisherAware and implement it's `setApplicationEventPublisher` method. First method is recommended, and here's how we are going to use that:

```
public class MyEvenPublisherBean{
       @Autowired
       ApplicationEventPublisher publisher;
       public void sendMsg(String msg){
           publisher.publishEvent(new MyEvent(this, msg));
       }
   }
```

Custom Events:

Introduction

Events are one of the more overlooked functionalities in the framework but also one of the most useful. And—like many other things in Spring—event publishing is one of the capabilities provided by ApplicationContext.

There are a few simple guidelines to follow:

- the event should extend **ApplicationEvent**
- the publisher should inject an ApplicationEventPublisher object
- the listener should implement the ApplicationListener interface

Application events are available since the very beginning of the Spring framework as a mean for loosely coupled components to exchange information.

A Custom Event

Spring allows you to **create and publish custom events** which—by default—are **synchronous**. This has a few advantages, for example the listener being able to participate in the publisher's transaction context.

A Simple Application Event

Let's make a simple event class—just a placeholder to store the event data. In this sample, the event class holds a String message:

```
public class DiscoverSpringEvent extends ApplicationEvent {

private String message;

   public DiscoverSpringEvent(Object source, String message) {
       super(source);
       this.message = message;
   }
   public String getMessage() {
       return message;
   }
}
```

A Publisher

Now let's create a publisher of that event. The publisher constructs the event object and publishes it to anyone who's listening.

To publish the events, the publisher can simply inject the ApplicationEventPublisher and use the publishEvent() API:

```
public class DiscoverSpringEventPublisher {
   @Autowired
   private ApplicationEventPublisher applicationEventPublisher;

   public void doStuffAndPublishAnEvent(final String message) {
       System.out.println("Publishing custom event. ");
       DiscoverSpringEvent discoverSpringEvent = new DiscoverSpringEvent(this, message);
       applicationEventPublisher.publishEvent(discoverSpringEvent);
   }
}
```

Alternatively, the publisher class can implement the ApplicationEventPublisherAware interface; this will also inject the event publisher on the application start-up. Usually, it's simpler to just inject the publisher with @Autowire.

A Listener

Finally, let's create the listener.

The only requirement for the listener is to be a bean and implement ApplicationListener interface:

```
@Component
public class DiscoverSpringEventListener implements ApplicationListener<DiscoverSpringEvent>
{
   @Override
   public void onApplicationEvent(DiscoverSpringEvent event) {
       System.out.println("Received spring custom event - " + event.getMessage());
   }
}
```

2. Publish and Listen Spring Application Events

2.1. Create Custom Application Event Class

All event classes must extend the `ApplicationEvent` class.

EmployeeEvent.java

```
public class EmployeeEvent extends ApplicationEvent
{
    private static final long serialVersionUID = 1L;

    private String eventType;
    private EmployeeDTO employee;

    //Constructor's first parameter must be source
    public EmployeeEvent(Object source, String eventType, EmployeeDTO employee)
    {
        //Calling this super class constructor is necessary
        super(source);
        this.eventType = eventType;
        this.employee = employee;
    }

    public String getEventType() {
        return eventType;
    }

    public EmployeeDTO getEmployee() {
        return employee;
    }
}
```

Here `EmployeeDTO` class looks like this:

EmployeeDTO.java

```
public class EmployeeDTO
{
    private Integer id;
    private String firstName;
    private String lastName;
    private String designation;

    public EmployeeDTO(String designation)
    {
        this.id = -1;
        this.firstName = "dummy";
        this.lastName = "dummy";
        this.designation = designation;
    }

    public EmployeeDTO() {
        // TODO Auto-generated constructor stub
    }

    //Setters and Getters

    @Override
    public String toString() {
        return "Employee [id=" + id + ", firstName=" + firstName
                + ", lastName=" + lastName + ", type=" + designation + "]";
    }
}
```

EmployeeDAO.java

```
public interface EmployeeDAO
{
    public EmployeeDTO createNewEmployee();
}
```

EmployeeDAOImpl.java

```
@Repository ("employeeDao")
public class EmployeeDAOImpl implements EmployeeDAO
{
    public EmployeeDTO createNewEmployee()
    {
        EmployeeDTO e = new EmployeeDTO();
        e.setId(1);
        e.setFirstName("Lokesh");
        e.setLastName("Gupta");
        return e;
    }

    public void initBean() {
        System.out.println("Init Bean for : EmployeeDAOImpl");
    }

    public void destroyBean() {
        System.out.println("Init Bean for : EmployeeDAOImpl");
    }
}
```

EmployeeManager.java

```
public interface EmployeeManager
{
    public EmployeeDTO createNewEmployee();
}
```

2.2. Publisher – Implement ApplicationEventPublisherAware Interface

Any bean, which implements `ApplicationEventPublisherAware` interface, can use its `publishEvent()` method to send event to listeners.

EmployeeManagerImpl.java

```
@Service ("employeeManager")
public class EmployeeManagerImpl implements EmployeeManager, ApplicationEventPublisherAware
{
    @Autowired
    private EmployeeDAO dao;

    private ApplicationEventPublisher publisher;

    //You must override this method; It will give you acces to ApplicationEventPublisher
    public void setApplicationEventPublisher(ApplicationEventPublisher publisher) {
        this.publisher = publisher;
    }

    public EmployeeDTO createNewEmployee()
    {
        EmployeeDTO employee =  dao.createNewEmployee();

        //publishing the veent here
        publisher.publishEvent(new EmployeeEvent(this, "ADD", employee));

        return employee;
    }
}
```

2.3. Subscriber/Listener – Implement ApplicationListener Interface

For a bean to listen to certain events, it must implement the `ApplicationListener` interface and handle the events in the `onApplicationEvent()` method. Actually, Spring will notify a listener of all events, so you must filter the events by yourself.

If you use generics, Spring will deliver only messages that match the generic type parameter. In this example, I am using generics code to listen only `EmployeeEvent`.

UserEventsProcessor.java

```
public class UserEventsProcessor implements ApplicationListener<EmployeeEvent>
{
    public void onApplicationEvent(EmployeeEvent event)
    {
        EmployeeEvent employeeEvent = (EmployeeEvent) event;

        System.out.println("Employee " + employeeEvent.getEventType()
                           + " with details : " + employeeEvent.getEmployee());

        // Do more processing as per application logic
    }
}
```

2.4. Configure the beans in application context file

applicationContext.xml

```
<context:component-scan base-package="com.howtodoinjava.demo" />

<bean class="com.howtodoinjava.demo.processors.UserEventsProcessor" />
```

2.5. Demo

TestSpringContext.java

```
public class TestSpringContext
{
    @SuppressWarnings("resource")
    public static void main(String[] args) throws Exception
    {
        ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

        EmployeeController controller = context.getBean(EmployeeController.class);

        controller.createNewEmployee();
    }
}
```