**Background**

One of the core technologies of the Spring Framework is the Inversion of Control (IoC) container.

IoC is also known as Dependency Injection (DI).

Objects will define their dependencies (that is, the other objects that they work with) through arguments and properties.

Then, the container will inject those dependencies when it creates the bean.

In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and managed by a Spring IoC container. Otherwise, a bean is simply one of many objects in your application.

The *org.springframework.beans* and *org.springframework.context* packages are the basis for Spring Framework’s IoC container.

The *BeanFactory* interface provides an advanced configuration mechanism capable of managing any type of object. *ApplicationContext* is a sub-interface of *BeanFactory*. It adds event publication functionality.

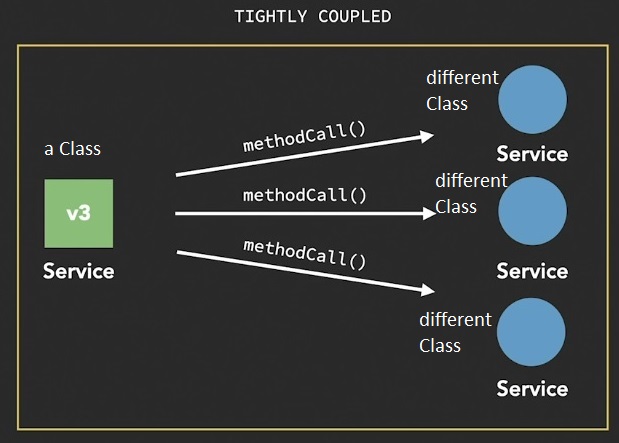
The *org.springframework.context.ApplicationContext* interface represents the Spring IoC container and is responsible for instantiating, configuring, and assembling the beans. The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata. The configuration metadata is represented in XML, Java annotations, or Java code.

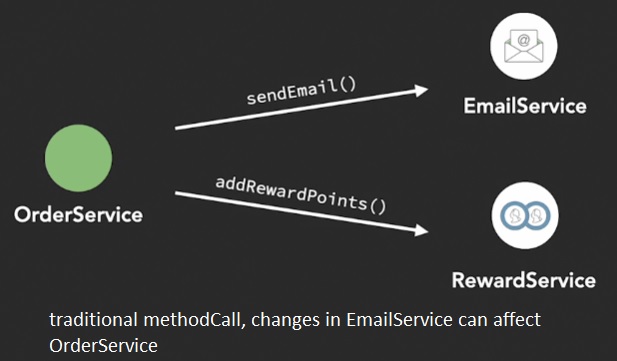
**Spring event-driven architecture**

Traditional approach:

Calling methods in another class results in the two classes being tightly coupled. Difficult to change 1 java file (i.e., class) without affecting others.

<https://www.delftstack.com/howto/java/how-to-call-a-method-in-another-class-in-java/>





Event-driven approach:

eventPublisher -> Spring event system -> many eventListener(s)/Subscriber(s)

more flexible and loosely-coupled than calling methods () in another class

"coupling": degree of knowledge one Object has of another Object that it interacts with.

Loose coupling design handles changes to codes better

- allows new functionalities to be added with no disruption of old ones

- reduces dependency between multiple different components

- reduces risk of unanticipated impact when changes made to one component

- simplifies testing, maintenance and trouble-shooting

- allows pub/sub design

**Pub/Sub model**

- 1 to 1, or 1 to many

- subscribers/listeners can be add or removed at any time

- event publishers and subscribers not tied to each other

- independent, means changes in publisher or listener do not affect each other

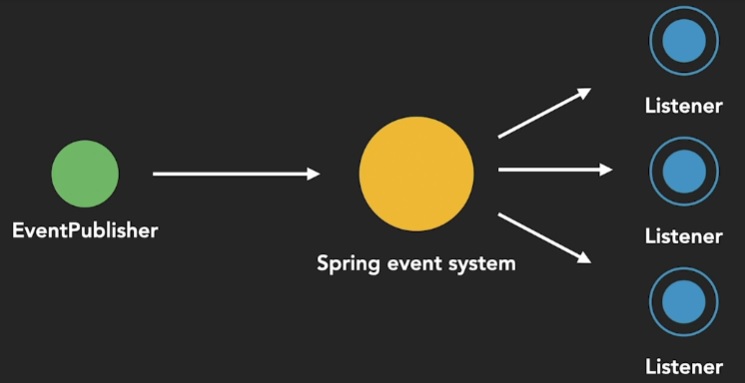
- allow data to be sent to other components effectively

- can invoking logic on multiple components at the same time

- essentially, an Observer pattern (a behavioral design pattern).

if a bean implements ApplicationListener interface, it is notified every time ApplicationEvent is published to ApplicationContext.

Note: Observable/Observer implementation deprecated since Java9, do NOT use.



**Open-Closed Principle**

- close eventPublisher to modifications

- open to modifications / new functionalities by implementing new eventListeners

- net effect: domain/functionalities expand, no disruption of old ones.

**Reference: https://spring.io/event-driven**

- event-driven microservices

- streaming data

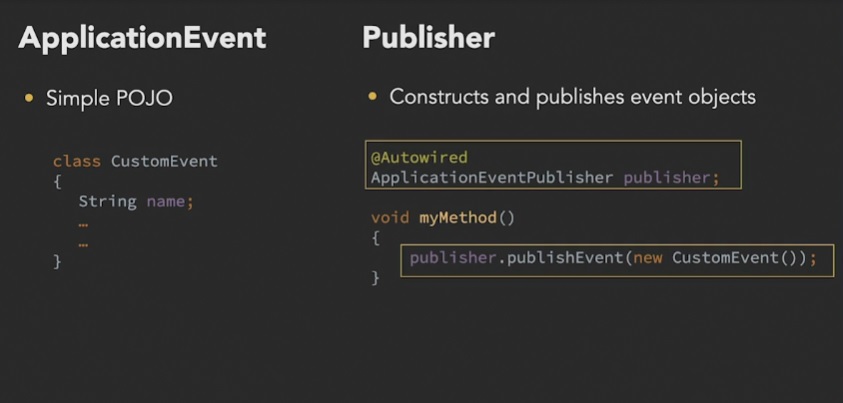
- integration with message handling

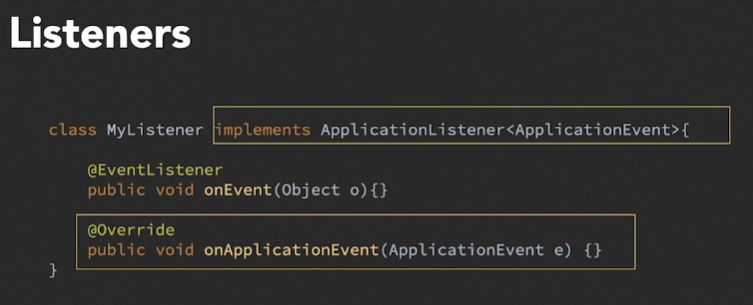
**Spring Events Highlight**

core capability of Spring Framework

publish and listen to specific events

key components: Events, Publishers, Listeners



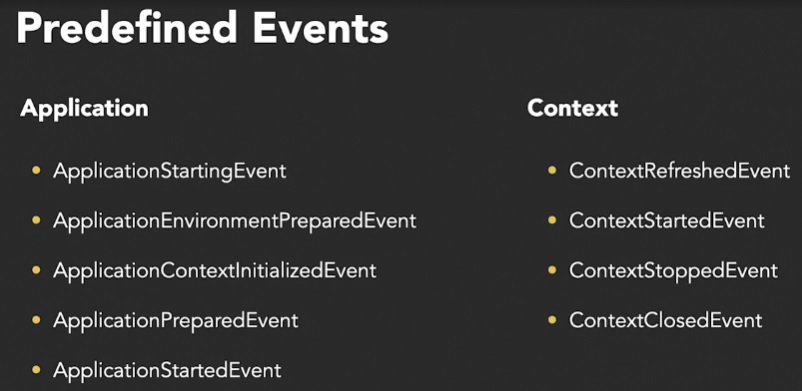


**ApplicationEvent**

* holds data(properties)
* passed from Publisher to Listeners

*Pre-defined Events*

* Built-in events tied to life-cycle of Spring ApplicationContext:



[https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-java-configuration-annotation](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html" \l "beans-java-configuration-annotation)

* If not suitable, you can create your own custom event, then call it using *publishEvent* method on an *ApplicationEventPublisher*.
* If listeners needed before ApplicationContext created, add manually



**Publishers**

* *Either @Autowired* in a *ApplicationEventPublisher* bean
* Or, have class implements *ApplicationEventPublisherAware* then

public void setApplicationEventPublisher (ApplicationEventPublisher publisher)

{ this.publisher = publisher;}

* Plus, call *publisher.publishEvent* method (), pass in Event as parameter

**Listeners**

Spring gives 2 ways to define:

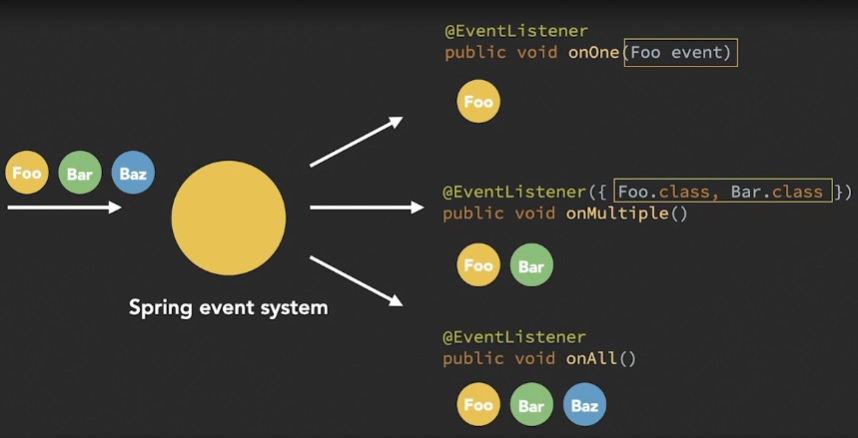
* either class implements *Application Listener<type of Event, can be customEvent> interface*
* Or, annotate method with *@EventListener*

Limitation of ApplicationListener interface

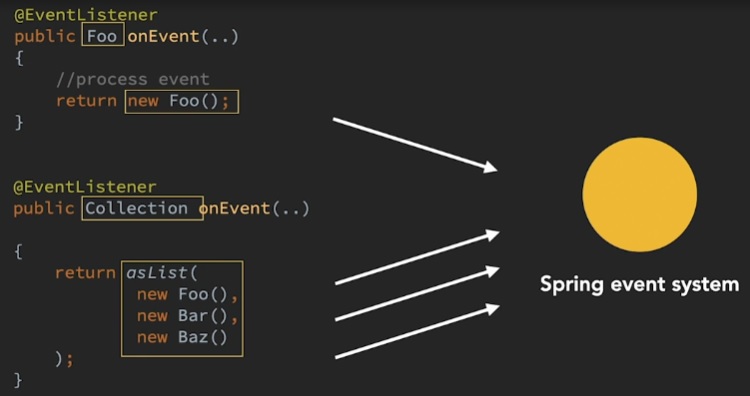
* can only be used to listen for Objects that extend ApplicationEvent class
* Listener can only process 1 event type
* only void return type. Cannot be used to trigger second event.

@EventListener annotation method

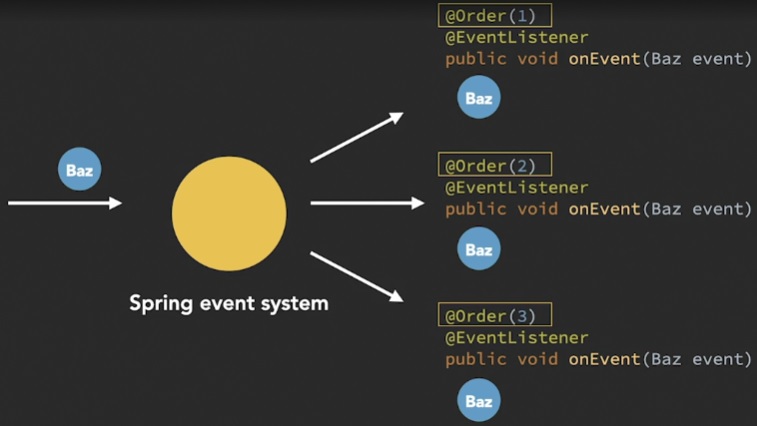
* annotate method(s). Can annotate more than 1 method in a class.
* listen for 1, multiple or all events. Indicate in method signature.



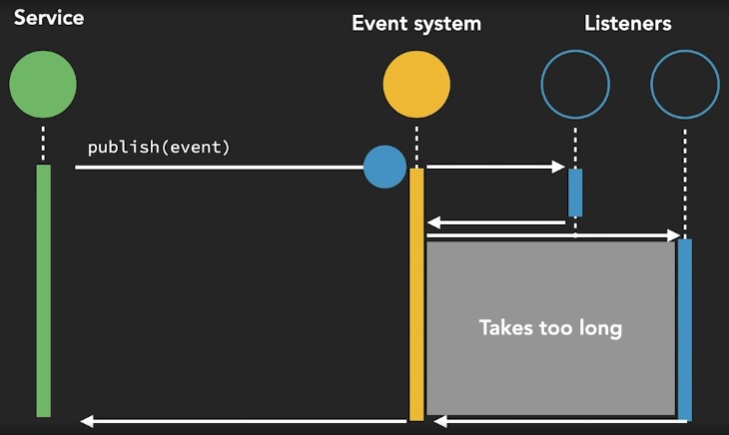
* can be non-void return type. If single return, sent as single NEW event. If return type is array/Collection, each element sent as new event, I.e., multiple new events.

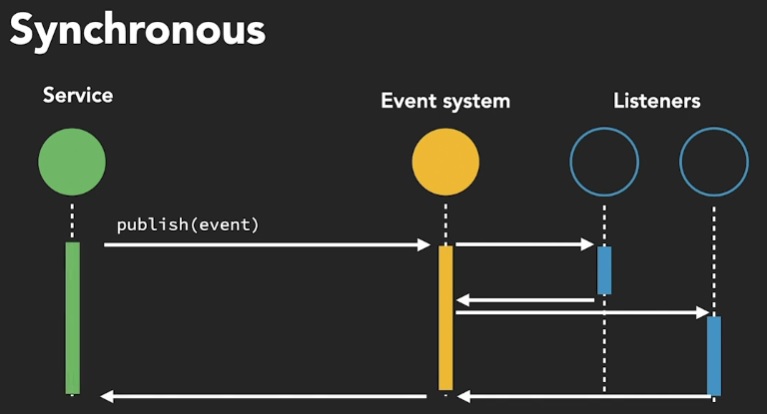


* The order that listeners are invoked can be prioritised using @Order annotation. In response to a single event only.



By default, eventListeners receive events **synchronously.**

* publishEvent() method is blocked until all listeners have finished processing the event.
* 
* when a listener receives an event, it operates inside the transaction context of the publisher.
* single thread
* See [ApplicationEventMulticaster](https://docs.spring.io/spring-framework/docs/5.3.15/javadoc-api/org/springframework/context/event/ApplicationEventMulticaster.html) interface and [SimpleApplicationEventMulticaster](https://docs.spring.io/spring-framework/docs/5.3.15/javadoc-api/org/springframework/context/event/SimpleApplicationEventMulticaster.html) implementation for multi listeners scenarios.



For particular eventListener to process events **asynchronously**, add *@Async* on top of @EventListener,

@Async

@EventListener

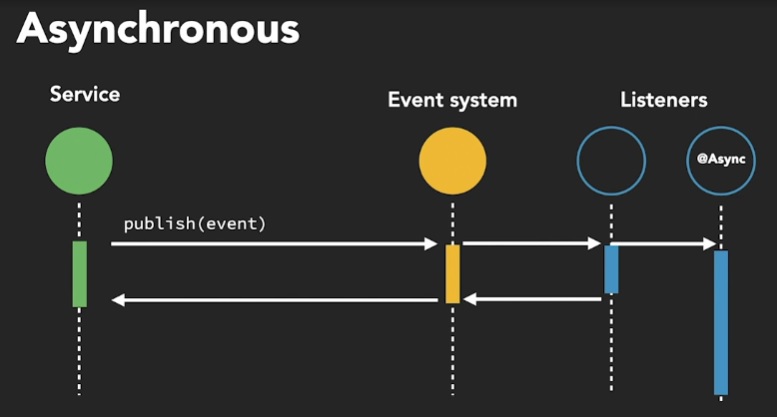
public void processAsyncEvent (EventPassedInParameter event) {

// event is processed in a separate thread from publisher}

And, add @EnableAsync on top of Spring Configuration.



* No return to caller if Exception. If need to catch exception, see [AsyncUncaughtExceptionHandler](https://docs.spring.io/spring-framework/docs/5.3.15/javadoc-api/org/springframework/aop/interceptor/AsyncUncaughtExceptionHandler.html)
* No value returned, so cannot publish subsequent event from processing
* If need to publish another event after processing, need to inject an [ApplicationEventPublisher](https://docs.spring.io/spring-framework/docs/5.3.15/javadoc-api/org/springframework/context/ApplicationEventPublisher.html) to publish the event manually. Use ApplicationEventPublisher.publisher().



**Filtering events**

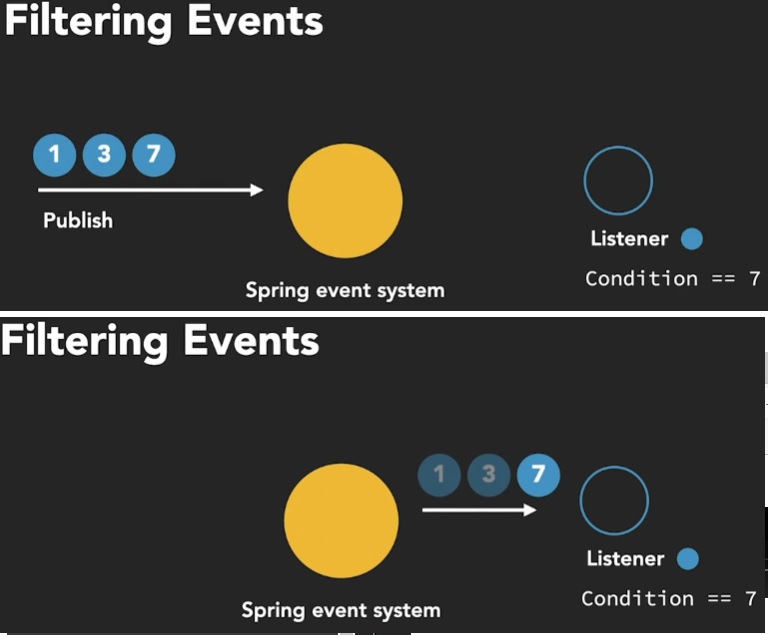
* Runtime filtering
* Event processed only if condition met
* Add **condition** attribute with Spring Expression Language (SpEL) to *@EventListener* annotation

@EventListener (condition = “blEvent.whateverAttribute == ‘value’”)

public void myMethod (someEvent blEvent) {

// body of method that will be invoked if there is a condition match}

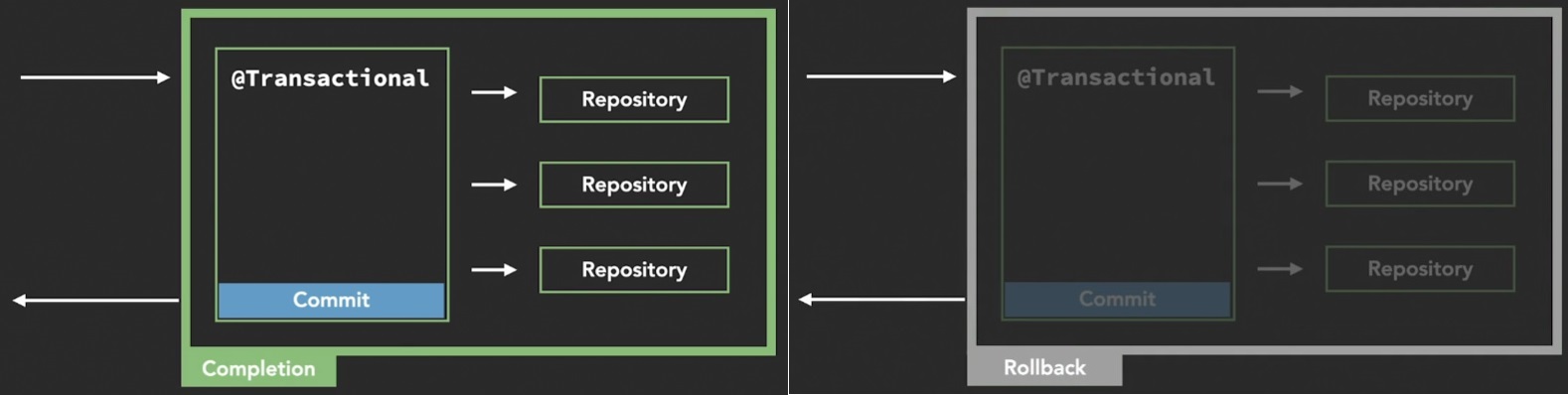
* Event will be handled if condition =
  + “ ” (empty String, always handled)
  + “true”, “on”, “yes”, “1” (these are Strings)
  + True (Boolean type)
* Examples of ways to refer for condition
  + #event – refers to an event
  + #event.customer.type eq ‘b2c’
  + @myBean.test(#event) - to refer to other Spring beans



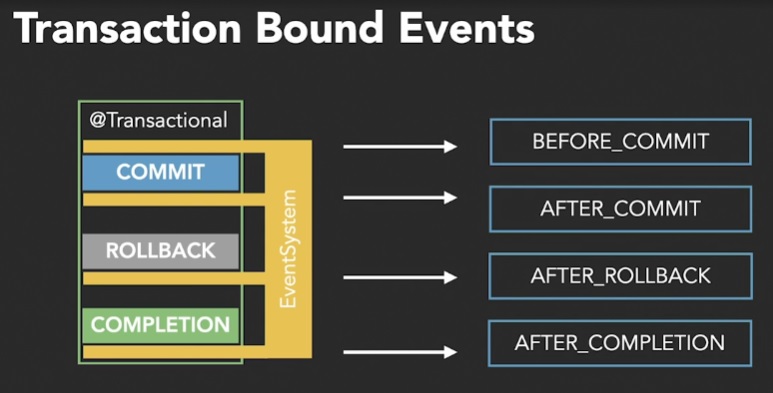
[https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-java-configuration-annotation](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html" \l "beans-java-configuration-annotation)

**Transaction-bound events**

* Spring transactions have to do with databases
* A transaction is a sequence of operations that is treated as a single unit. Entire set of operations is all or none. Atomicity.
* Annotate with @Transactional if want class/method to be executed for transaction. Outcome of all statements to be either successful or nothing.



* *@**TransactionalEventListener(phase=TransactionPhase.PHASE\_NAME\_HERE)* binds event listeners to particular phases of a current transaction
* Used when outcome of phases of transaction matters to event listener, for flexibility.
* Valid phases: BEFORE\_COMMIT, AFTER\_COMMIT (default), AFTER\_ROLLBACK, AFTER\_COMPLETION
* Use AFTER\_COMMIT if want event listener to only run after current transaction successful.
* Avoid making changes to database infrastructure using event listeners. Slower and problematic when transactions roll back.



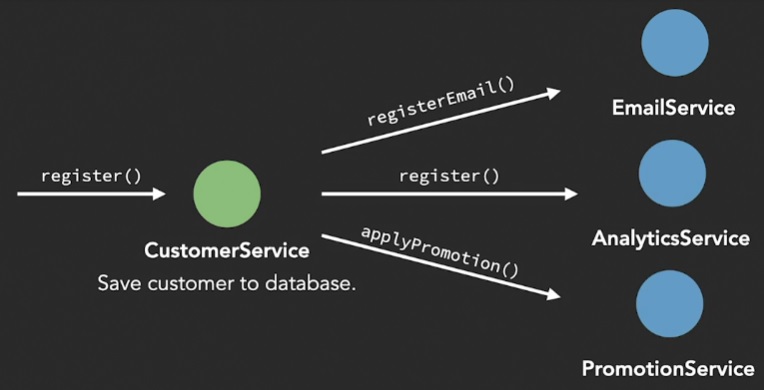
Aside on Transaction Management

* ACID vs BASE
* Local vs global transaction managers
* Programmatic vs declarative (..AOP) transaction management
* [https://docs.spring.io/spring-framework/docs/current/reference/html/data-access.html#transaction](https://docs.spring.io/spring-framework/docs/current/reference/html/data-access.html" \l "transaction)
* <https://www.dineshonjava.com/transaction-management-in-spring/>
* https://www.baeldung.com/java-transactions

**Example of Application:** eCommerce backend



Possible activities in eCommerce backend:



New Customer -> save info to database

-> add to email mailing list

-> register for external analytics service

-> apply for promotion

Spring bean annotations

@Component: EmailService, CustomerService, AnalyticsService, OrderService, PromotionService, TicketService

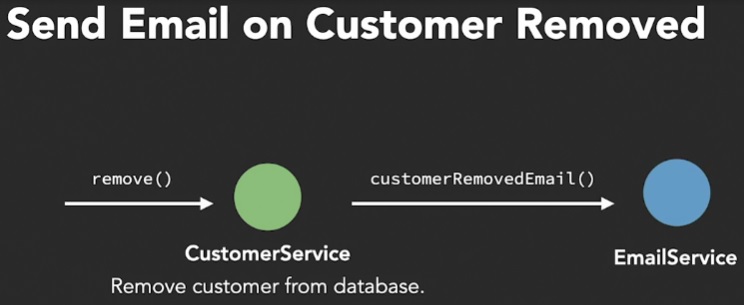
@Repository: CustomerRepository, OrderRepository

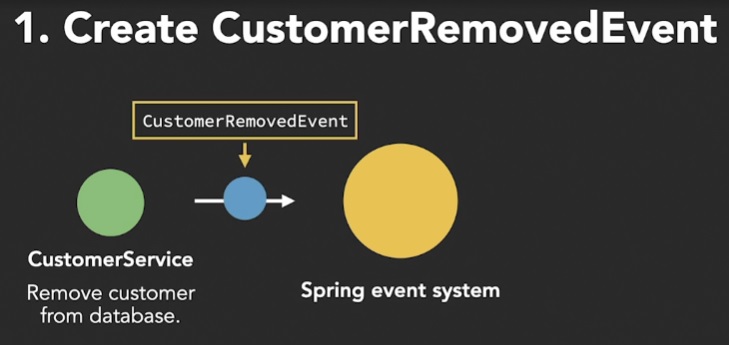
Situation 1: Synchronous event

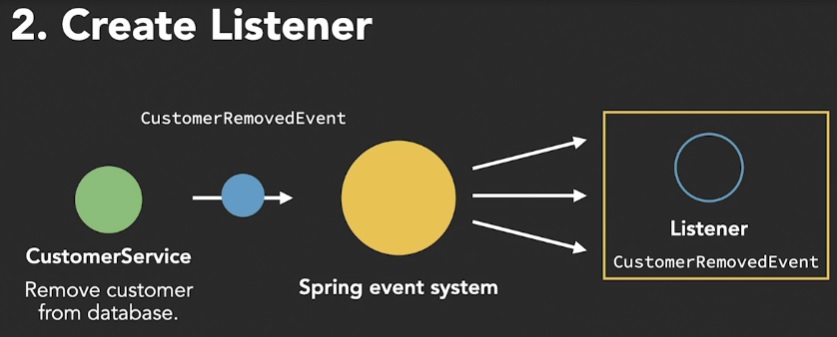
Relevant packages: com.linkedin.events.customer and com.linkedin.events.email

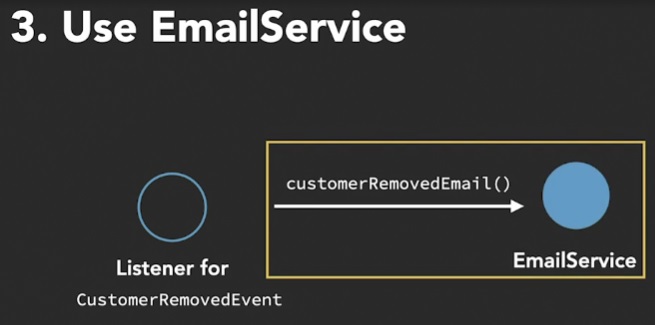
Flow: Customer (DAO) -> CustomerRegisteredEvent / CustomerRemovedEvent (the event) -> CustomerService (publishEvent) -> EmailListeners (event is argument) => EmailService (implementation)

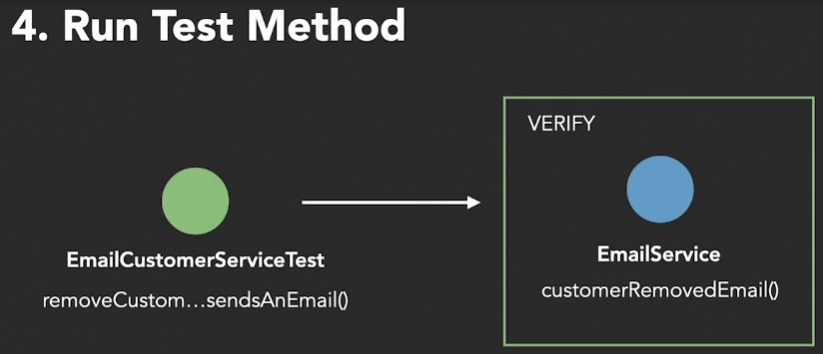
Challenge 1: Refactor codes to send email to Customer to inform his email has been removed.











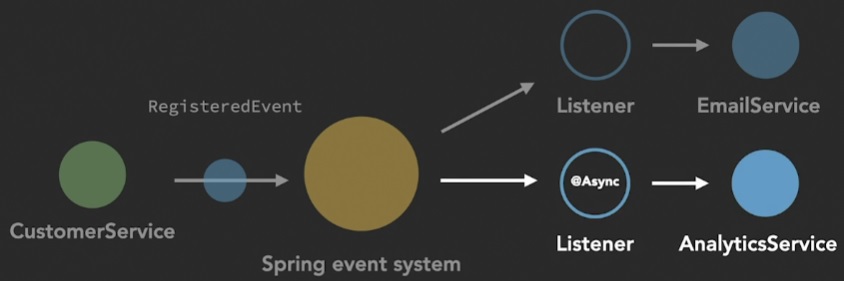
Situation 2: Asynchronous event

Call to external Analytics System may be slower, so use async. Will not hold up thread.

Relevant packages: com.linkedin.events.customer, com.linkedin.events.analytics

Flow: Customer (DAO) -> CustomerRegisteredEvent (the event) -> AnalyticsCustomerRegisteredListener (async EventListener) -> AnalyticsService (implementation)





Situation 3: Conditional event handling (filtering for specific events)

Run promotion for new customers who have signed up for newsletter

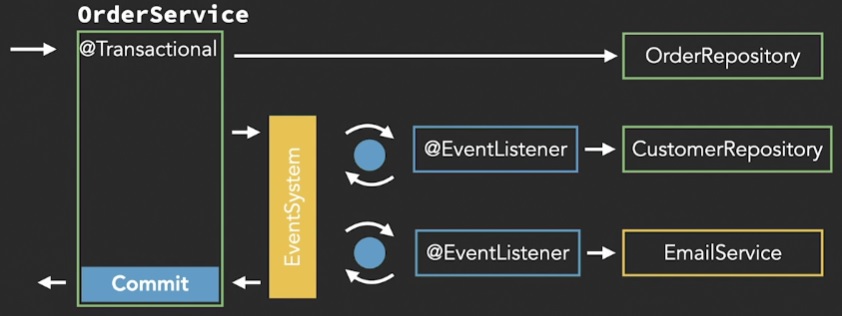
Relevant package: com.linkedin.events.promotion, com.linkedin.events.customer

Flow: Customer (DAO, true for newsletter) -> CustomerRegisteredEvent (event) -> PromotionListener (has condition/filter for newsletter is true) -> PromotionService (implementation)

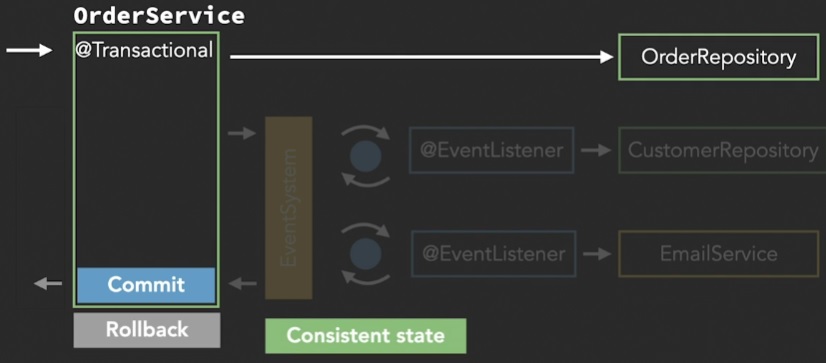
Situation 4: Transactional event listener

If we annotate with @Transactional, all steps are completed or rolled back.

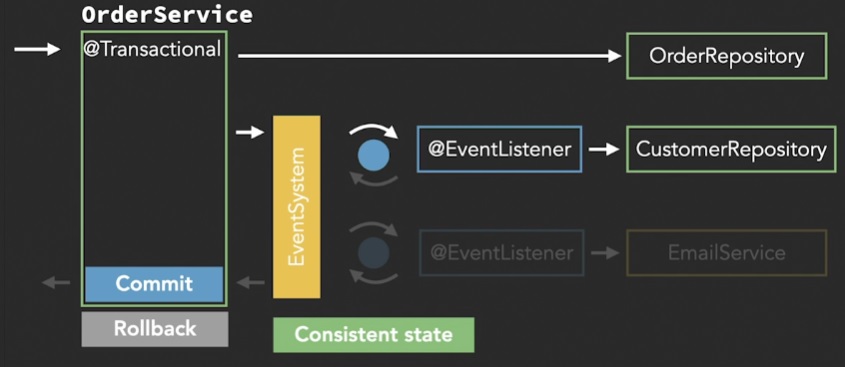
Order placed -> OrderService placeOrder method has @Transactional annotation. Order completed -> order status in orderRepository updated and saved. Completed order published as event to application context. Event listeners triggered, update reward points in customerRepository, send order email to customer. Consistent with desired outcome for all steps.



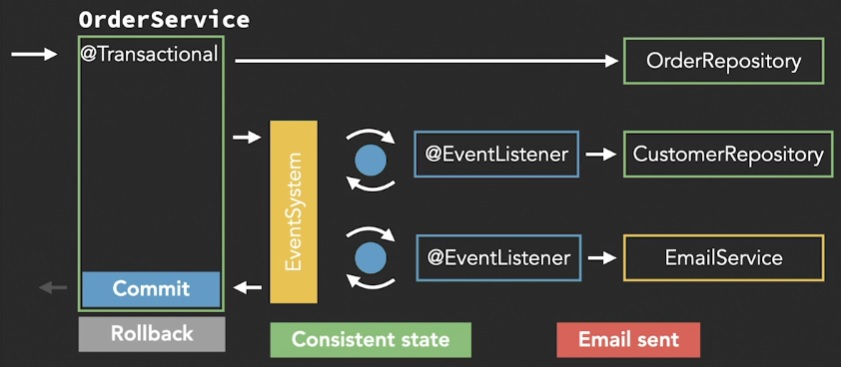
Exception situations:



Error before event listeners triggered. Rollback transaction, consistent state. OK.



Error after 1st event listener invocation. Roll back commits to OrderRepository and CustomerRepository. No email sent. Consistent state. OK.



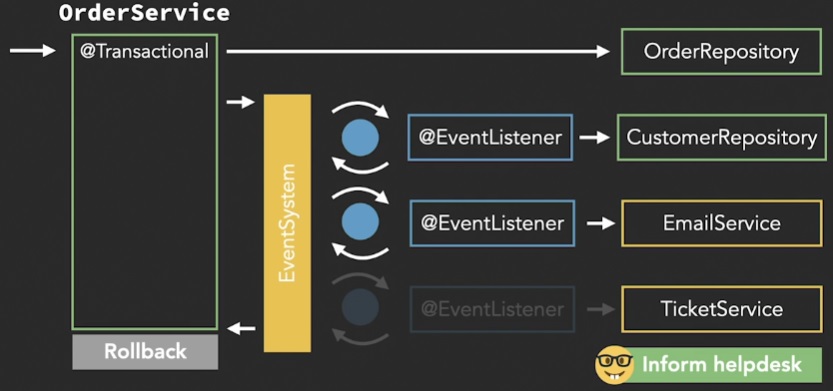
Error after implementation of all event listeners. Can roll back all steps. Consistent state in database. However, email sent to customer. Not OK.

If we annotate with @TransactionalEventListener(phase = TransactionPhase.NAME\_OF\_PHASE), method invoked only after named phase (e.g. AFTER\_COMMIT).

Relevant package: com.linkedin.events.orders, com.linkedin.events.promotion

Flow: Order (POJO) -> OrderCompletedEvent (event) -> OrderService (ApplicationEventPublisher) -> EmailListener -> EmailService (implementation)

Challenge 4: To inform helpdesk if rollback. Need @TransactionalEventListener for ticketService.



**JPEG pictures**

|  |  |
| --- | --- |
| registerCustomer\_forNewCustomer\_sendsAnEmail Test | Test result for synchronous event processing. Object -> event -> publish Event -> EventListener -> implementation |
| removeCustomer\_forExistingCustomer Test | Same as above. Instead of register new Customer, this is to remove Customer. |
| AnalyticsService Sync EventListener 5s simulated delay in event processing | Test result with simulated 5s processing delay. No async annotation. Sync processing. |
| AnalyticsService Async EventListener | Test result for with/without @Async. Processing time reduced to 204ms. |
| Conditional Transactional EventListener Test | Test result for @EventListener(condition = “”). If newsletter==TRUE for customer, promotion event listener sends promotion email. |
| EmailOrderServiceTest send email or rollback transaction | Test result for @Transactional. All-or-none. Rollback with exception (3-digit rewardPoints), or successful completion (2-digit rewardPoints). |
| forPlaceOrder\_whenRolledBacked\_createTicket Test TransactionalEventListener for after\_rollback | Test result for @TransactionalEventListener (phase = TransactionPhase.AFTER\_ROLLBACK). Invoke TicketService only after rollback. |

**References**

Spring documentation

1.15.2 Standard and Custom Events (standard Observer design pattern)

https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#context-functionality-events

Project Lombok

https://projectlombok.org/

-@Data

https://projectlombok.org/features/Data

- @Log

https://projectlombok.org/features/log

- @NoArgsConstructor, @RequiredArgsConstructor

https://projectlombok.org/features/constructor

- @SneakyThrows

https://projectlombok.org/features/SneakyThrows

Configure Log4j for logging

https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto.logging.log4j