A Deep Dive into Spring Application Events

Oliver Drotbohm /odrotbohm odrotbohm@vmware.com

FUNDAMENTALS

Programming model

TRANSACTIONS

Consistency & error scenarios

ARCHITECTURE

Components & relationships

FUNDAMENTALS

Programming model

Programming model

`ApplicationEventPublisher`

`ApplicationListener<T>`

`@EventListener`

`@TransactionalEventListener`

DEMO

Application events fundamentals

Application Events with Spring Data

Application events with Spring (Data)

- **Powerful mechanism to publish events in Spring applications**
  - Application event – either a general object or extending `ApplicationEvent`
  - `ApplicationEventPublisher` – injectable to manually invoke event publication

- **Spring Data event support**
  - Spring Data's focus: aggregates and repositories
  - Domain-Driven Design aggregates produce application events
  - `AbstractAggregateRoot<T>` – base class to easily capture events and get them published on `CrudRepository.save(…)` invocations.
  - No dependency to infrastructure APIs

```java
class Order {

  private List<Object> events;

  @DomainEvents
  Collection<Object> events() { … }

  @AfterDomainEventPublication
  void reset() { … }
}
```

Aggregate accumulates event instances

Method to expose them

Wipe events after publication

```
class Order extends AbstractAggregateRoot<Order> {

  Order complete() {
    registerEvent(OrderCompleted.of(this));
    return this;
  }
}
```

Method to register event.
Super class takes care of exposure.

Spring Data provided base class

```
@Component
class OrderManagement {

  private final OrderRepository orders;

  @Transactional
  void completeOrder(Order order) {
    orders.save(order.complete());
  }
}
```

No references to Spring infrastructure APIs

Intermediate summary

Events for Bounded Context interaction

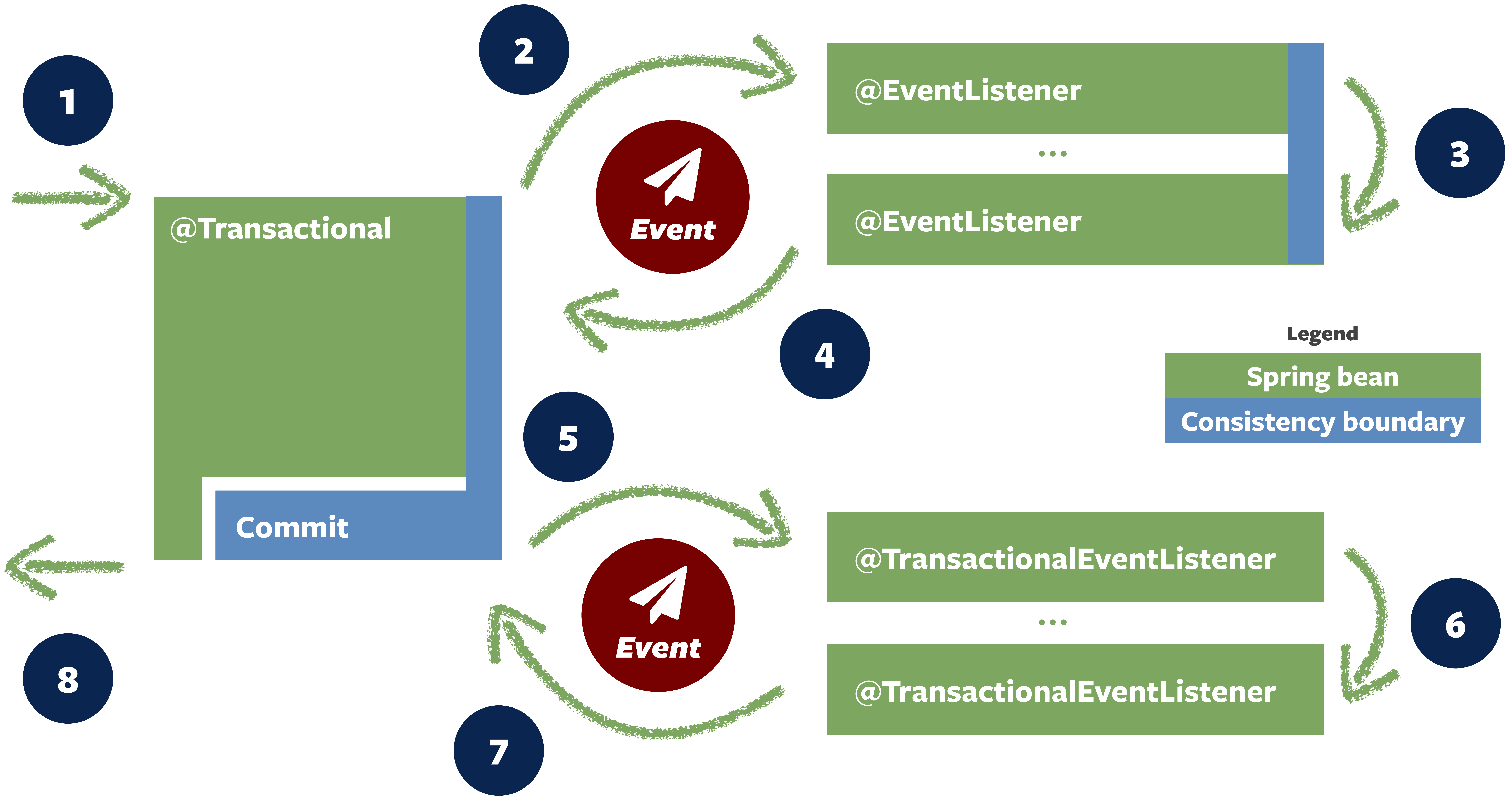
Spring's application events are a very light-weight way to implement those domain events. Spring Data helps to easily expose them from aggregate roots. The overall pattern allows loosely coupled interaction between Bounded Contexts so that the system can be extended and evolved easily.

Externalize events if needed

Depending on the integration mechanism that's been selected we can now write separate components to translate those JVM internal events into the technology of choice (JMS, AMQP, Kafka) to notify third-party systems.

TRANSACTIONS

Consistency & error scenarios


1
2
3
4
5
6
7
8
@Transactional
Commit
Event
@EventListener
...
@EventListener
Event
@TransactionalEventListener
...
@TransactionalEventListener
Legend
Spring bean
Consistency boundary

Application events in a Spring application

1. We enter a transactional method

Business code is executed and might trigger state changes on aggregates.

2. That transactional method produces application events

In case the business code produces application events, standard events are published directly. For each transactional event listener registered a transaction synchronization is registered, so that the event will eventually be published on transaction completion (by default on transaction commit).

3. Event listeners are triggered

By default, event listeners are synchronously invoked, which means they participate in the currently running transactions. This allows listeners to abort the overall transaction and ensure strong consistency. Alternatively, listeners can be executed asynchronously using `@Async`. They then have to take care of their transactional semantics themselves and errors will not break the original transaction.

Application events in a Spring application

4. Service execution proceeds once event delivery is completed

Once all standard event listeners have been invoked, the business logic is executed further. More events can be published, further state changes can be created.

5. The transaction finishes

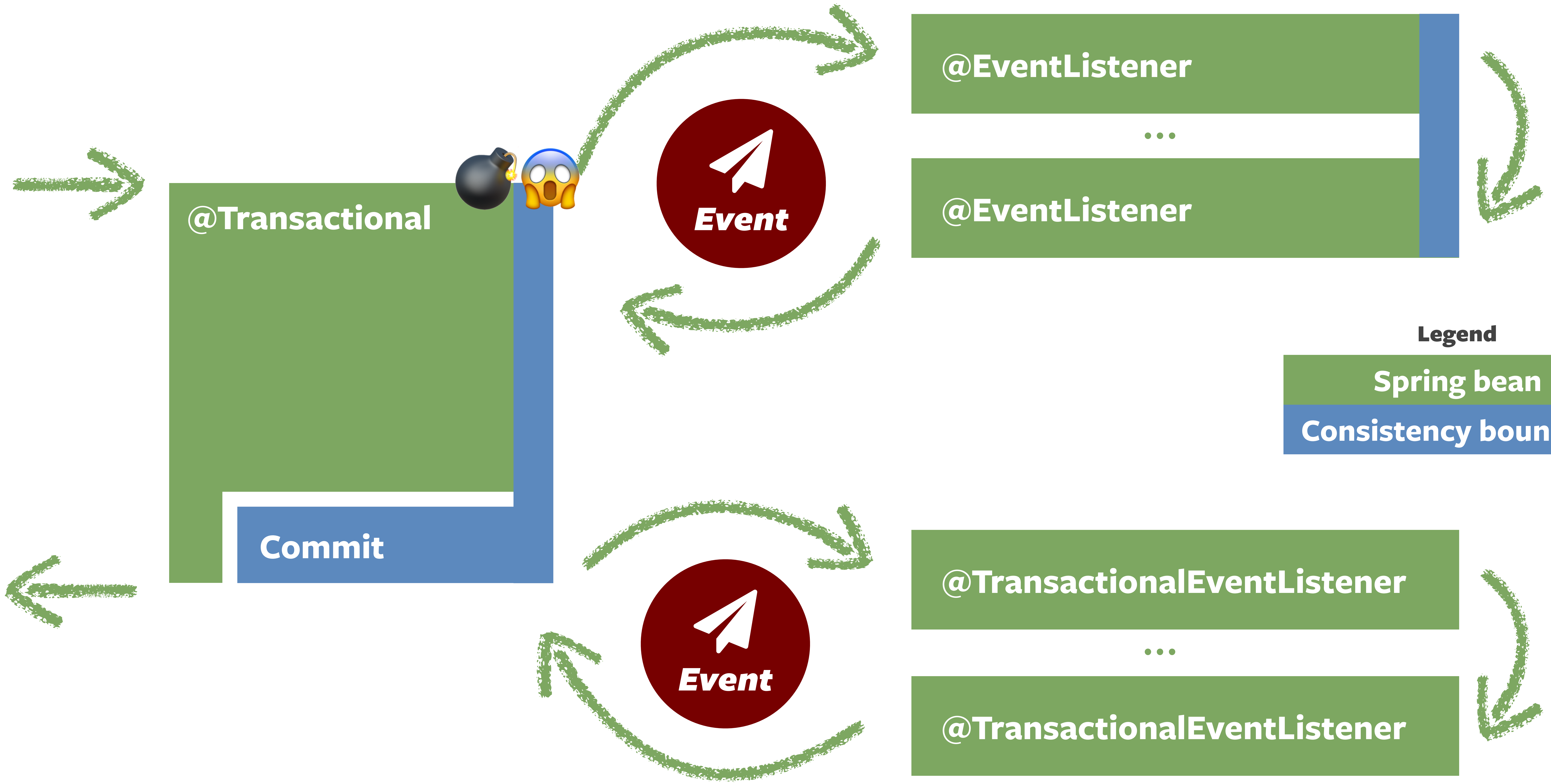
Once the transactional method is done, the transaction is completed. Usually all pending changes (created by the main business code or the synchronous event listeners) are written to the database. In case inconsistencies or connection problems, the transaction rolls back.

6. Transactional event listeners are triggered

Listeners annotated with `@TransactionalEventListener` are triggered when the transaction commits, which means they can rely on the business operation the event has been triggered from having succeeded. This allows the listeners to read committed data. Listeners can be invoked asynchronously using `@Async` in case the functionality to be invoked might be long-running (e.g. sending an email).

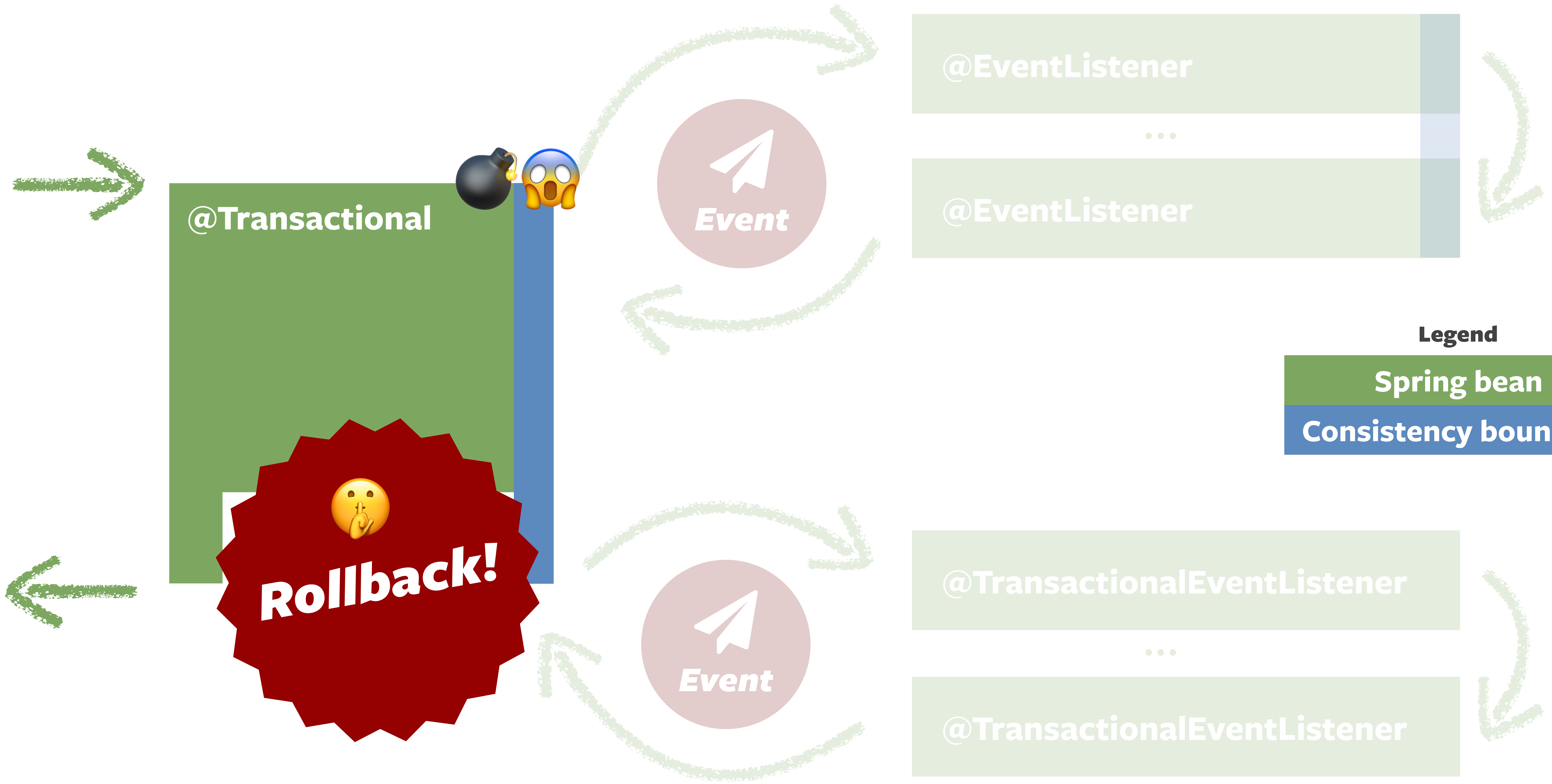
Error scenarios

What if the service fails? 🤔



Legend

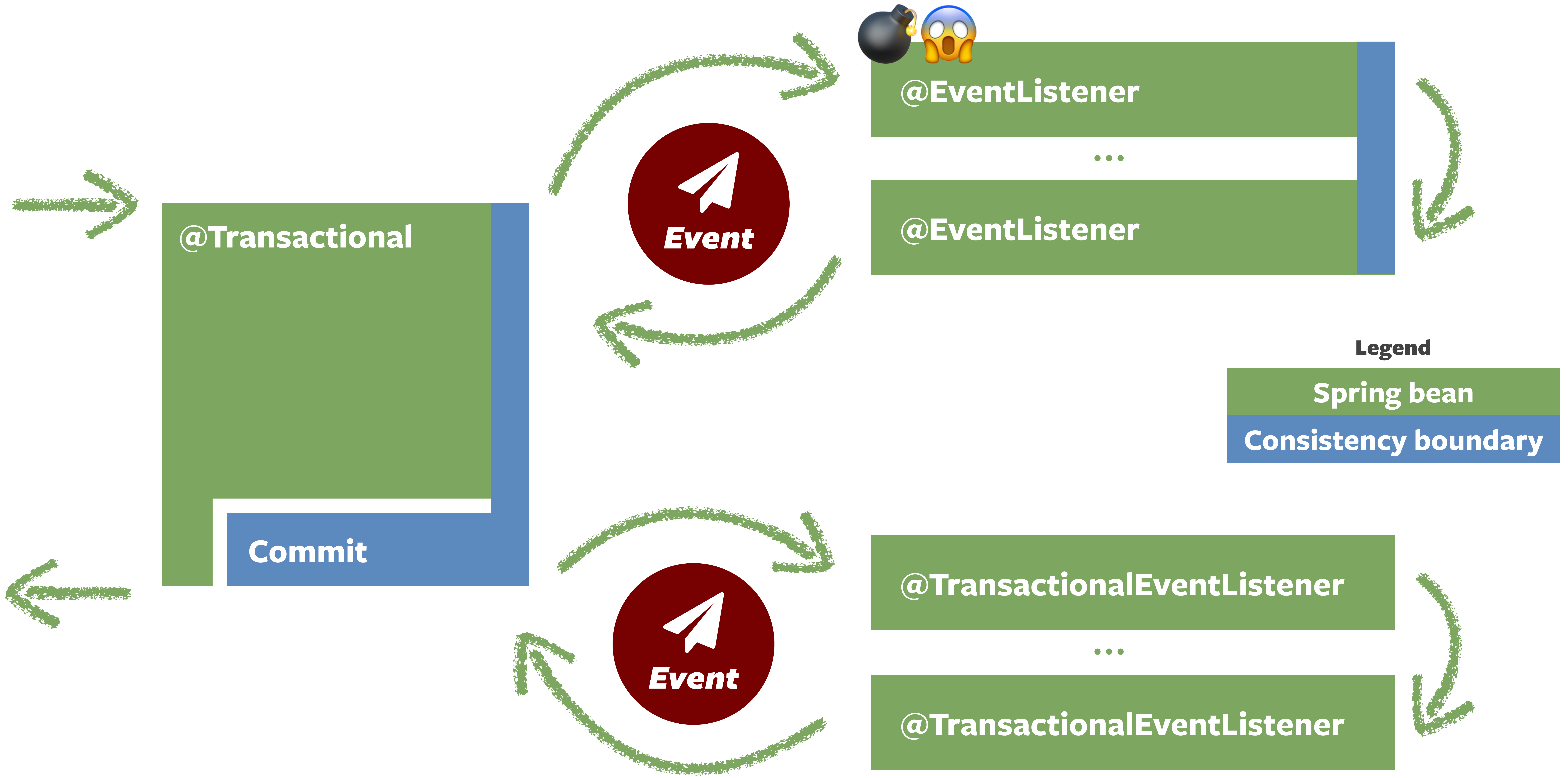
Spring bean
Consistency boundary

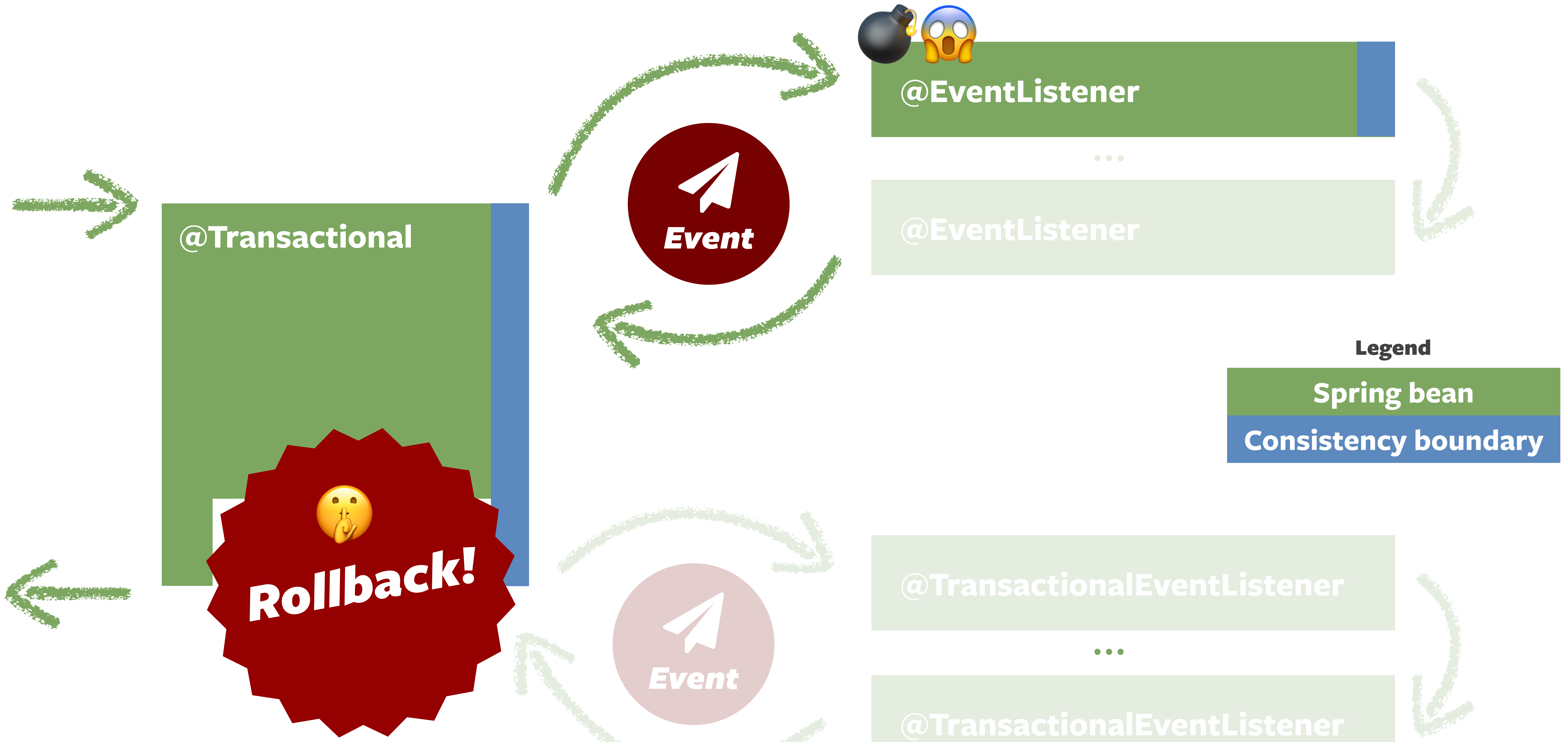


Legend

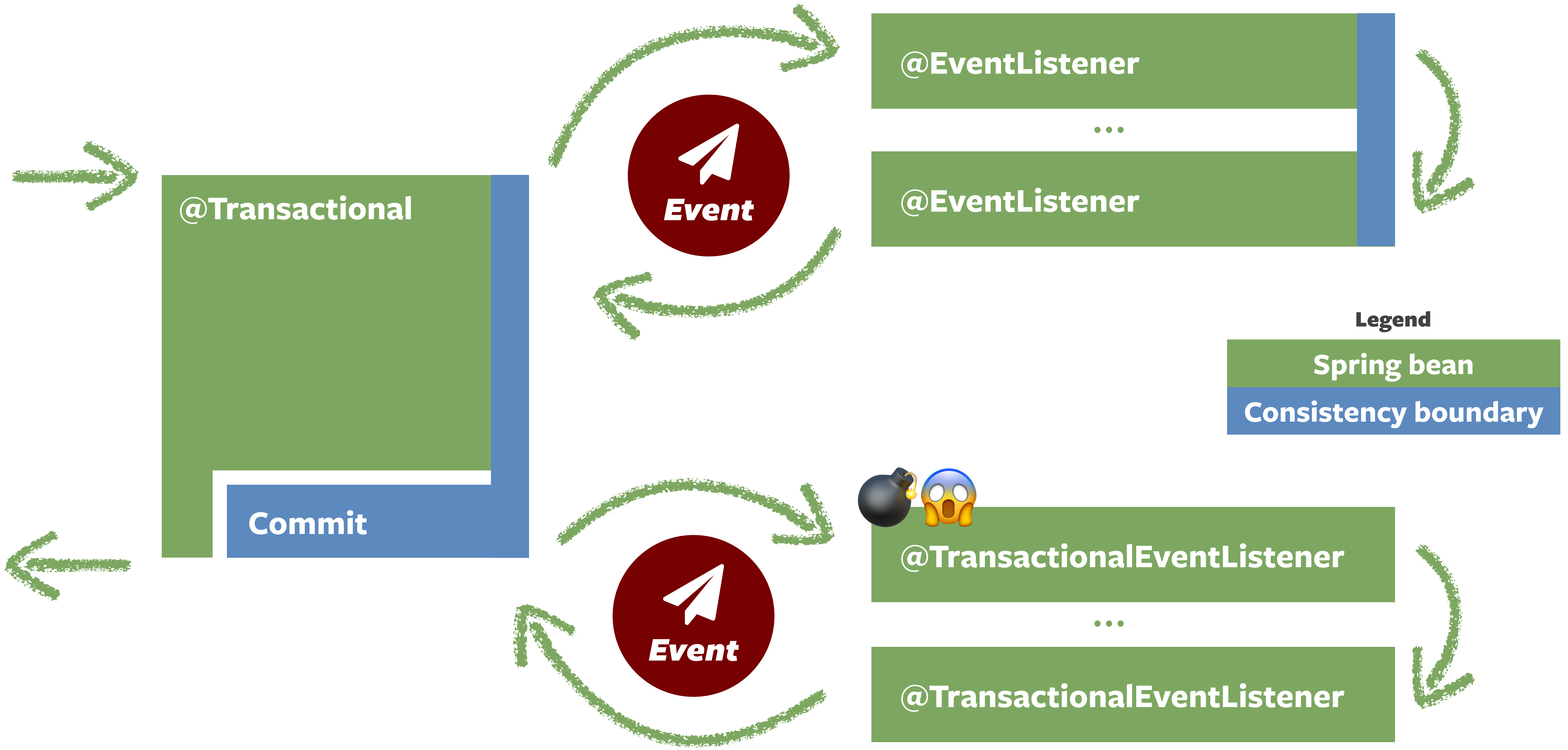
Spring bean
Consistency boundary

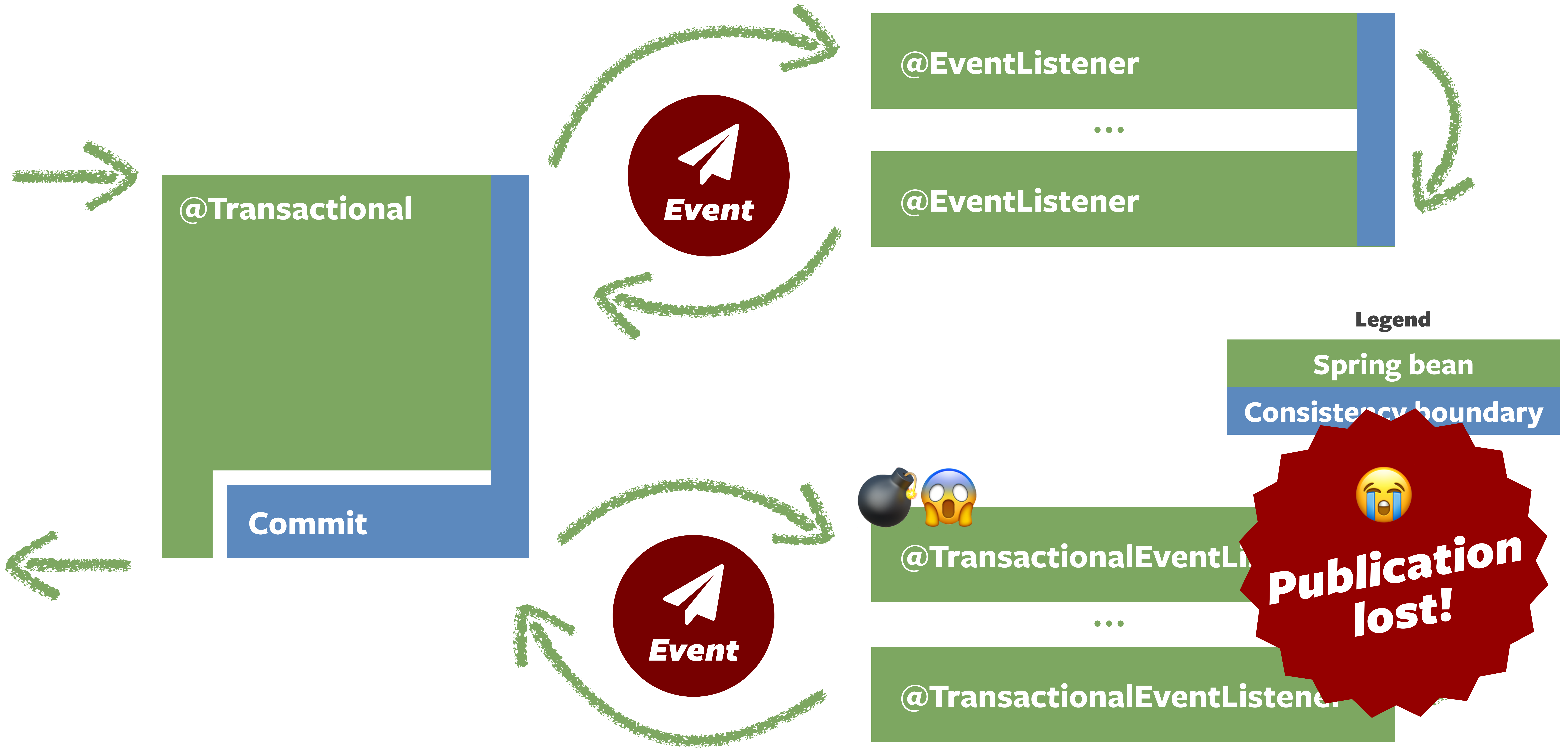
What if an event listener fails? 🤔


@Transactional
Commit
Event
@EventListener
...
@EventListener
Event
@TransactionalEventListener
...
@TransactionalEventListener
Legend
Spring bean
Consistency boundary


@Transactional
Event
@EventListener
...
@EventListener
Rollback!
Event
@TransactionalEventListener
...
@TransactionalEventListener
Legend
Spring bean
Consistency boundary

What if a transactional event listener fails? 🤔


@Transactional
Commit
Event
@EventListener
...
@EventListener
Event
@TransactionalEventListener
...
@TransactionalEventListener
Legend
Spring bean
Consistency boundary


@Transactional
Commit
Event
@EventListener
…
@EventListener
Event
@TransactionalEventLi
…
@TransactionalEventListene
Legend
Spring bean
Consistency boundary
Publication lost!

DEMO

Transactional application events

Application events – Error Scenarios

The service fails

Only standard event listeners up until the failure will have been executed. Assuming the already triggered event listeners also execute transactional logic, the local transaction is rolled back and the system is still in a strongly consistent state. Transactional event listeners are not invoked in the first place.

A synchronous event listener fails

In case a normal event listener fails the entire transaction will roll back. This enables strong consistency between the event producer and the listeners registered but also bears the risk of supporting functionality interfering with the primary one, causing the latter to fail for less important reasons. The tradeoff here could be to move to a transactional event listener and embrace eventual consistency.

Application events – Error Scenarios

A transactional event listener fails

In case a transactional event lister fails or the application crashes while transactional event listeners are executed, the event is lost and functionality might not have been invoked. Other transactional event listeners will still be triggered.

BONUS: An asynchronous event listener fails

The event is lost but the primary functionality can still succeed as the event is handled in a separate thread. Retry mechanisms can (should?) be deployed in case some form of recovery is needed.

Event Publication Registry

@TransactionalEventListener

...

@TransactionalEventListener

...

@TransactionalEventListener

...

@TransactionalEventListener

@TransactionalEventListener

...

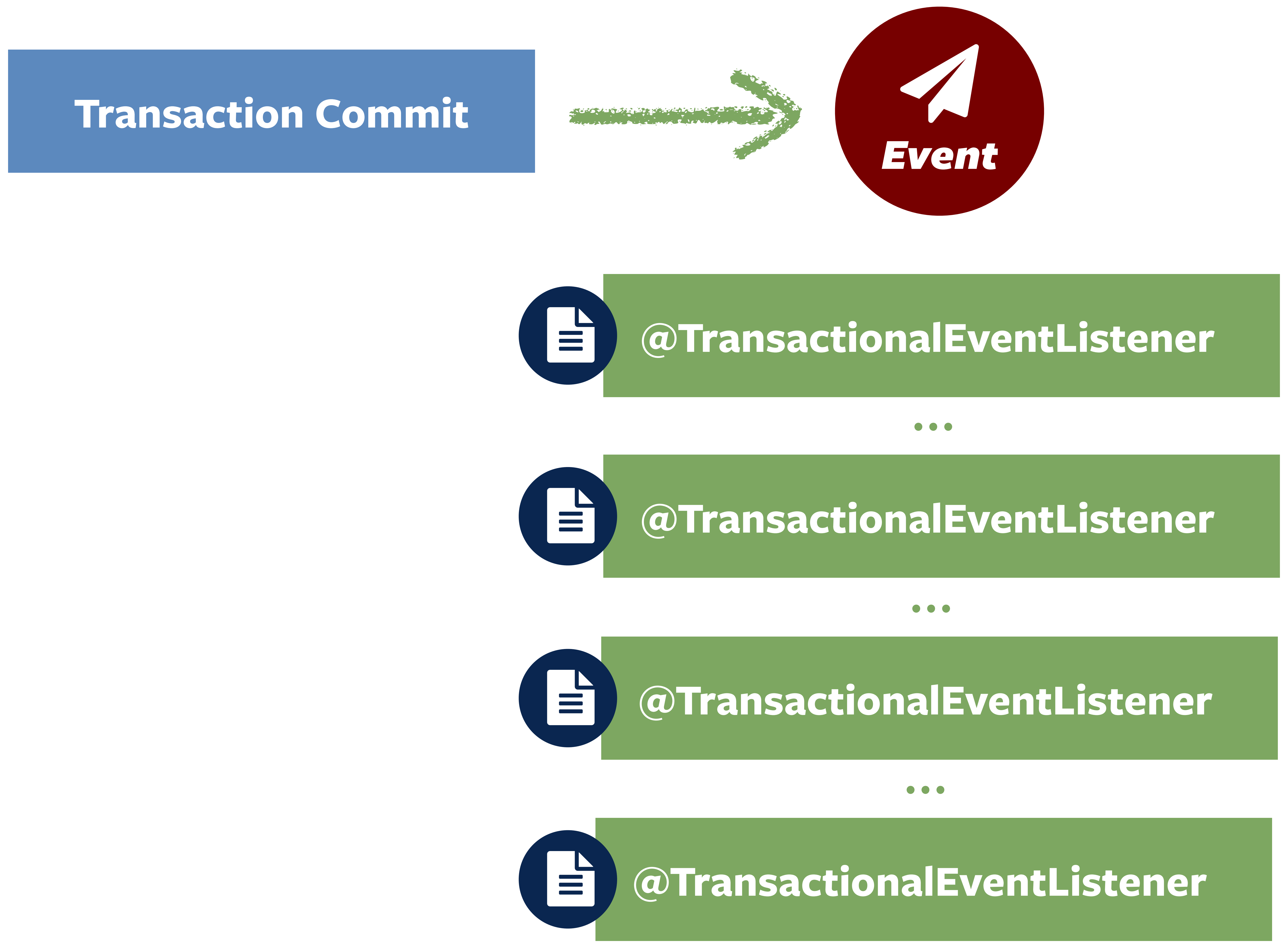
@TransactionalEventListener

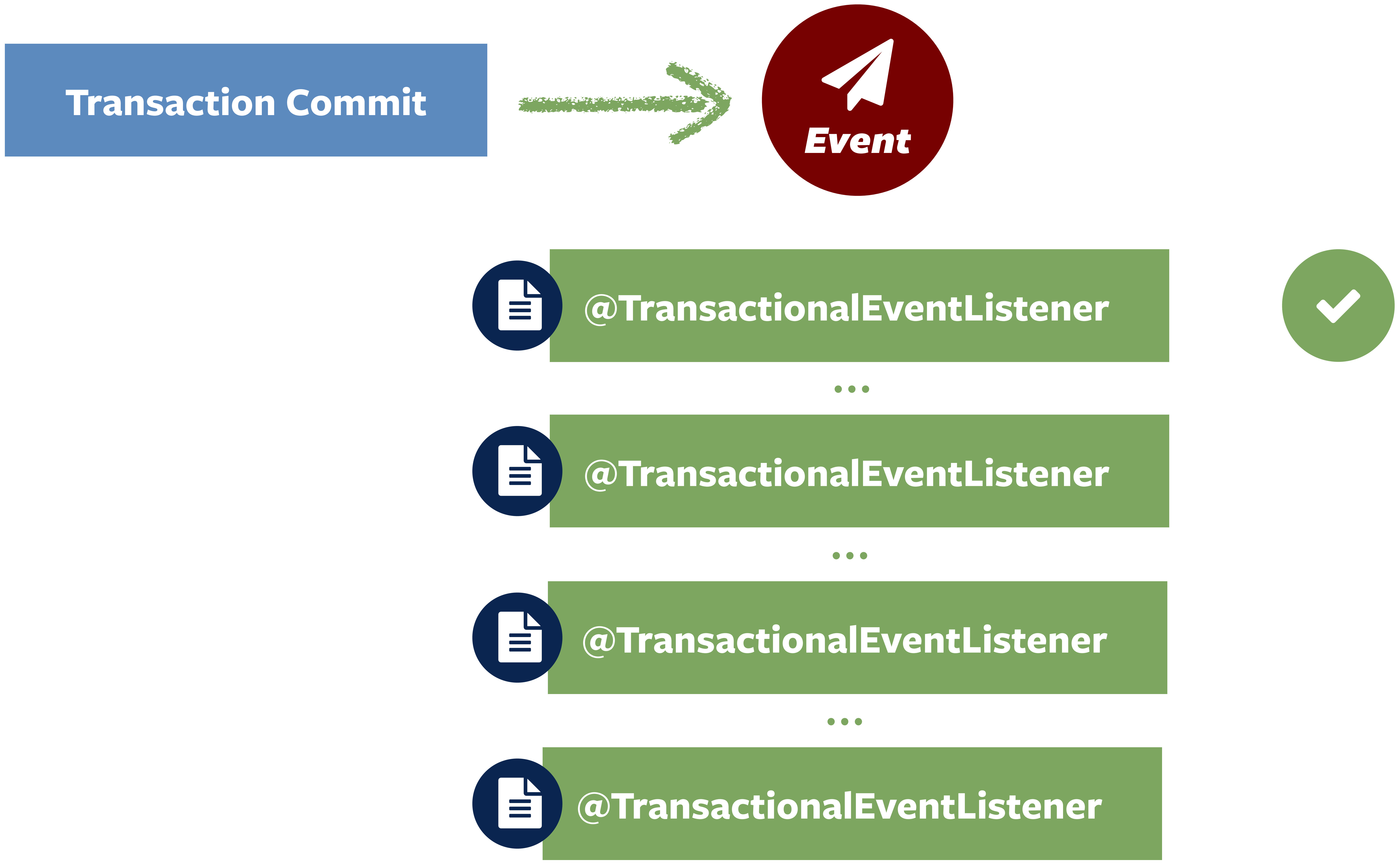
...

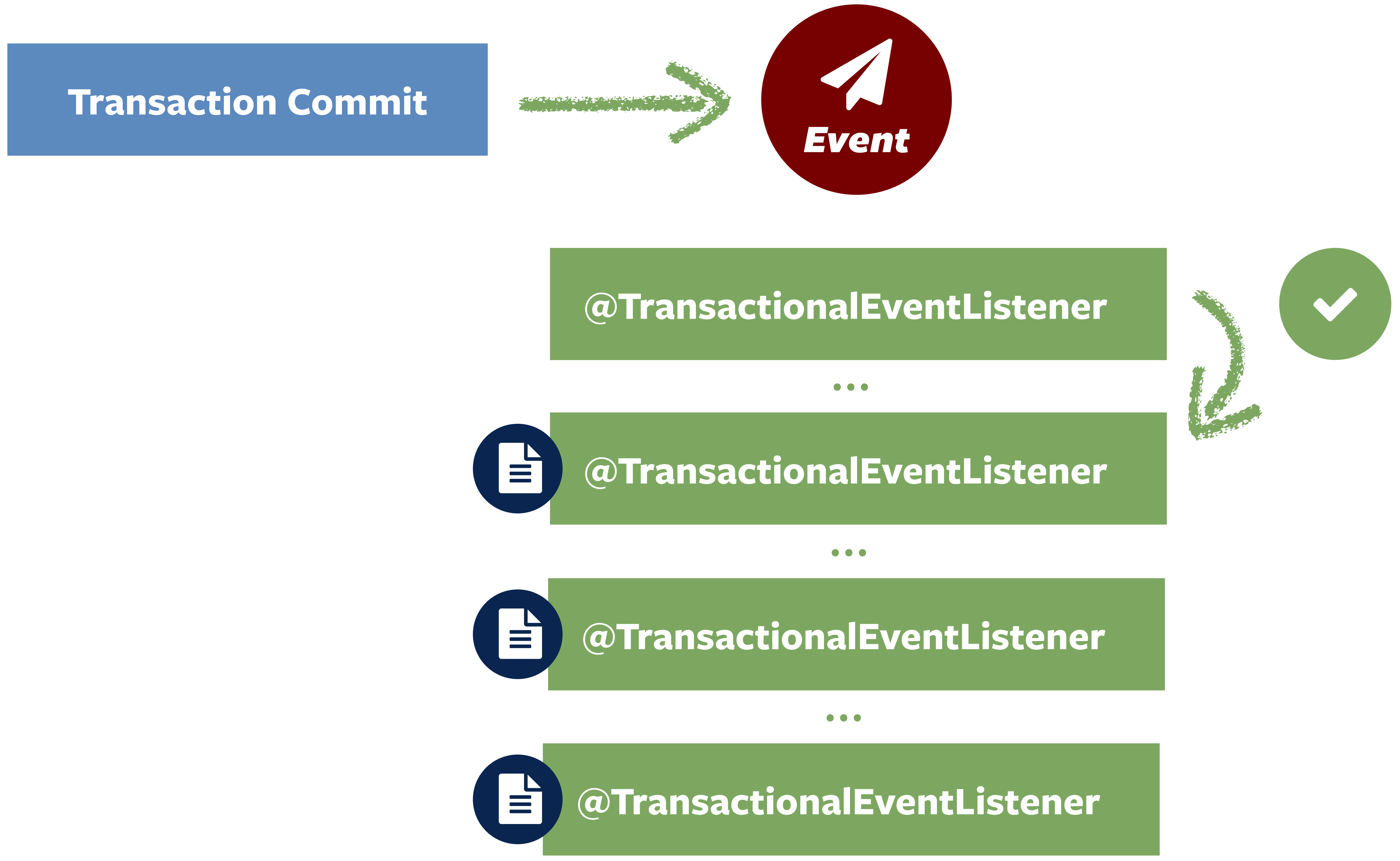
@TransactionalEventListener

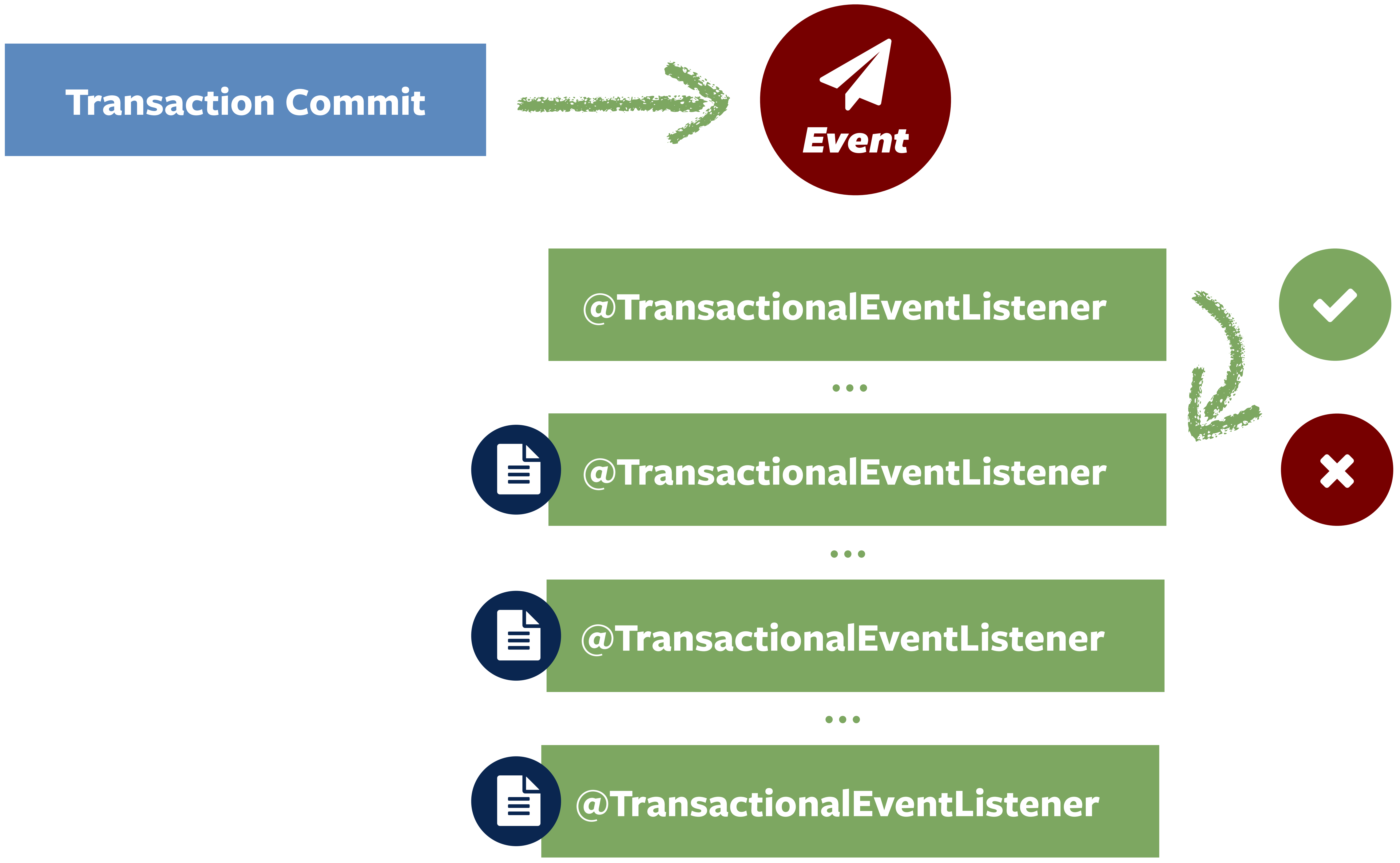
...

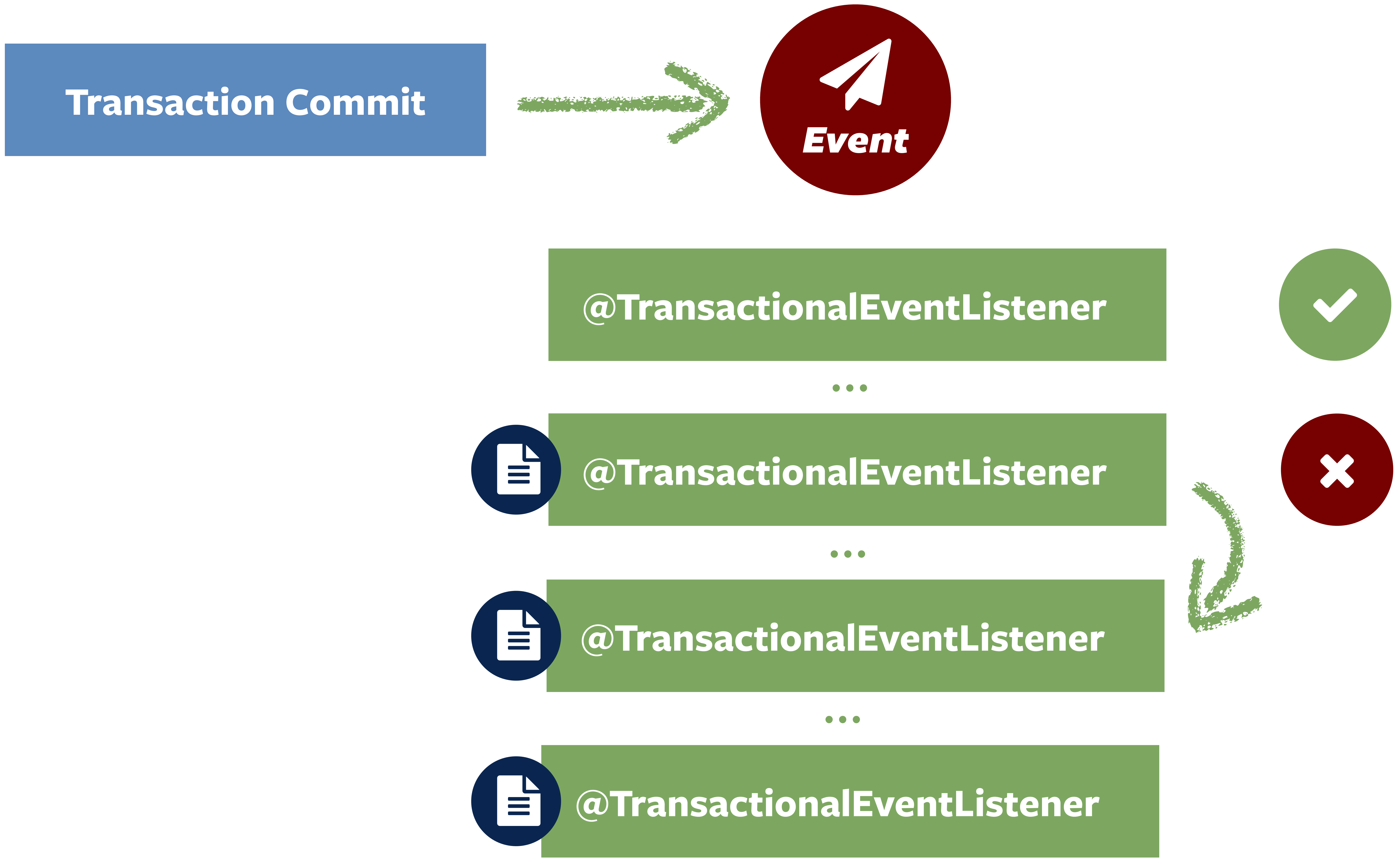
@TransactionalEventListener

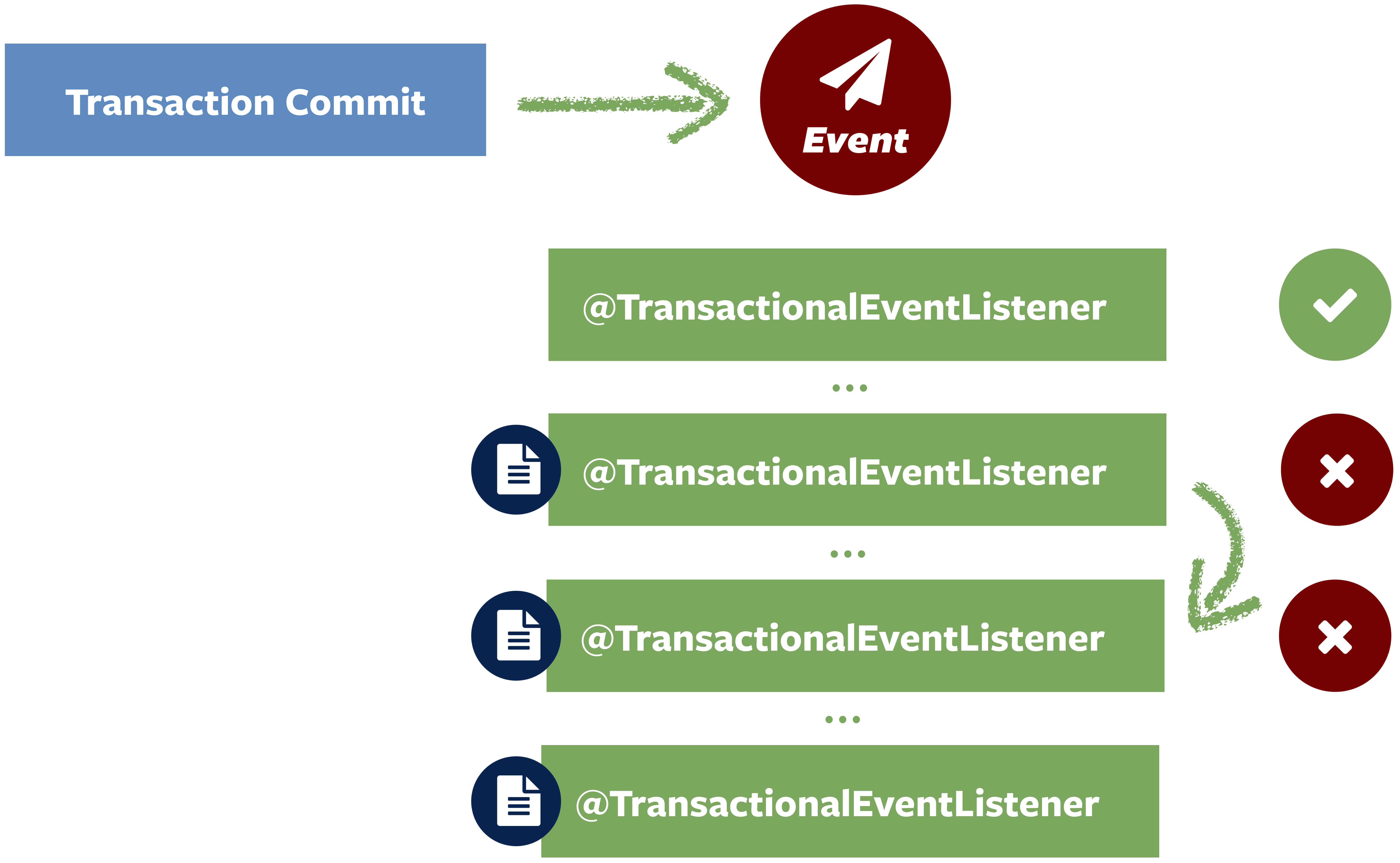

Transaction Commit
Event
@TransactionalEventListener
…
@TransactionalEventListener
…
@TransactionalEventListener
…
@TransactionalEventListener

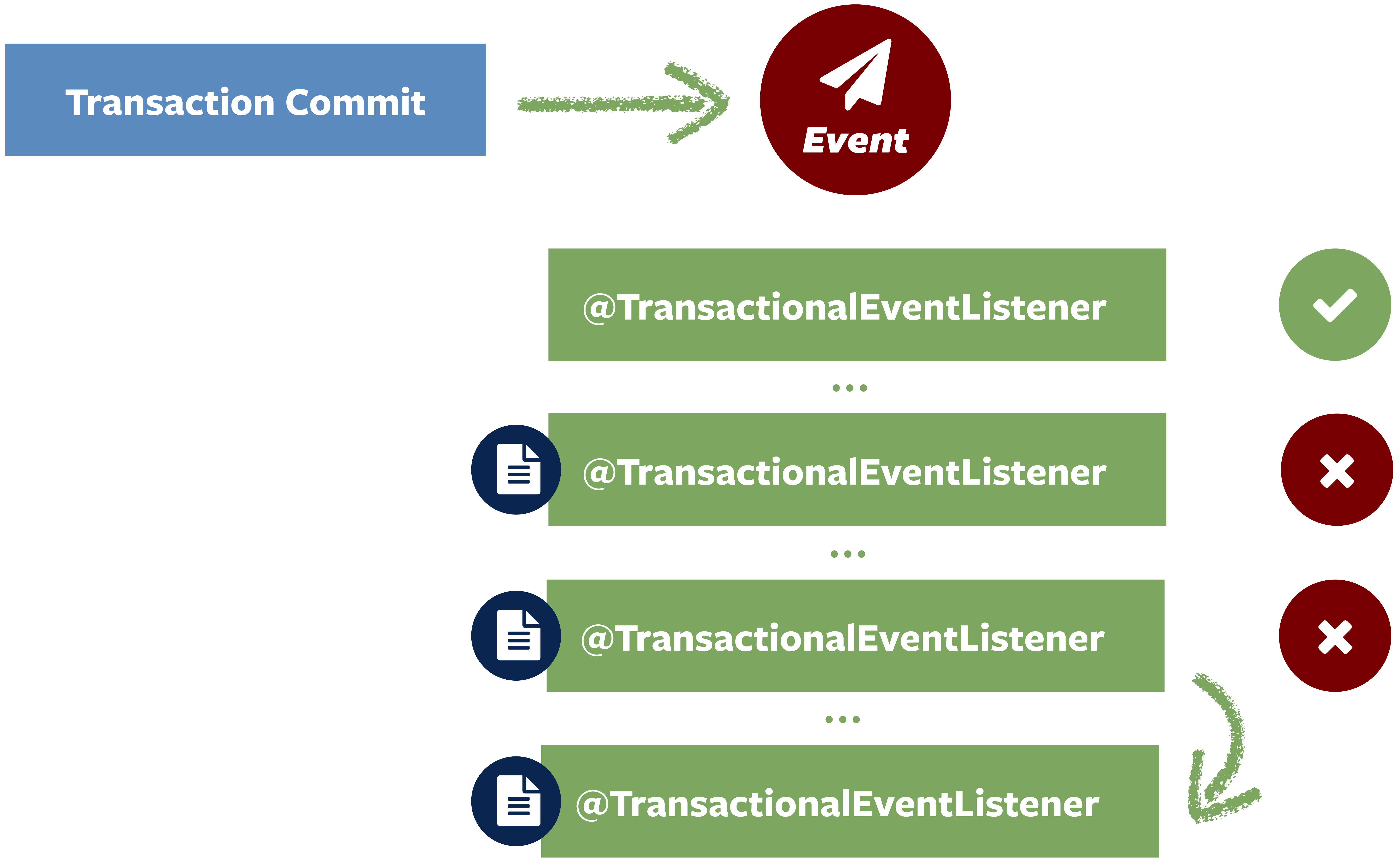

Transaction Commit
Event
@TransactionalEventListener
...
@TransactionalEventListener
...
@TransactionalEventListener
...
@TransactionalEventListener

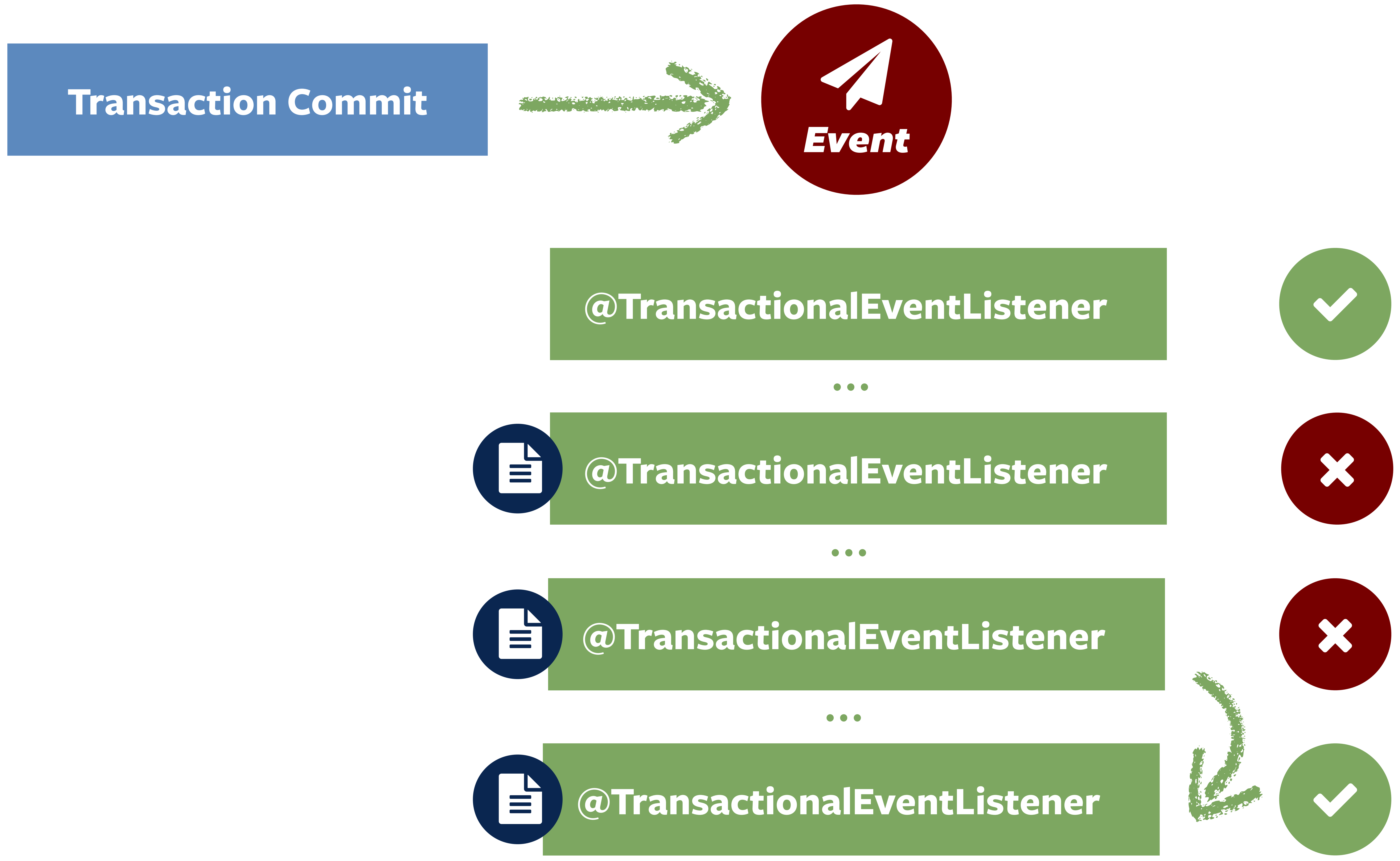

Transaction Commit
Event
@TransactionalEventListener
...
@TransactionalEventListener
...
@TransactionalEventListener
...
@TransactionalEventListener

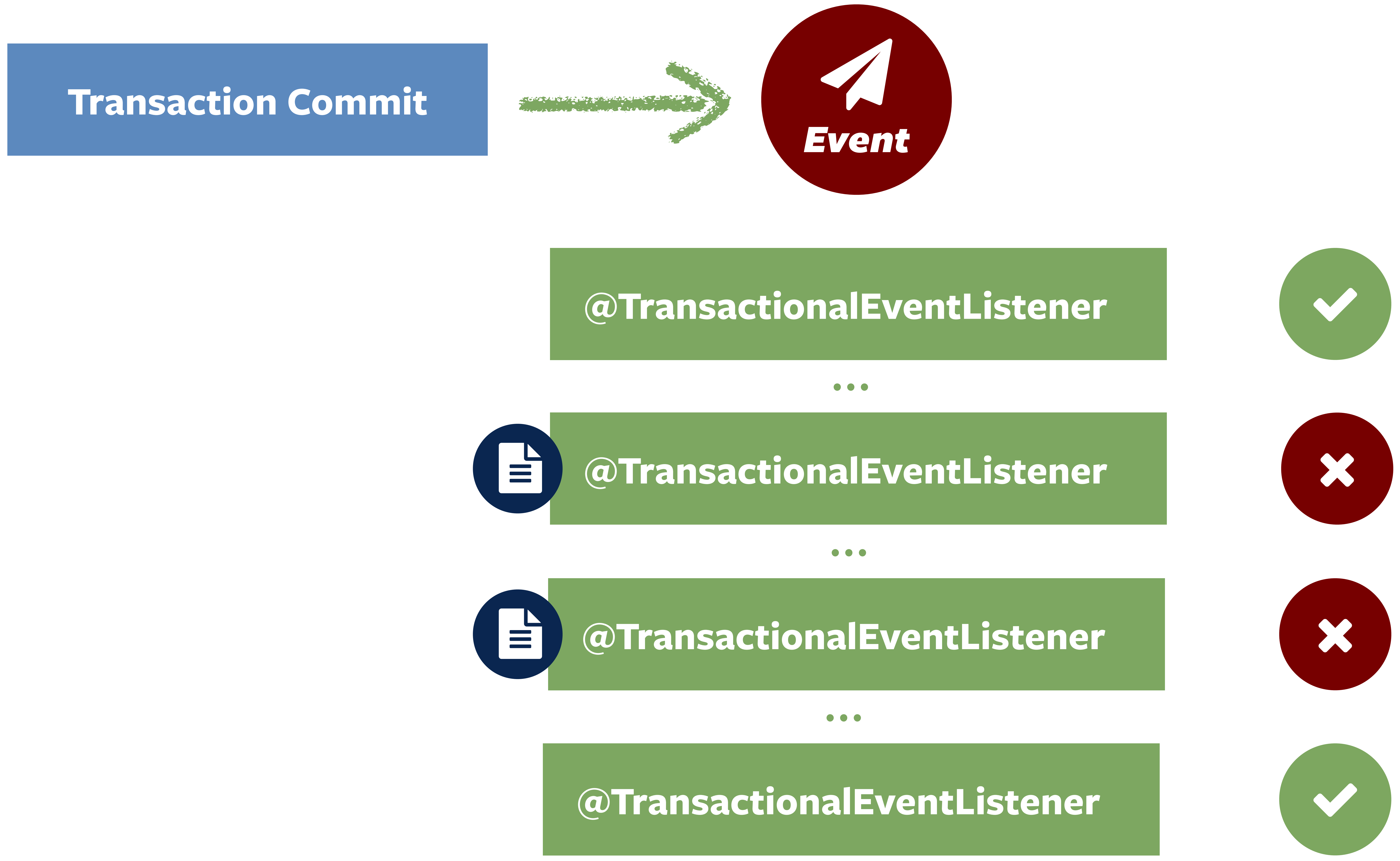

Transaction Commit
Event
@TransactionalEventListener
...
@TransactionalEventListener
...
@TransactionalEventListener
...
@TransactionalEventListener


Transaction Commit
Event
@TransactionalEventListener
...
@TransactionalEventListener
...
@TransactionalEventListener
...
@TransactionalEventListener


Transaction Commit
Event
@TransactionalEventListener
...
@TransactionalEventListener
...
@TransactionalEventListener
...
@TransactionalEventListener


Transaction Commit
Event
@TransactionalEventListener
...
@TransactionalEventListener
...
@TransactionalEventListener
...
@TransactionalEventListener


Transaction Commit
Event
@TransactionalEventListener
...
@TransactionalEventListener
...
@TransactionalEventListener
...
@TransactionalEventListener


Transaction Commit
Event
@TransactionalEventListener
✓
…
@TransactionalEventListener
✗
…
@TransactionalEventListener
✗
…
@TransactionalEventListener
✓

Event Publication Registry

1. Write application event publication log for transactional listeners

On application event publication a log entry is written for every event and transactional event listener interested in it. That way, the transaction remembers which events have to be properly handled and in case listener invocations fail or the application crashes events can be re-published.

2. Transaction listeners are decorated to register successful completion

Transactional event listeners are decorated with an interceptor that marks the log entry for the listener invocation on successful listener completion. When all listeners were handled, the log only contains publication logs for the ones that failed.

3. Incomplete publications can be retried

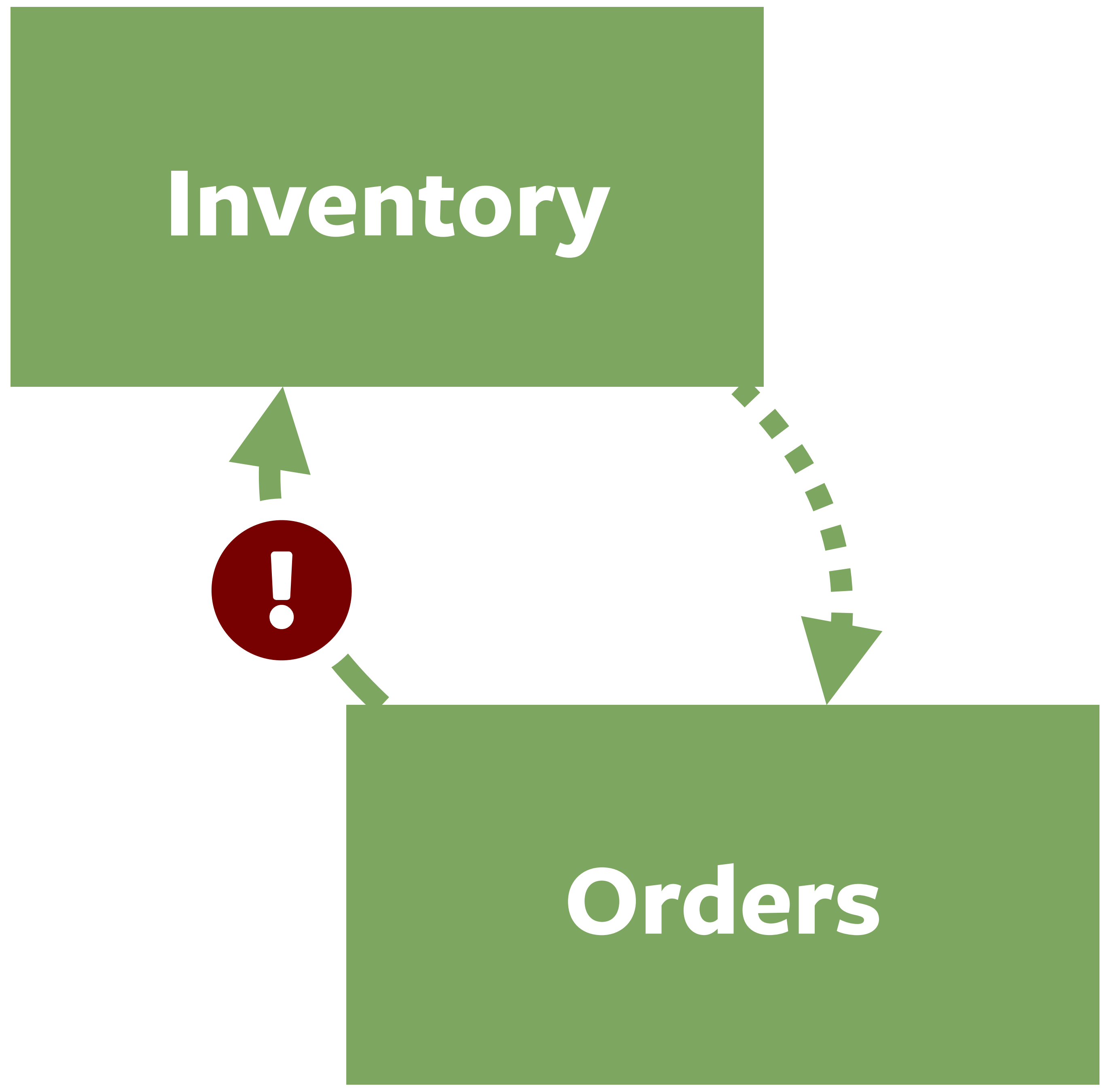
Either periodically or during application restarts.

DEMO

Event publication registry

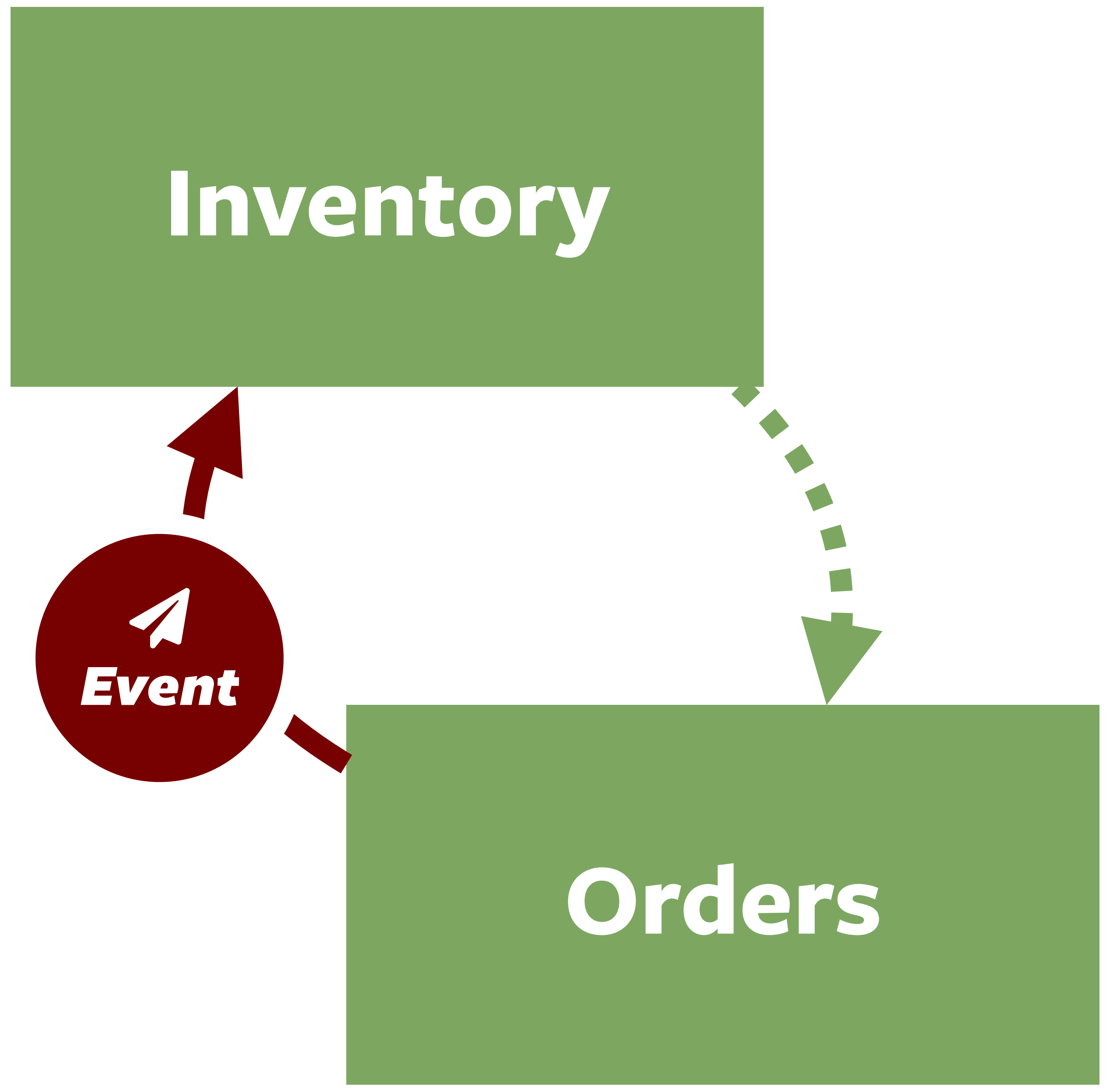
ARCHITECTURE

Components & relationships



Component Dependency

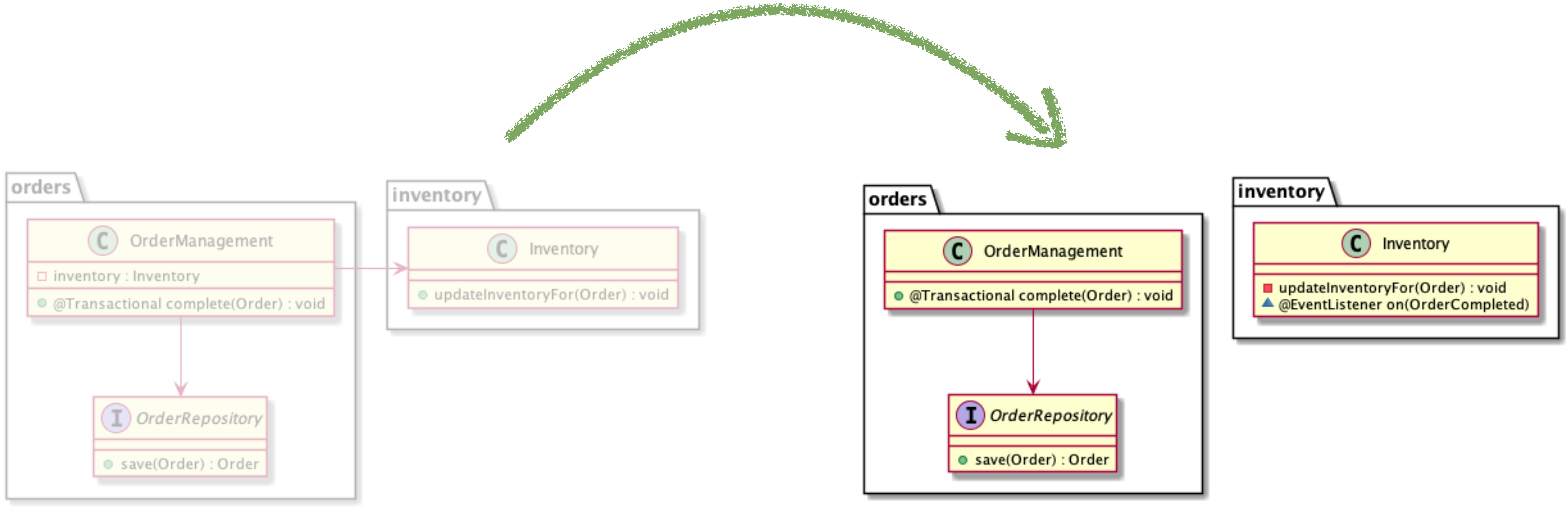
Type dependency



Component Dependency

Type dependency

Event publication


orders
OrderManagement
inventory : Inventory
@Transactional complete(Order) : void
OrderRepository
save(Order) : Order
inventory
Inventory
updateInventoryFor(Order) : void
orders
OrderManagement
@Transactional complete(Order) : void
OrderRepository
save(Order) : Order
inventory
Inventory
updateInventoryFor(Order) : void
@EventListener on(OrderCompleted)

Imperative style

1. Spring bean invocations across Bounded Contexts

The `OrderManagement` actively invokes other Spring Beans to trigger peripheral functionality. By that, it becomes a spot of gravity that is likely to become more complex over time and pollute the transaction (i.e. it's doing too much). There's no explicit representation of the business incident "completion of an order" except the method name.

2. Testing focussed on explicit interaction

Unit testing usually works with mocks and requires complex setup of expected behavior and verifications. For integration tests, dependency beans need to be available, too, which makes it hard to test the module in isolation.

3. Post-commit functionality hard to integrate

As the entire method is running inside a transaction, it becomes harder to integrate logic that is supposed to run *after* the transaction has committed, for example sending a confirmation email. Especially interactions with external resources that can take a while should be held outside the main transaction.

Event based implementation

1. Less knowledge in the order module and explicit integration point

Introducing the `OrderCompleted` event allows the order module to get rid of the knowledge about which peripheral functionality to integrate and thus allows for easier extension. Whether that downstream functionality is executed synchronously as part of the main transaction, asynchronously or after the transaction is now determined by the event listener.

2. Testing shifts to verification of events published

The reduced number of dependencies of a component significantly eases testing as these dependencies don't have to be mocked and prepared (in the case of a unit test) nor do they have to be available as Spring bean in an integration test. In the former case, Modulith's `PublishedEvents` API allows to write readable assertions on the events that were produced during the test execution. The latter enables Modulith's support for `@ModuleTest` to integration test a single module.

Resources

- **Sample code**

  Project website on GitHub

- **Moduliths**

  Spring Boot extension to support building modular monoliths, incl. support for events, testing etc., Project website on GitHub

- **Spring Domain Events**

  Prototype implementation of an event publication registry, likely to be integrated into Moduliths, Project website on GitHub

- **Refactoring to a System of Systems**

  Video on YouTube

Thank you!

Oliver Drotbohm /odrotbohm odrotbohm@vmware.com