Events

In this lesson, we'll learn all about events.

We'll cover the following

- Introduction
- Events and DDD
- Patterns from strategic design
  - Specific events
  - Published language
- Sending minimal data in an event

Introduction

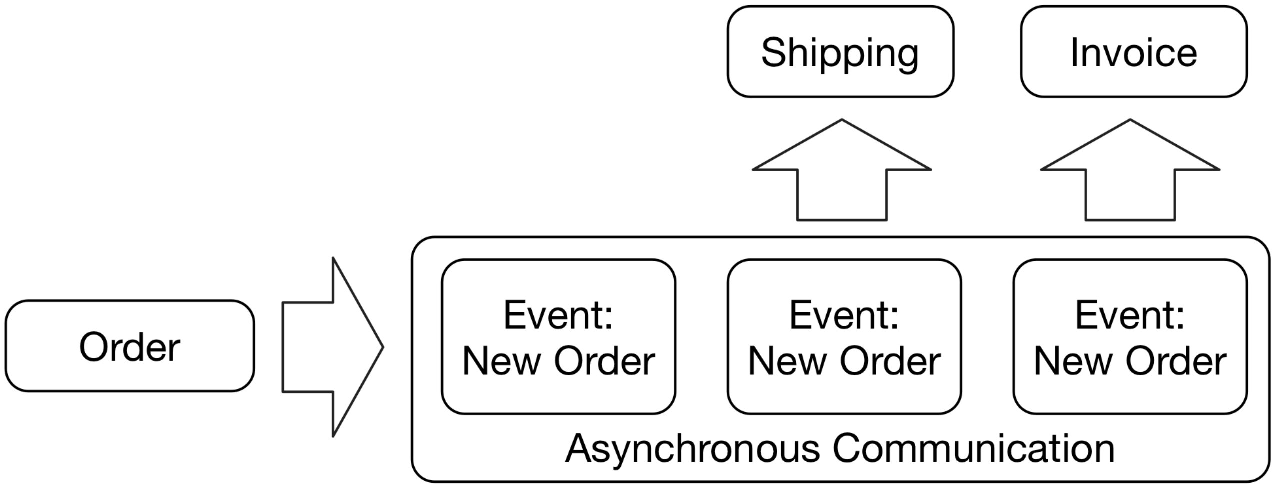
With asynchronous communication, the coupling of systems can be driven to different lengths. As already mentioned:

- The system for **order processing** could inform the **invoice** system asynchronously that an invoice has to be written.
- The ordering system thus determines exactly what the invoicing system has to do: generate an invoice.
- It also sends a message to the bounded context **shipping** to trigger the delivery.

The system can also be set up differently. The focus will then be on an event like, “There is a new order.” Every microservice can react appropriately to this:

- The **invoicing** system can write an invoice.
- The **shipping** system can prepare the goods for delivery.

Each microservice decides for itself how it reacts to the events, see the drawing below.



Asynchronous Events

This leads to **better decoupling**. If a microservice has to react differently to a new order, the microservice can implement this change on its own.

It is also possible to **add a new microservice** that generates statistics when a new order is placed.

Events and DDD

However, this approach is not quite as easy to implement. The crucial question is **what data is transferred with the event**?

If the data is to be used for such different purposes as the writing of an invoice, statistics, or recommendations, then a large number of different attributes must be stored in the event.

This is problematic because domain-driven design shows that **each domain model is valid only in a bounded context**:

- For invoicing, prices and tax rates have to be known.
- For shipping, size and weight of the goods are needed to organize a suitable transport.

Patterns from strategic design

The views of *invoice* and *shipping* on the data of an order represent two bounded contexts, which can receive data from a third bounded context, the *order process*.

Domain-driven design defines patterns for this. For example, with customer/supplier, the team for invoicing and shipping can define what data it needs to receive. The team that provides the data for the order must meet these requirements.

This pattern defines the interaction of the teams that develop the bounded contexts that participate in the communication relationship. Such coordination is necessary regardless of whether events are sent, or whether communication between components takes place by a synchronous call.

> In other words: Events may seem to decouple the system, but coordination regarding the necessary data must still take place.

This means that events do not necessarily lead to a truly decoupled system. In extreme cases, events can even lead to hidden dependencies. Who reacts to an event? If this question can no longer be answered, the system is hardly changeable anymore because the changes to the events have had unforeseeable consequences.

Specific events

A solution might be to provide specific types of events for each receiver. Each type of event would contain the information that this receiver needs.

If a new order appears in the system, an event is sent to the invoicing system with the data it needs. Another event is sent to shipping with the data for that system.

The two systems are truly decoupled: a change in the interface to one of the systems does not influence the other system. This can be the result of a customer/supplier relationship.

Published language

Another solution would be to use a published language. In that case, a common data structure exists that contains all information for all receivers. This might make it hard to understand which receivers use what as changes to the data structure might lead to unforeseen problems. However, there is just one data structure, so it is somewhat easier to implement the system.

A very important matter is the difference in information that each receiver needs. If it is mostly the same, a published language might be better. If it is very different, it might make more sense to have separate data structures. For the case of invoicing and shipping, there is probably not too much overlap, so two separate data structures might be the better alternative.

Sending minimal data in an event

There is yet another way to deal with this problem. In this event, only the number of a new order is sent along.

Afterwards every bounded context can decide for itself how to get the necessary data. There can be a special interface for each bounded context that provides the appropriate data for that specific bounded context.

QUI
Z

1 Suppose there is no overlap between the data that two microservices require in order to respond to an event. Which of the following patterns does NOT suit this scenario well?

○ A) Published language

○ B) Specific events

○ C) Minimal data

Submit Answer

Question 1 of 2

0 attempted

Reset Quiz

In the next lesson, we'll study event sourcing.

Back

Data Replication, Bounded Contexts, ...

Next

Event Sourcing

✓ Mark as Completed

Report an Issue