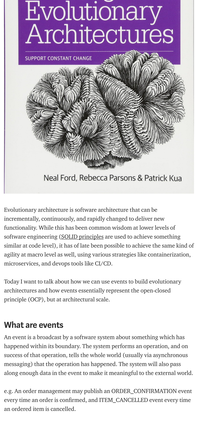
Using Events to build evolutionary architectures

Kislay Verma | Follow
Jun 26, 2019 · 7 min read



Evolutionary architecture is software architecture that can be incrementally, continuously, and rapidly changeset to deliver new functionality. While this has been common wisdom at lower levels of software engineering (SOLID principles are used to achieve something similar at code level), it has of late been possible to achieve the same kind of agility at macro level as well, using various strategies like containerization, microservices, and devops tools like CI/CD.

Today I want to talk about how we can use events to build evolutionary architectures and how events essentially represent the open-closed principle (OCP), but at architectural scale.

What are events

An event is a broadcast by a software system about something which has happened within its boundary. The system performs an operation, and on success of that operation, tells the whole world (usually via asynchronous messaging) that the operation has happened. The system will also pass along enough data in the event to make it meaningful to the external world.

e.g. An order management may publish an ORDER_CONFIRMATION event every time an order is confirmed, and ITEM_CANCELLED event every time an ordered item is cancelled.

Events vs Messages

While they are often used inter-changeably by developers who are building asynchronous communication between two system, *events* and *messages* are fundamentally different and give rise to very different kinds of behaviours in software systems.

An event is a record of a certain action having happened in a system and is therefore defined in the language of the publishing system. The publisher cares not at all about who might be listening and merely guarantees that a certain set of event data will be emitted over a certain medium of transmission.

Top highlight

A message, on the other hand, is a peer-to-peer construct. The publisher of the message targets the message at a specific consumer system and the contents must be defined in the language of the consumer. Such a message would not be meaningful to others, even if they were to listen in. In a sense, a message sent by system A to system B is API invocation done asynchronously.

Event based architecture

If both event and message travel over an asynchronous transport medium (e.g. Kafka, RabbitMQ), how does it matter which one is which? It matters when we think about interactions between many distributed systems and who knows about who in such a world.

If we use events to propagate information across our distributed system, we come up with a very loosely coupled architecture where there is minimal knowledge of each other across systems. All systems either broadcast events corresponding to activities in their world or consume events from other systems to trigger workflows in their own world. As a publisher, a system does not know who will consume its events. As a consumer, a system is not aware of where the event came from, just that it should perform something when it receives such an event.

e.g. An order system might emit an ORDER_CONFIRMED event, which may be consumed by an invoicing system and an accounting system. The invoicing system will now generate an invoice and emit INVOICE_GENERATED event. Listening to the INVOICE_GENERATED event, the order system may send an email to customer. The order system sees one publish and one consume but does not trace a causality between the two.

Workflow in event driven systems

In the micro-service world, events give rise to the *choreography style* of building workflows. Essentially, this is no explicitly defined workflow at all but service are mapped to respond to certain set of events. The interaction described above is an example. An end-to-end workflow is achieved without describing it as such because we are able to compose it from independent event-service interactions. No one needs to know the complete flow as it does not really exist.

Message based architecture

In a message based architecture, the order system would emit two messages : GENERATE_INVOICE (to the invoicing service) and BOOK_REVENUE (to the accounting system) with order identifier as reference and then wait (callback based) on the invoicing system response. The invoicing system, after generating the invoice, sends back an acknowledging message for the GENERATE_INVOICE message, on receiving which the order system sends an email to the customer.

Workflows in a message driven architecture

Note how systems are aware of each other in this paradigm. They may be decoupled in time due to the use of asynchronous messaging, but they are coupled at the domain handover boundary. However, since systems are aware of each other, we can build nuanced experiences around handshakes (the ack sent by the invoicing system in our example above is such an example) and error handling which would not be possible in the event driven world.

In the micro-service world, messages give rise to *orchestration style workflows*. A service or an orchestrating system (often a workflow engine like JBPM or its more modern avatars like Conductor and Cadence) captures the sequence in which a set of services should be invoked to achieve an end-to-end output and it invokes them via messages (or APIs, as the case may be). ESB based systems are a version of messaging architectures.

Events are OCP

Now it should be clearer why I think of events as a form of open-closed principle (OCP). OCP says that our code should be open to extension but closed to change. i.e. anyone who wants to add additional functionality to existing code should be able to do so *from the outside*, without having to touch the code itself. In an event based architecture, all a system is responsible for is performing its function and emitting the corresponding events. It doesn't know which other systems are consuming these events or how.

So if we were to change the implementation of our current invoicing system, or to build different invoicing systems for different types of orders, or don't want to send notifications for some types of invoices, we could do it all without touching the order system itself. Whole new things could be developed outside of the order system to enrich the order management platform without touching the order system. The is the open-closed principle at work on an architectural scale.

Evolving an event based architecture

Let's talk a little more about how we would evolve an architecture based on events. We have already seen how we can change everything around a system without touching the system itself. Now what would we do if we wanted to change this system itself (the order system in our previous example)? How to manage the impact on other system?

As it turns out, there is no/minimal impact. As far as all the other systems are concerned, this system does not exist. For them, the event stream *IS* the fact of life, and as long as the events continue to flow in, it doesn't matter to them whether they are coming from the same system or from the next version of it or from a entirely new system. Even if we build a new system which does not abide by the current event structure or semantics, it is often only a matter of understanding the new event-data and massaging it into the consuming systems own language.

This kind of decoupling is very powerful when we want to quickly move around our technical constructs. A widely employed strategy for building new versions of software is the strangler pattern where you progressively migrate and deploy functionality from one version of a software to the next one, all the while keeping the structure of the events same. As long as we keep the event flow backward compatible, no one need to know that somehing is changing. This pattern is often used in migrating from monoliths to micro-services.

Event-ful Pitfalls!

However, this degree of decoupling comes at a price. There are some problems that must be kept in mind when adopting event driven architectures.

The most important problem is one of tracking business workflows. Since systems do not collaborate with each other but rather with events, it becomes difficult to track what the status of any business process is. Long pipelines like order processing become very difficult to track and manage. Answering "define the complete process of order fulfilment" can have you running all over the engineering department!

The other, lesser problem is around error handling. If one system suffers from an outage and loses some messages, there is no straightforward way to re-generate/replay them. The publisher gives no guarantee that it can re-publish them. Persistent messaging system like Kafka help to a certain extent, but guaranteeing their uptime and resilience, even more than the core services, becomes a mission critical problem for the whole company.

In some cases, we are all right accepting a little coupling in order to attain business cohesion and debuggability. Simply enabling an event stream for your system will decouple you from all other use-cases which you do not deem to be "core" for your domain, while you retain the freedom to sign up for message driven use-cases when you have. How to distinguish these situations will, of course, vary from use-case to use-case. The key thing, as always, is tradeoffs.

Sprinkle carefully for a juicy architecture!

. . .

If you liked this article, you can subscribe to my mailing list to stay up to date on the latest.

. . .

WRITTEN BY
Kislay Verma
Code, products, platforms, books, music
Follow

Noteworthy - The Journal Blog
The Official Journal Blog
Follow