O que você quer dizer com "Event-Driven"?

Martin Fowler

DESIGN
EVENT ARCHITECTURES

07 February 2017

Towards the end of last year I attended a workshop with my colleagues in Thoughtworks to discuss the nature of “event-driven” applications. Over the last few years we've been building lots of systems that make a lot of use of events, and they've been often praised, and often damned. Our North American office organized a summit, and Thoughtworks senior developers from all over the world showed up to share ideas.

The biggest outcome of the summit was recognizing that when people talk about “events”, they actually mean some quite different things. So we spent a lot of time trying to tease out what some useful patterns might be. This note is a brief summary of the main ones we identified.

Event Notification

This happens when a system sends event messages to notify other systems of a change in its domain. A key element of event notification is that the source system doesn't really care much about the response. Often it doesn't expect any answer at all, or if there is a response that the source does care about, it's indirect. There would be a marked separation between the logic flow that sends the event and any logic flow that

I spoke about this topic in more depth in my opening keynote at goto Chicago in April 2017.

Event notification is nice because it implies a low level of coupling, and is pretty simple to set up. It can become problematic, however, if there really is a logical flow that runs over various event notifications. The problem is that it can be hard to see such a flow as it's not explicit in any program text. Often the only way to figure out this flow is from monitoring a live system. This can make it hard to debug and modify such a flow. The danger is that it's very easy to make nicely decoupled systems with event notification, without realizing that you're losing sight of that larger-scale flow, and thus set yourself up for trouble in future years. The pattern is still very useful, but you have to be careful of the trap.

A simple example of this trap is when an event is used as a passive-aggressive command. This happens when the source system expects the recipient to carry out an action, and ought to use a command message to show that intention, but styles the message as an event instead.

An event need not carry much data on it, often just some id information and a link back to the sender that can be queried for more information. The receiver knows something has changed, may get some minimal information on the nature of the change, but then issues a request back to the sender to decide what to do next.

Event-Carried State Transfer

This pattern shows up when you want to update clients of a system in such a way that they don't need to contact the source system in order to do further work. A customer management system might fire off events whenever a customer changes their details (such as an address) with events that contain details of the data that changed. A recipient can then update it's own copy of customer data with the changes, so that it never needs to talk to the main customer system in order to do its work in the future.

An obvious down-side of this pattern is that there's lots of data schlepped around and lots of copies. But that's less of a problem in an age of abundant storage. What we gain is greater resilience, since the recipient systems can function if the customer system is becomes unavailable. We reduce latency, as there's no remote call required to access customer information. We don't have to worry about load on the customer system to satisfy queries from all the consumer systems. But it does involve more complexity on the receiver, since it has to sort out maintaining all the state, when it's usually easier just to call the sender for more information when needed.

Event-Sourcing

The core idea of event sourcing is that whenever we make a change to the state of a system, we record that state change as an event, and we can confidently rebuild the system state by reprocessing the events at any time in the future. The event store becomes the principal source of

truth, and the system state is purely derived from it. For programmers, the best example of this is a version-control system. The log of all the commits is the event store and the working copy of the source tree is the system state.

Event-sourcing introduces a lot of issues, which I won't go into here, but I do want to highlight some common misconceptions. There's no need for event processing to be asynchronous, consider the case of updating a local git repository - that's entirely a synchronous operation, as is updating a centralized version-control system like subversion. Certainly having all these commits allows you to do all sorts of interesting behaviors, git is the great example, but the core commit is fundamentally a simple action.

Another common mistake is to assume that everyone using an event-sourced system should understand and access the event log to determine useful data. But knowledge of the event log can be limited. I'm writing this in an editor that is ignorant of all the commits in my source tree, it just assumes there is a file on the disk. Much of the processing in an event-sourced system can be based on a useful working copy. Only elements that really need the information in the event log should have to manipulate it. We can have multiple working copies with different schema, if that helps; but usually there should be a clear separation between domain processing and deriving a working copy from the event log.

When working with an event log, it is often useful to build snapshots of the working copy so that you don't have to process all the events from scratch every time you need a working copy. Indeed there is a duality here, we can look at the event log as either a list of changes, or as a list of states. We can derive one from the other. Version-control systems often mix snapshots and deltas in their event log in order to get the best performance. [1]

Event-sourcing has many interesting benefits, which easily come to mind when thinking of the value of version-control systems. The event log provides a strong audit capability (accounting transactions are an event source for account balances). We can recreate historic states by replaying the event log up to a point. We can explore alternative histories by injecting hypothetical events when replaying. Event sourcing make it plausible to have non-durable working copies, such as a Memory Image.

Event sourcing does have its problems. Replaying events becomes problematic when results depend on interactions with outside systems. We have to figure out how to deal with changes in the schema of events over time. Many people find the event processing adds a lot of complexity to an application (although I do wonder if that's more due to poor separation between components that derive a working copy and components that do the domain processing).

CQRS

Command Query Responsibility Segregation (CQRS) is the notion of having separate data structures for reading and writing information. Strictly CQRS isn't really about events, since you can use CQRS without any events present in your design. But commonly people do combine CQRS with the earlier patterns here, hence their presence at the summit.

The justification for CQRS is that in complex domains, a single model to handle both reads and writes gets too complicated, and we can simplify by separating the models. This is particularly appealing when you have a difference in access patterns, such as lots of reads and very few writes. But the gain for using CQRS has to be balanced against the additional complexity of having separate models. I find many of my colleagues are deeply wary of using CQRS, finding it often misused.

Making sense of these patterns

As a sort of software botanist, keen to collect samples, I find this a tricky terrain. The core problem is confusing the different patterns. On one project the capable and experienced project manager told me that event sourcing had been a disaster - any change took twice the work to update both the read and write models. Just in that phrase I can detect a potential confusion between event-sourcing and CQRS - so how can I figure out which was culprit? The tech lead on the project claimed the main problem was lots of asynchronous communications, certainly a known complexity-booster, but not one that's a necessary part of either event-sourcing or CQRS. Furthermore we have to beware that all these patterns are good in the right place and bad when put on the wrong terrain. But it's hard to figure out what the right terrain is when we conflate the patterns.

I'd love to write some definitive treatise that sorts all this confusion out, and gives solid guidelines on how to do each pattern well, and when it should be used. Sadly I don't have the time to do it. I write this note in the hope it will be useful, but am quite aware that it falls well short of what is really needed.

Further Reading

I prepared a talk on this topic which was the keynote at goto Chicago 2017.

Back in 2006, I wrote a bunch of proto-patterns with the thought of making a further volume of my P of EAA book. Sadly, even a decade later, I haven't had time to continue with this work. The stuff I did write then, however, is there to be read. For events I'd start with Focusing on Events, which summarizes my thinking on using events at the time. Although it is a while ago, I think most of what I wrote then is still valid.

The most influential of these articles is the on Event Sourcing. It primarily talks about the value of using replay to form historic and alternative states.

The article on Event Collaboraton touches on the patterns of Event Notification and Event-Carried State Transfer, but conflates the patterns (it was only during the workshop that I started thinking of them as separate).

I have a bliki post on CQRS.

There's a lot more material on the web about these topics, so have fun exploring that. I don't have any comments on it here as I haven't spent time looking through it and picking out some recommendations.

Footnotes

1: I sometimes hear people say that git isn't an example of event sourcing because it stores states of files and trees in `.git/objects`. But whether a system uses changes or snapshots for its internal storage doesn't affect whether it's event sourced. Git happily will summon up a list of changes for me on demand. And when it compresses data into packfiles, it does use a combination of snapshots and changes, with the mix chosen for performance reasons.

Updates

2017-02-08: Tweaked discussion of use of event log versus application state and added a footnote to clarify role of git and snapshots.