Concept/Topic: *Event Sourcing and Command Query Responsibility Segregation (CQRS)*

Motivation and Overview

Event Sourcing and CQRS follow a different approach to state management than that followed by most information systems that center on a rich domain model whose application state is updated continuously while processing and responding to incoming client requests. Such state might be captured as the content of entities modeled with tactic Domain-Driven Design (DDD); the entity modifiers implementing business logic then update the attributes of the entities (which usually is hidden behind the interface of the entity).

Event sourcing and *Command Query Responsibility Segregation (CQRS)* deal with state differently:

- A stream of events is the only means to transfer/initiate state changes. All changes come in as events and are handled via *event processing*.
- To get current values, the application cannot simple read the property of an entity, but has to start from empty application state and use the change log to recalculate the current state (unless certain snapshots are taken, which leads to a hybrid approach).
- All events are immutable (they do not have any setter methods, and do not implement any business logic that might have side effects). There are two types of such immutable events, those describing an absolute/full new value vs. those reporting differences between old and new values; the latter type is more reversible than the first one.
- In CQRS, different (domain) models are used for read and write access; this has to be handled with care (as it is powerful but also complex and therefore risky).

This approach is different from the request-reply paradigm in which state changes are side effects of incoming calls (e.g., calls to entity modifiers in tactic DDD).

Usage scenarios/capabilities of such event-based state management include:

- Complete rebuilds, temporal queries, and event replay can easily be supported; full audit logs become available without extra effort.
- Events can be used to implement asynchronous, eventually consistent updates across DDD aggregates/bounded contexts (with instances of the DDD pattern Service acting as event handlers).

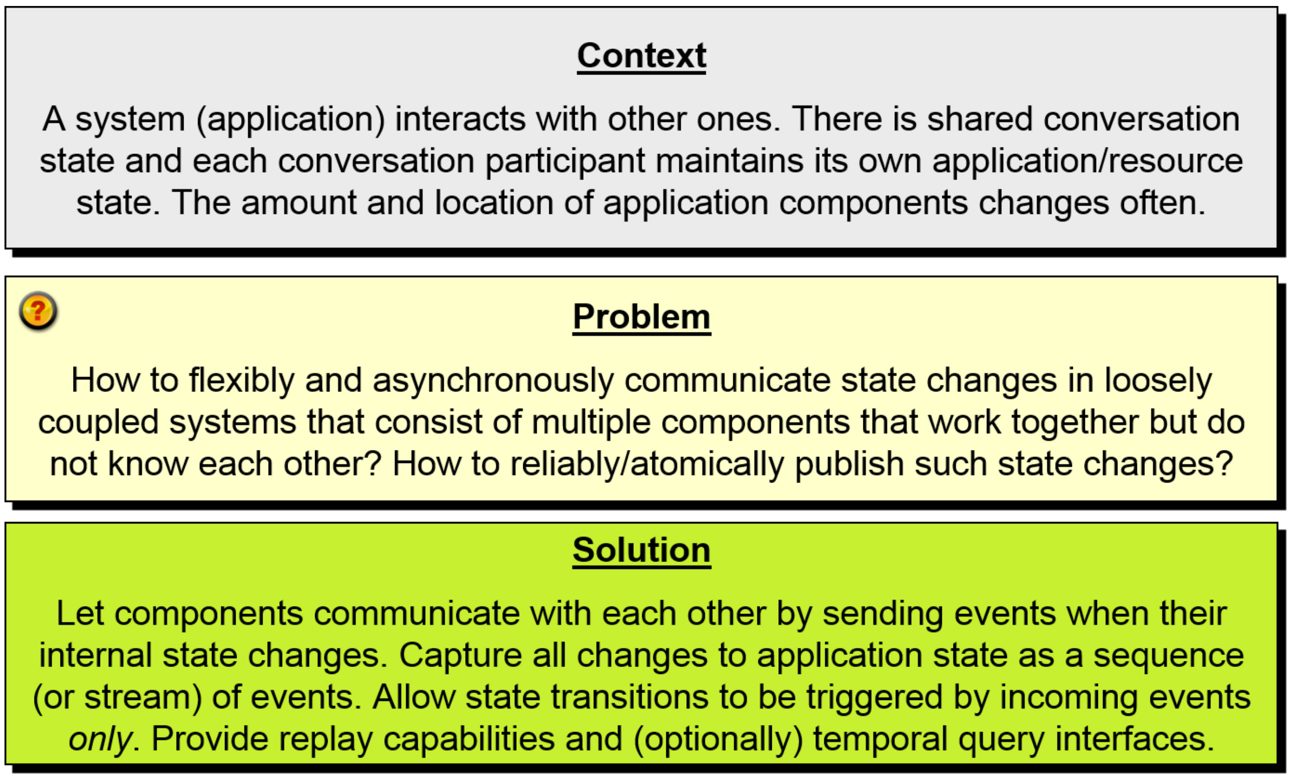
Events are often sent via reliable messaging (queues), stored in special databases (time series databases, key-value stores), and used to populate a dedicated query datastore (in CQRS).

Definition(s)

Events and Event Sourcing

An *event* is something that happened that affects application/resource state. Its name must indicate that the event took place in the past. For instance, a naming convention could be "domain model element (noun) + action verb (in -ed form)": `CustomerVerifiedEvent`, `ShipmentArrivedEvent`, `AddressUpdatedEvent` make good names that become part of the DDD ubiquitous language. The event content must be complete enough to indicate update needs; typically, an Id, a timestamp as well as entity references (i.e., ids) are sent. Entity properties (or class attributes) are often codified as self-descriptive markup (JSON, XML).

See Figure "Event Sourcing pattern" for a pattern summary. The pattern is fully described by M. Fowler on his bliki[1] and C. Richardson (in draft form) on the microservices.io website[2].



Context

A system (application) interacts with other ones. There is shared conversation state and each conversation participant maintains its own application/resource state. The amount and location of application components changes often.

Problem

How to flexibly and asynchronously communicate state changes in loosely coupled systems that consist of multiple components that work together but do not know each other? How to reliably/atomically publish such state changes?

Solution

Let components communicate with each other by sending events when their internal state changes. Capture all changes to application state as a sequence (or stream) of events. Allow state transitions to be triggered by incoming events *only*. Provide replay capabilities and (optionally) temporal query interfaces.

Figure 1: *Event Sourcing pattern*

Events are never deleted, but additional events are sent to report the business-level end of life of an entity instance (unless regulatory requirements mandate permanent data removal). W.r.t. the (loose) coupling dimensions (i.e., time, reference, platform, format autonomy), the pattern supports time, reference, and platform autonomy; the degree of format autonomy depends on the event content.

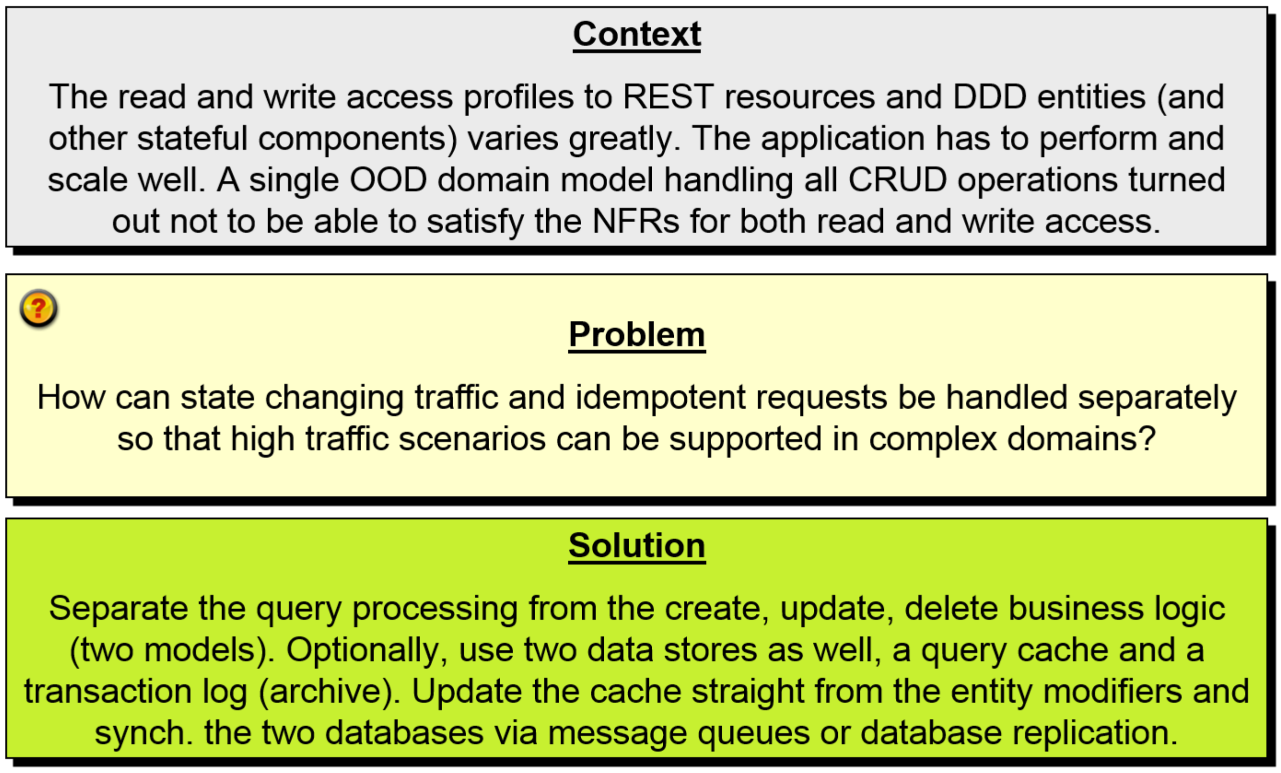
Command Query Responsibility Segragation (CQRS)

Command Query Responsibility Segregation (CQRS) suggests to use one (domain, communication) model to update information and another model to read information; queries are free of side effects, commands (a.k.a. modifiers) change state. If create, update, delete processing is separated from read access this way, specialized patterns and middleware can be used for each channel. The cost includes the increased integration/coordination effort on data access layer/database level (consistency guarantees and management?).

CQRS is neither an individual architecture nor an architectural style, but an architectural pattern according to G. Young, who came up with the pattern name (see this online article[3]). This pattern has its roots in Command Query Separation[4] described by B. Meyer in his seminal book "Object-Oriented Software Engineering". It does not constitute an end goal but is "a stepping stone towards event sourcing"[5].

See figure "CQRS pattern" for a pattern summary. An article in M. Fowler's bliki[6] contains a more detailed pattern description. C. Richardson's vision for using DDD, microservices and event sourcing/CQRS is "Mi-

[1] https://martinfowler.com/eaaDev/EventSourcing.html
[2] http://microservices.io/patterns/data/event-sourcing.html
[3] http://codebetter.com/gregyoung/2012/09/09/cqrs-is-not-an-architecture-2/
[4] https://martinfowler.com/bliki/CommandQuerySeparation.html
[5] http://codebetter.com/gregyoung/2010/02/16/cqrs-task-based-uis-event-sourcing-agh/
[6] http://martinfowler.com/bliki/CQRS.html



Context

The read and write access profiles to REST resources and DDD entities (and other stateful components) varies greatly. The application has to perform and scale well. A single OOD domain model handling all CRUD operations turned out not to be able to satisfy the NFRs for both read and write access.

Problem

How can state changing traffic and idempotent requests be handled separately so that high traffic scenarios can be supported in complex domains?

Solution

Separate the query processing from the create, update, delete business logic (two models). Optionally, use two data stores as well, a query cache and a transaction log (archive). Update the cache straight from the entity modifiers and synch. the two databases via message queues or database replication.

Figure 2: *CQRS pattern*

croservice architecture as a Web of Event-Driven Aggregates”[7]. He makes a draft pattern text available here[8]. DDD aggregates (and/or their root entities) often handle events, not individual entities or entire bounded contexts.

Example(s)

See presentation video by M. Ploed[9] and lecture slides for an example from the incident management domain. Also see an online article by U. Dahan[10] for examples and pros and cons discussion.

Application of the Concept in Products and Projects

CQRS and event sourcing are not mainstream, but a popular topic in the software architecture community (and beyond) at present. Successful projects are being reported on at conferences and in online articles; supporting middleware exists. The two patterns are (somewhat) popular in the cloud computing, adaptive systems, and Internet of Things communities.

Event Sourcing and CQRS can be seen as the foundation of *Event-Driven Architectures (EDAs)* when being combined with more patterns, which are outlined and/or described on M. Fowler’s website Development of Further Patterns of Enterprise Application Architecture[11]:[12]

- Domain Event

[7]https://www.infoq.com/articles/microservices-aggregates-events-cqrs-part-1-richardson
[8]http://microservices.io/patterns/data/cqrs.html
[9]https://www.youtube.com/watch?v=A0goyZ9F4bg
[10]http://www.udidahan.com/2009/12/09/clarified-cqrs%3E
[11]https://martinfowler.com/eaaDev/
[12]but unfortunately not in a Volume 2 of “Patterns of Enterprise Application Architecture”.

- Event Collaboration
- Parallel Model
- Temporal Property
- Retroactive Event
- etc. (complex event processing, event stream processing)

Critique (Pros and Cons of CQRS and Event Sourcing)

The patterns are suited for complex domains (according to M. Fowler). The read and write models in CQRS can be optimized independently to perform/scale well. Event sourcing goes well with functional programming and cloud computing. CQRS is good for occasional connectivity scenarios and when availability more important than consistency. Time becomes a crucial element of the domain knowledge (G. Young).

That said, it is hard to keep the data consistent (e.g., when dealing with parallel, conflicting updates involving multiple event sources and sinks). Moreover, it is hard(er) to validate/test and maintain systems that apply these patterns in comparison to systems that only apply a single model, request-reply style.

Hence, usage advice and critique include:

- “Despite these benefits, you should be very cautious about using CQRS.” (M. Fowler)
- “A hell of a lot to give up, on highly dubious rationale.” (D. Haywood)
- “A Whole System Based on Event Sourcing is an Anti-Pattern”[13](J. Stenberg); therefore a hybrid style that only uses CQRS and Event Sourcing in isolated places with weaker consistency requirements or moderate read/write access profiles (but a need for audit and temporal analysis capabilities).

See the video tutorial by M. Ploed also referenced in the lecture[14] and this InfoQ summary and vision article by J. Stenberg[15] for additional benefits and liabilities discussions.

More Information

Related Topics and Concepts

SOA, REST, microservices; state management

Miscellaneous Links:

- Two articles by C: Richadson on InfoQ: Part 1[16] and Part 2[17].
- M. Fowler’s bliki has a related tag *event architectures*[18].
- An award-winning presentation at SATURN 2017[19] also focused on Event Sourcing and CQRS[20].

[13] https://www.infoq.com/news/2017/04/event-sourcing-anti-pattern
[14] https://www.youtube.com/watch?v=A0goyZ9F4bg
[15] https://www.infoq.com/news/2017/11/event-sourcing-microservices
[16] https://www.infoq.com/articles/microservices-aggregates-events-cqrs-part-1-richardson
[17] https://www.infoq.com/articles/microservices-aggregates-events-cqrs-part-2-richardson
[18] https://martinfowler.com/tags/event%20architectures.html
[19] http://eltjopoort.nl/blog/2017/05/14/saturn-2017/
[20] https://insights.sei.cmu.edu/saturn/2017/05/saturn-2017-awards-conferred.html