Rill/Wheel - Design and Implementation

Joost Diepenmaat, <joost@zeekat.nl>

2016-12-21

Design space

Message-based, streaming architectures

- Kafka
- Elastic stack
- AWS Lambda

Explicit domain

Extract domain from mechanism and protocols.

- Hexagonal architecture, ports
- Domain Driven Design

Distributed multi-node services

Horizontal scaling of individual services; multiple nodes per service.

State local to (sharded) service.

Constrain side effects, state updates

FP, Clojure lessons: unrestricted side effects are complex.

Unrestricted state modifications are hard to reason about.

Events as truth

Events are immutable data; ideal for sharing.

CQRS

- Commands
- Queries
- Responsibilities
- Separation

Domain Events

Domain

> *late Middle English (denoting heritable or landed property): from French domaine, alteration (by association with Latin dominus 'lord') of Old French demeine 'belonging to a lord'*

Description of what the system is about; its responsibilities and invariants, independent of mechanisms.

Event

> *late 16th century: from Latin eventus, from evenire 'result, happen', from e- (variant of ex-)'out of' + venire 'come'.*

Thing that happened, an outcome

- Past tense
- Immutable; cannot "unhappen"
- Unconditional; in the past

Domain Event

An event that describes something relevant to the domain.

- Preferably self-describing, independent
- Appropriate abstraction level for the domain
- 5 Ws - Who What Where When Why

Abstraction level

When in doubt, abstract "up"

"User clicked button X"

- Describes mechanism (user pressed "h" "e" "l" "l" "o")
- Not independent - doesn't capture enough information (other form fields...)
  May still be useful to track for improving mechanism

"Attributes X,Y of Entity E changed to values V,W"

As in relational transaction, datomic

Doesn't capture what actually happened in the domain; focused on effect instead of intent.

"User's address was corrected" is fundamentally different from "User moved to a different address".

"User moved to new address X"

Captures meaning and describes properties; the new address, and the fact that the user moved there.

Who, What, Where, When, Why

Generating events in a distributed system

The problem with one big log

If you take events as truth, where do you do your coordination?

Resolving conflicting events is hard, but we cannot lock the whole log.

How do we quickly retrieve relevant earlier events when new events should be generated?

Streams as a coordination mechanism

- Split the log into streams; independent, append-only sequences of events
- Serialize commits to each stream
- Reduce the stream's events into just the data needed for generating new events (aggregates).

Streams are consistancy boundaries

Transactions can only affect the single stream that events will be committed to.

The invariants are described by the command model.

Streams are concurrency boundaries

Other streams are readable in a transation but may yield stale data (missing latest/concurrent events); you can only look into the past and not prevent the future.

Queries and views

Derived data

Queries are read only; do not generate domain events

Can be implemented as separate services

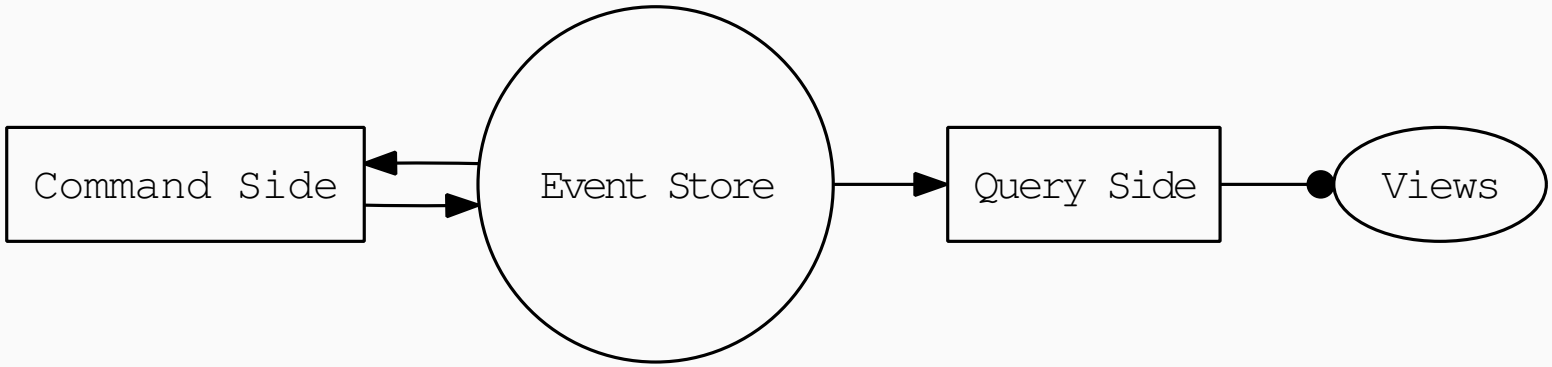
On separate nodes and/or client-side

Ephemeral; can be rebuild based on events

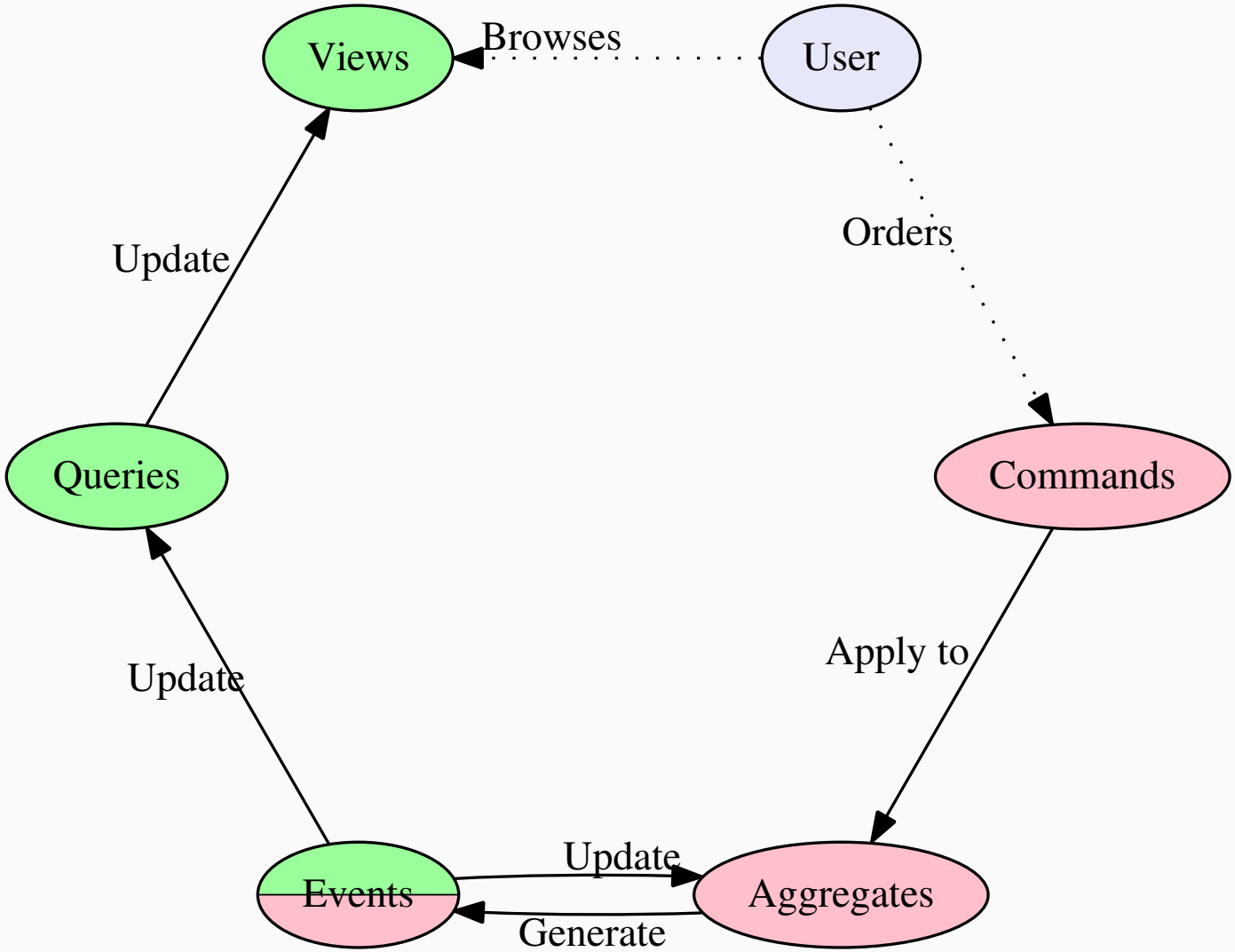
Caching to improve deployment, scaling speed

Wheel Implementation

CQRS / Event Sourcing



Event Sourcing



What does Rill/Wheel provide?

Event Store

- Durable event storage.
- Per-stream transactions; append events if stream unchanged.

Aggregate description

```
(defaggregate authorization
  "Controls authorization from consumer to app"
  [app-id consumer-name]
  {:pre [(account-name? consumer-name) app-id]})
```

Domain Event descriptions

```
(defevent revoked ::authorization
  [authorization]
  ; pre-post goes here, when useful
  "previously granted authorization was revoked"
  (dissoc authorization :granted))
```

Command descriptions

```
(defcommand revoke ::authorization
  "revoke access"
  [authorization]
  ; pre-post map goes here
  (if-not (:granted authorization)
    (rejection authorization :not-granted)
    (revoked authorization)))
```

Repository

Maps event streams to aggregates. Transactions on aggregates.

```
(defprotocol Repository
  (commit! [repo aggregate]
    "Commit changes to ‘aggregate‘.
    Applications should use ‘rill.wheel/commit!‘
    instead.")
  (update [repo aggregate]
    "Returns updated aggregate by applying
    new committed events."))
```

Data and function-based command invocation

```
(defaggregate a [prop1])
(defevent some-event ::a
  [a prop2]
  (assoc a :p2 prop2))
(defcommand some-command ::a
  [a prop2]
  (if some-condition
    (rejection a :some-reason)
    (some-event a prop2)))

(transact! repository (->some-command prop1 prop2))
(commit! (some-command aggregate prop1 prop1))
(some-command! repository prop1 prop2)
```

What is out of scope?

Query mechanism

Can be implemented using many different tools. Datascript, hand-build indexes, relational system.

Keep track of last applied event in index to make query service restartable.

Event publishing at scale

Built-in solutions poll the event store for updates. This will not scale when readers multiply.

Possible to publish to "big log" solutions like Kafka, no batteries included solution

Open problems

Retractions

Right to be forgotten type stuff maybe should not be published as an event. Restrict privacy sensitive data to its own silo(s).

Effective use

Design tips

- Event names should be past tense: "user-registered"
- Command names should be imperative: "register-user"
- Small events; no big payloads
- Side effects happen after commit

Consistency vs Concurrency

Choose wisely

- Smaller aggregates mean better concurrency
- Larger aggregates means more consistency

Migrations

Defining a domain model

Goals

The goal of a Rill/Wheel model is to allow efficient generation of useful domain events.

System constraints

- Aggregates are only accessible by identifier (primary key lookup)
- Commands & events must not run queries or have side effects

Design strategy

- Start with a list of events
- Determine invariants & consistency boundaries: "it should not be possible to register two users with the same account name"
- Determine commands that will generate the event
- Group commands and events in small aggregates (boundaries)

Aggregate definition

```
(defaggregate user ; name
  [account-name]) ; key properties

;; => generates
(get-user repository account-name) ; lookup by key prop
(user account-name) ; generate descriptor/key from prop
```

Aggregate key properties are fixed and merged into the events & commands for the aggregate.

Event definitions

```
(defevent registered ::user  ; event name & aggregate type
  ;; aggregate + additional event properties
  [user full-name]
  ;; empty body means leave aggregate as is
  )
;; => generates...
;; plain event message
;; includes key properties of aggregate
(->registered account-name full-name)
;; update aggregate with event - can be chained
;; aggregate can be committed later
;; used in command bodies
(registered user full-name)
;; and a few other bits and bobs
```

Command definition

```
(defcommand register ::user
  [{:keys [account-name] :as user} full-name]
  (cond
    (string/blank? account-name)
    ;; rejection data is arbitrary
    (rejection user :account-name-blank)
    (wheel/exists user)
    (rejection user :account-already-exists)
    (string/blank? full-name)
    (rejection user :full-name-blank)
    :else
    (registered user full-name)))
```

Command definition generates

```
(defcommand register ::user ...)
;; => generates
;; command message, can be passed to ‘transact!‘
(->register account-name full-name)
;; call command and commit against repository given props
(register! repository account-name full-name)
;; apply command to aggregate,
;; result can be passed to ‘commit!‘
(register user full-name)
```

Common questions 1/2

Q: What properties to include in an event?

A: 5 Ws; include at least relevant actors

Q: What properties are aggregate key props

A: As many as possible if available and fixed

Q: Queries / joins?!

A: Joins only by primary key! Solve queries in the view and only validate in the command

Common questions 2/2

Q: Uniqueness constraints

A1: Can be enforced in the model by using aggregate keys

A2: Solve on the view with queries (non-consistent), be prepared to revert events (using compensating events).

Q: Migrations?

A: Use `rill.wheel.wrap-upcasts`. Define new event types (keep old ones) when upcasts don't work.