Data Persistence

MICROSERVICES

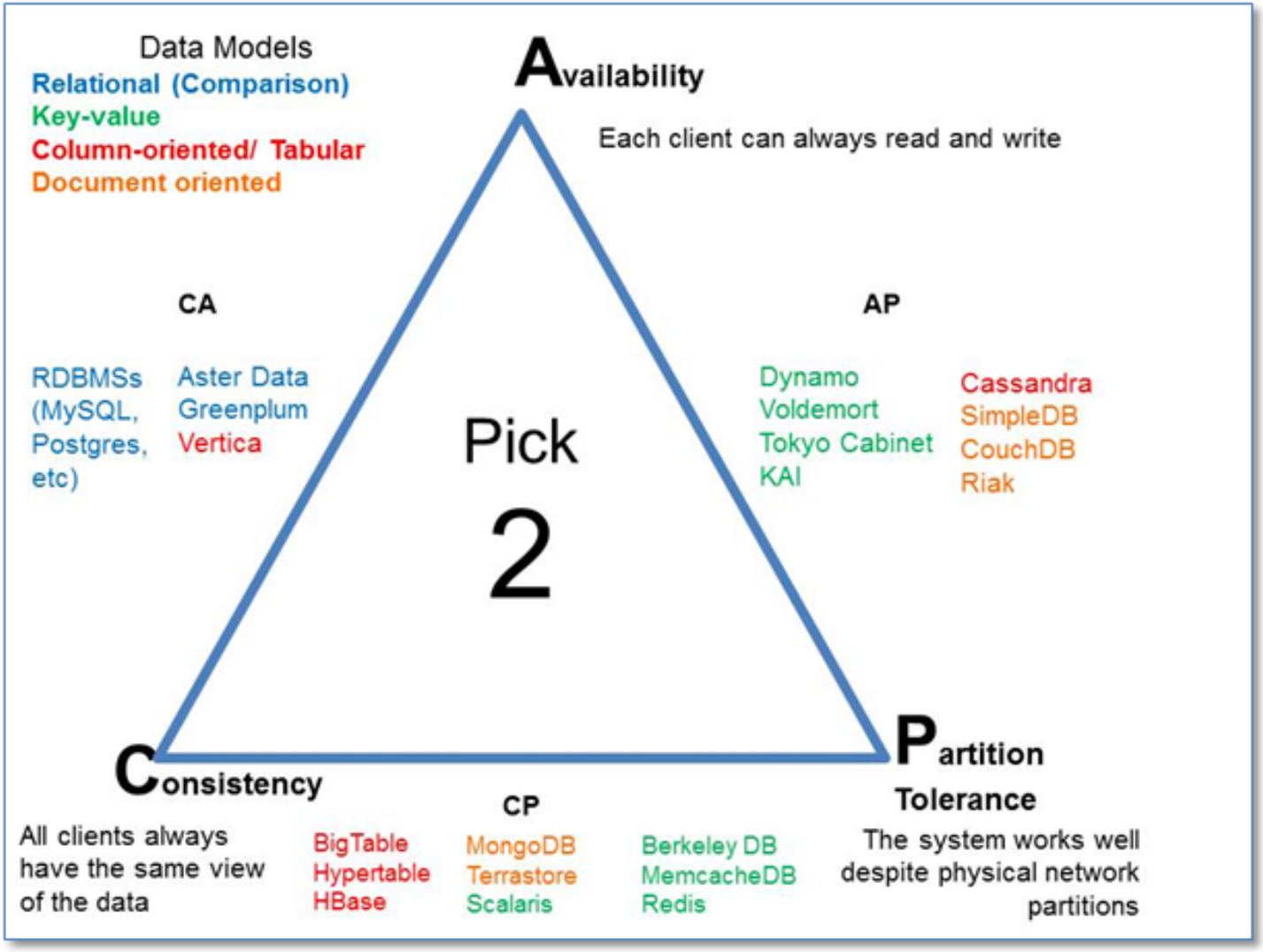
Content

- Overview
- Pattern and Anti-Pattern
- Caching
- Event Sourcing / CQRS / Sagas

Overview

- People try to copy Netflix, but they can only copy what they see. They copy the results, not the process. – Adrian Cockcroft, former Chief Cloud Architect, Netflix
- Data is the hardest part in microservices!
- CRUD (Create, Read, Update, Delete) is not enough for microservices
- You cannot do ACID (atomicity, consistency, isolation, durability) over multiple datasources transactions → Better use BASE (Basically Available, Soft state, Eventual consistency)

CAP

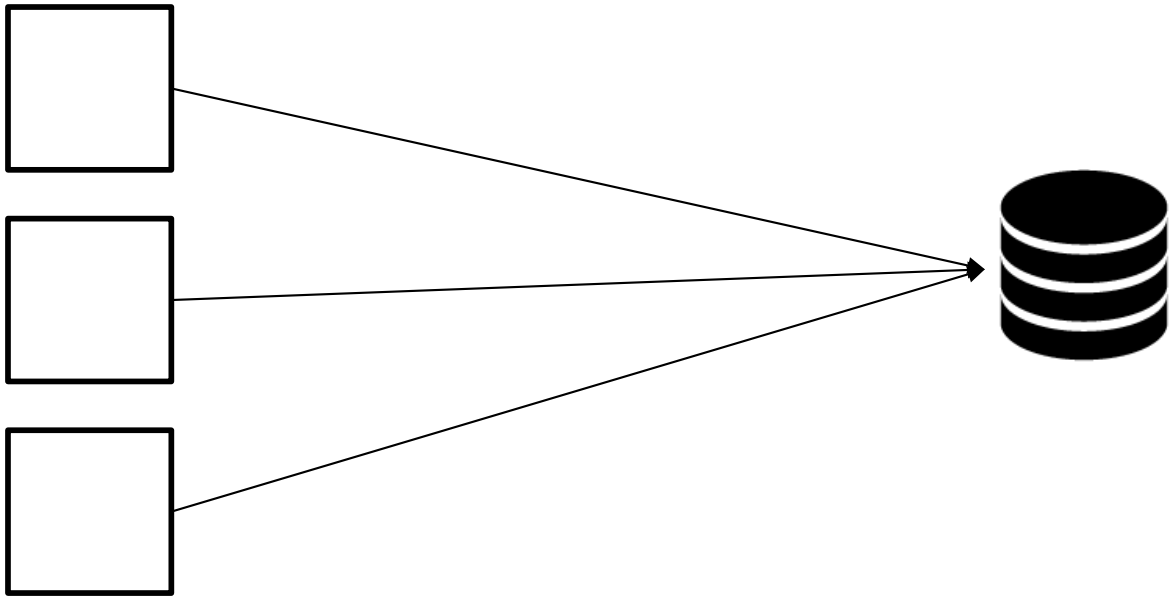


➔ More @Datenbanken 2 (Master)

http://nosql-guide.com/wp-content/uploads/1970/01/CAP-Theorem.png

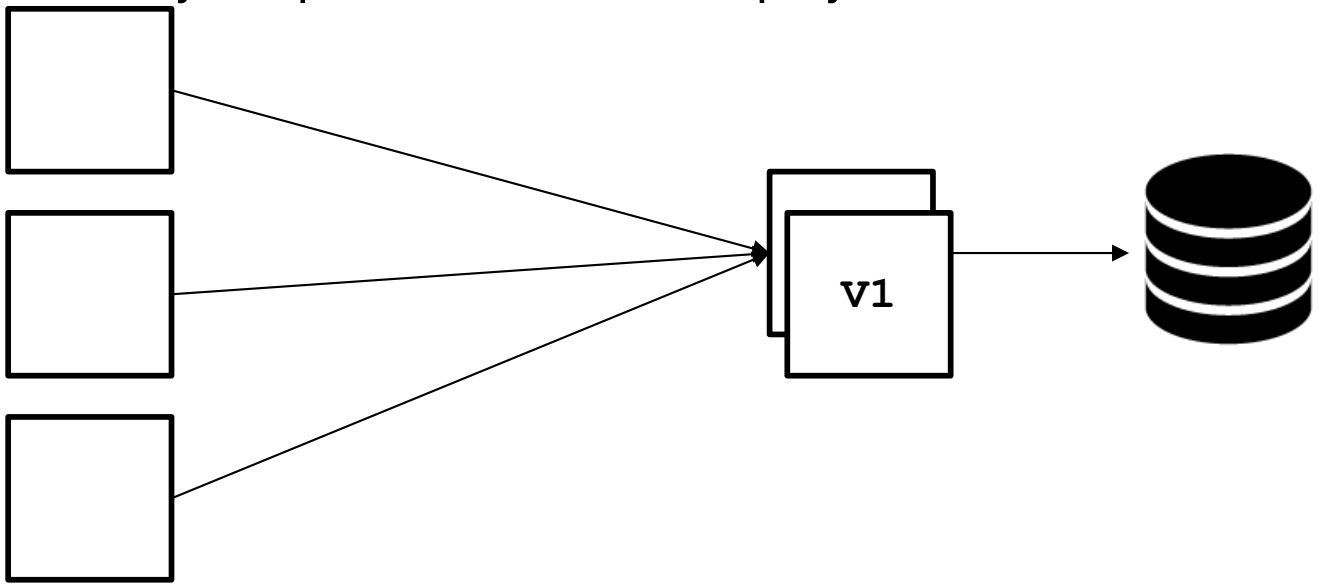
Pattern and Anti-Pattern

- Anti-Pattern: Shared Database
  - While microservices appear independent, transitive dependencies in the data tier all but eliminate their autonomy



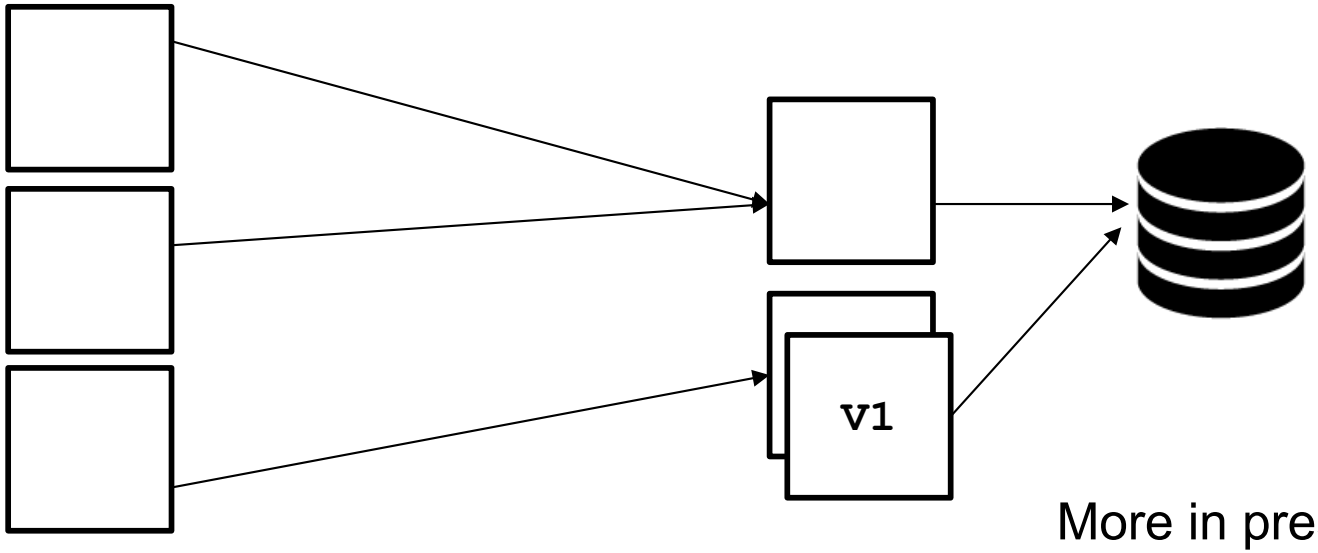
Pattern and Anti-Pattern

- Pattern: Data API
  - Microservices do not acces data layer directly
  - Expect for the microservice that implement the data API
  - A Surface area to
    - Implement access control
    - Implementing throttling
    - Perform logging
    - Other policies
  - Possibly coupled with Parallel deployment



Pattern and Anti-Pattern

- Pattern: Bounded Context
  - Domain-Driven-Design
  - Each bounded context has a signle, unified model
  - Relationships between models are explicitly defined
  - A product team usually has a strong correlation to a bounded context
  - Ideal pattern for Data APIs – do not fail into the trap of simply projecting current data models
  - Model transactional boundaries as aggregates
  - ➔ Focus on domain models, not data models

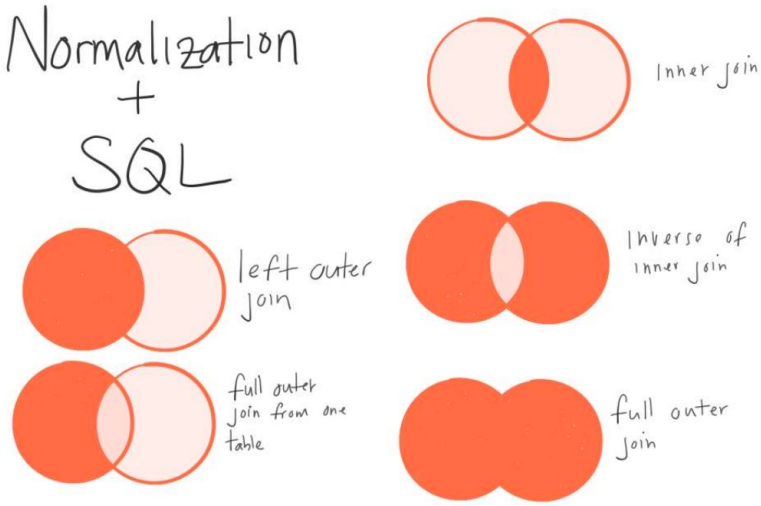


More in presentation from 07.12

Pattern and Anti-Pattern

- Pattern: Database per Service and Client Side Joins
  - Support polyglot persistence
  - Independent availability, backup/restore, access patterns, etc.
  - Services must be loosely coupled so that they can be developed, deployed and scaled independently

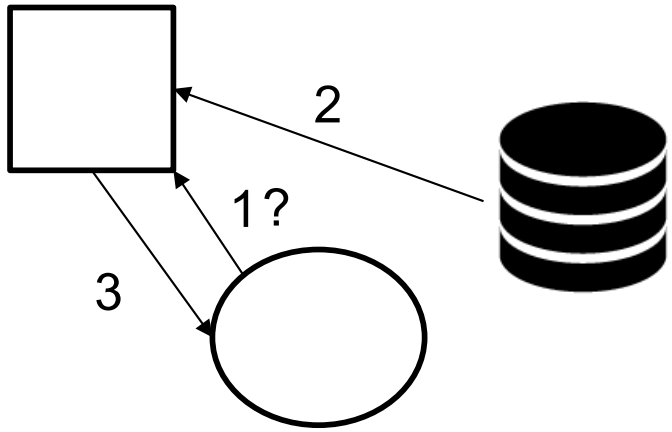
➔ Microservices need a Cache! & Materialized Views



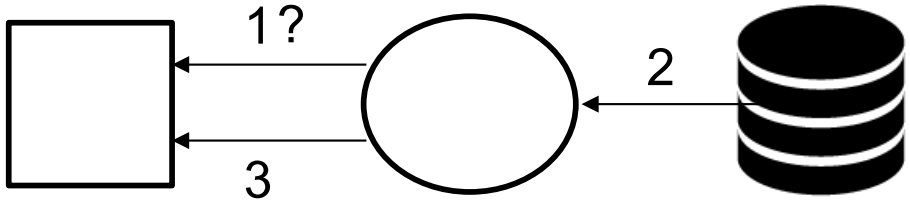
https://www.slideshare.net/ceposta/the-hardest-part-of-microservices-your-data?qid=249549ff-4a01-4951-8565-9fbffceb7805&v=&b=&from_search=2

Caching

- Caching Patterns
  - Look Aside
    - Attempt retrieval from cache
    - Client retrieves from source
    - Write into cache

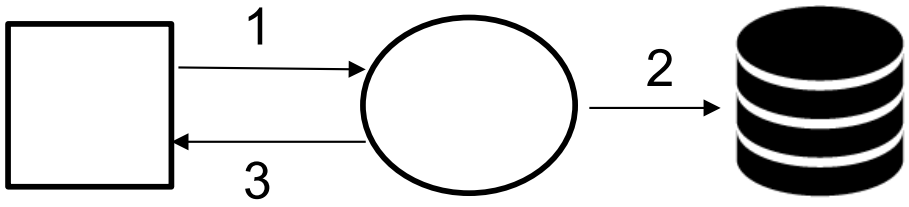


  - Read-through
    - Attempt retrieval from cache
    - Cache retrieves from source and stores in cache
    - Return value to client

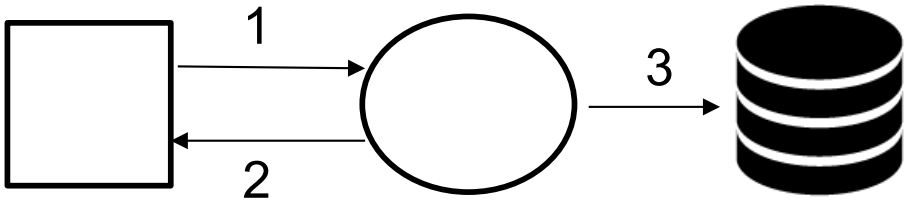


Caching

- Caching Patterns
  - Write-through
    - Write to cache
    - Cache writes to source
    - Ack sent to client



  - Write-behind
    - Write to cache
    - Ack sent to client
    - Cache writes to source asynchronously



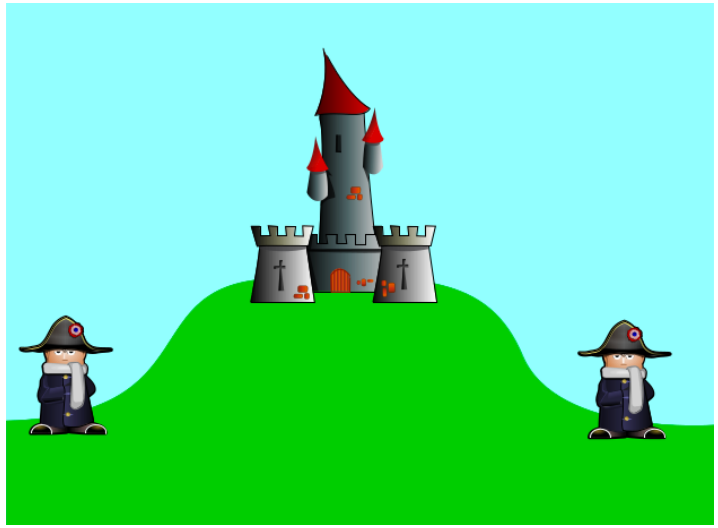
Caching

- Requirements
  - Distributed
    - Over various failure boundaries AZ (Availability Zones), Regions (Data Centers)
  - Data replication
    - Tunable consistency
  - Available
    - Multi-node
    - Recovery process protects against any data loss
  - Scalable
    - In-memory performance
    - Horizontally scalable
  - Ease of Provisioning

- ➔ So perhaps think about the pattern: Cache Warming

Two Generals Problem

- Sooner or later you will face the two generals problem in a microservice architecture
- So you will see that CRUD is the wrong approach for microservices
- It is proven to be unsolvable, so you need another aproach
- In different databases the application cannot simply use a local ACID transaction



https://www.youtube.com/watch?v=holjbuSbv3k

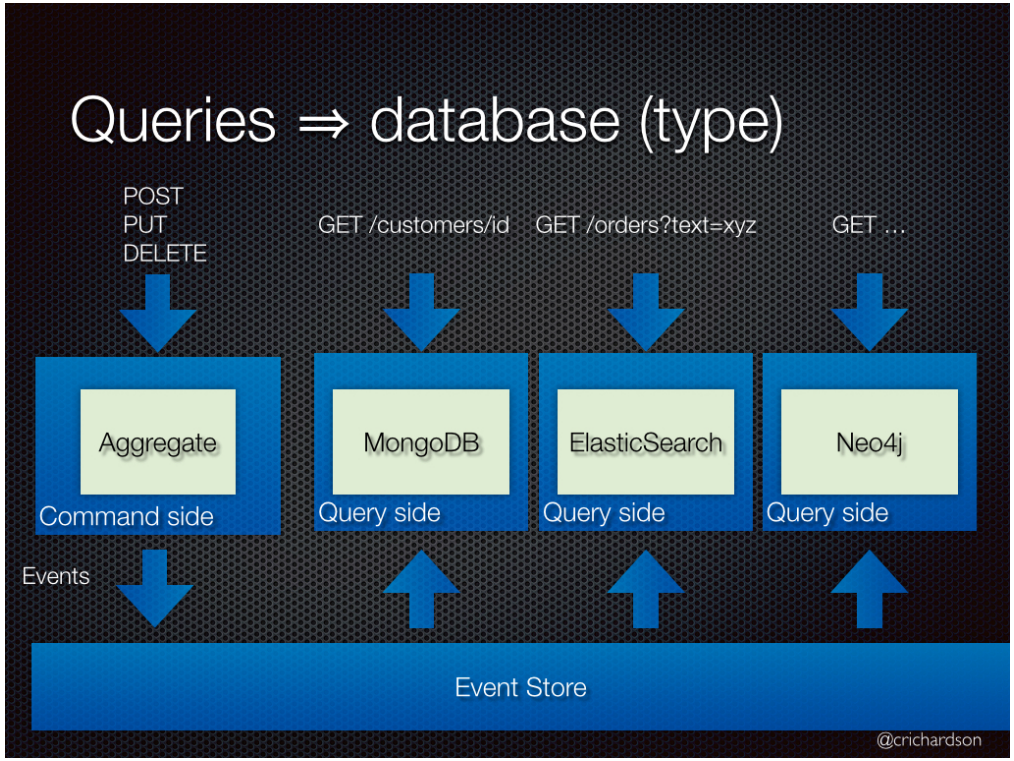
Event Sourcing

- Event sourcing persists the state of a business entity such an Order or a Customer as a sequence of state-changing events. Whenever the state of a business entity changes, a new event is appended to the list of events. Since saving an event is a single operation, it is inherently atomic. The application reconstructs an entity's current state by replaying the events.
- Applications persist events in an event store, which is a database of events. The store has an API for adding and retrieving an entity's events. The event store also behaves like a message broker. It provides an API that enables services to subscribe to events. When a service saves an event in the event store, it is delivered to all interested subscribers.
- Some entities, such as a Customer, can have a large number of events. In order to optimize loading, an application can periodically save a snapshot of an entity's current state. To reconstruct the current state, the application finds the most recent snapshot and the events that have occurred since that snapshot. As a result, there are fewer events to replay.
- ➔ Good Frameworks for doing this: akka persistence or lagom

http://microservices.io/patterns/data/event-sourcing.html

Command Query Responsibility Segregation (CQRS)

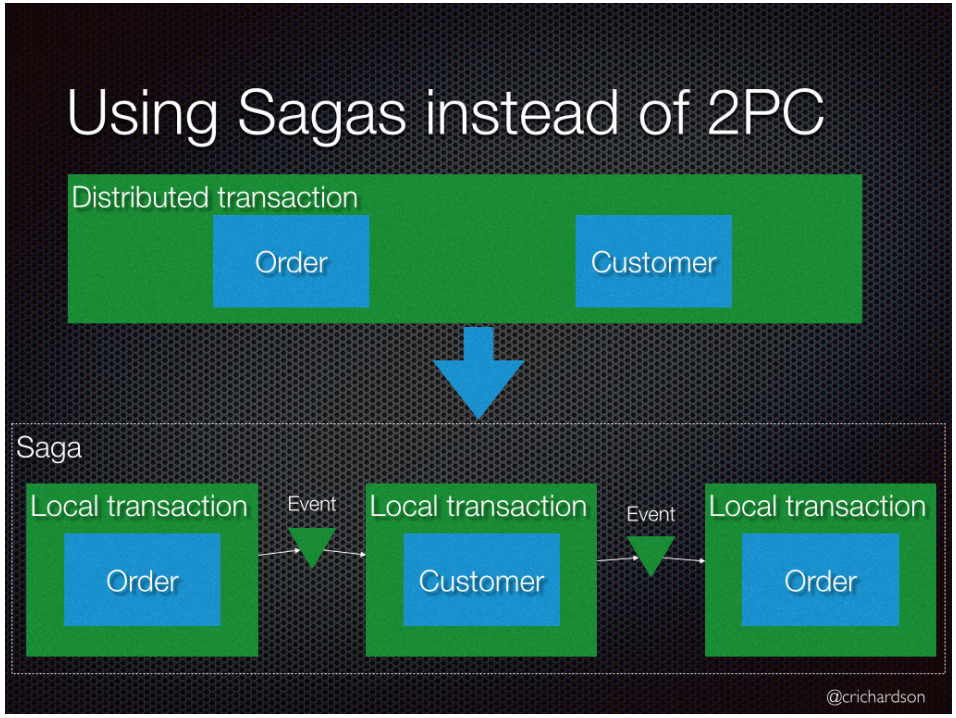
- Split the application into two parts: the command-side and the query-side. The command-side handles create, update, and delete requests and emits events when data changes. The query-side handles queries by executing them against one or more materialized views that are kept up to date by subscribing to the stream of events emitted when data changes.



http://microservices.io/patterns/data/cqrs.html

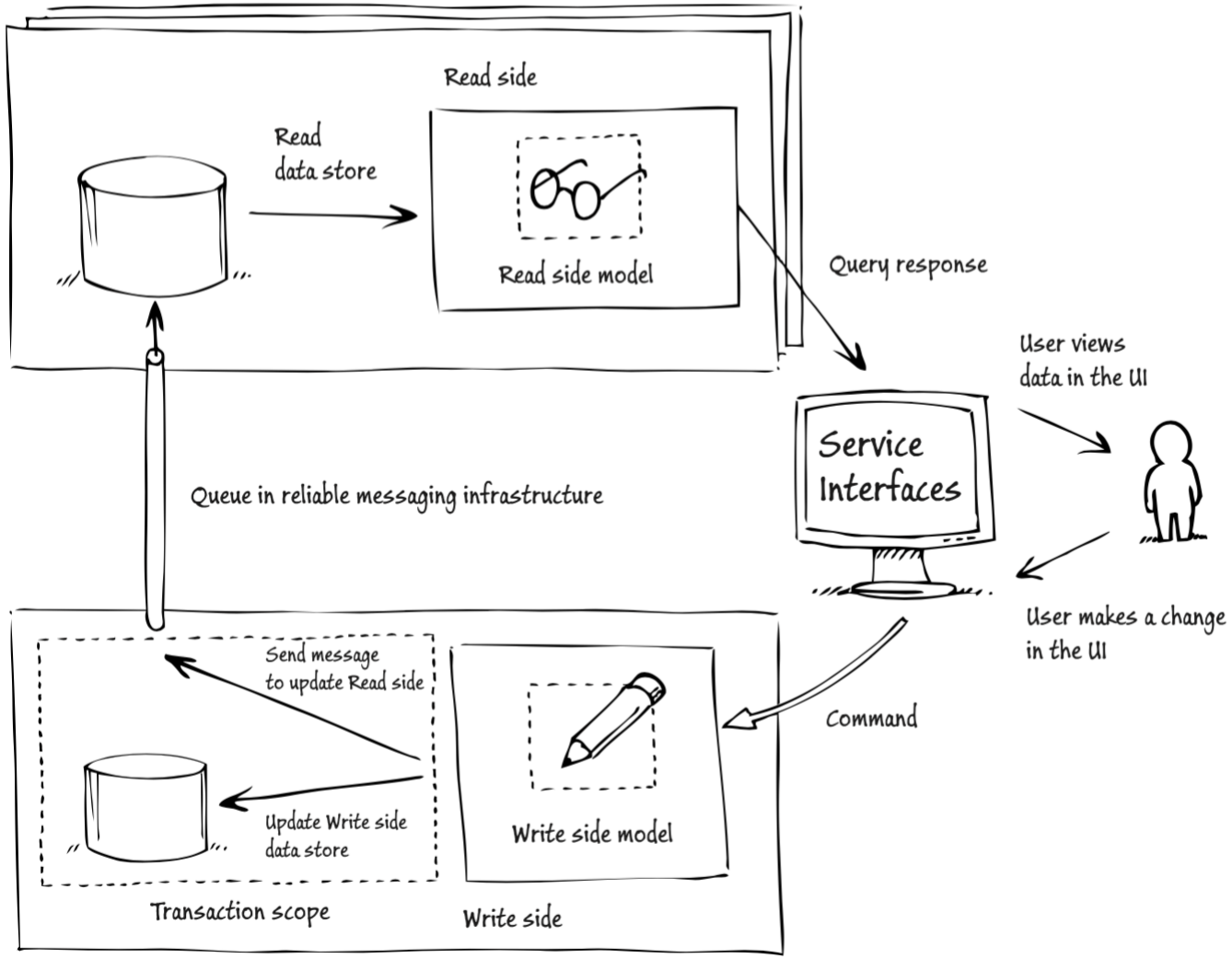
Saga

- Implement each business transaction that spans multiple services as a saga. A saga is a sequence of local transactions. Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga. If a local transaction fails because it violates a business rule then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions.



http://microservices.io/patterns/data/saga.html

Event Sourcing and CQRS



https://msdn.microsoft.com/en-us/library/jj554200.aspx

https://www.slideshare.net/jboner/the-road-to-akka-cluster-and-beyond