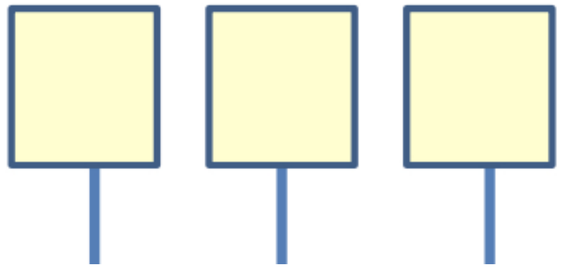
Service Oriented Architecture

- Advantages
  - Independent services
  - Separation of business processes and service logic
  - Architecture is optimized for the business
  - Reuse of services
  - Architecture flexibility



Service Oriented Architecture

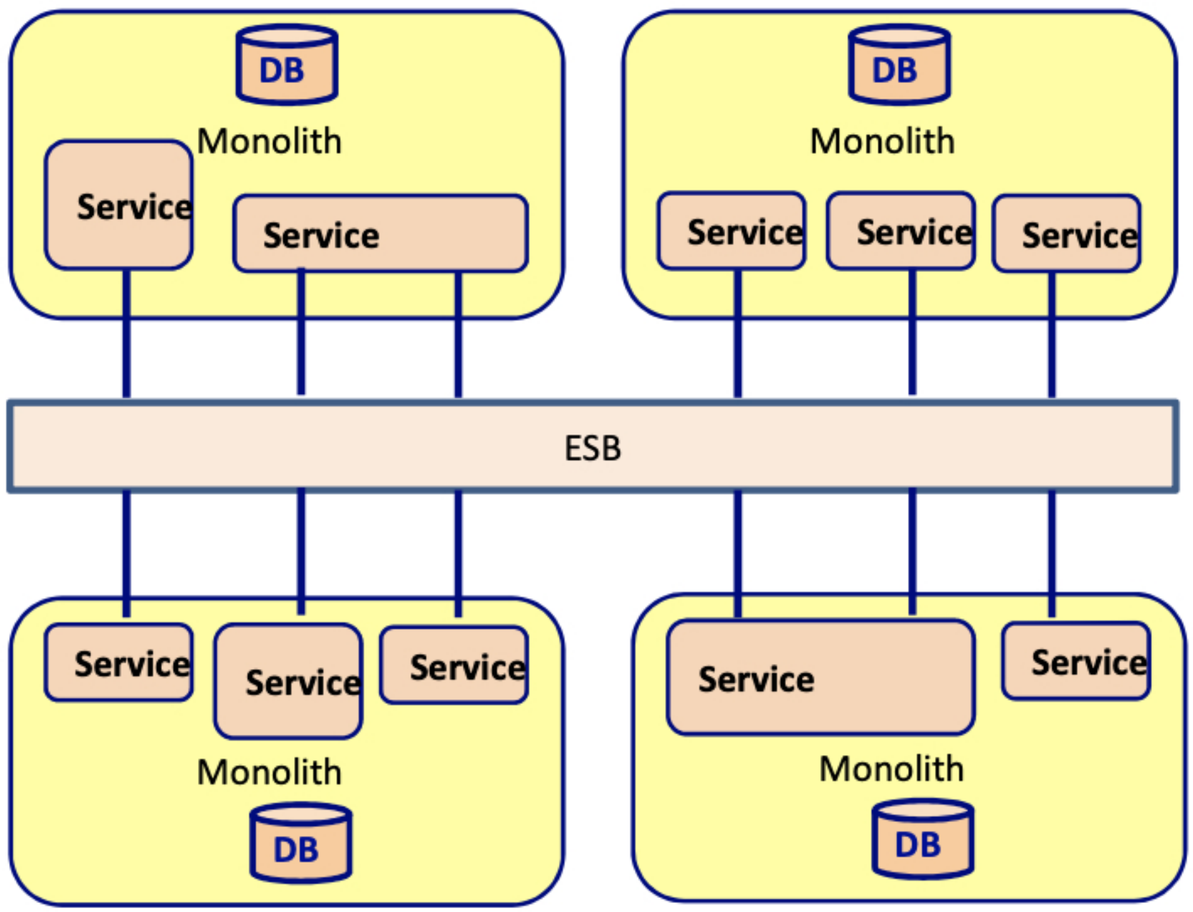
- Disadvantages
  - Complex ESB
  - Changing the business process while still business processes are running is very difficult
  - Most SOA's are build on top of monoliths

Problems with a monolith architecture

- Can evolve in a big ball of mud
- Limited re-use is realized across monolithic applications
- All or nothing scaling
- Single development stack
- Does no support small agile scrum teams
- Deploying a monolith takes a lot of ceremony

SOA vs Microservice

SOA

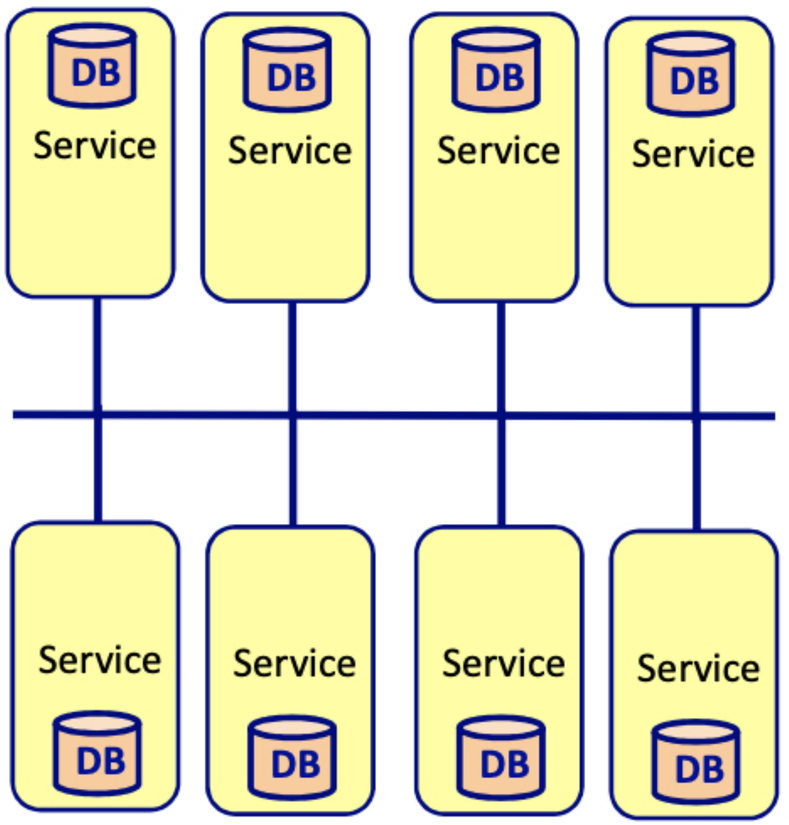


Microservice

small independent service

easy to build, understand and test

do not use enterprise service bus



Microservices

- Small independent services
  - Simple and lightweight
  - Runs in an independent process
  - Technology agnostic
  - Decoupled

Appropriate boundaries

- DDD bounded context
  - Isolated domains that are closely aligned with business capabilities
- Autonomous functions
  - Accept input, perform its logic and return a result
    - Encryption engine
    - Notification engine
    - Delivery service that accept an order and informs a trucking service

Appropriate boundaries

- Size of deployable unit
  - Manageable size
- Most appropriate function or subdomain
  - What is the most useful component to detach from the monolith?
    - Hotel booking system: 60-70% are search request
      - Move out the search function
- Polyglot architecture
  - Functionality that needs different architecture
    - Booking service needs transactions
    - Search does not need transactions

Appropriate boundaries

- Selective scaling
  - Functionality that needs different scaling
    - Booking service needs low scaling capabilities
    - Search needs high scaling capabilities
- Small agile teams
  - Specialist teams that work on their expertise
- Single responsibility

Appropriate boundaries

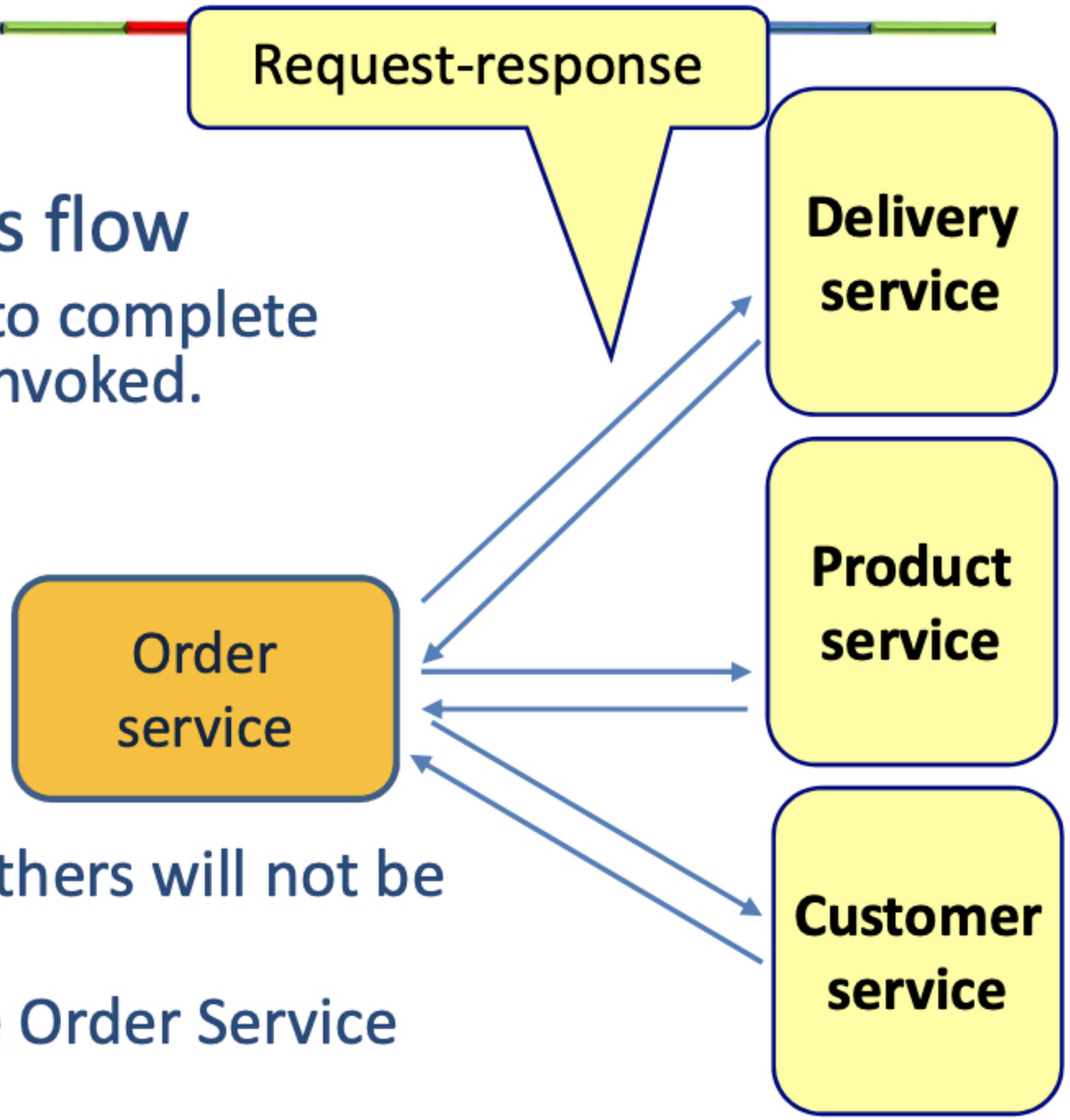
- Replicability or changeability
  - The microservice is easy detachable from the overall system
  - What functionality might evolve in the future?
- Coupling and cohesion
  - Avoid chatty services
  - Too many synchronous request
  - Transaction boundaries within one service

Orchestration

One central brain

- Advantage
  - Full control of the synchronous flow
    - For example, if Service A needs to complete successfully before Service B is invoked.
  - Easy to monitor the process

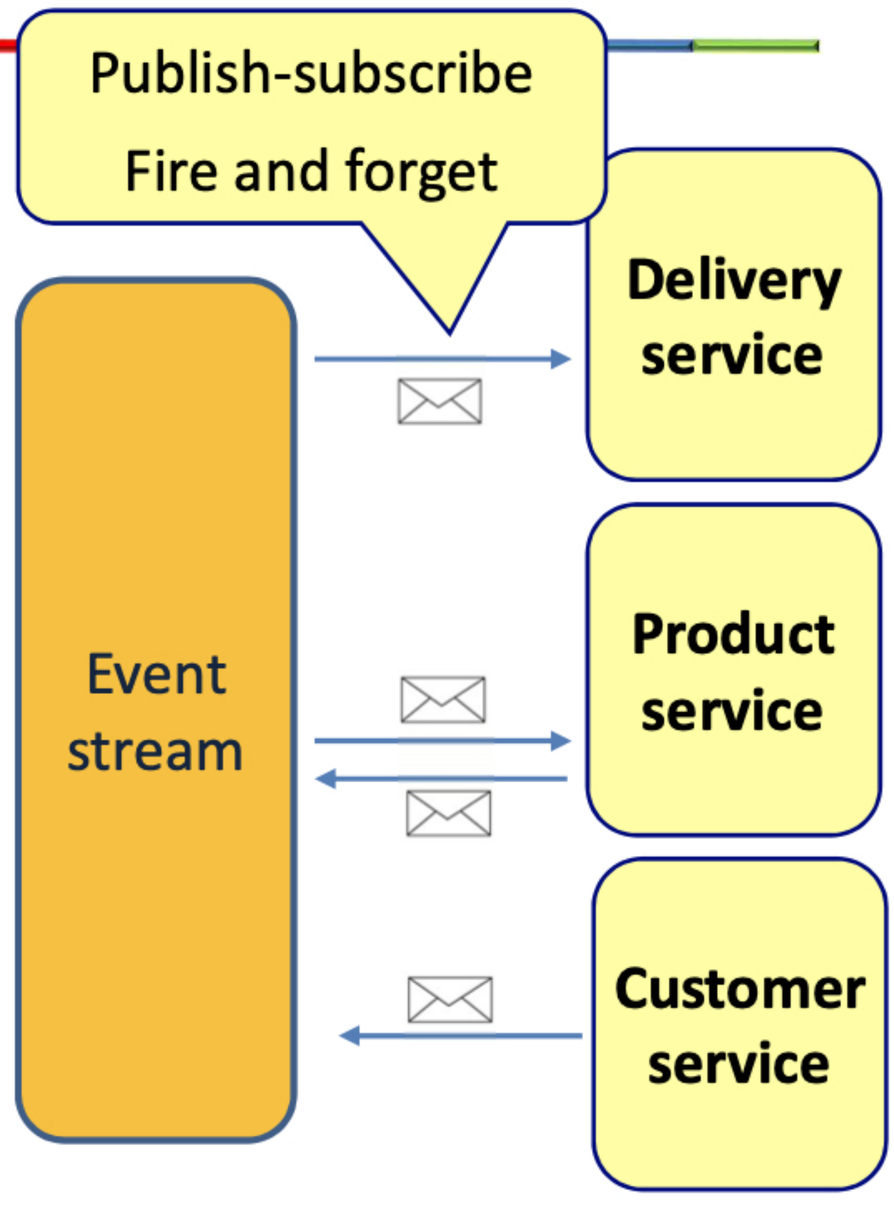
- Disadvantages
  - Coupling
    - If the first service is down, the others will not be called
    - If you add/remove a service, the Order Service needs to change
  - Orchestrator is single point of failure
  - No parallel processing



Choreography

no central brain

- Advantage
  - Less Coupling
    - Easy to add/remove services without impact on other services
    - Fast: parallel processing
    - No single point of failure
- Disadvantages
  - Harder to monitor the process



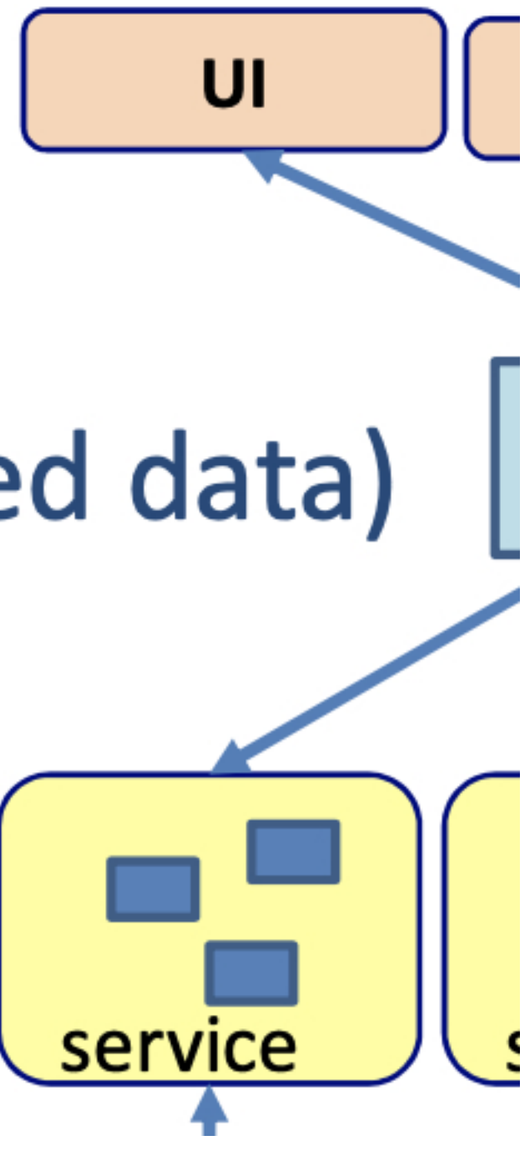
Stateful vs Stateless

- Stateful
  - The service contains in-memory state
  - State is maintained between requests

- Stateless
  - No in-memory data
  - All data is stored outside the service

Stateful

- Advantage
  - Fast
- Disadvantages
  - Synchronization issues (shared data)
  - Hard to scale
    - State need to be replicated



Stateless

- Advantage
  - Database takes care of synchronization
  - Easy to scale
- Disadvantages
  - Performance issue

Service deployment

- Service are written using different languages, frameworks, framework versions
- Run multiple service instances of a service for throughput and availability
- Building and deploying should be fast
- Instances need to be isolated
- Constrain the resources a service may consume (CPU, memory, etc.)
- Deployment should be reliable

NETFLIX

EUREKA

SERVICE REGISTRY: EUREKA

Service Registry

- Like the phone book for microservices
  - Services register themselves with their location and other meta-data
  - Clients can lookup other services
- Netflix Eureka

Why service registry/discovery?

1. Loosely coupled services
   - Service consumers should not know the physical location of service instances.
     - We can easily scale up or scale down service instances
2. Increase application resilience
   - If a service instance becomes unhealthy or unavailable, the service discovery engine will remove that instance from the list of available services.

NETFLIX

RIBBON

LOAD BALANCING: RIBBON

NETFLIX

ZUUL

API GATEWAY: ZUUL

is the enter to the syste, which allow different clients like browser, app, etc to consume microservices

It has the tasks of routing, filterins, logging, security

ZIPKIN

DISTRIBUTED TRACING: ZIPKIN

Distributed Tracing

- One central place where one can see the end-to-end tracing of all communication between services

Zipkin

- Centralized tracing server
  - Collects tracing information
- Zipkin console shows the data

DISTRIBUTED LOGGING: ELK

We need collect all log data from all services to know what has happened. It is centralized

RESILIENCE

The ability to recover from failures

© 2018 ICT Intelligence

Fallacies of distributed computing

- The network is reliable
- Latency is zero
- Bandwidth is infinite
- The network is secure
- Topology doesn’t change
- There is one administrator
- Transport cost is zero
- The network is homogeneous

Resilience patterns

- Timeouts
- Circuit breaker
- Bulkheads

Hystrix thread pool

- Hystrix uses a common thread pool for all remote calls

Hystrix bulkheads

- Hystrix uses a common thread pool for all remote calls

Command Query Responsibility Segregation (CQRS)

- Separates the querying from command processing by providing two models instead of one.
  - One model is built to handle and process commands change the state
  - One model is built for presentation needs (queries)

for strict consistency, CQRS is not good, this cause eventual consistency

Architectural properties

- Command and query side have different architectural properties
  - Consistency
    - Command: needs consistency
    - Query: eventual consistency is mostly OK
  - Data storage
    - Command: you want a normalized schema (3rd NF)
    - Query: denormalized (1st NF) is good for performance (no joins)
  - Scalability
    - Command: commands don't happen very often. Scalability is often not important.
    - Query: queries happen very often, scalability is important

Reactive systems

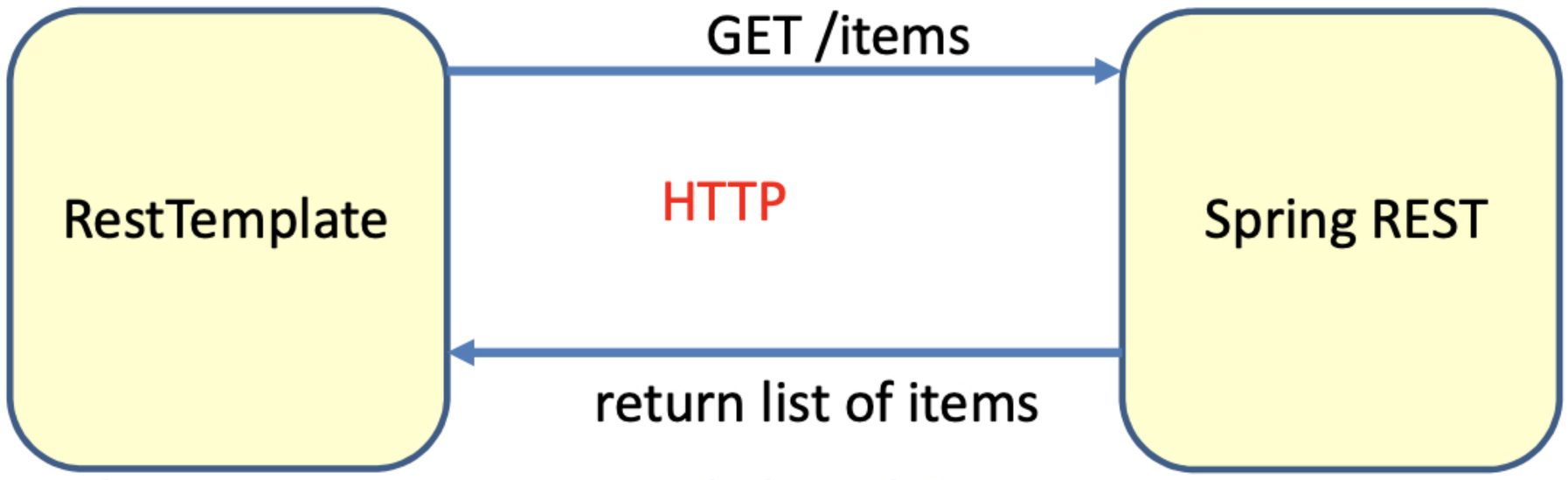
- Advantage
  - Performance
    - No need to wait till all results are available
  - Scaling
    - Less threads needed
- Disadvantage
  - The whole calling stack needs to be reactive
    - Client <->controller<->data access
  - Harder to debug

Spring WebFlux

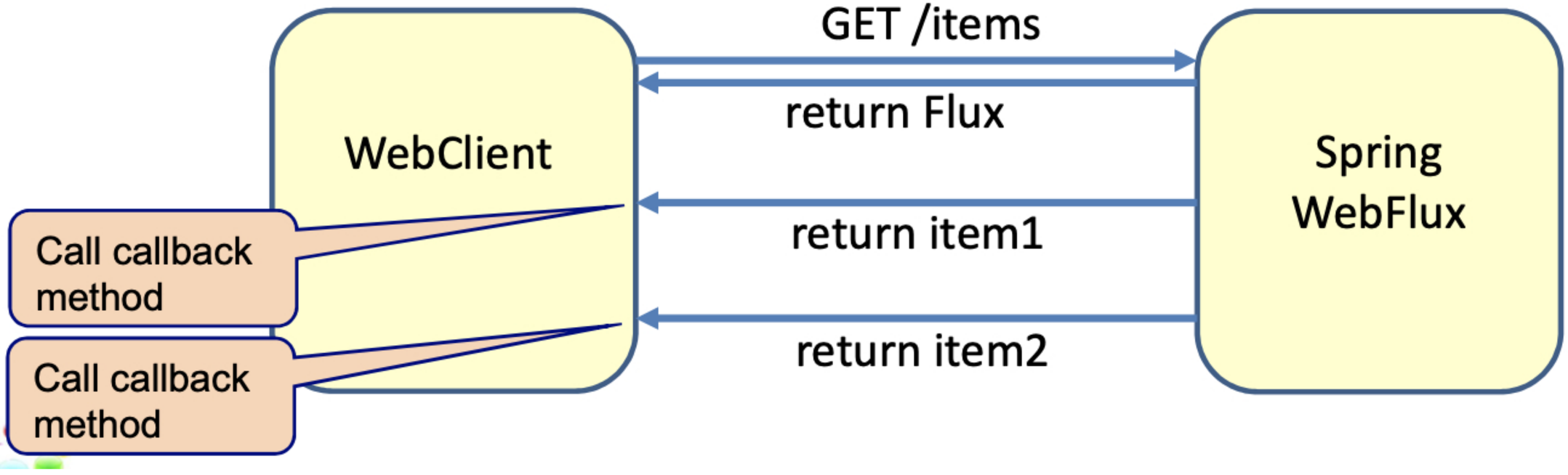
- Allows to build reactive web(REST) applications
- Uses Netty as embedded webserver no tomcat

Reactive Web

- Synchronous, blocking



- Asynchronous, non-blocking



Transactions

- A Transaction is a unit of work that is:
  - **ATOMIC:** The transaction is considered a single unit, either the entire transaction completes, or the entire transaction fails.
  - **CONSISTENT:** A transaction transforms the database from one consistent state to another consistent state
  - **ISOLATED:** Data inside a transaction can not be changed by another concurrent processes until the transaction has been committed
  - **DURABLE:** Once committed, the changes made by a transaction are persistent

Saga with events/choreography

- Advantages
  - More control over transaction
  - Easier to monitor transaction
  - No cyclic dependencies

  work better in simple cases
- Disadvantages
  - Need an extra orchestrator service
  - Orchestrator becomes too complex

Saga is way to manage data consistency across microservices in distributed transactio scenarios

Saga with events/choreography

- Advantages
  - Simple
  - Loosely coupled services

work better in complex cases

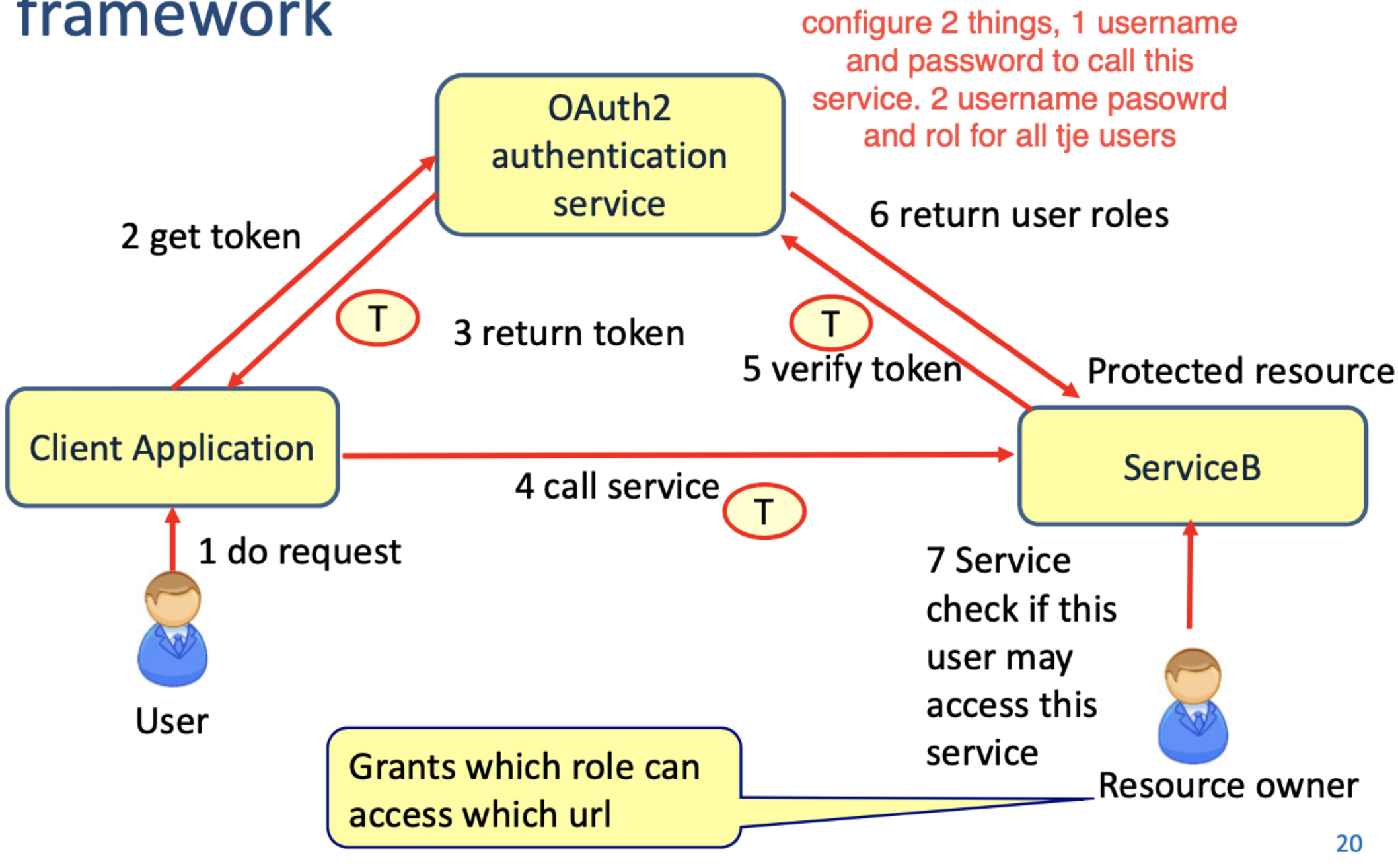
- Disadvantages
  - Difficult to track who listen to which events
  - Possibility of cyclic events

Aspects of security

- Authentication: are you who you say you are?
  - Login with username/password
- Authorization: what are you allowed to do?
  - Make url's and/or methods secure
- Confidentiality: No one may look into this request/response
  - Encryption
- Data integrity: No one may change this request/response
  - Encryption, hashcode,…

How does OAuth2 work

- Token based authentication and authorization framework



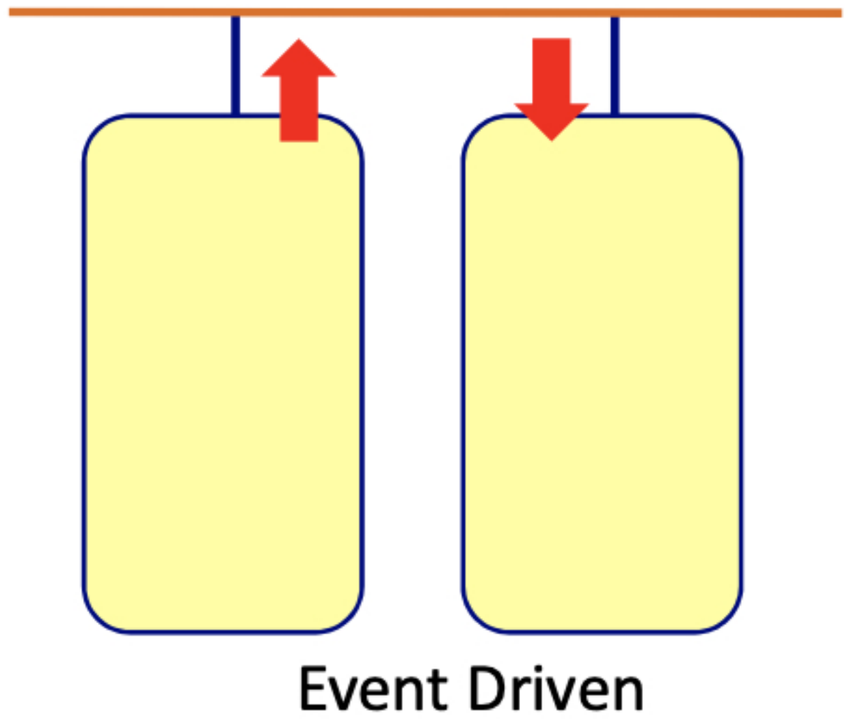
JWT tokens

- Oauth2 is a token based authorization framework bus does provide a standard for tokens
- JWT (JavaScript Web Tokens) provides a standard structure for OAuth tokens
  - Small
  - Cryptographically signed
  - Self contained
  - Extensible

Main point

- With OAuth2 and JWT we can make microservices secure without providing security credentials to the services, and without storing information into sessions
- By transcending into Pure Consciousness one gets access to all intelligence of creation.

Event Driven Architecture (EDA)



- Applications publish events
- Applications subscribe to events

message broker

Apache Kafka

- Created by Linked In
- Characteristics
  - High throughput
  - Distributed
  - Unlimited scalable
  - Fault-tolerant
    - Reliable and durable
  - Loosely coupled Producers and Consumers
  - Flexible publish-subscribe semantics

High Volume:

- Over 1.4 trillion messages per day
- 175 terabytes per day

High Velocity:

- Peak 13 million messages per second
- 2.75 gigabytes per second

Apache Zookeeper

- Maintains metadata about a cluster of distributed nodes
  - Configuration information
  - Heath status
  - Group membership

Partition

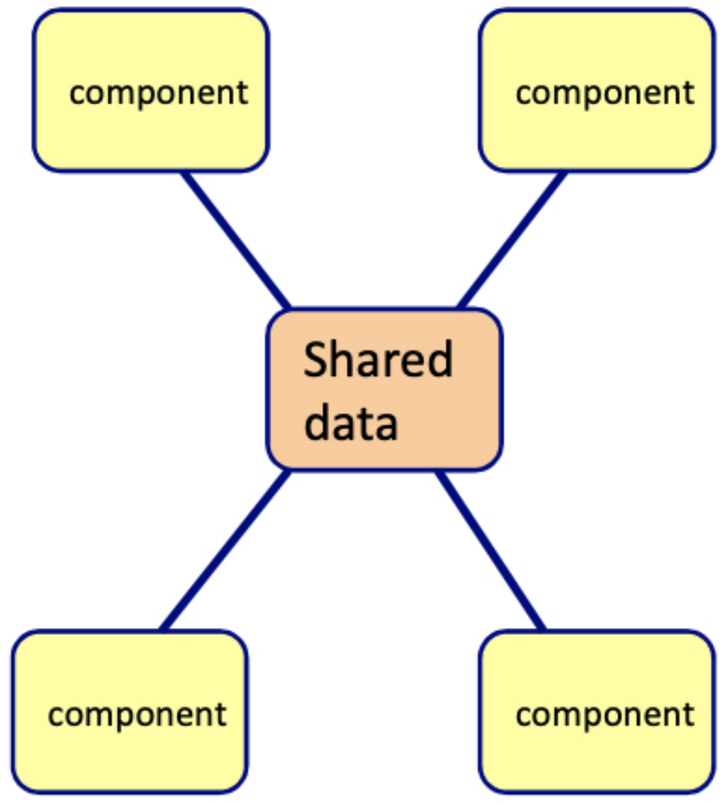
- Each topic has one or more partitions
  - This is configurable
- Each partition is maintained on 1 or more brokers

Blackboard pattern

- Used for non deterministic problems no algorithm way to solve
  - There is no fixed straight-line solution to a problem
- Every component adds her information on the blackboard

Blackboard

- Common data structure
  - Extension is no problem
  - Change is difficult
- Easy to add new components
- Tight coupling for data structure
- Loose coupling for
  - Location
  - Time
  - Technology(?)
- Synchronisation issues



Blackboard

- Benefits
  - Easy to add new components
    - Components are independent of each other
    - Components can work in parallel
- Drawbacks
  - Data structure is hard to change
    - All components share the same data structure
  - Synchronization issues

Event sourcing

- Instead of storing the state of an entity in a database, you store the series of events that lead up to the state.
- Storing all of the events increases the analytical capabilities of a business.
- Instead of just asking what the current state of an entity is, a business can ask what the state was at any time in the past

Event sourcing

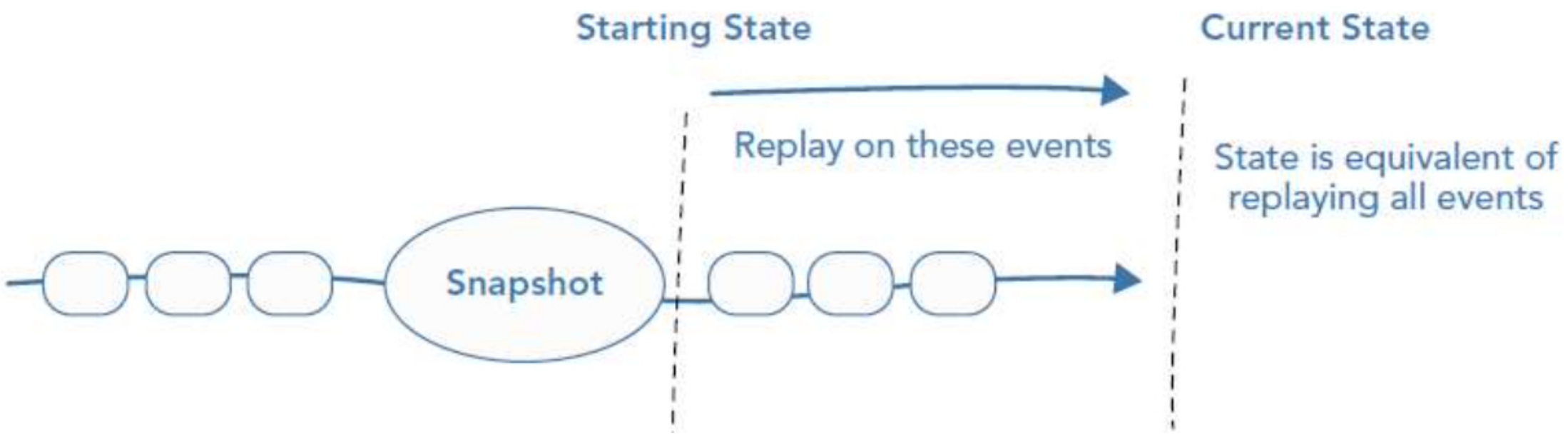
- For each aggregate
  - Identify (state changing) domain events
  - Define event classes
- Example:
  - Shopping cart
    - ItemAddedEvent
    - ItemRemovedEvent
    - CheckedOutEvent
  - Order
    - OrderCreated
    - OrderApproved
    - OrderShipped

Advantages of storing events

- You don’t miss a thing
  - Business can analyze history of events
  - Bugs can be solved easier
- Can be replayed
- Events are immutable

Snapshots

- Intermediate steps in an event stream that represent the state after replaying all previous events
  - Can increase performance when streams are very long



Stream based systems

- Continuous stream of data
  - Stock market systems
  - Social networking systems
  - Internet of Things (IoT)systems
  - Systems that handle sensor data
  - System that handle logfiles
  - Systems that monitor user clicks
  - Car navigator software

But also

- Stream of purchases in web shop
- Stream of transactions in a bank
- Stream of actions in a multi user game
- Stream of bookings in a hotel booking system
- Stream of user actions on a web application
- …

Batch processing

- First store the data in the database
- Then do queries (map-reduce) on the data
- Queries over all or most of the data in the dataset.
- Latencies in minutes to hours

Stream processing

- Handle the data when it arrives
- Handle event (small data) by event
- Latencies in seconds or milliseconds

Stream based architecture

Works good for applications with:

1. high volume data
2. high frequency changes

Stream

- Stream is just a sequence of events
- Implemented with a distributed messaging system
  - Kafka
  - MapR