Zurich University
of Applied Sciences

School of
Engineering
InIT Institute of Applied
Information Technology

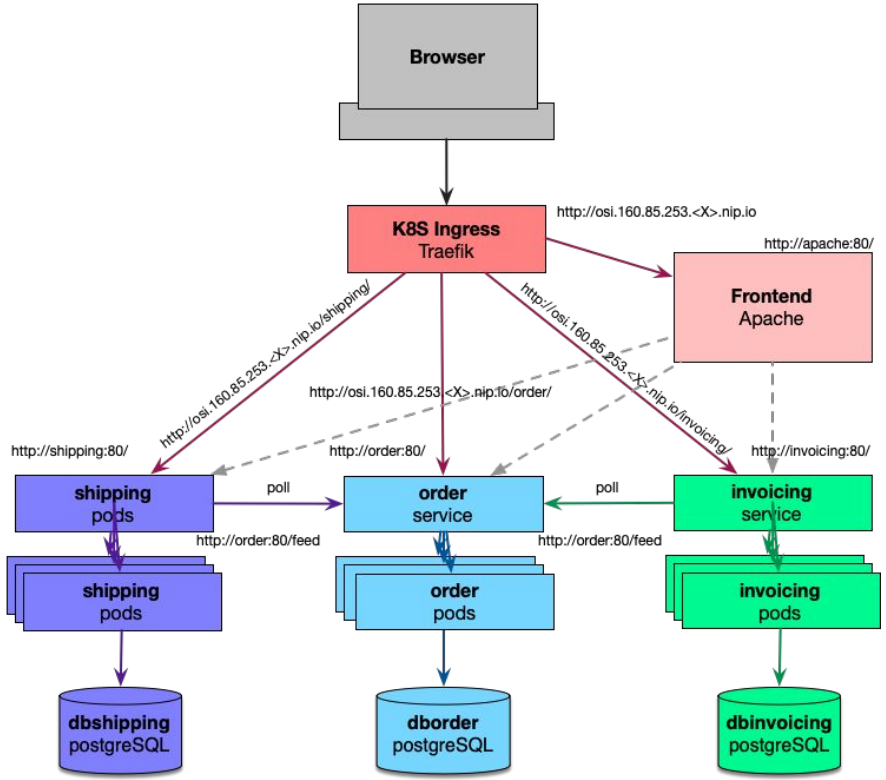
CNA2 - Cloud Native Applications
Cloud Patterns

Prof. Dr. Thomas M. Bohnert
Christof Marti

Patterns for Cloud-Native Applications

- Cloud-Native (CNA) Application design is a consequence resulting from principles of Cloud Computing Service Models and the basic tenets of Distributed Systems.
- CNA design can be applied to software architecture by implementing specific Cloud Computing architectural patterns
- In other words, an application is cloud-native if it implements Cloud Patterns
- Cloud Patterns are
  - Service Registry, Circuit Breaker, Load Balancer, API Gateway, Endpoint Monitoring (Health, Metering), and many more ...

Order-Shipping-Invoicing Example App (see lab)

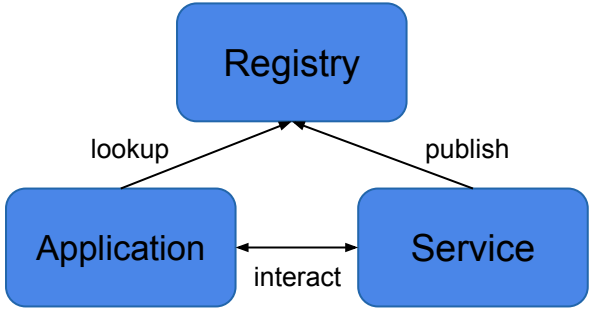


Service Registry

- Service-Orientation at scale means using/running many services
  - Challenge is how to keep track of them
- A **Service** has
  - A (business) functionality
  - A configuration space
  - An interface, a protocol, plus an information model
  - Operational features and requirements
  - 0-N service instances and their respective state
- A **Service Instance** is an instantiation of a service
  - A Service Instance has a state (functional and operational)
  - State? The business function may be implemented stateless but the service itself has a state (sleeping, runnable, running, terminated, etc.)

Service Registry

- A Service Registry maintains state information of Service Instances
  - What type of Service provided by a Service Instance
  - What is the function offered
  - How to reach a Service Instance
  - How to talk to/use the Service Instance
  - What is the state of the Service Instance
- Service Instances register (create) initial state at startup and maintain it (update) until disposal of the Service Instance (delete)



Technologies implementing a Service Registry function

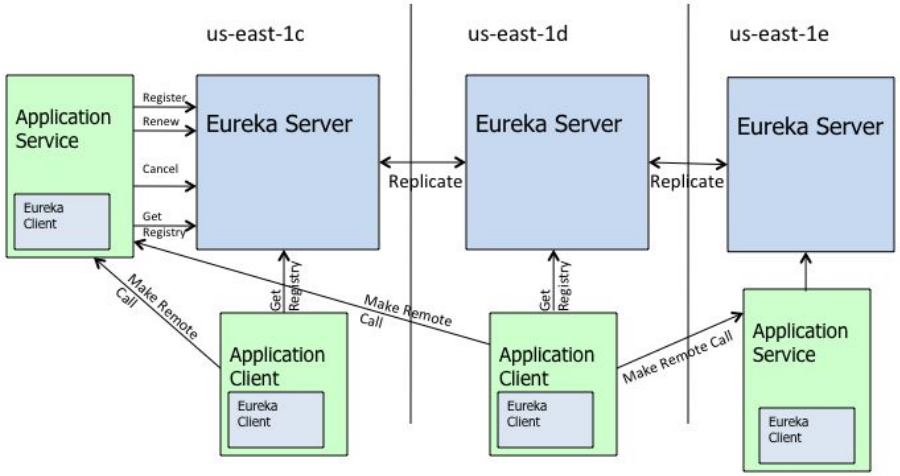
- Consul - consensus
- Zookeeper - consensus
- Netflix Eureka - broadcast
- OpenStack Keystone - master/slave
- Serf - gossip
- **Etcd - consensus**

Service Registry Technology: Netflix Eureka

NETFLIX OSS

https://github.com/Netflix/eureka/wiki

- REST based service registry
  - Server & Client side library provided for Java
- Designed for:
  - service (instance) registration and lookup
  - defined data structure (https://github.com/Netflix/eureka/wiki/Eureka-REST-operations)
- Operational features
  - replication of server configuration
  - easy integration with Spring Framework (https://cloud.spring.io/spring-cloud-netflix)
- Actions:
  - register
    service instance registers its connection endpoint and data
  - renew
    service instance does renew using a heartbeat mechanism
  - cancel
    service instance explicitly informs the registry it is shutting down
  - getRegistry:
    client is looking up service (instance) information



Service Registry Technology: Netflix Eureka

Netflix Eureka offers a REST-based interface and a Java based server & client library

- register new instance
  - `curl http://$HOST:8888/eureka/v2/apps/<serviceID> -X POST -d {....}`
- renew instance (heartbeat)
  - `curl http://$HOST:8888/eureka/v2/apps/<serviceId>/<instanceID> -X PUT`
- cancel instance
  - `curl http://$HOST:8888/eureka/v2/apps/<serviceId>/<instanceID> -X DELETE`
- lookup service instance info (getRegistry)
  - All instances of all services
    `curl http://$HOST:8888/eureka/v2/apps -X GET`
  - All instances of a specific service
    `curl http://$HOST:8888/eureka/v2/apps/<serviceId> -X GET`
  - Info for a specific instance
    `curl http://$HOST:8888/eureka/v2/apps/<serviceId>/<instanceID> -X GET`

https://github.com/Netflix/eureka/wiki/Eureka-REST-operations

Service Registry Technology: etcd

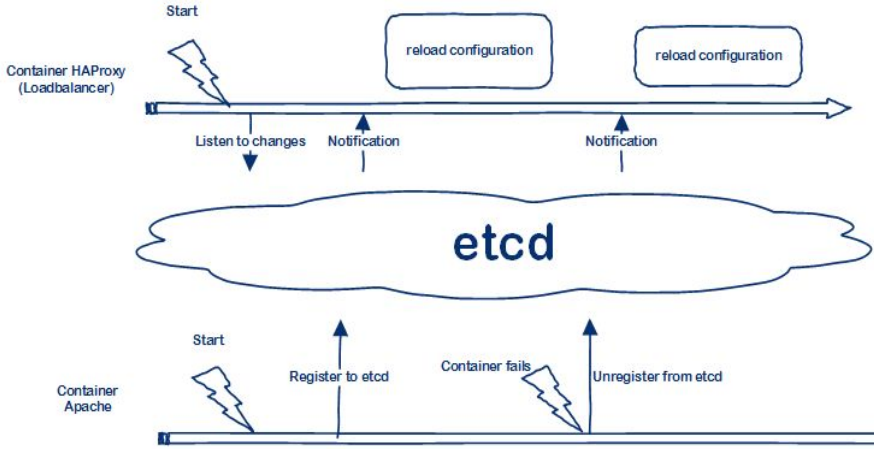
- etcd: Distributed key value store
- Designed for:
  - Any Key-Value Content
    - Service Instance Information and Configuration
- Operational features
  - Can be run as highly-available cluster
  - Raft consensus algorithm
    - master election, re-election due to machine failures
- Usage features
  - Actions: read, write, listen
  - Data expiry mechanism
  - Notification mechanism
  - Shallow data structure to organize keys into folders
    - /folder
    - /folder/key

https://github.com/coreos/etcd

Service Registry Technology: etcd

An etcd-Instance/cluster offers a REST-based interface and a command-line tool named etcdctl

- create/write and read a value using the REST interface directly
  - `curl http://$HOST:2379/v2/keys/message -XPUT -d value="Hello world"`
  - `curl http://$HOST:2379/v2/keys/message -XGET`
- read/create directory using command line tool etcdctl
  - `etcdctl mkdir /folder`
  - `etcdctl ls /folder`
- listen to changes, using one of both means
  - `curl http://$HOST:2379/v2/keys/message?watch=true -XGET`
  - `etcdctl exec-watch /folder/key -- /bin/bash -c “touch /tmp/test”`



Kubernetes DNS Service

https://www.digitalocean.com/community/tutorials/an-introduction-to-the-kubernetes-dns-service

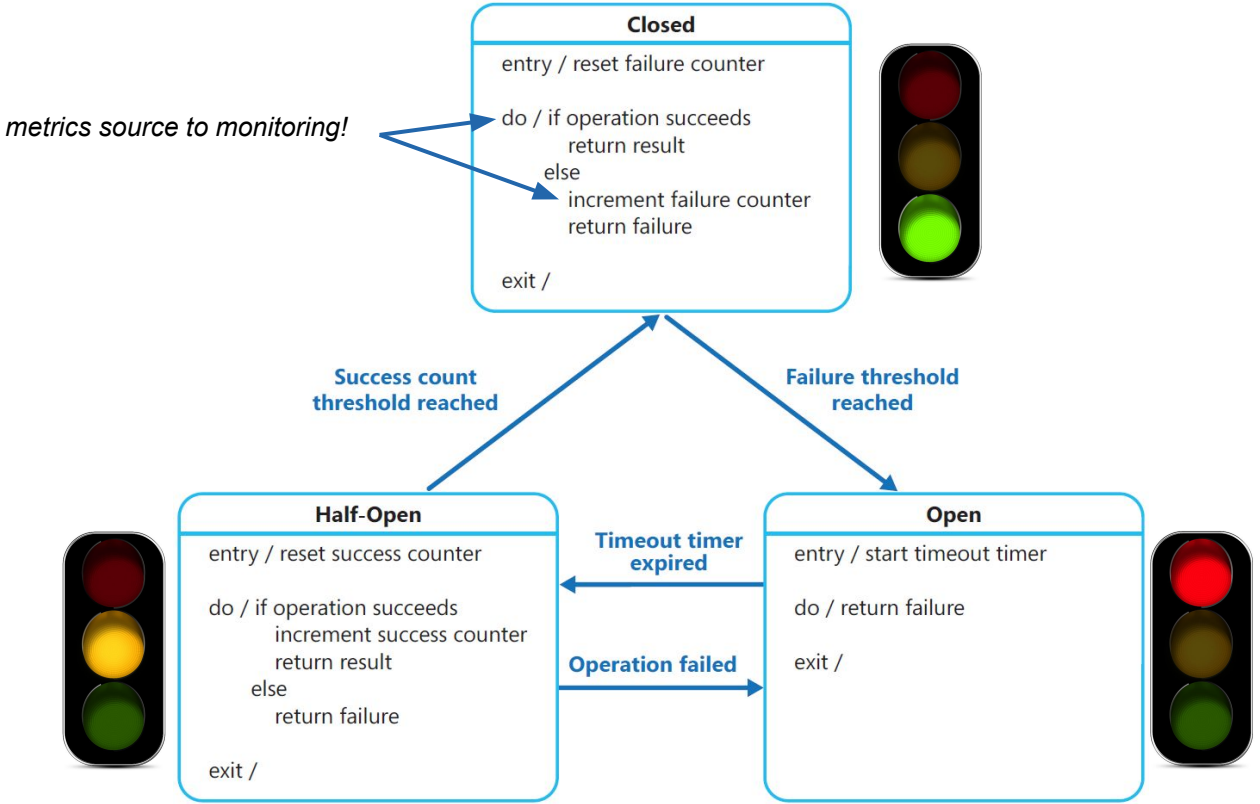
https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/

Circuit Breaker

- Service-Orientation at scale means using/running many services
  - **Challenge is to deal with unavailable or unresponsive services**
    - **Unavailable/Unresponsive: Service may not exist at all/ Service does not reply**
- CNA should be able to handle unreliable dependencies
- A temporarily unavailable service (down, network slow, overloaded, etc) should have minimal influence on a CNA.
- Conceptual idea:
  - Consumer perspective: Detect failure fast, temporarily hold requesting, do not block indefinitely and allow invoking service to react
  - Provider perspective: Amount of requests will be limited in case of a failure, allows provider to recover.
- Approach
  - Establish an intermediate that mediates request intensity between consumer and provider

Circuit Breaker

Circuit Breaker: State Machine



Circuit Breaker: State Machine

Closed:

- Circuit Breaker will pass every request.
- If a request fails a failure counter will be increased.
- If the failure counter surpasses a certain threshold the circuit breaker will change its state to open.
- The failure counter will be reset after a certain amount of time.
- (Prevents state from changing to open if only sporadic failures occur).

Open:

- Circuit Breaker will immediately return an error when asked to forward a request.
- After a certain amount of time it will change to the half-open state.
- Another possibility is that the circuit breaker sporadically sends some request/pings by himself to see if the service is responsive again and change to the half-open state after confirming it.

Half-Open:

- Circuit Breaker will let some request pass while still responding with an immediate error to most requests. The reason for this is to not just overwhelm a maybe recuperating service.
- It will count the successful requests and will change its state to close if a certain amount of successful requests have passed.
- If a request fails the state will immediately change to open again.

Load Balancer

- Service-Orientation at scale means using/running many services
  - **Challenge is to implement elasticity**
- CNA should be able to scale up and down with increasing/decreasing load, i.e. scale horizontally or vertically
- Horizontal scaling (elasticity) means to add/remove parallel resources (Service Instances) as load increases/decreases
- A consequence of this is the requirement to distribute current load to a pool of Service Instances at a certain point in time
- A Load Balancer is a mechanism to distribute the aggregate load to the individual members a pool of Services Instances according to a certain distribution algorithm

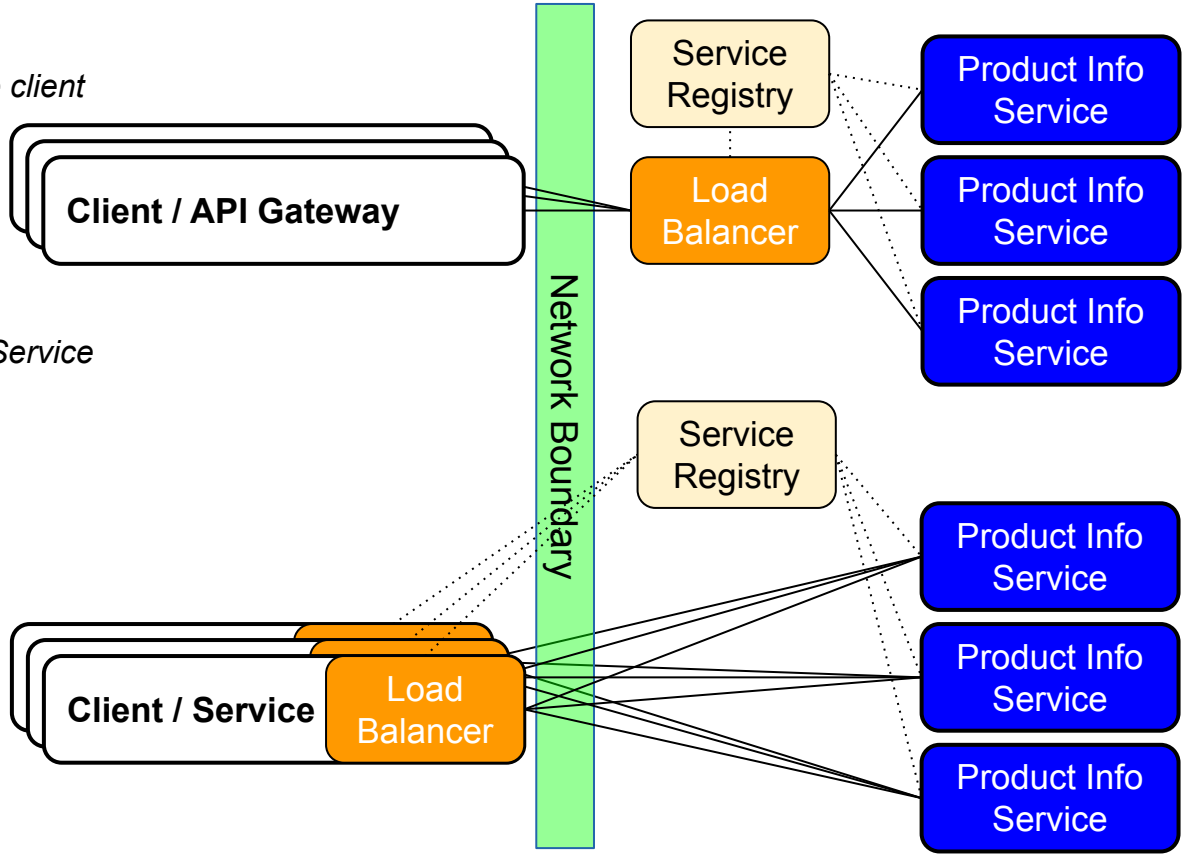
Load Balancer

Server-side Load Balancer:

- *A "Service" used **transparently** by the client*
- *Runs in separate process*
- *Shared by clients*
- *Often shared for multiple services*
- *Possible bottle-neck*
- *Enterprise: hardware, Cloud: software*
- ***Examples**: HAProxy, nginx, F5, K8S-Service*

Client-Side Load Balancer:

- *A "Library" within the client*
- *Runs in client process*
- *Client decides which service instance to connect to*
- *Required to keep in sync (needs a registry)*
- ***Examples**: Netflix Ribbon*



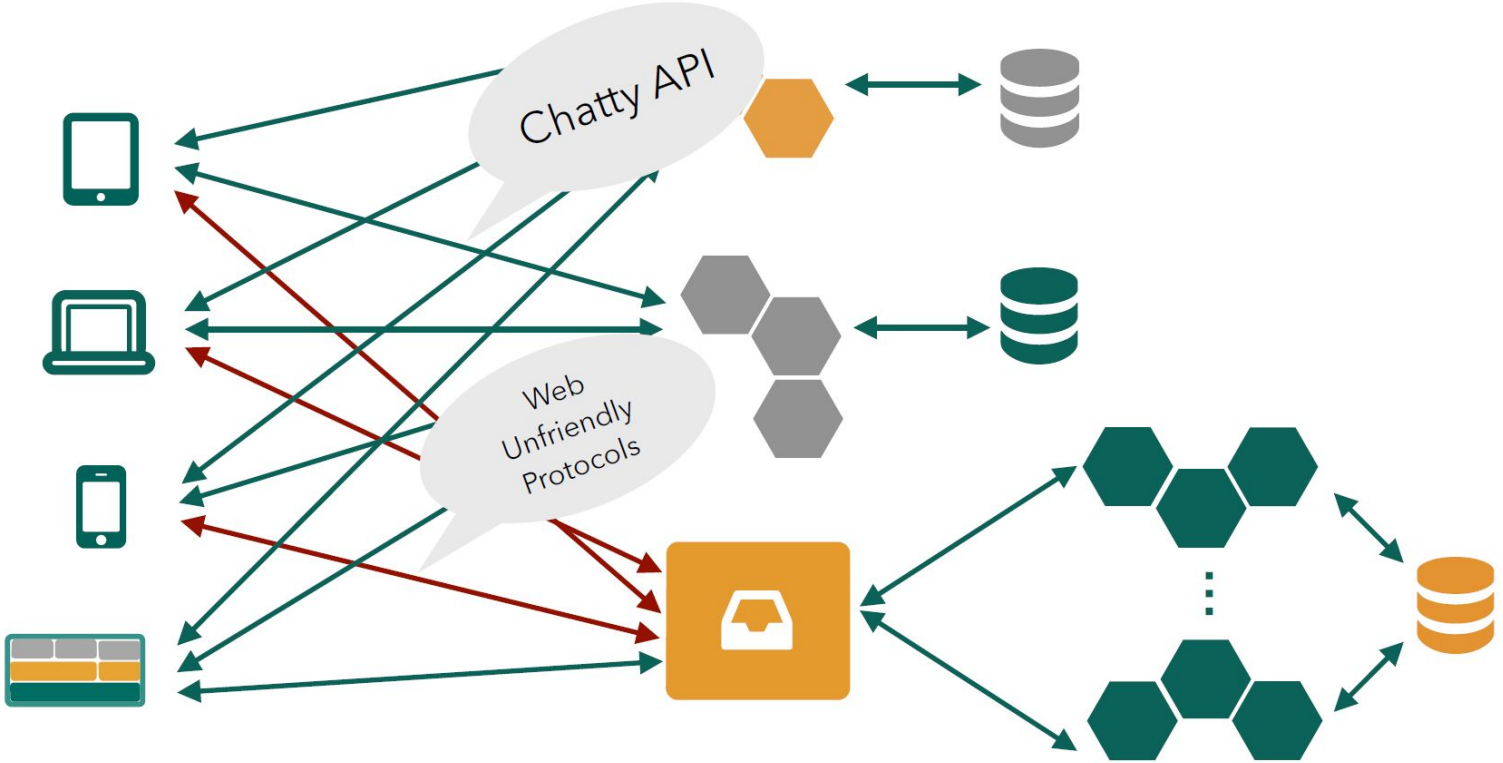
Load Balancer

- There are multiple distribution algorithms in accordance to different goals, e.g:
  - Fairness, equal load
  - Speed, unequal load distribution in relation to capacity planning
  - Economics, unequal load distribution to save expenses
- Some popular algorithms:
  - Round-Robin: Distribute load equally by assigning request to Service Instances in turns (fairness)
  - Least-Connection: Request is assigned to Service Instance with the least on-going requests (performance)
  - Source: Request is assigned to same Service Instance as former request (stickiness)
- Theoretical limit to the number of replicas behind a LB
  - Amdahl's Law, Universal Scaling Law (Gunter)

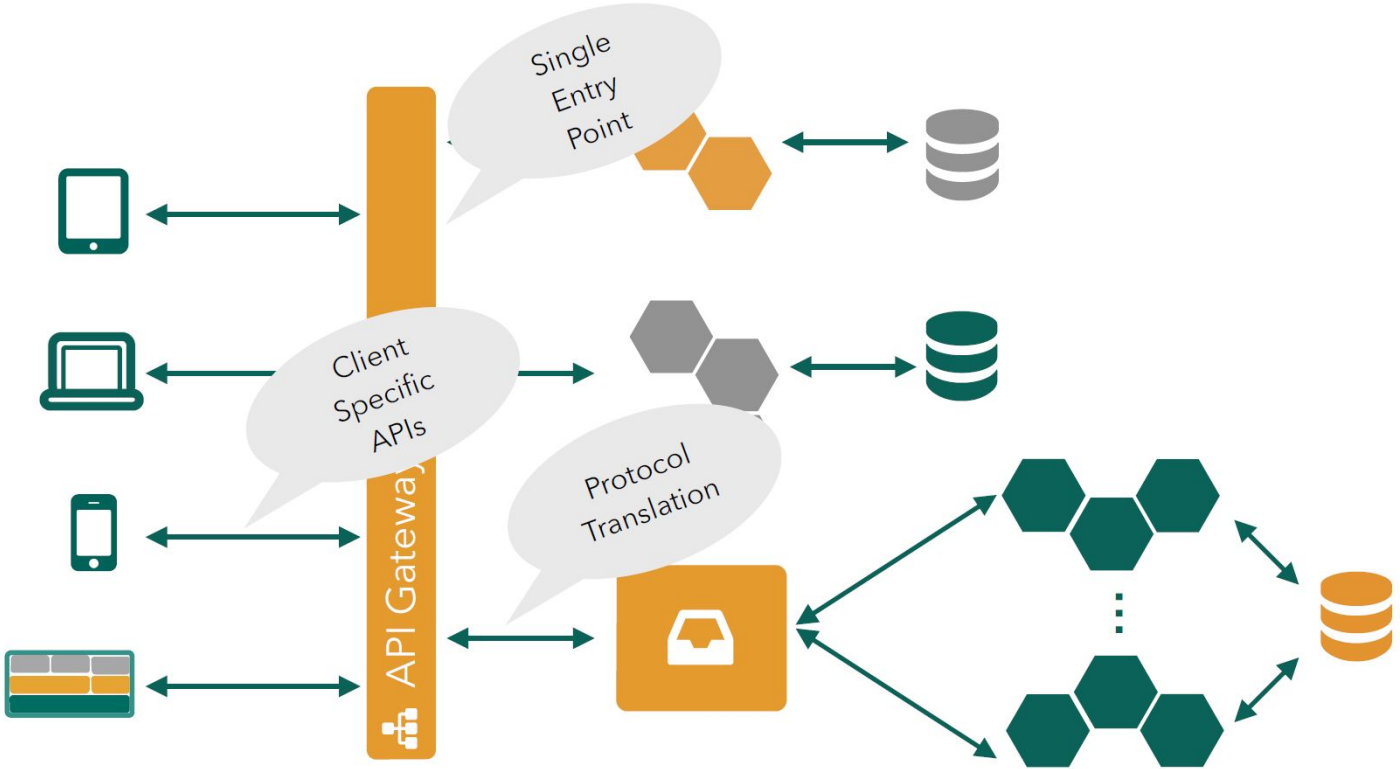
API Gateway

- Service-Orientation at scale means using/running many services
  - **Challenge is how to present them consistently to consumers**
- Each service is providing specific endpoints and interfaces
- For consumers a CNA should provide a clear and consistent interface.
- Internal structure of the CNA should not be exposed to the consumers
- Approach:
  - Implement a gateway service providing one unified interface
  - It forwards requests to the respective internal service
  - Implementation of the facade pattern (GOF) for distributed systems
- Advanced usage
  - provide client-specific variants (Browser, Mobile App, IoT devices)
  - aggregate responses from concurrent requests to internal services
  - convert from / to consumer unfriendly protocols

Without API-Gateway



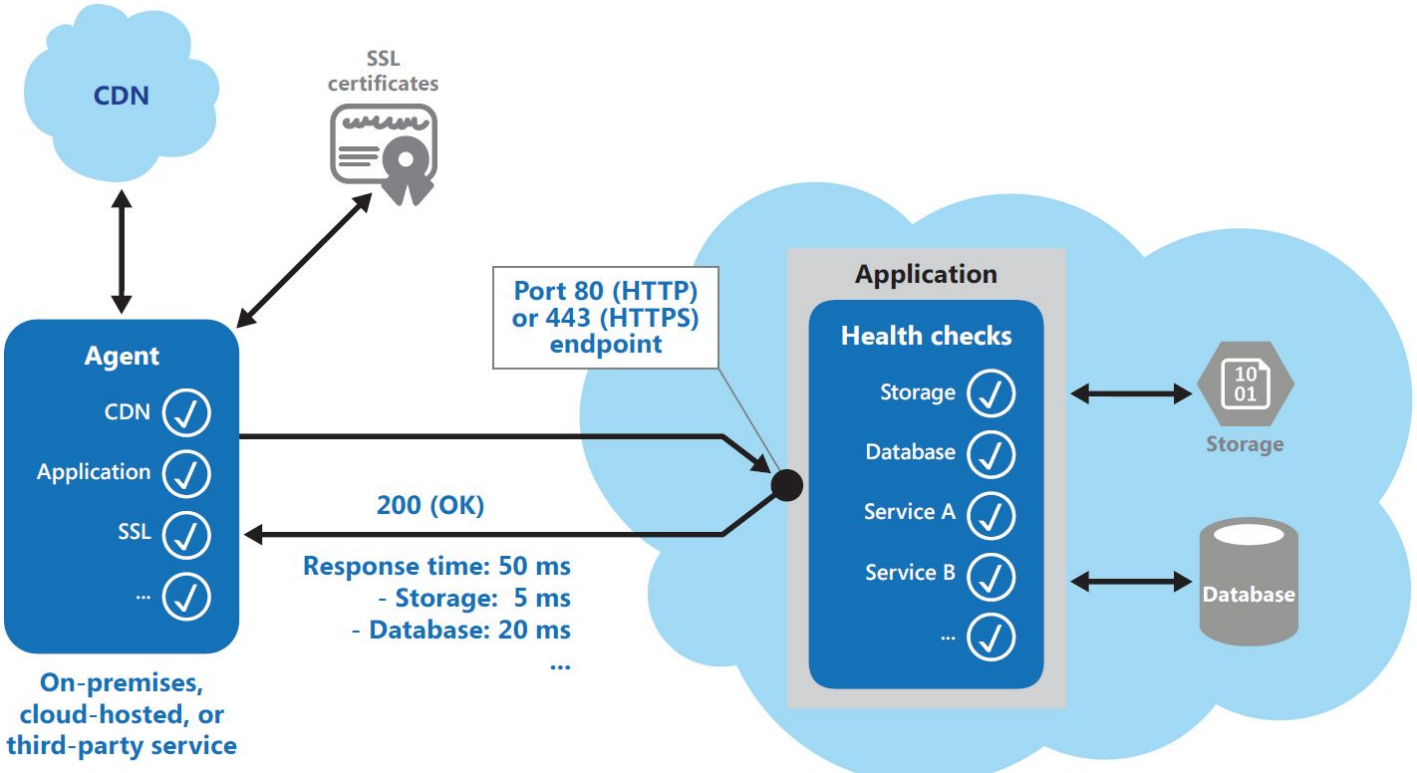
With API Gateway



Endpoint Monitoring

- Service-Orientation at scale means using/running many services
  - **Challenge is to track operational status**
- How to detect/report the status of your application?
- Even if the process is still running, the application could be crashed or stuck
- Conceptual Idea:
  - CNA implements functional checks and instances make specific metrics available through a dedicated verification interface
  - External Tool gathers and analyses the metrics and decides the health status
- Typical Checks
  - Verifying internal process status (resource consumption, load, ...)
  - Checking presence of external backing services (e.g Database, CDN,...).
  - Measuring the response times (network latency, application processing time)
  - Checking for expiration of SSL certificates.
  - Validating the (HTTP) response code.
  - Checking the content of the response

Endpoint Monitoring



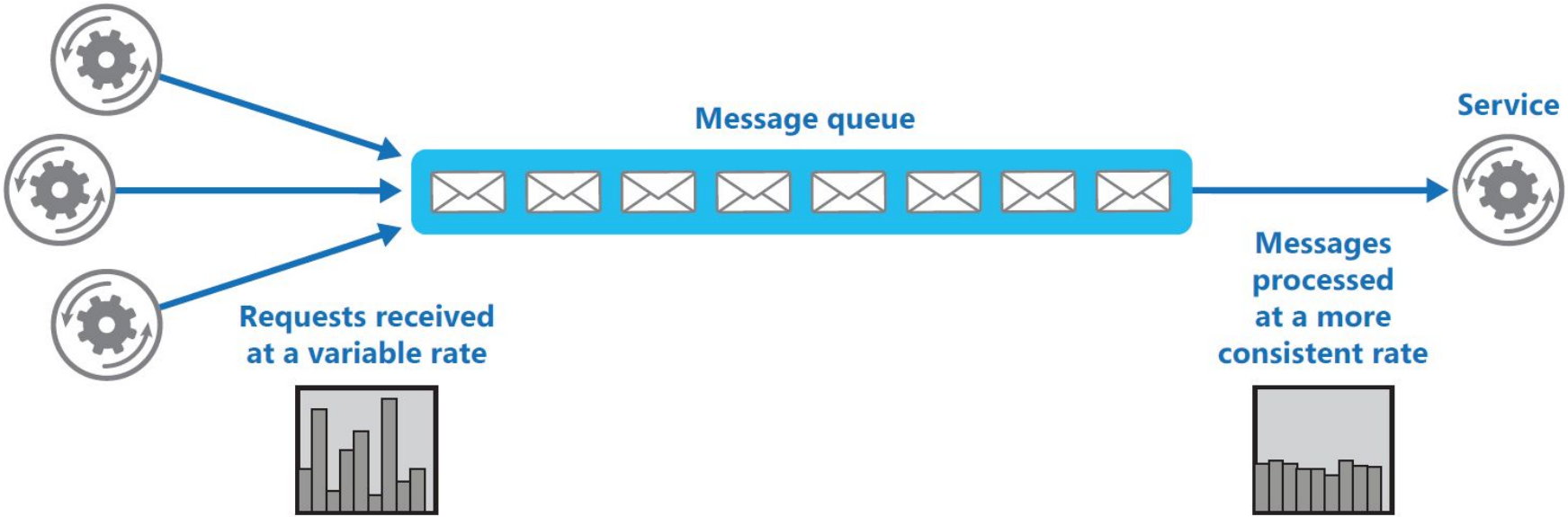
Health Manager

- Applications may fail at any time. Hardware defect, VM failure, programming error like memory leaks, etc.
  - **Challenge is how to make sure, that all the desired number of instances of each programs and services are operational?**
- Conceptual Idea
  - Monitor the application processes and status (health) (e.g. by checking the health endpoints)
  - Assures that the set of deployed resources for a service are:
    - consistent with the desired state of the service and
    - functioning correctly
- Implementation
  - It uses a specification of the “desired state” and compares it with the "actual state"
  - It automatically restarts failed components
- Examples
  - Fleet (Docker)
  - Kubernetes - Deployments/Deployment Controller
  - CloudFoundry - Diego (BBS&CellRep)

Queue Load-leveling

- Service-Orientation at scale means using/running many services
  - **Challenge is to deal with variable levels of load**
- Load (arrival processes) in distributed processes are random
- Random arrival of requests superimposed cumulate into load peaks that are difficult to handle (temporal system overload)
- Conceptual idea
  - Implement a queue for requests that smoothes request up to a certain range of variability
  - A predefined amount of Service Instances handles requests
  - If peaks are excessive, requests are dropped (policing)
  - Keep server-side in a predefined load-range

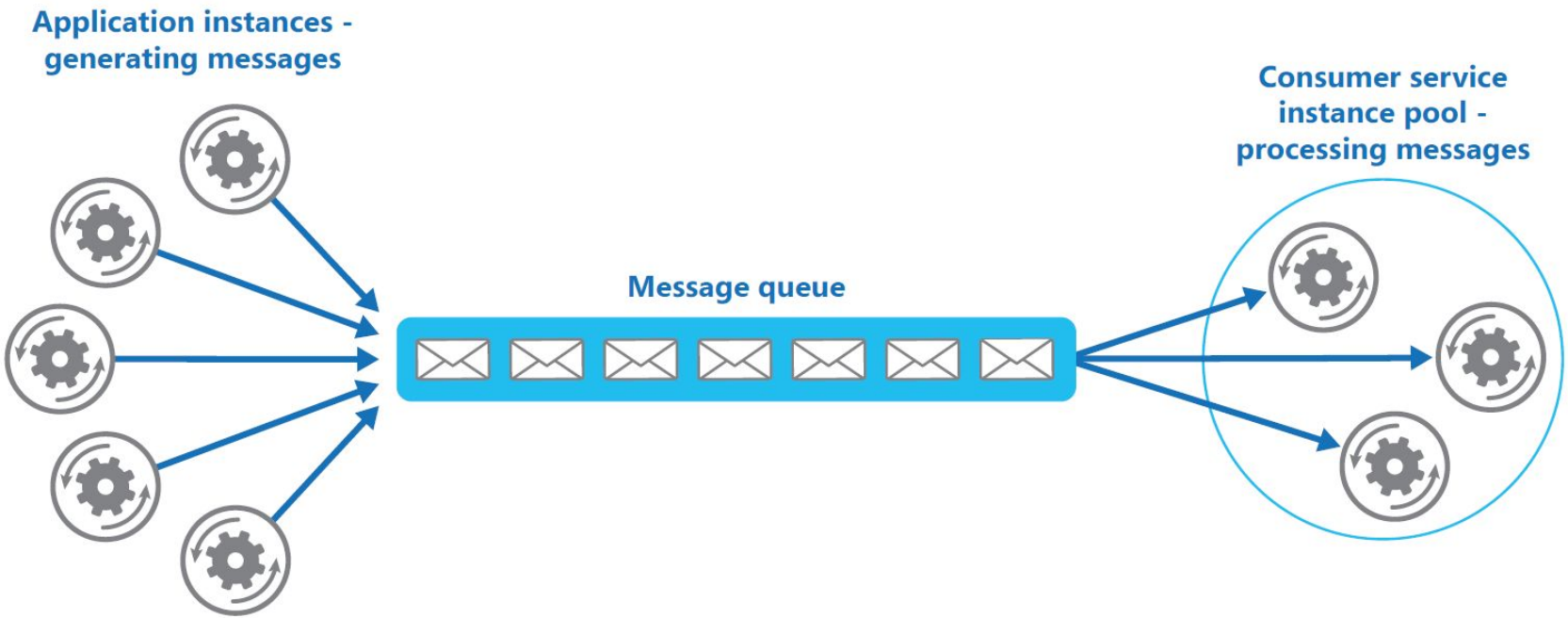
Queue Load-leveling



Competing Consumers / Producers

- Service-Orientation at scale means using/running many services
  - **Challenge is to provide elasticity**
- Conceptual idea
  - Implement messaging queue
  - Buffer requests and distribute them to a Service Instance (a consumer service) waiting in a pool
  - Enable handling of requests asynchronously
  - Enable handling of a variable amount of requests
  - Using a single instance of the consumer service might cause that instance to become flooded with requests or the messaging system may be overloaded by an influx of messages coming from the application.

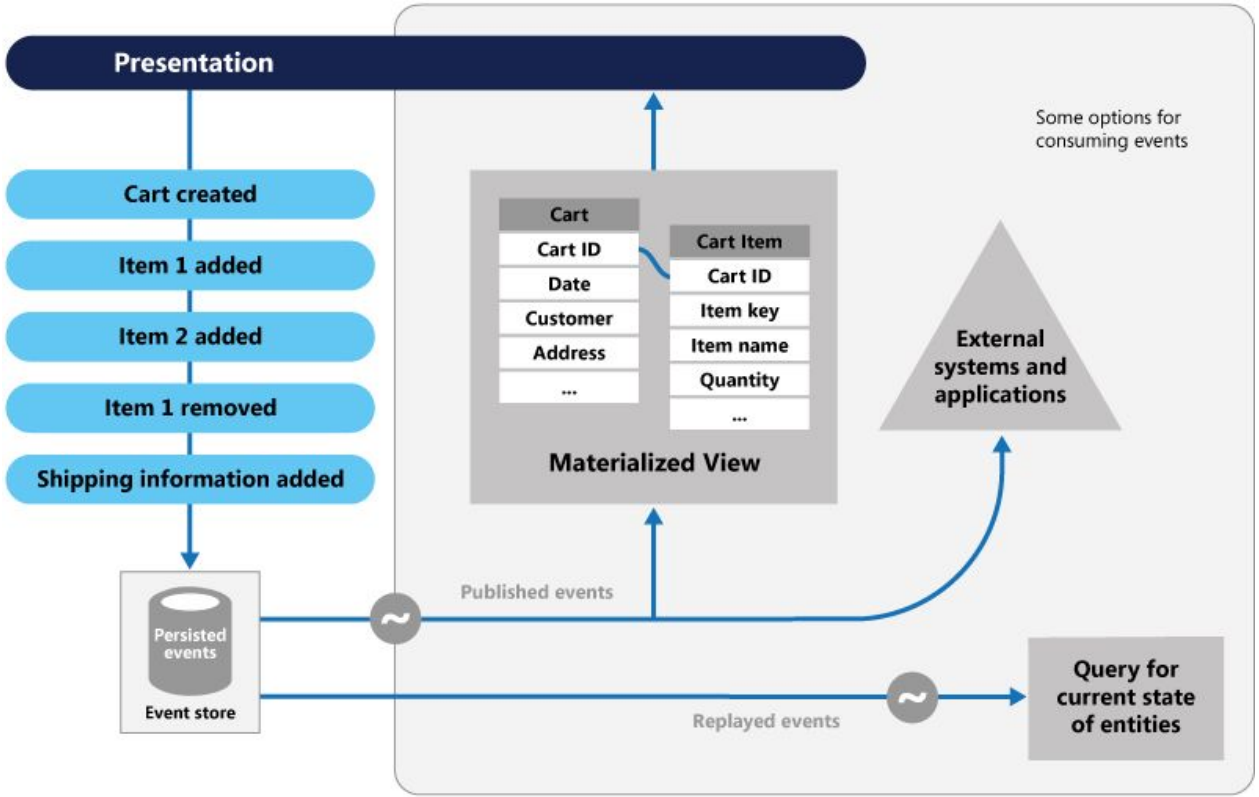
Competing Consumers / Producers



Event-Sourcing

- Service-Orientation at scale means using/running many services
  - **Challenge is to track state and provide transactional features in a highly distributed system**
- Conceptual idea
  - Store the Stream of events, instead of directly updating current state
  - Events are updating the state of the component(s) → materialized view(s)
    - Each component may have its own logic and state
  - Recreate the state(s) for any point in the past by sourcing (replaying) the event stream
  - Extendable by adding/updating new materialized views at any time and replay the event stream
- Advantages
  - Provide eventual consistency for transactional data
  - Maintain full audit trails and history that may enable compensating actions.

Event-Sourcing

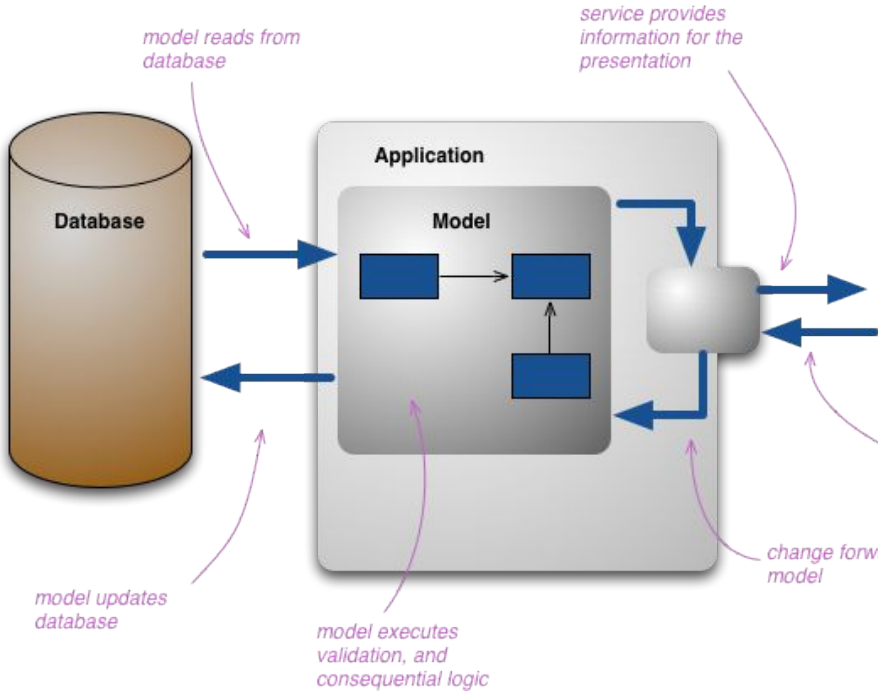


Command Query Response Segregation (CQRS)

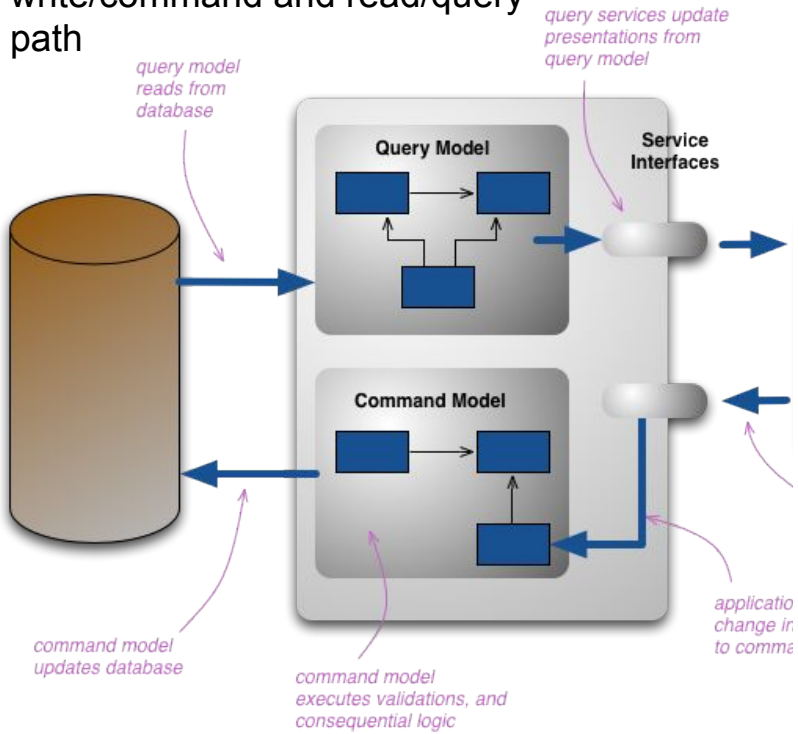
- Service-Orientation at scale means using/running many services
  - **Challenge is to optimize for read and write operations**
- Conceptual idea
  - Provide separate paths (interfaces) for write and read operations
  - Write operations (commands) are modifying the model / state
  - Read operations (query) provide the data from separate query models, which are optimized for read
  - Overload of one of the paths should not influence the other path.
  - Often used in combination with Event Sourcing
- Advantages
  - Each side is simpler to implement, and has only to concern the specific function
  - Read and Write can be scaled independently on demand.

CRUD vs CQRS

CRUD using single path for write and read



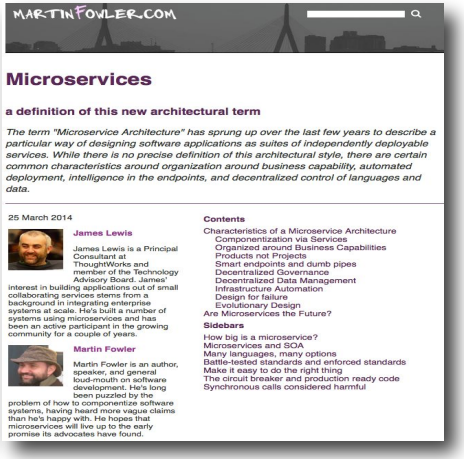
CQRS using separate write/command and read/query path



References

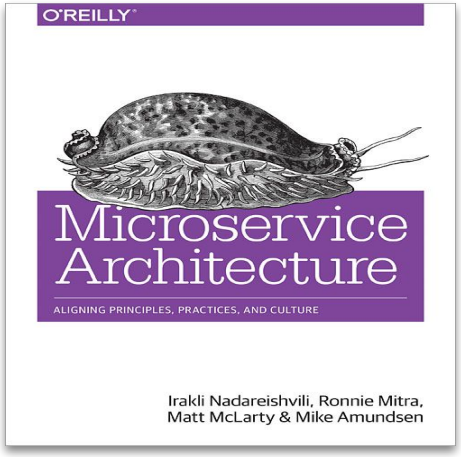
Matt Stine, “Migrating to Cloud-Native Application Architectures”

2015-02, O'Reilly Media



Martin Fowler, “Microservices”

2014-03,
http://www.martinfowler.com/articles/microservices.html



Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, Mike Amundsen, “Microservice Architecture”

2016-07, O’Reilly Media

Alex Homer, John Sharp, Larry Brader, Masashi Narumoto, Trent Swanson, “Cloud Design Patterns”

2014-02, Microsoft Press

Customer-Store Example App (complete)