

# Resilient Software Design Patterns

Thorsten Maier

Version: 21.1

BASEL | BERN | BRUGG | BUKAREST | DÜSSELDORF | FRANKFURT A.M. | FREIBURG I.B.R. | GENÈVE  
HAMBURG | KOPENHAGEN | LAUSANNE | MANNHEIM | MÜNCHEN | STUTTGART | WIEN | ZÜRICH

**trivadis**



# Thorsten

- Trivadis / früher OIO
- Vater
- Faustballer
- Hausumbauer
- Smarthome Begeisterter



@ThorstenMaier



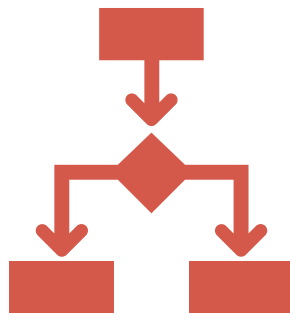
blog.oio.de

## ***Resilient Software?***

*Resilient Software?*

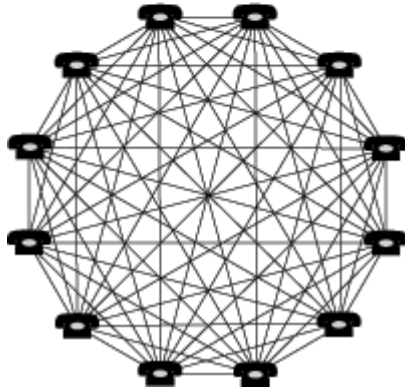
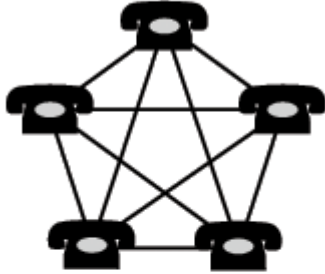
***Widerstandsfähig  
gegen Fehler***

*Warum?*



**Post-Monolit**





Metcalfe'sche Gesetz

***„Der Wert eines Netzwerkes ist proportional zum Quadrat der Anzahl der Teilnehmer“***



# “Big Data”



<https://pixabay.com/de/photos/lego-figuren-spielzeug-menge-viele-1044891/>



25. März 2018, 11:51 Uhr Digitale Privatsphäre

## Datensammelwut gefährdet die Demokratie



Jeden Tag werden viele Terabytes von Daten über Milliarden Menschen verarbeitet und gespeichert. (Foto: Daniel Reinhardt/picture alliance / dpa)

Die Daten werden in riesigen Serverfarmen gespeichert, die oft in kühlen, dunklen Räumen stehen. Die Daten werden in riesigen Serverfarmen gespeichert, die oft in kühlen, dunklen Räumen stehen. Die Daten werden in riesigen Serverfarmen gespeichert, die oft in kühlen, dunklen Räumen stehen.

## ***2 Millionen Google-Server***

*Unrealistische Annahme*

***MTTF = 30 Jahre***

***~ alle 8 Minuten fällt ein Server aus***

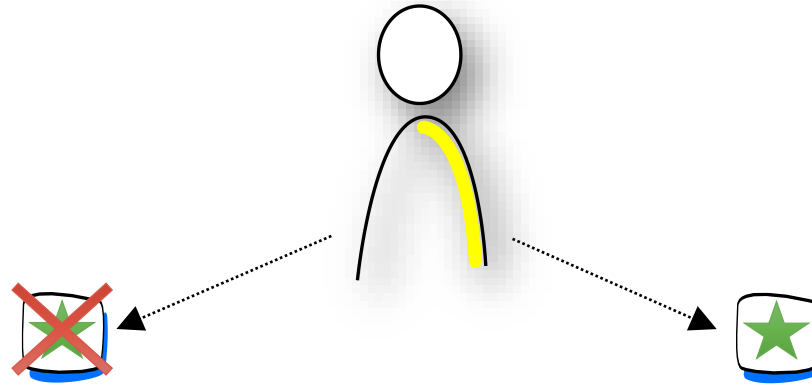
***Things will crash.  
Deal with it!***

*→ Resilient Software Design*

***Wie machen wir eine Software „resilient“?***

*Wie machen wir eine Software „resilient“?*

## ***Redundanz***

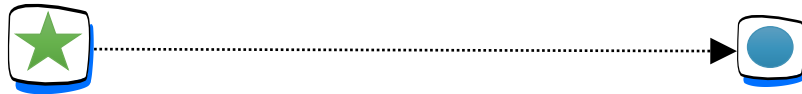




*Wie machen wir eine Software „resilient“?*

***Redundanz***

***Isolation***



*Wie machen wir eine Software „resilient“?*

*Redundanz*

*Isolation*

***Lose Kopplung***



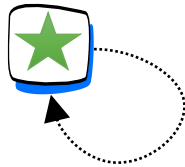
*Wie machen wir eine Software „resilient“?*

*Redundanz*

*Isolation*

*Lose Kopplung*

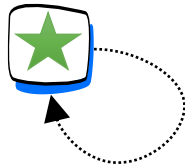
***Fallback***



*Wie machen wir eine Software „resilient“?*

*Redundanz*

*Isolation*  
**Noch Fragen?**  
*Lose Kopplung*

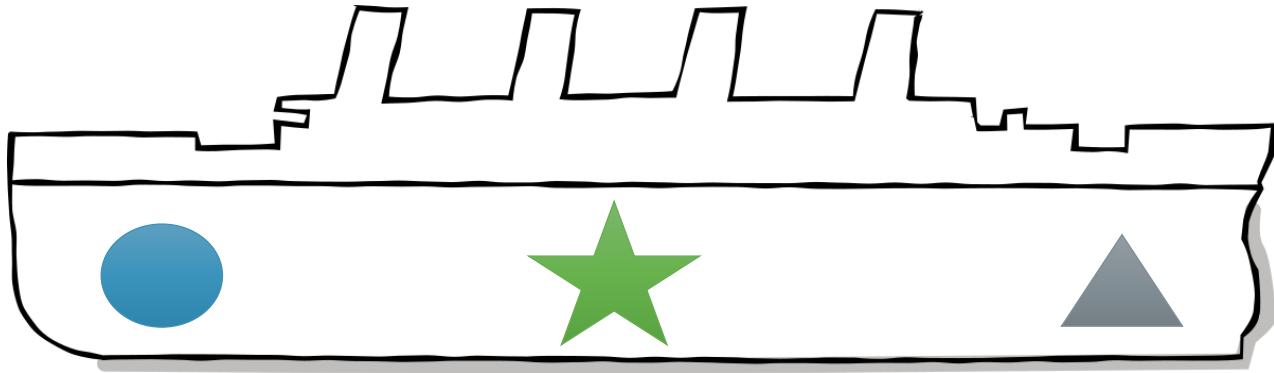


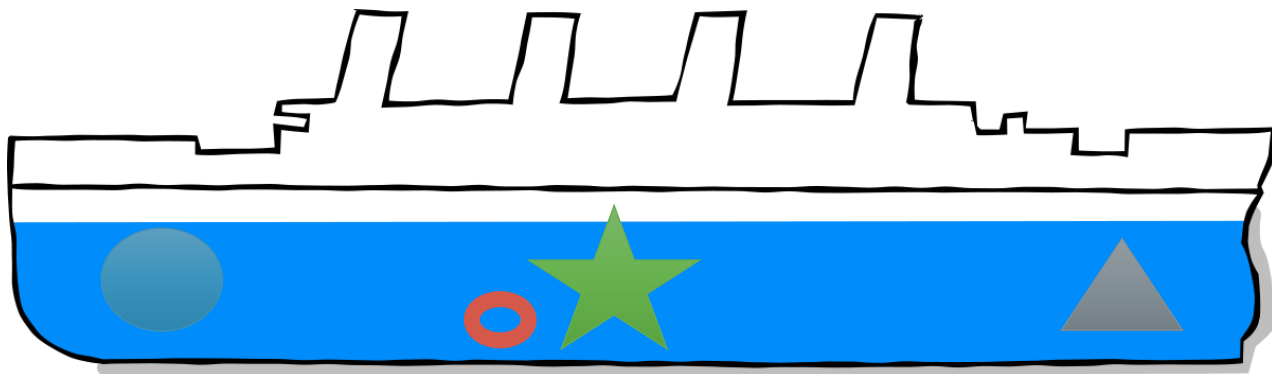
**Fallback**

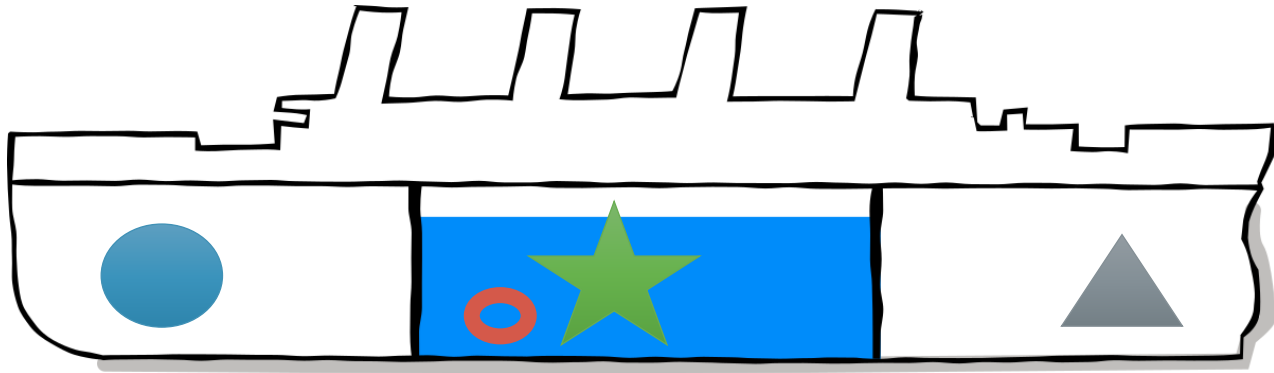


# ***ISOLATION***









# ***BULKHEADS***

*Bulkheads im Kleinen:*

# ***Methodenaufrufe***

## **Validierung der Aufrufparameter**

*Datentypen korrekt?*

*Wertebereiche eingehalten?*

*Vorbedingungen erfüllt?*

*!= null*

*Kreditkartennummer valide*

...

## **„Freundliche“ Rückgabewerte**

*Niemals „null“ zurückliefern*

*Datenmengen beschränken*

...



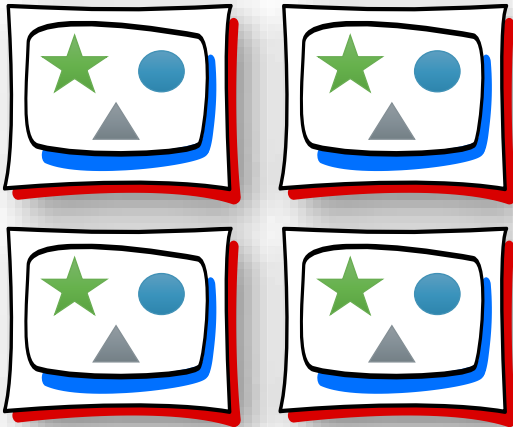
*Bulkheads im Großen:*  
***Software-Bausteine***




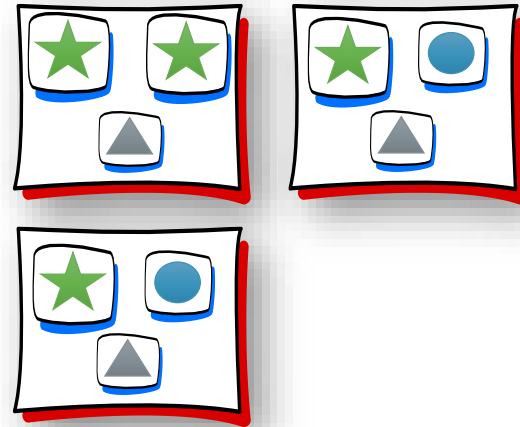
Anwendung mit 3 Bausteinen



Unabhängige Artefakte



Baustein  wird 4 mal benötigt



**Flexible** Skalierung auf 3 Server

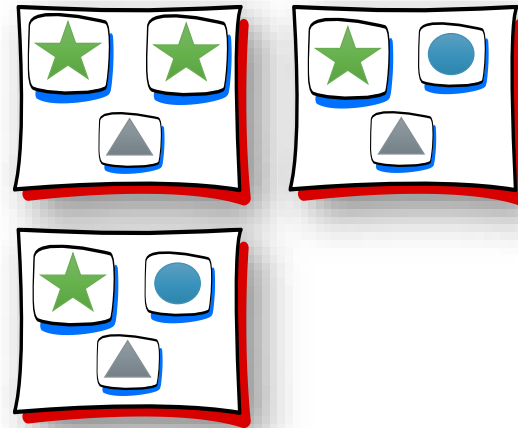
## Vorteile

**Isolierte Entwicklung**  
**Isolierte Fehler**  
**Isoliertes Deployment**

...



Unabhängige Artefakte



**Flexible** Skalierung auf 3 Server

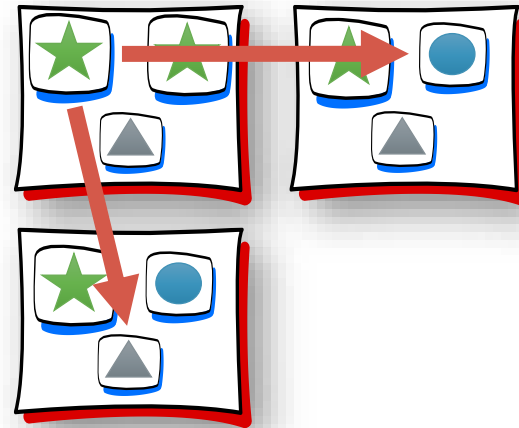
## Nachteil

Kommunikation über  
Prozess- und Netzwerkgrenzen

→ Verteilte Datenverarbeitung



Unabhängige Artefakte



Flexible Skalierung auf 3 Server

## *8 Irrtümer der verteilten Datenverarbeitung*

***Netzwerk ist ausfallsicher***

***Latenzzeit = 0***

***Datendurchsatz  $\infty$***

***Netzwerk ist sicher***

***Netzwerktopologie ist stabil***

***1 Netzwerkadministrator***

***Kosten des Datentransports = 0***

***Netzwerk ist homogen***



***Verhalten und Standorte der  
Komponenten unseres Systems  
verändern sich ständig***



*Komponenten liefern*  
***unzuverlässige Daten*** *oder*  
***verschwinden völlig***





*Ok, wir brauchen ISOLATION*

***Aber wie kommen die Teile  
wieder zusammen?***

# ***LOSE KOPPLUNG***

# ***LOSE KOPPLUNG***

***asynchron***

*synchron*

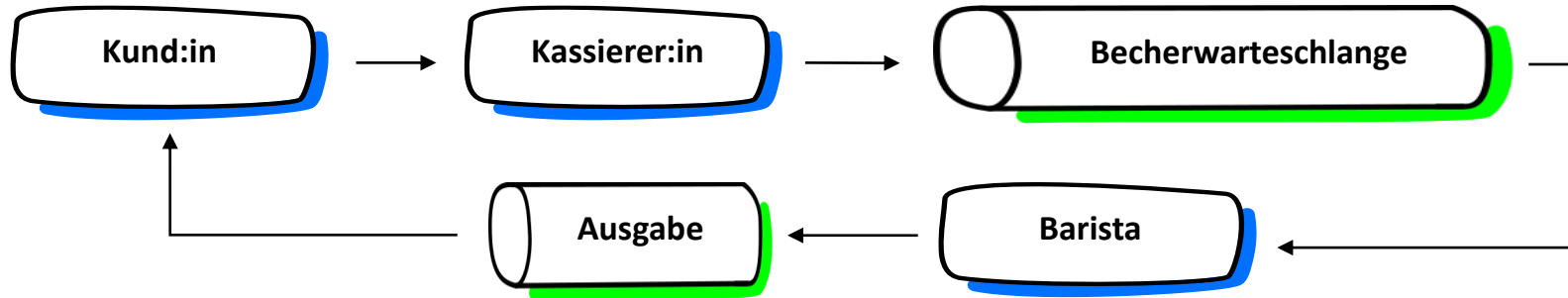
# ***Asynchrone Kommunikation***

*Entkoppelt Sender und Empfänger*

*Verhindert Fehlerketten*

*Sender muss nicht warten*

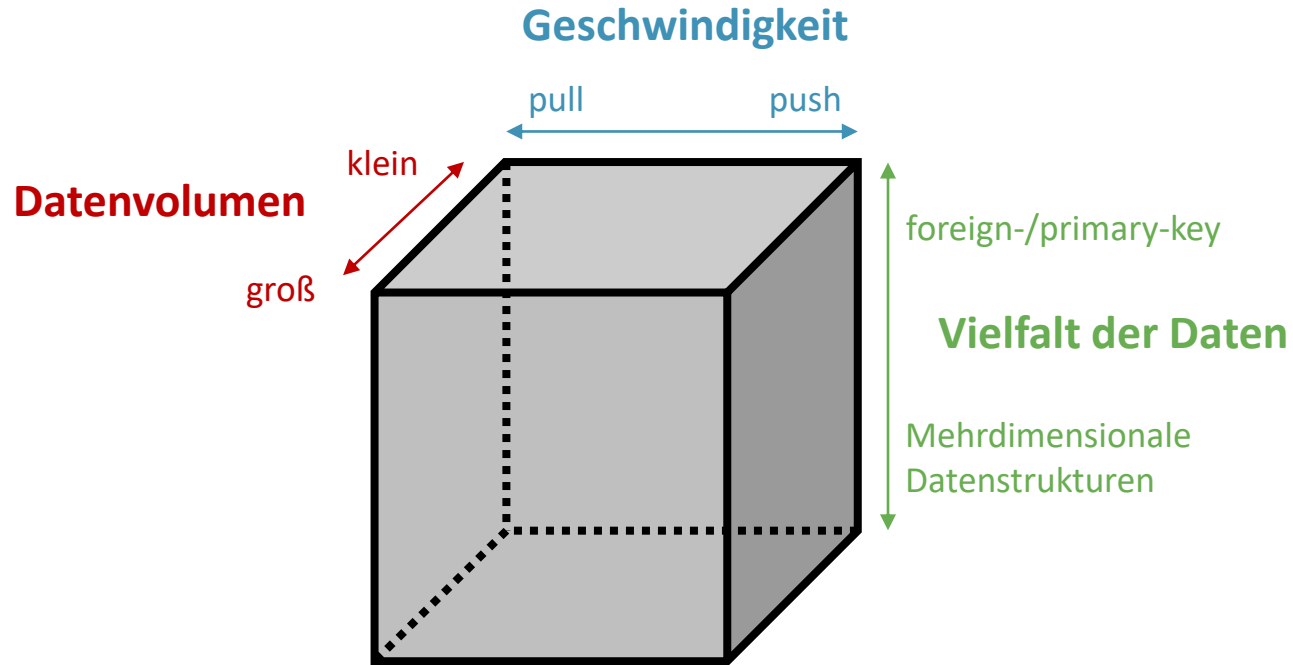
## *„Starbucks Does Not Use Two-Phase Commit”*

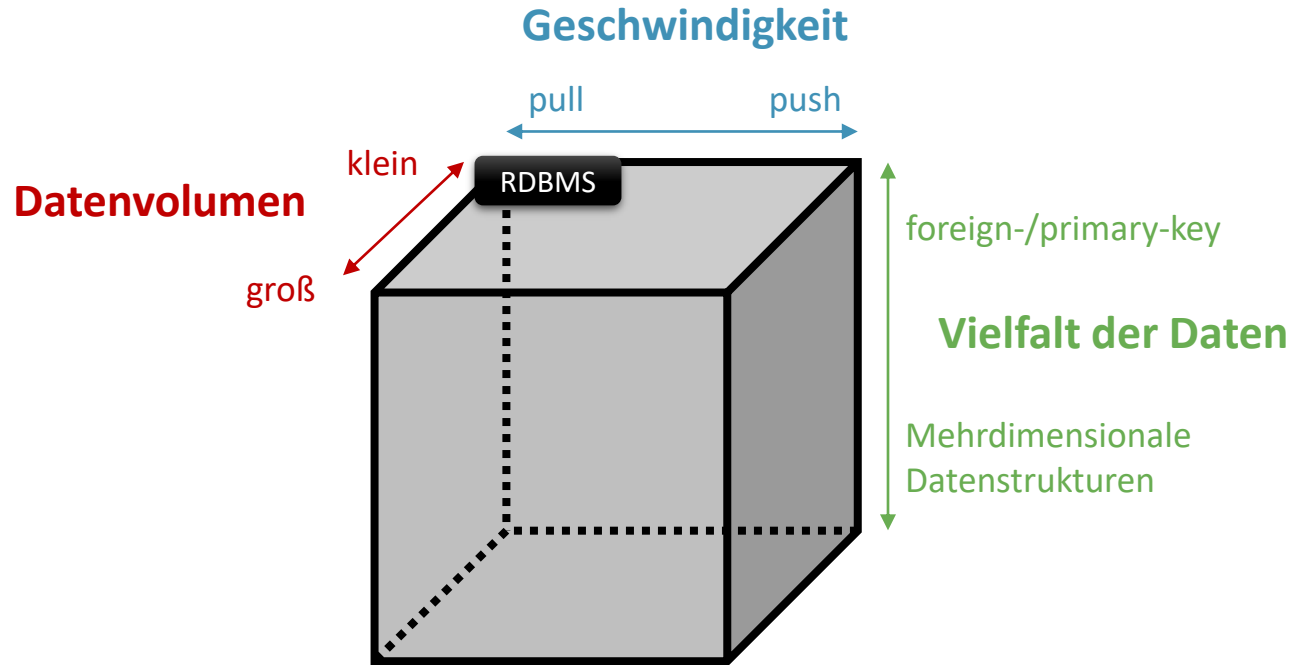


```
JmsTemplate jmsTemplate = context.getBean(JmsTemplate.class);  
jmsTemplate.convertAndSend("coffeeOrderQueue", new CoffeeOrder("Thorsten", "Cappuccino"));
```

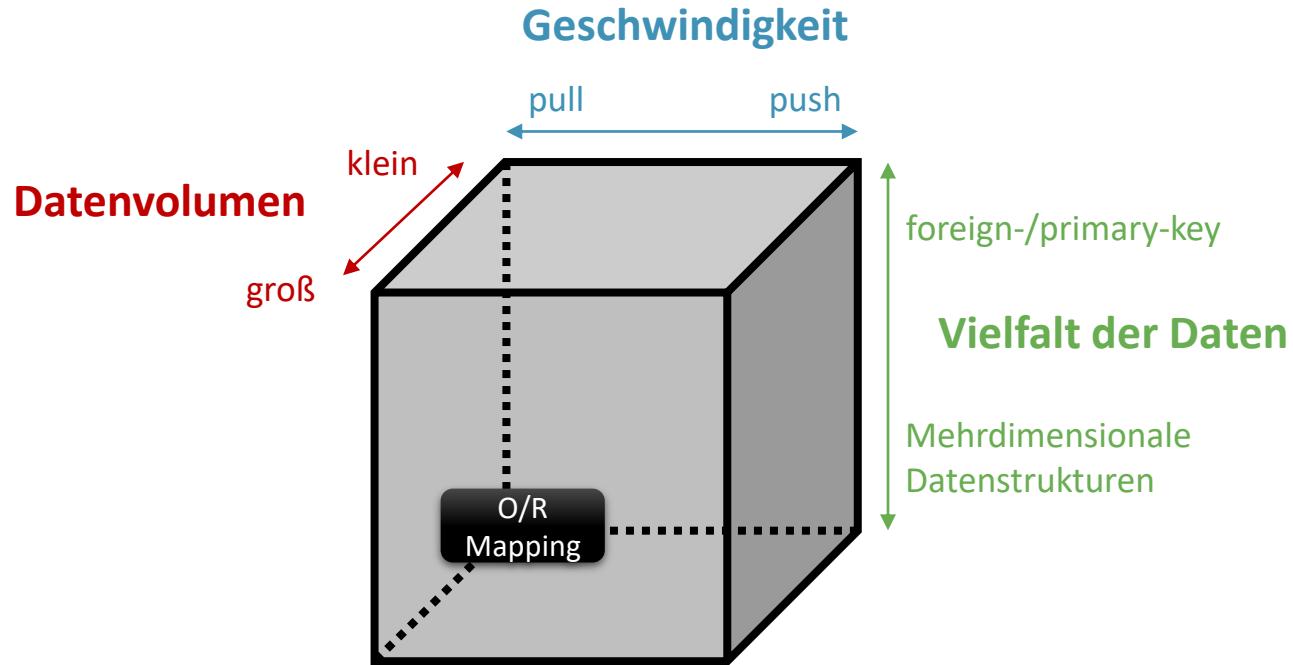


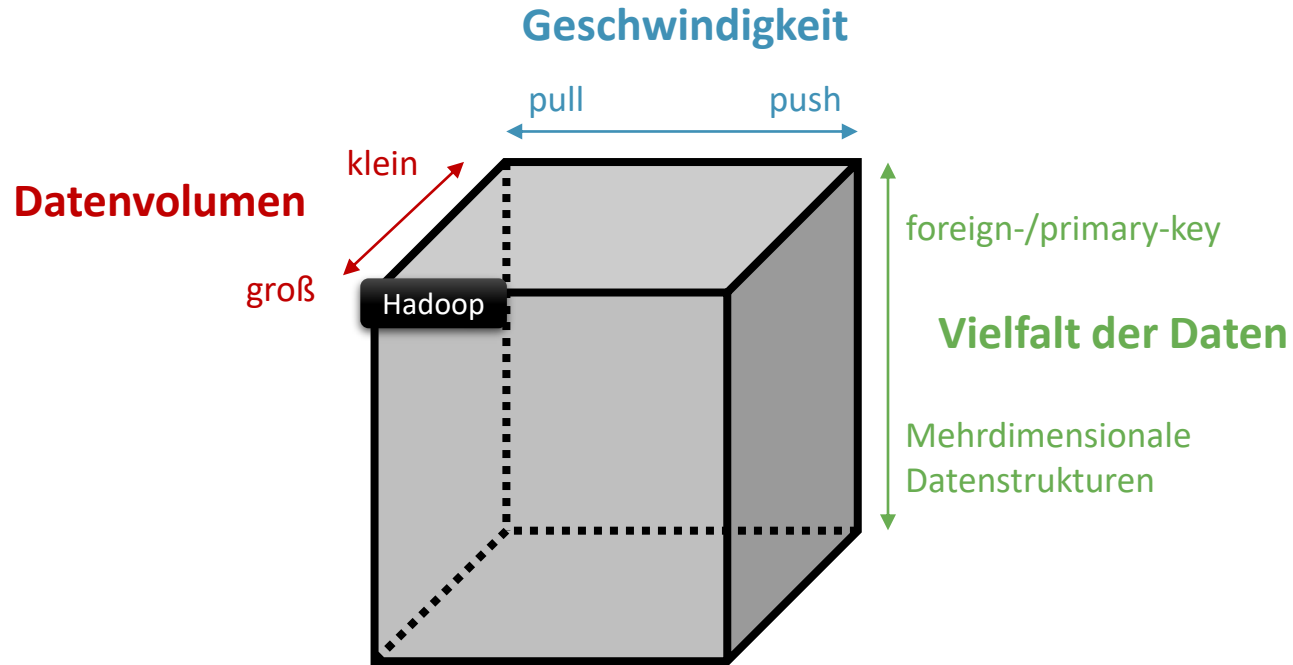
```
@Component  
public class Barista {  
  
    @JmsListener(destination = "coffeeOrderQueue")  
    public void receiveMessage(CoffeeOrder coffeeOrder) {  
        System.out.println("Received <" + coffeeOrder + ">");  
    }  
}
```

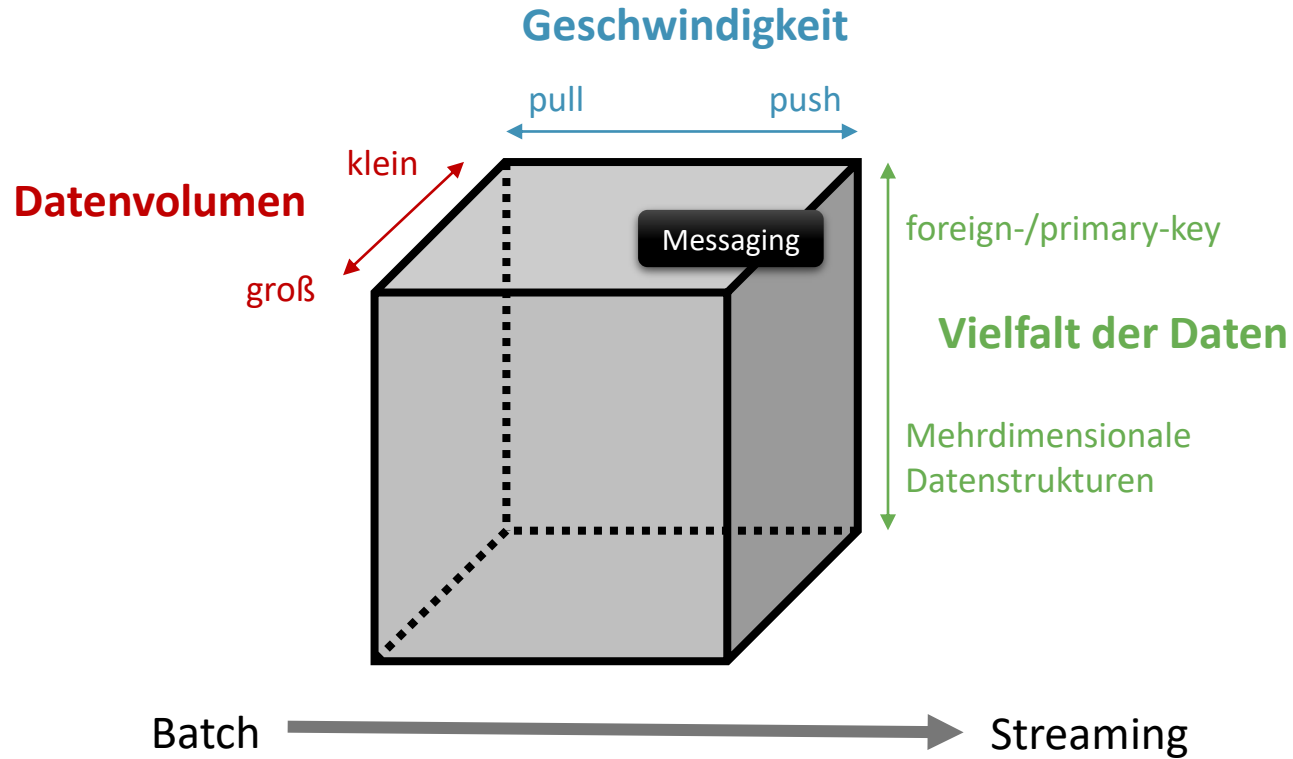












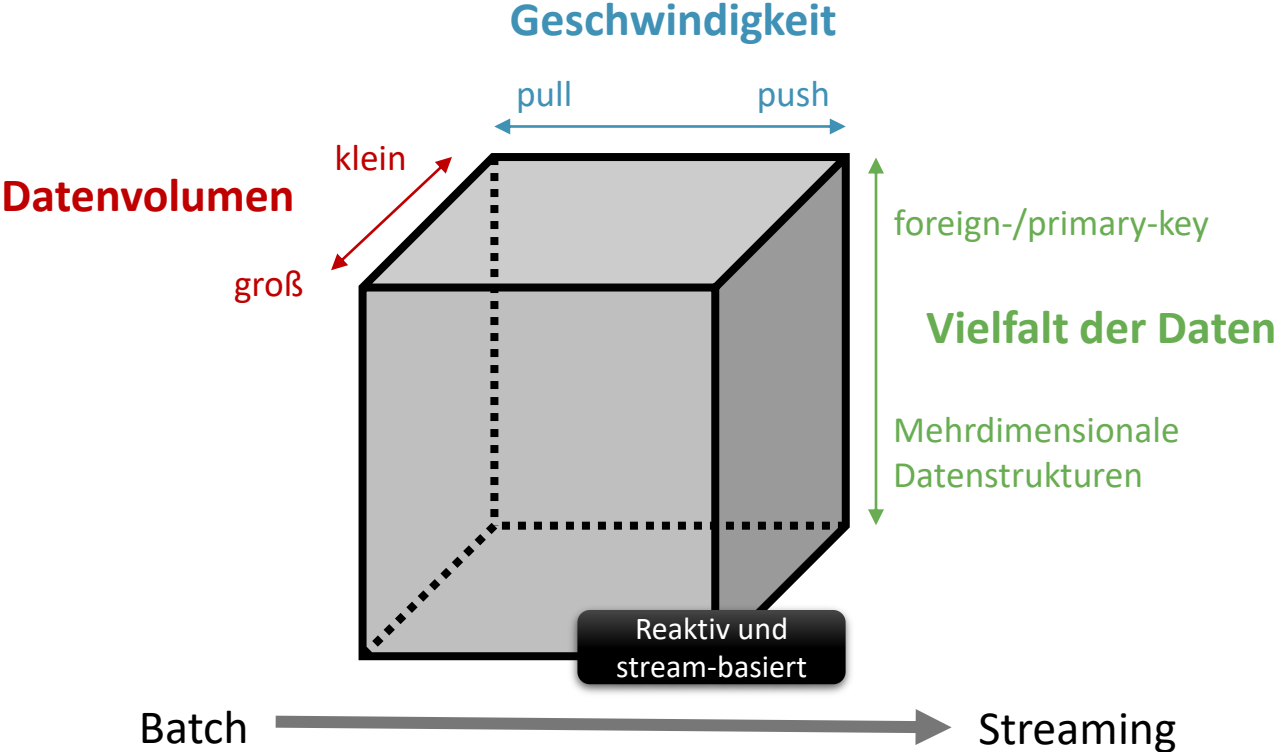




Bild von [Roland Mey](#) auf [Pixabay](#)

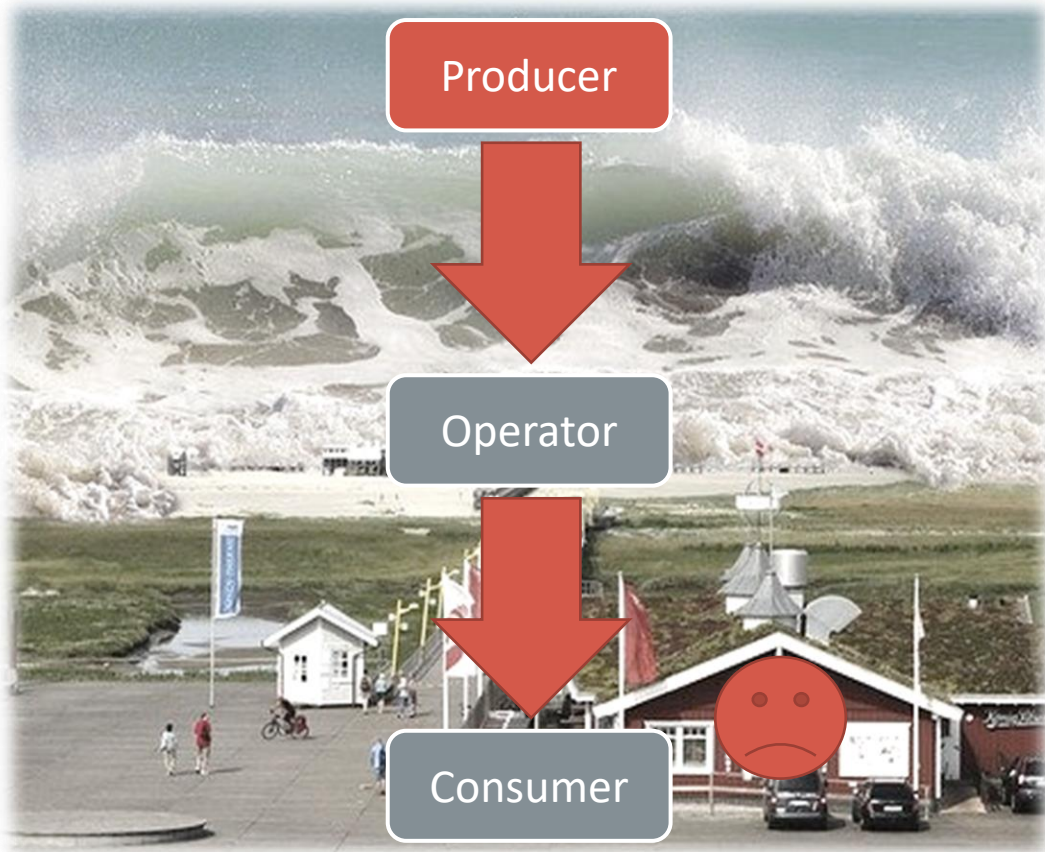
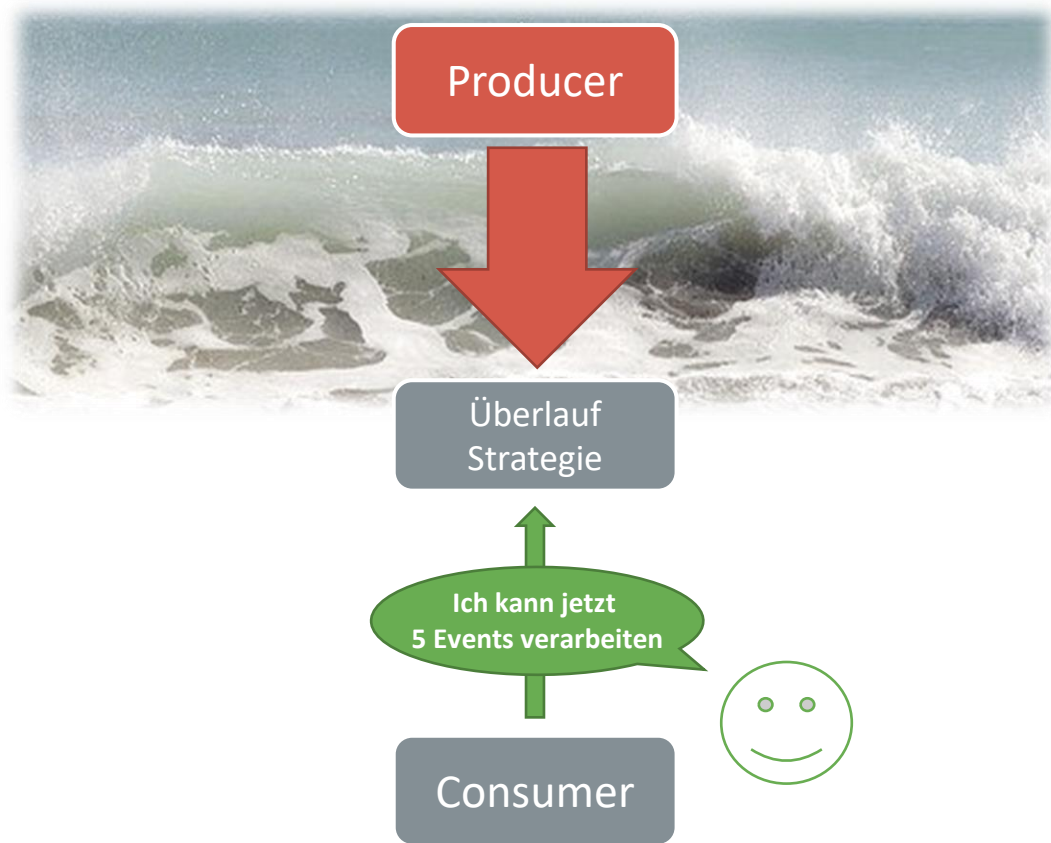
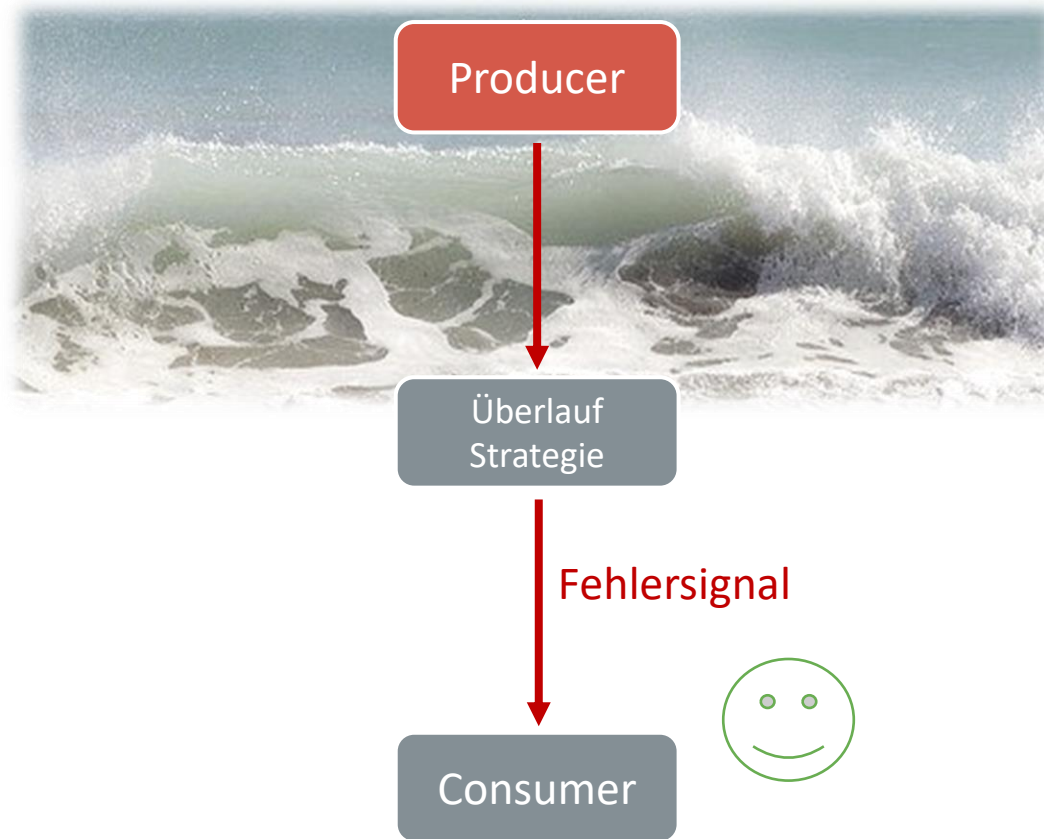
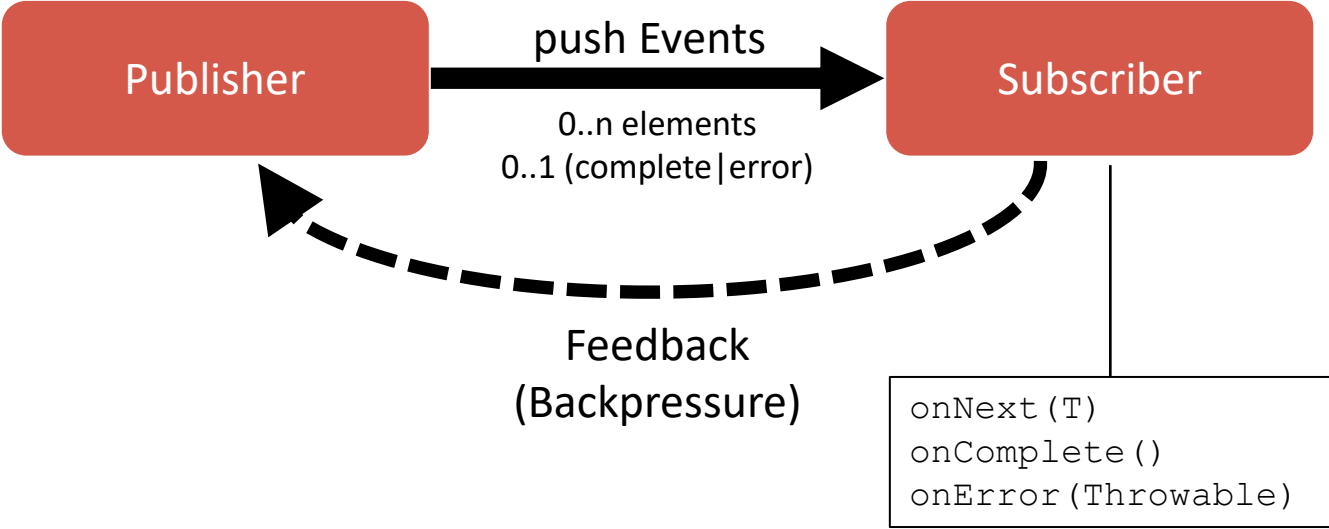


Bild von [Roland Mey](#) auf [Pixabay](#)










Name

Sendername bereits vergeben

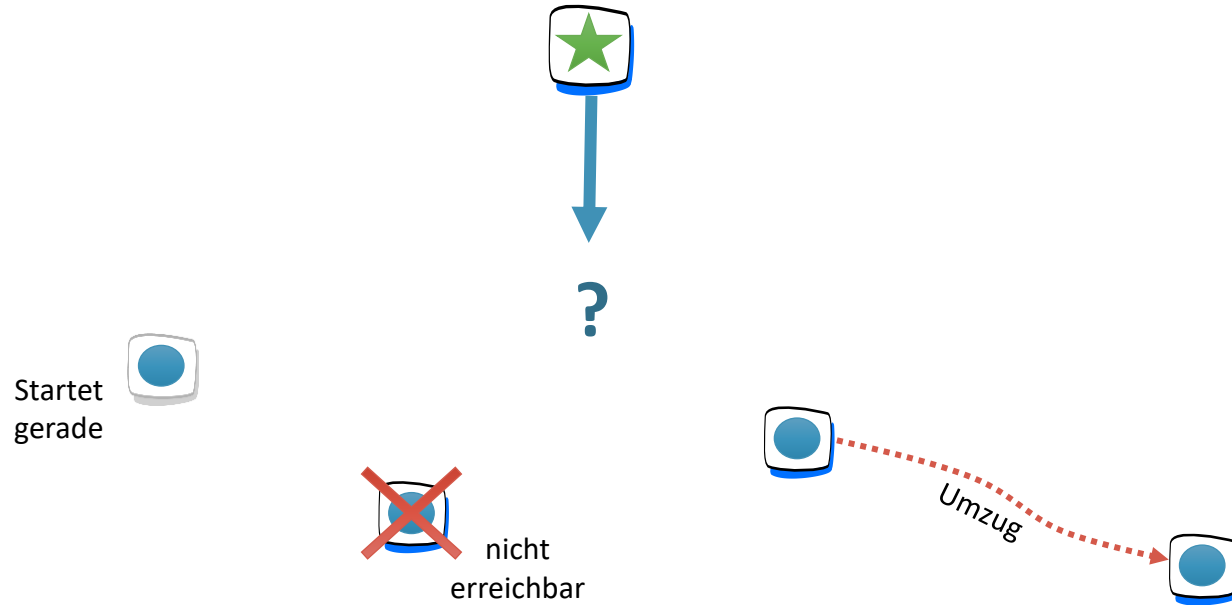
 Sender anlegen

```
this.unique$ = this.nameChangedEvent.pipe(  
  debounceTime(200), // 200ms auf geänderte Usereingabe warten  
  map(station => JSON.parse(JSON.stringify(station))),  
  flatMap(station => this.httpClient.post(...),  
    share(),  
    filter(serverResult => serverResult && serverResult.name === this.selectedStation.name),  
    map(serverResult => serverResult.unique),  
    startWith(true),  
  );
```

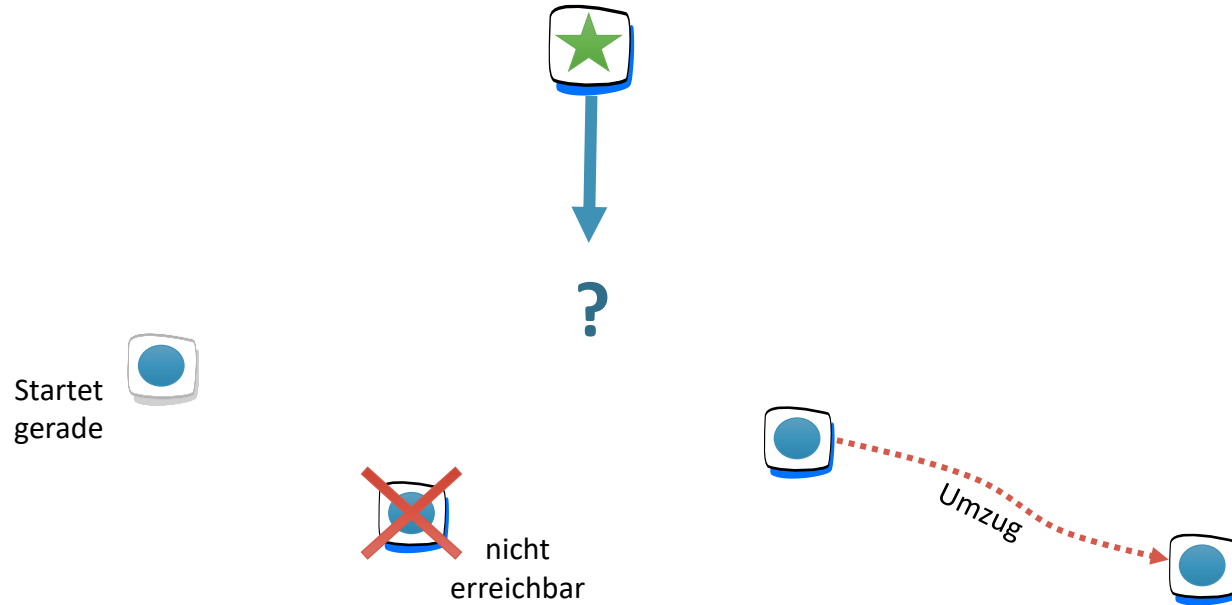
# ***LOSE KOPPLUNG***

*asynchron*

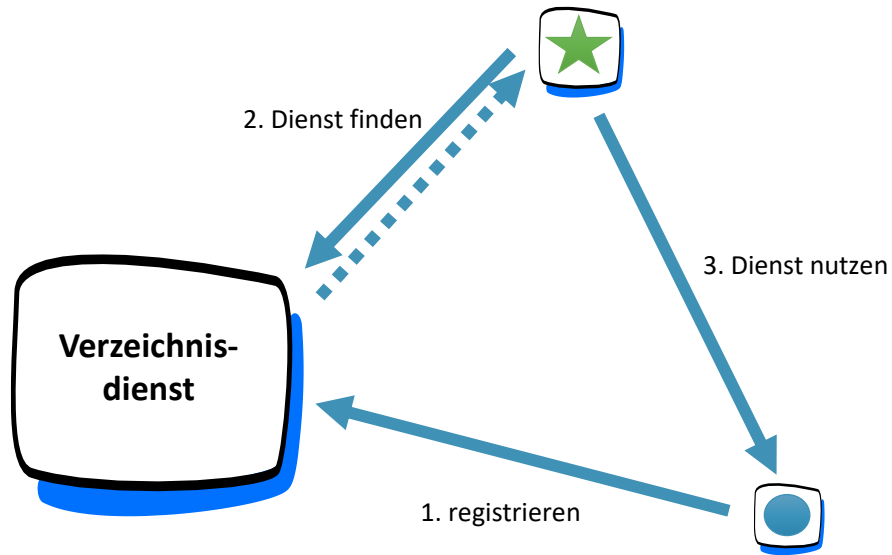
***synchron***



***Mit wem kann / soll ich kommunizieren?***



Als **Architekt** lassen sich alle Probleme mit  
**„Boxes and Lines“** lösen 😊



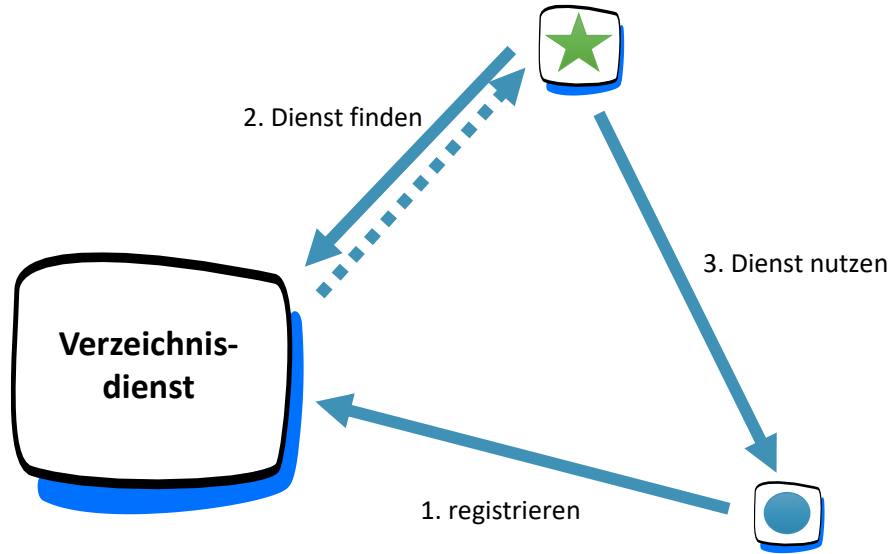
z.B.:

*Netflix Eureka*

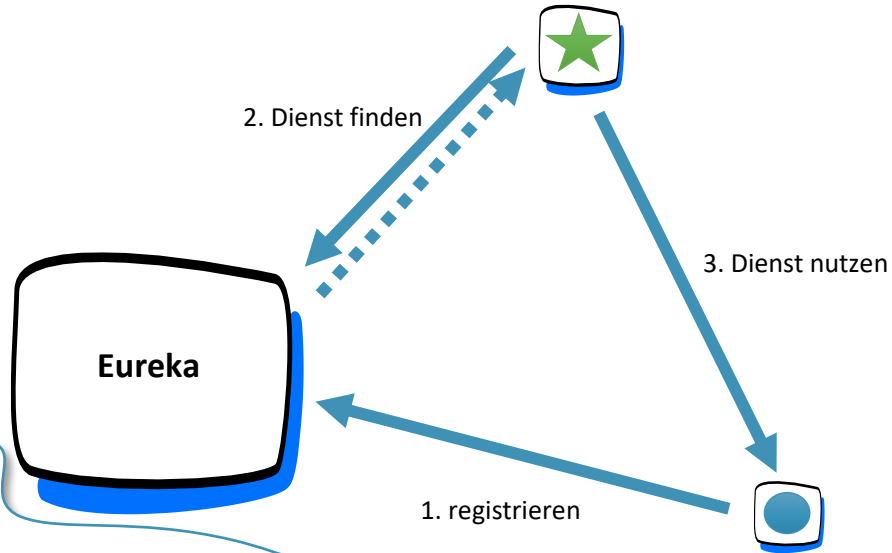
*Native Kubernetes Service Discovery*

*Apache ZooKeeper*

...

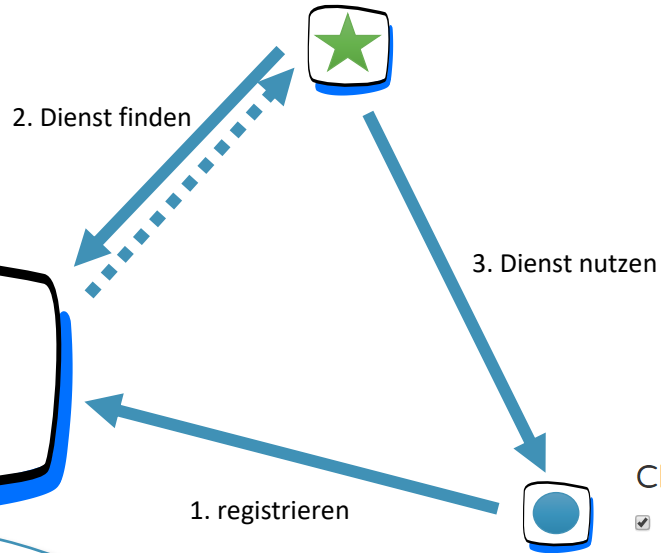


Als **Entwickler** brauchen wir **etwas mehr**



```
@SpringBootApplication
@EnableEurekaServer
public class Application {
    public static void main(String[] args) {
        // ...
    }
}
```



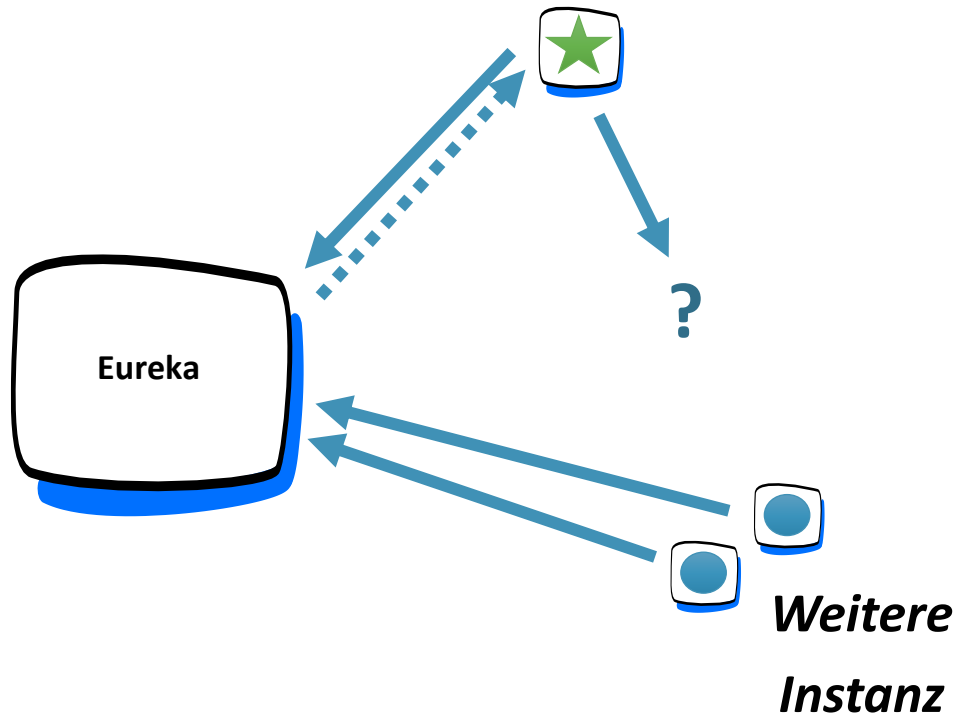


```
@SpringBootApplication
@EnableEurekaServer
public class Application {
    public static void main(String[] args) {
        // ...
    }
}
```

Cloud Discovery

☒ Eureka Discovery

Service discovery using spring-cloud-netflix and Eureka



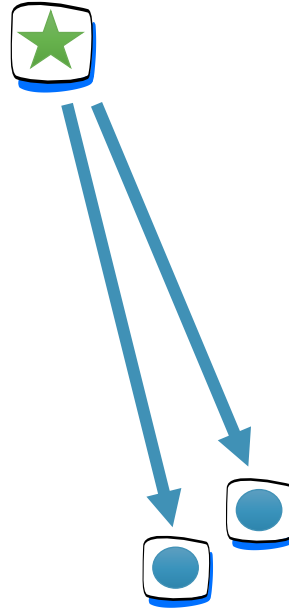
## Algorithmen

*Simple Round Robin*

*Zone Aware Round Robin*

*Random*

*Weighted Response Time*



*Resiliente Lösung:*  
**Clientseitiges  
Load-Balancing**


*Resiliente Lösung:*

## Clientseitiges Load-Balancing

```
@RibbonClient(name = "blauerService")
@RestController
public class Application {

    @LoadBalanced
    @Bean
    RestTemplate restTemplate() {
        return new RestTemplate();
    }

    @RequestMapping("/")
    public String serviceCall() {
        return restTemplate().getForObject("http://blauerService/", String.class);
    }
}
```



The diagram illustrates client-side load balancing. At the top, a client icon (a square with a green star) sends two separate requests, represented by blue arrows, to two different service instances. Each service instance is represented by a square containing a blue circle. The entire diagram is enclosed within a blue rounded rectangle that also contains the Java code snippet on the left.

```
🐋 docker-compose.yml ●  
1  version: "3"  
2  services:  
3    myService:  
4      image: my/application:latest  
5      restart: unless-stopped  
6      ports:  
7      - 8080:8080
```

```
prometheus.yml X
1  scrape_configs:
2    - job_name: "configserver"
3      metrics_path: "/actuator/prometheus"
4      static_configs:
5        - targets: ["configserver:8888"]
6    - job_name: "pdfcreator"
7      metrics_path: "/actuator/prometheus"
8      static_configs:
9        - targets: ["pdfcreator:8088"]
10  rule_files:
11    - "rules.yml"
12  alerting:
13    alertmanagers:
14      - static_configs:
15        - targets:
16          - alertmanager:9093
```

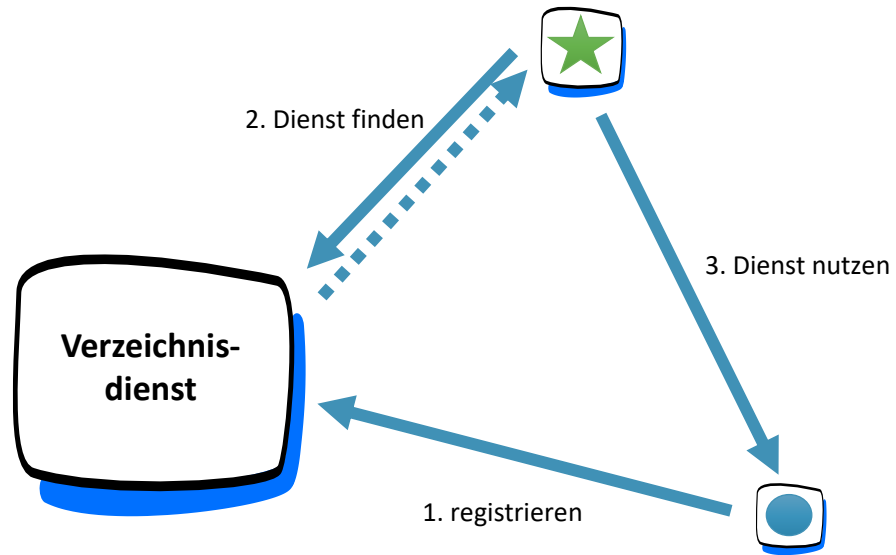
oder gleich der dicke Hammer



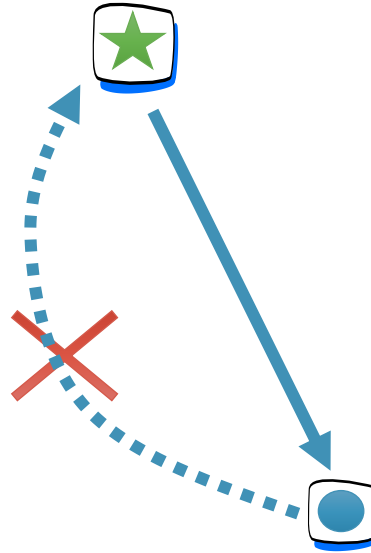
**kubernetes**

***FALLBACK***

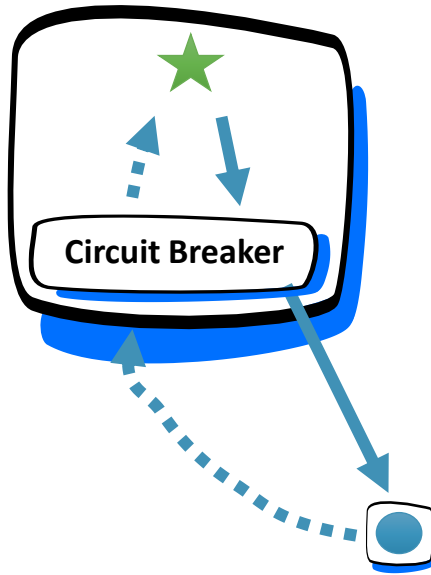


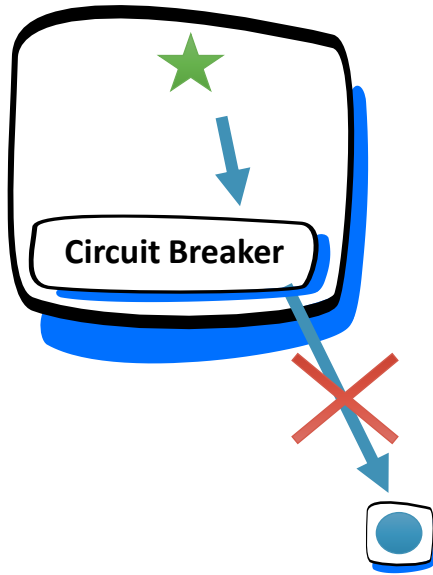


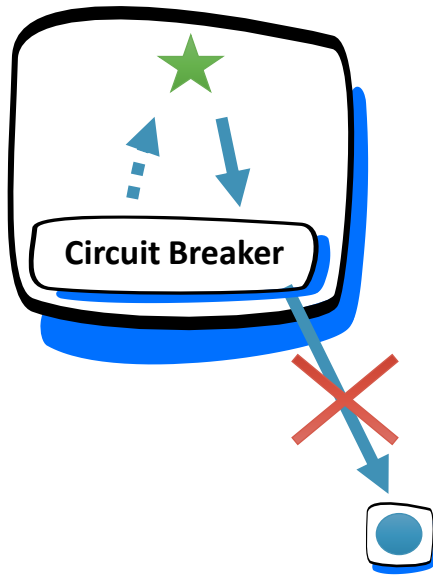
*zur Erinnerung*

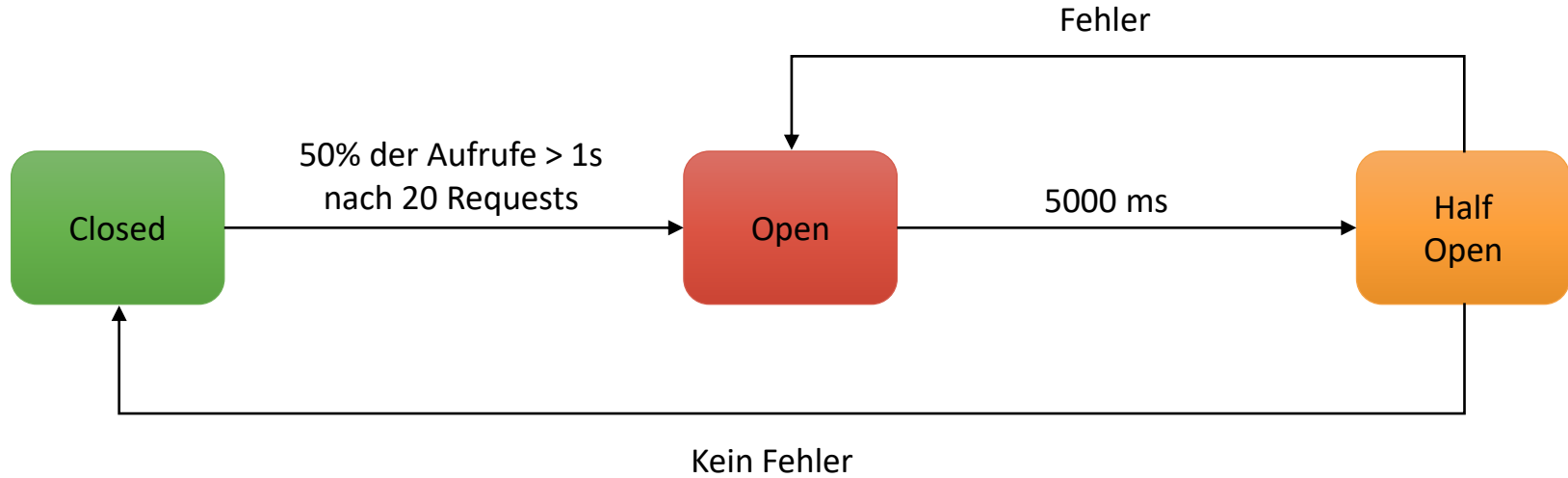


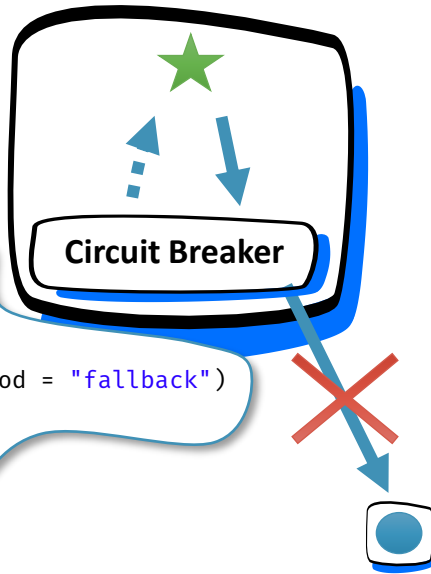
***Instanz fällt kurz  
NACH der Auswahl aus***











```
@CircuitBreaker(name = "BACKEND", fallbackMethod = "fallback")  
public String method(String param1) {  
    // ...  
}  
  
private String fallback(String param1) {  
    // ...  
}
```

# resilience4j

```
@CircuitBreaker(name = BACKEND, fallbackMethod = "fallback")
@RateLimiter(name = BACKEND)
@Bulkhead(name = BACKEND)
@Retry(name = BACKEND, fallbackMethod = "fallback")
@TimeLimiter(name = BACKEND)
public Mono<String> method(String param1) {
    return Mono.error(new NumberFormatException());
}

private Mono<String> fallback(String param1, IllegalArgumentException e) {
    return Mono.just("test");
}

private Mono<String> fallback(String param1, RuntimeException e) {
    return Mono.just("test");
}
```



```
@CircuitBreaker(name = BACKEND, fallbackMethod = "fallback")
```

Der Aufrufer misst Antwortzeiten / Fehlerraten

```
@RateLimiter(name = BACKEND)
```

Maximal 5 Aufrufe in 2 Sekunden

```
@Bulkhead(name = BACKEND)
```

Maximal 10 Aufrufe zur gleichen Zeit

```
@Retry(name = BACKEND, fallbackMethod = "fallback")
```

Der Aufrufer wiederholt die Anfragen  
(ähnlich zu Circuit Breaker aber ohne Zustand)

```
@TimeLimiter(name = BACKEND)
```

Ein Aufruf darf maximal 3 Sekunden dauern

```
public Mono<String> method(String param1) {  
    return Mono.error(new NumberFormatException());  
}
```

Reaktiv

# resilience4j

```
@CircuitBreaker(name = BACKEND, fallbackMethod = "fallback")
@RateLimiter(name = BACKEND)
@Bulkhead(name = BACKEND)
@Retry(name = BACKEND, fallbackMethod = "fallback")
@TimeLimiter(name = BACKEND)
public Mono<String> method(String param1) {
    return Mono.error(new NumberFormatException());
}

private Mono<String> fallback(String param1, IllegalArgumentException e) {
    return Mono.just("test");
}

private Mono<String> fallback(String param1, RuntimeException e) {
    return Mono.just("test");
}
```



# Fragen?

BASEL | BERN | BRUGG | BUKAREST | DÜSSELDORF | FRANKFURT A.M. | FREIBURG I.B.R. | GENÈVE  
HAMBURG | KOPENHAGEN | LAUSANNE | MANNHEIM | MÜNCHEN | STUTTGART | WIEN | ZÜRICH

**trivadis**





**Vielen Dank für Ihre  
Aufmerksamkeit!**

BASEL | BERN | BRUGG | BUKAREST | DÜSSELDORF | FRANKFURT A.M. | FREIBURG I.B.R. | GENÈVE  
HAMBURG | KOPENHAGEN | LAUSANNE | MANNHEIM | MÜNCHEN | STUTTGART | WIEN | ZÜRICH

**trivadis**