Resilience-Patterns in Cloud-Anwendungen

Kristian Köhler

sourcefellows

Die Frage ist nicht OB ein Fehler auftritt, sondern WANN ein Fehler auftritt

Bei 40000 Requests gibt es mindestens 40000 mögliche Fehler

Begri der Resilience in der Softwareindustrie

- **Ursprünge in der Materialwissenschaft**
  - Nach Verformung wieder in ursprüngliche Form zurückzukehren
- **Trotz Fehler oder Impulse können Transaktionen durchgeführt werden**
  - Kurzzeitige Ausfälle, Lastspitzen, etc
  - Nicht reine Stabilität des Systems im Fokus
  - Ziel: Anwender können weiter Arbeit erledigen – „Unit of Work“
- **Die Fähigkeit eines Systems auf unerwartete Fehler zu reagieren**
  - Ohne dass der Anwender es merkt
  - Eventuell Abschaltung/ Degradierung eines Service

Ausbreitung verhindern

- **Probleme beginnen meist mit kleinen „Rissen“ / Cracks**
  - „System X reagiert nicht schnell genug“
  - „Datenbank Y ist ausgefallen“
  - „Message-Verarbeitung läuft auf einen Fehler“
- **„Cracks propagate“**
- **Entstehende Fehler müssen eingedämmt werden**
  - Ausbreitung verhindern
  - „Crackstoppers“ - James R. Chiles
  - Sollbruchstellen einbauen

Pattern Kataloge

- **Release It! - Design and Deploy Production-Ready Software**

  Michael T. Nygard

- **Microsoft Azure – Microservice Patterns**

  https://learn.microsoft.com/en-us/azure/architecture/patterns/

- **Well architectured frameworks**

  AWS, Google, Microsoft

- **...**

Resiliency Patterns in Microservices

Resiliency Pattern	Short Description
Circuit Breaker Pattern	Fail fast in case of errors and enables you to perform the default or fallback operations.
Retry Pattern	Making several attempts to execute a failed remote operation before giving up and reporting it as an issue.
Timeouts/Time Limits	Set a time limit for a remote operation instead of indefinitely waiting for response.
Fallback Mechanism	Fallback mechanisms provide an alternative response or behaviour when a remote operation is failing. This can be like returning cached results or default values.
Bulkhead Pattern	The Bulkhead pattern involves isolating components of a system so that the failure of one component does not lead to the failure of the entire system.
Health Checks	Monitor the remote services and remove from the load balancer automatically or stop routing requests when it is unhealthy.
Failover and Redundancy	Redundancy and failover capabilities ensures that if one instance or component fails, another can take over.
Event/message-based communications	Adopt event/message-based communication wherever is possible during service-to-service communications. This decouples services and enables them to react to events at their own pace, improving overall resilience.

Quelle: https://anjireddy-kata.medium.com/architecture-and-design-101-resiliency-patterns-in-microservices-71029bbb92b7

Wer ich bin

Kristian Köhler

Source Fellows GmbH
https://www.source-fellows.com
https://www.linkedin.com/in/kristian-köhler/

25+ Jahre in Softwareentwicklung

Java Enterprise Hintergrund
Javascript, Python, C#, etc etc

Timeouts

sourcefellows

Timeouts

- **Timeout steuert Abbruch der Verarbeitung**
  - Blockierende Threads können System lahmlegen
    - Resource Pools können ausgeschöpft werden
    - Deadlocks
- **Timeouts bieten Isolation von Fehlern**
  - Externe Systeme sind über Netzwerk angebunden
    - Netzwerke können ausfallen (Router, Switch, Firewall, Kabel …)
    - Externe Systeme selbst können instabil sein
  - Externer Fehler nicht auf eigenes System durchschlagen
    - Es wird keine Antwort mehr erwartet (Server- und Clientseitig)

> The Timeouts pattern is useful when you need to protect your system from someone else's Failure.
>
> Michael T. Nygard

Timeouts in Bibliotheken

- **Default-Werte in Bibliotheken meist suboptimal**

  Oftmals kein Timeout Blockierender Thread

- **Bibliotheken bieten meist gute Einstellmöglichkeiten**

  Prüfen welche Werte eingestellt werden können

  Passende Werte für Anwendungsfall verwenden (Beispiel: HTTP-Streaming)

- **Jeder Resource Pool sollte kon guriert werden**

  Nicht ewig warten! Blockierender Thread

Timeouts in Go

Go Context - API

- **Deadline oder Cancelation Steuerung**
- **Weitergabe von aufrufabhängigen Werten**
  - Request-Scoped Values
  - “ThreadLocal”
- **Context sollte bei jedem Call mitgegeben werden**
  - Umsetzung in Standardbibliothek
  - erster Parameter namens ctx
  - Propagation durch Anwendung

Go Context – Gut zu wissen...

- **Erzeugte Objekte sind nicht veränderbar - „Immutable Objects“**
  - Weitergabe in Go-Routinen ohne Probleme möglich
  - Keine Synchronisierung nötig
- **Context Objekte bilden Hierarchie**
  - Propagation des Status (Timeout, etc) durch Hierarchie
- **Information über Ablauf eines Context über Channel**

```
ctx := context.Background()
ctx, cancel := context.WithTimeout(ctx, 2*time.Second)
defer cancel()

<- ctx.Done()
```

HTTP-Client in Go

Go HTTP-Client – Mit und ohne context.Context

```
response, err := http.Get("http://source-fellows.com")
if err != nil {
      log.Fatal(err)
}
defer response.Body.Close()
```

```
ctx:= context.Background()
req, err := http.NewRequestWithContext(ctx, http.MethodGet, "url", nil)
...
response, err := http.DefaultClient.Do(req)
if err != nil {
      return
}
defer response.Body.Close()
```


HTTP-Client Timeouts

- **Standard-Timeout-Wert nicht für Produktion geeignet**

  Timeout-Wert bei 0 unendlich!

  Context scha t Abhilfe wenn Timeout dort angegeben

- **Timeout Werte können an Client Objekt gesetzt werden**

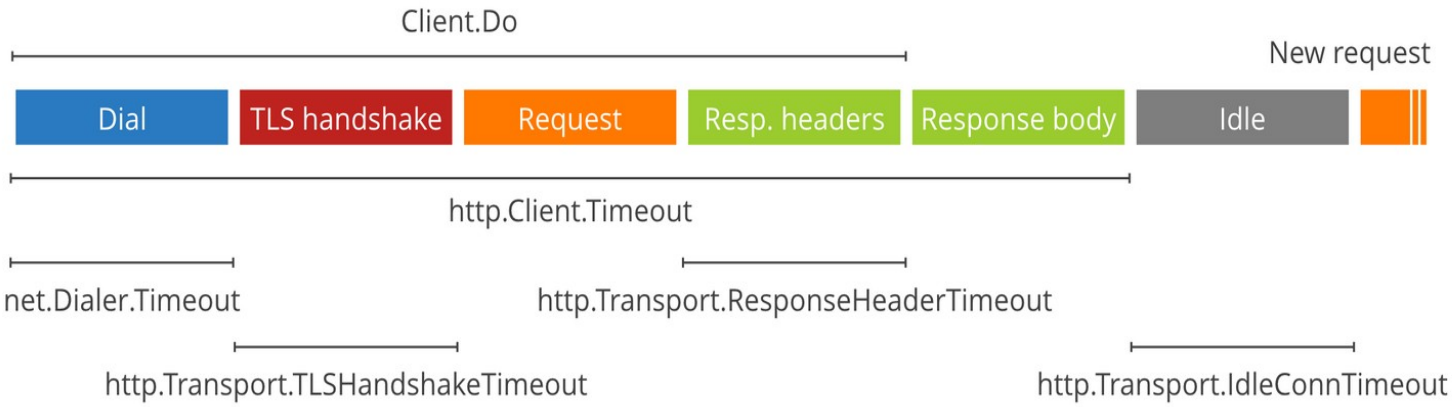
  Kon guration des „DefaultClient“ global

  ```
  http.DefaultClient.Timeout = 3*time.Second
  ```

  Nutzen eines separaten Clients für jedes Backend

  ```
  client := http.Client{Timeout: 3*time.Second}
  client.Do(...)
  ```

Genauere Kon guration der Timeouts des HTTP-Clients



Quelle: https://blog.cloud are.com/the-complete-guide-to-golang-net-http-timeouts/

ToxiProxy – Test Harness

- **„Toxiproxy is a framework for simulating network conditions“**
  - A TCP proxy written in Go
  - Manipulate the health via HTTP
- **Entstanden bei Shopify**
  - OpenSource - lizensiert mit MIT License
  - https://github.com/Shopify/toxiproxy
- **Go und andere Client-Bibliotheken vorhanden**

- Toxics
  - latency
  - down
  - bandwidth
  - slow_close
  - timeout
  - reset_peer
  - slicer
  - limit_data

Beispiel

sourcefellows

Immer über Timeouts nachdenken und entsprechend setzen. Testen.

sourcefellows

Circuit Breaker

sourcefellows

Analog zur elektrischen Sicherung.

sourcefellows

Circuit Breaker – Die Sicherung für Backends

- **Integrationspunkte nicht mehr aufrufen wenn sie Probleme haben**

  Zu viele oder bestimmte Fehler

- **Nutzung zusammen mit sinnvollen Timeouts**

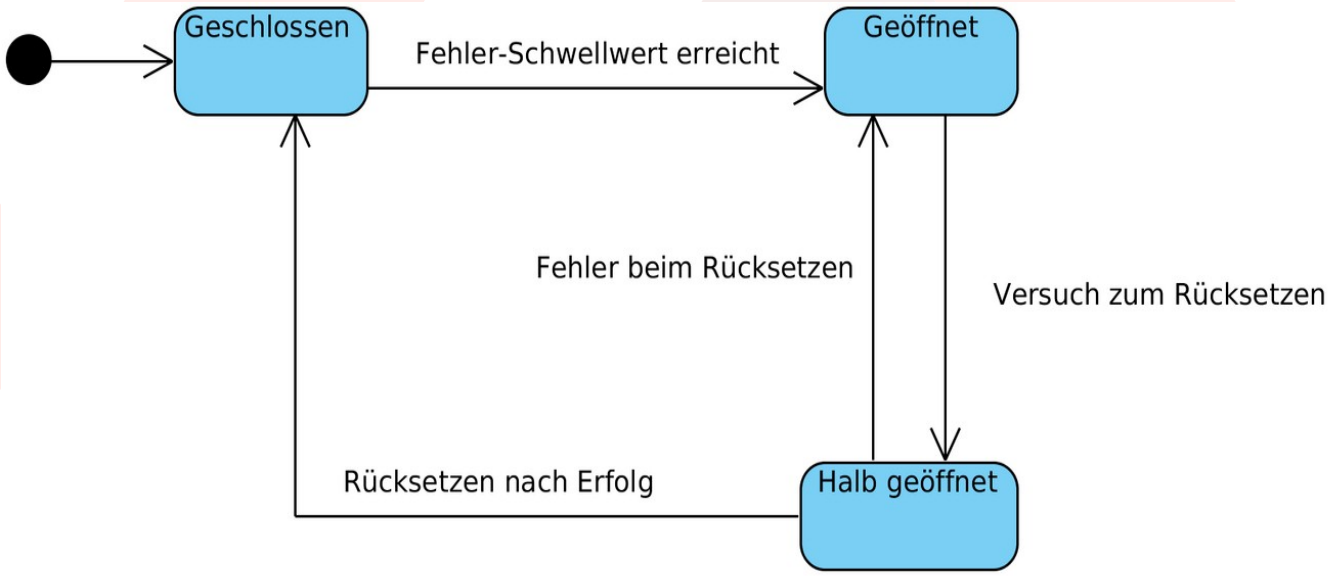
  Bei Abbruch eines Aufrufs ist Problem vorhanden

  Blockierende Aufrufe werden ohne Timeout nicht als Fehler gesehen

- **Statusänderungen am Circuit Breaker transparent machen**

  Es treten gravierende Fehler auf

Circuit Breaker Pattern



Circuit Breaker in Go

- **GoBreaker – OpenSource Bibliothek**

  Circuit Breaker implemented in Go - MIT Lizenz

  https://github.com/sony/gobreaker

- **Wrapping für Methoden**

  Eventuell auftretende Fehler werden für Status verwendet

```
func (cb *CircuitBreaker[T]) Execute(req func() (T, error)) (T, error)
```


Beispiel

Circuit Breaker – Vorsicht beim Einsatz...

- **Abhängigkeiten beachten**
  - Was bedeutet der Ausfall für andere Komponenten?
  - Sind andere Komponenten auf Fehler vorbereitet?
- **Mögliche Kettenreaktionen überdenken**
  - Backendaufruf liefert immer Fehler
    - Welche Auswirkungen hat das?
    - Fehler kommt eventuell schneller...
  - Eventuell ganze Komponenten stoppen
    - Einsatz eines context.Context

Bulkhead

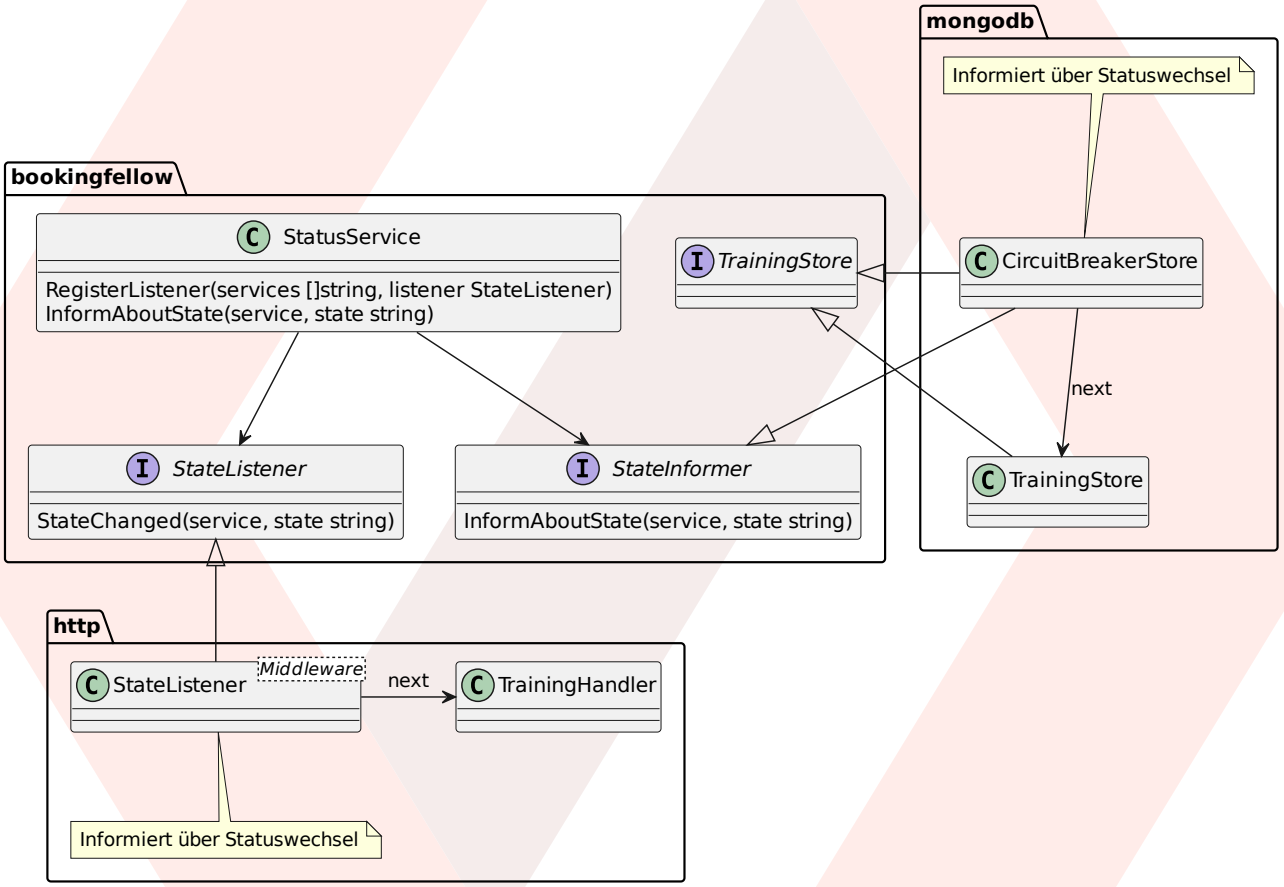
sourcefellows

Bulkhead

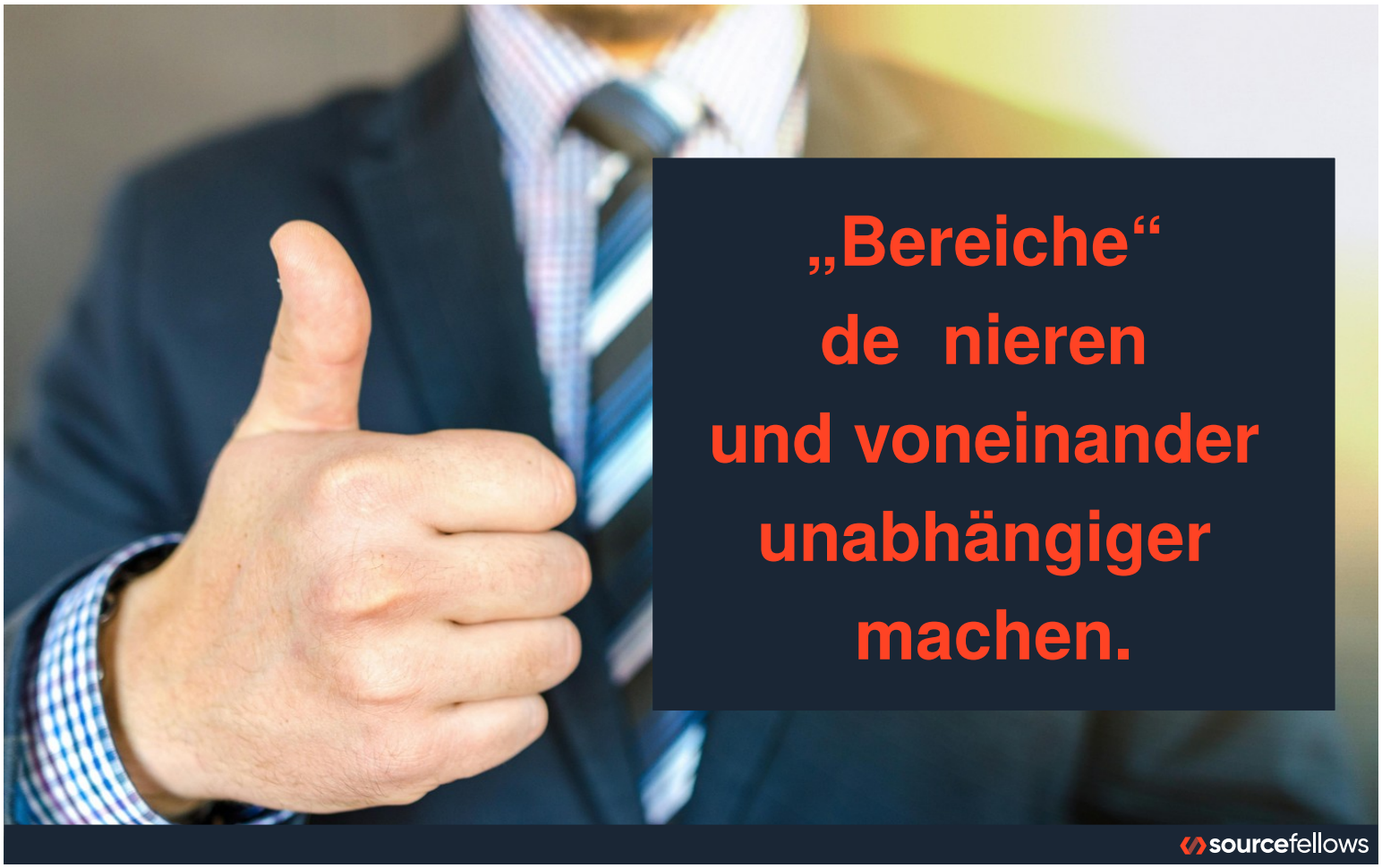
- **Partitionierung des Systems**
  - Redundanz von Systemen ist einfachste Möglichkeit
  - Anwendung: Server bestimmten Aufgaben zuordnen
  - Trennung innerhalb von Anwendungen
- **Abhängigkeiten zwischen Anwendungen über Drittanwendungen**
  - Trennung innerhalb von Anwendung nötig

Bulkhead in Go

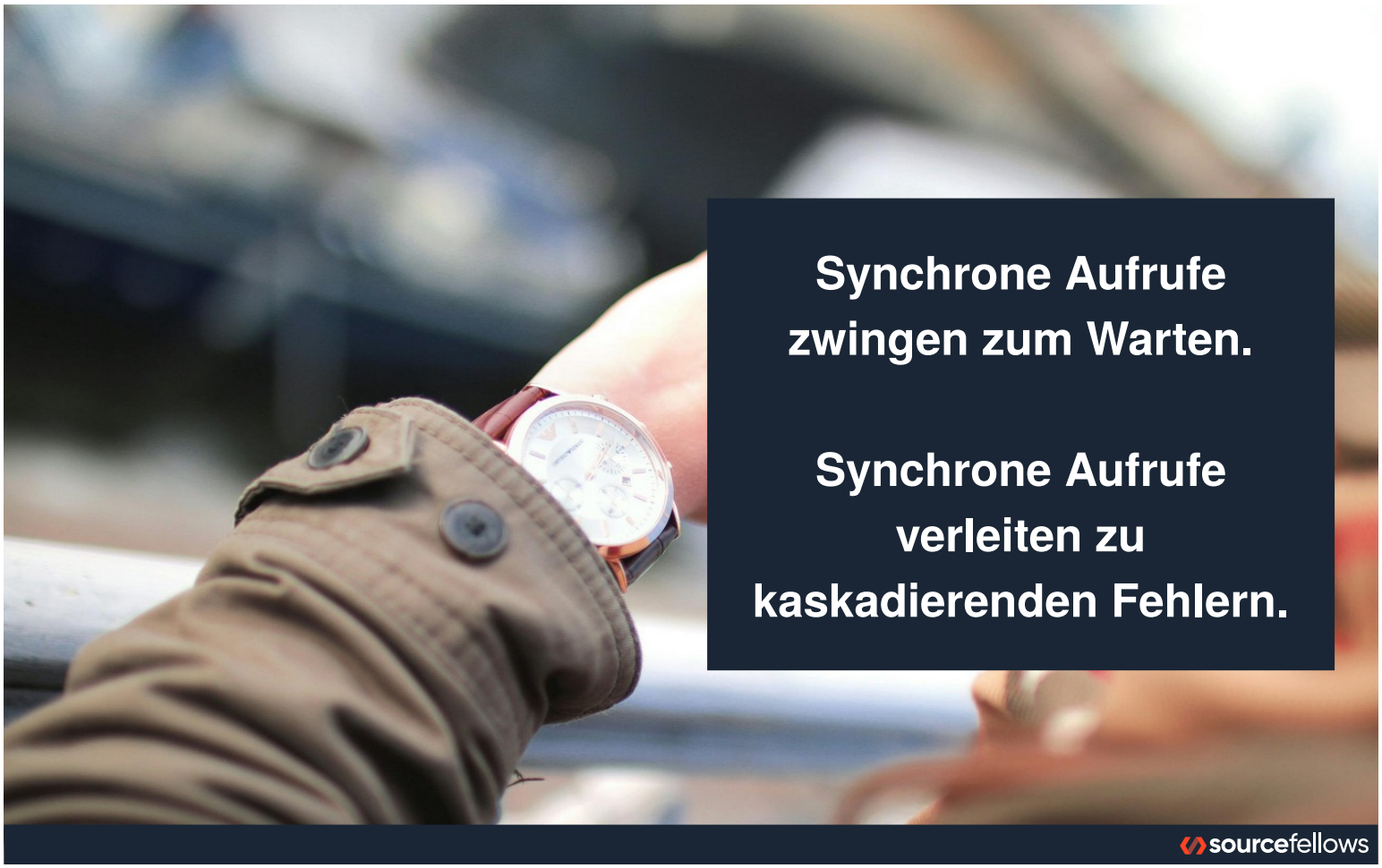
- **Separate Pools für Integration Points verwenden**
  - HTTP-Client besitzt Pool für Serververbindungen
  - Eigene HTTP-Clients
- **Trennung eingehender Verbindungen**
  - Unterschiedliche Ports
  - Eigener Listener für Pooling verwenden
    - Standardimplementierung wird über Betriebssystem limitiert
    - Bsp https://pkg.go.dev/golang.org/x/net/netutil#LimitListener
- **Serverstart auch ohne Backend ermöglichen**
  - Lazy-Init, Wiederholungen, Circuit-Breaker

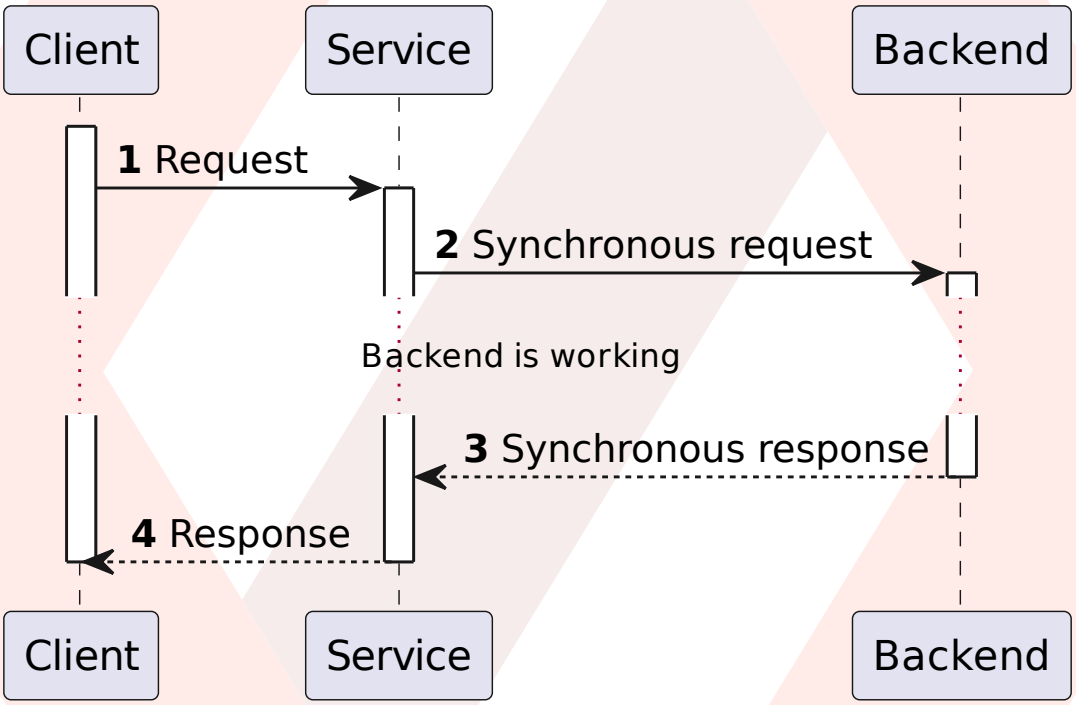

mongodb
Informiert über Statuswechsel
bookingfellow
StatusService
RegisterListener(services []string, listener StateListener)
InformAboutState(service, state string)
TrainingStore
CircuitBreakerStore
next
StateListener
StateChanged(service, state string)
StateInformer
InformAboutState(service, state string)
TrainingStore
http
StateListener
Middleware
next
TrainingHandler
Informiert über Statuswechsel

Beispiel

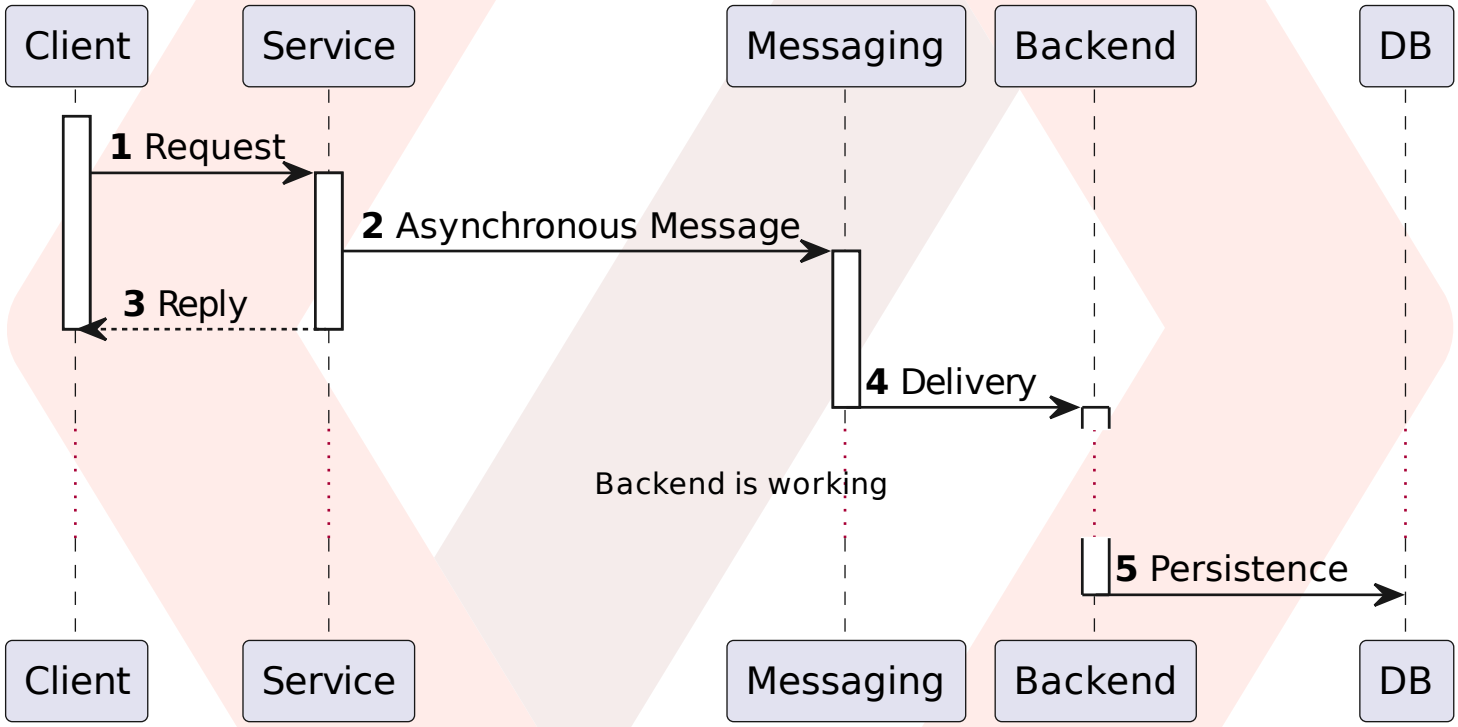


Decoupling via Messaging


Synchrone Aufrufe
zwingen zum Warten.
Synchrone Aufrufe
verleiten zu
kaskadierenden Fehlern.
sourcefellows


Client
Service
Backend
1 Request
2 Synchronous request
Backend is working
3 Synchronous response
4 Response
Client
Service
Backend

Asynchrone Kommunikation



Messaging führt zur Entkopplung

- **Zentraler Broker übernimmt Nachrichten-Verwaltung**
  - Sender und Consumer senden bzw. empfangen Nachrichten
- **Messaging führt zur zeitlichen Entkopplung (Resilience)**
  - Sender muss nicht auf eine Antwort warten
  - Vermeidung von kaskadierten Fehlern
- **Architektur- und eventuell fachliche Anpassungen nötig**
- **Komplexität der Anwendung steigt**
  - Antwort (auch Fehler) muss asynchron verarbeitet werden
  - Neue Infrastrukturkomponenten

Beispiel

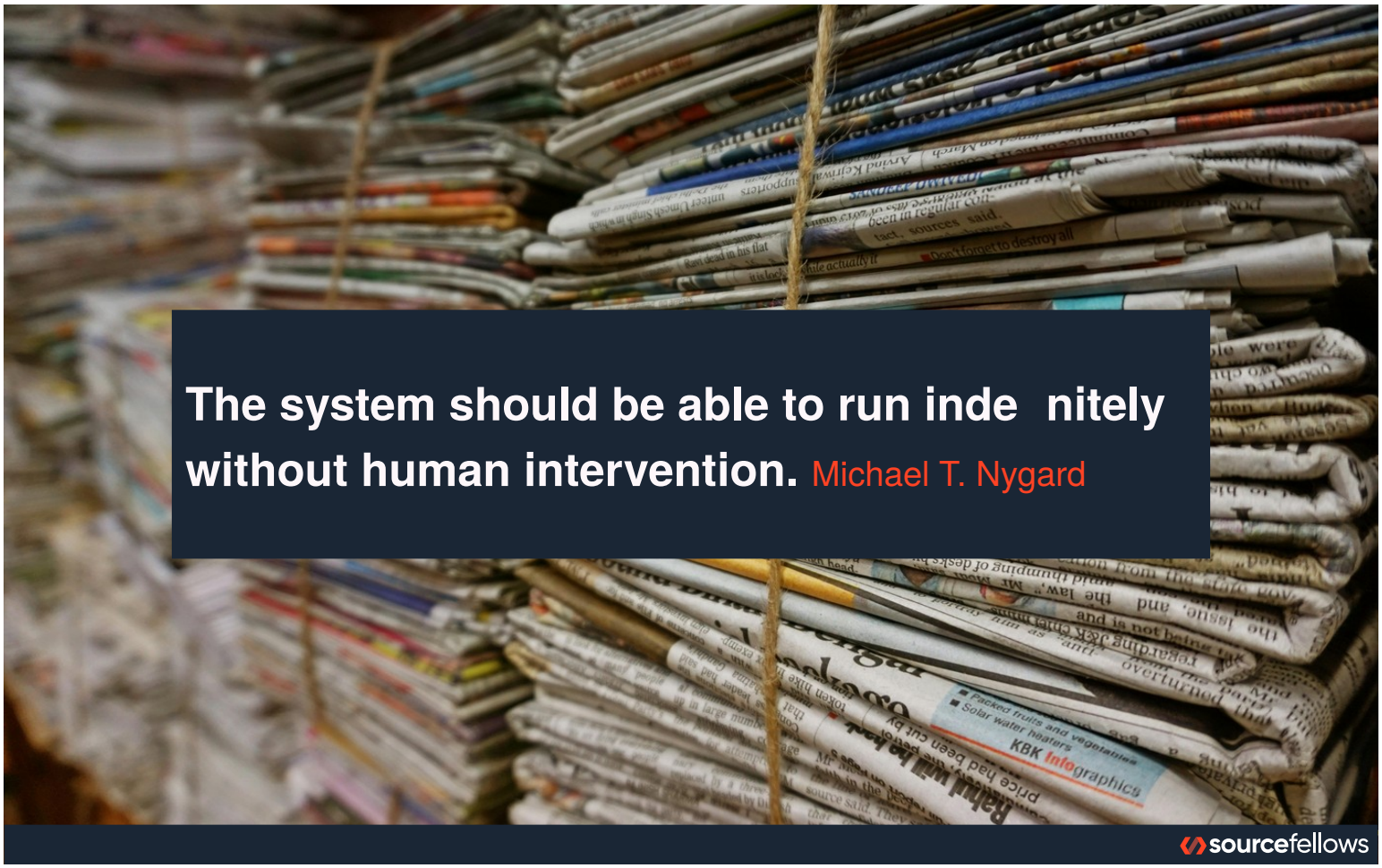
sourcefellows

Lose Kopplung mittels Messaging.

sourcefellows

Steady State

sourcefellows



sourcefellows

Steady State

- **Systeme sammeln Daten – manchmal ohne Cleanup**

  Logs, Datenbank, Benutzer-Uploads, etc
- **Cloud-Speicher verleitet zum Halten der Daten**
- **Für jeden Sammelmechanismus einen Cleanup einrichten**

  Alte Daten löschen, komprimieren, archivieren
- **Zu viele Daten können zu Instabilität führen**

  Lange Ladezeiten, Höhere Latenz, Höhere Last

  Speicher geht beim Laden aus

  ...

It sometimes seems that you'll be lucky if the system ever runs at all in the real world. The notion that it will run long enough to accumulate too much data to handle seems like a "high-class problem"—the kind of problem you'd love to have.

Keine Auswirkung durch Steady State!

- **Cleanup Jobs integrieren**

  Zu Beginn einplanen, umsetzen und kontrollieren

  Sinnvolle Lebensdauer und Datenvolumen bestimmen
- **Memory-Caches in Anwendungen**

  Obergrenzen für Mengengerüst festlegen
- **Queries Beschränken bzw. Paging beim Laden**

  Zur Absicherung der Stabilität falls sich Daten sammeln

Vielen Dank für Ihre Aufmerksamkeit!

sourcefellows

Fragen?