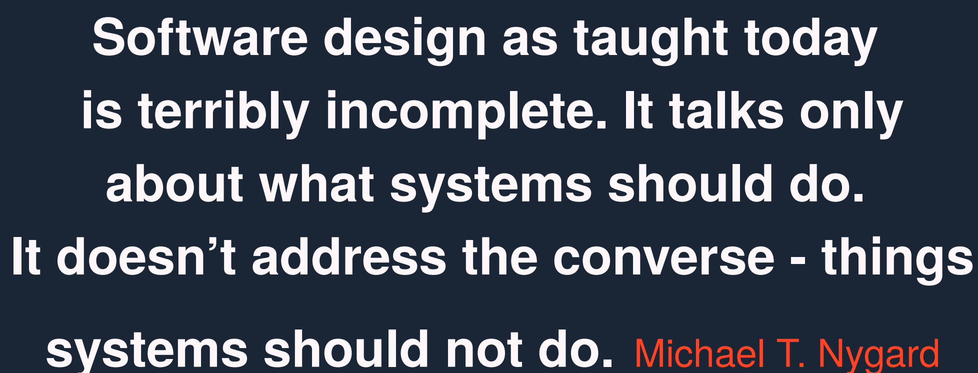
Resilience-Patterns in Cloud-Anwendungen

Kristian Köhler

DATEV Coding Festival 2024



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/ Druck 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

Timeouts

- **Timeout steuert Abbruch der Verarbeitung**
  - Es wird keine Antwort mehr erwartet
  - Blockierende Threads können System lahmlegen
    - Deadlocks
    - Resource Pools können ausgeschöpft werden
- **Timeouts bieten Isolation von Fehlern**
  - Externe Fehler sollen eigenes System nicht gefährden
    - Netzwerkfehler (Ausfall von Router, Switch, Firewall, Kabel …)
    - Externe Systeme selbst können instabil sein
      - Ereignisse zu jeder Zeit möglich

> 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 kon guriert   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 sinnvoll kon guriert werden**
  - Nicht ewig warten!   Blockierender Thread
  - Langsame Antworten können ebenfalls zu Problemen führen
  - Eventuell Anfrage in Warteschlange stellen und später ausführen

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
- **Verschiedene Client-Bibliotheken vorhanden**
  - Go, Java, etc.

Timeouts in Java Beispiel HttpClient

Apache HttpClient 5

- **Connection Timeout**

  Zeitspanne für Verbindungsaufbau (`http.connection.timeout`)
- **Socket Timeout**

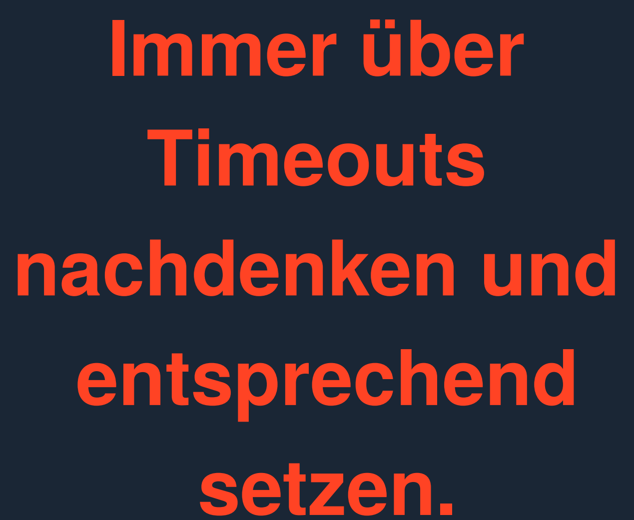
  Zeitspanne um auf Daten zu warten (`http.socket.timeout`)
- **Connection Manager Timeout**

  Zeitspanne die auf eine Verbindung aus dem Verbindungspool gewartet wird (`http.connection-manager.timeout`)
- **Vorsicht mit automatischem Retry!! (RetryStrategy)**
- **Allerdings kein „übergreifender“ Request-Timeout vorhanden**

  Beispiel DNS-Abfragen haben Probleme

  Eventuell Abbruch von Anfragen mit Abort-Funktionalität

Beispiel



Circuit Breaker

sourcefellows

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**

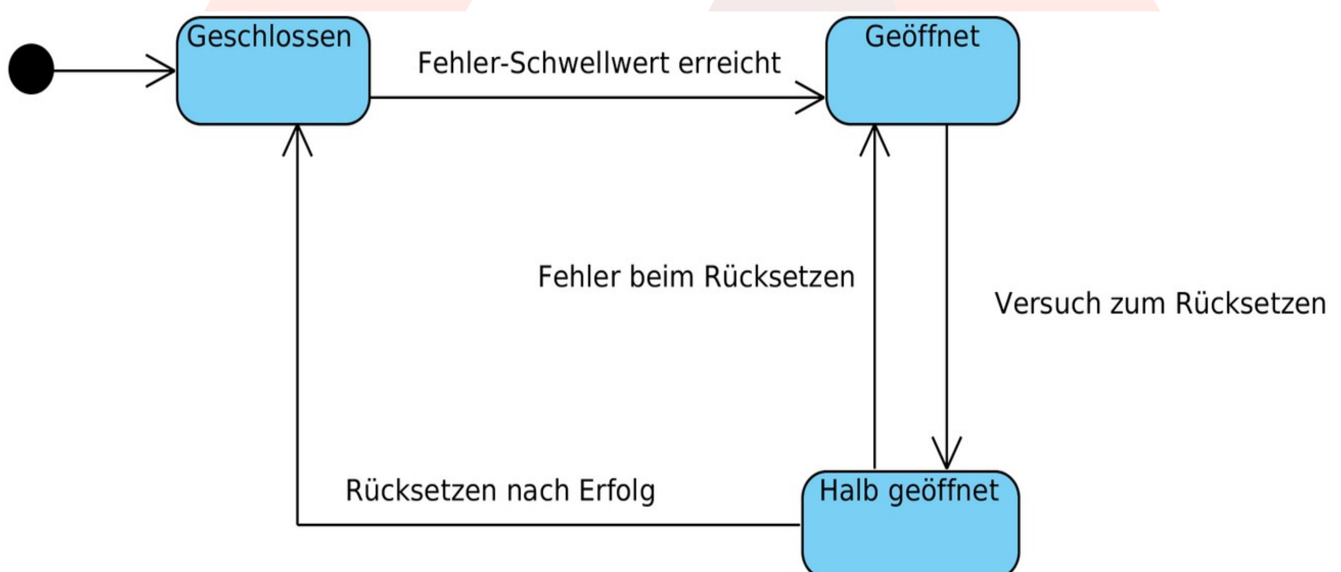
  Bei Abbruch eines Aufrufs ist Problem vorhanden

  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)
```


Circuit Breaker in Java mit Bibliotheken (Beispiele)

- **Net ix Hystrix**

  Wrapping in HystrixCommand, Last Stable Release 2018

  „*Hystrix is no longer in active development, and is currently in maintenance mode.*“

- **Resilience4j**

  Open Source Projekt unter Apache Lizenz

  - https://github.com/resilience4j/resilience4j

  Dekoratoren für Funktionalität

  Integration in Spring oder Micronaut vorhanden

Resillient4J und Spring Framework Beispiel

Java

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

private Mono<String> fallback(String param1, CallNotPermittedException e) {
    return Mono.just("Handled the exception when the CircuitBreaker is open");
}

private Mono<String> fallback(String param1, BulkheadFullException e) {
    return Mono.just("Handled the exception when the Bulkhead is full");
}

private Mono<String> fallback(String param1, NumberFormatException e) {
    return Mono.just("Handled the NumberFormatException");
}

private Mono<String> fallback(String param1, Exception e) {
    return Mono.just("Handled any other exception");
}
```

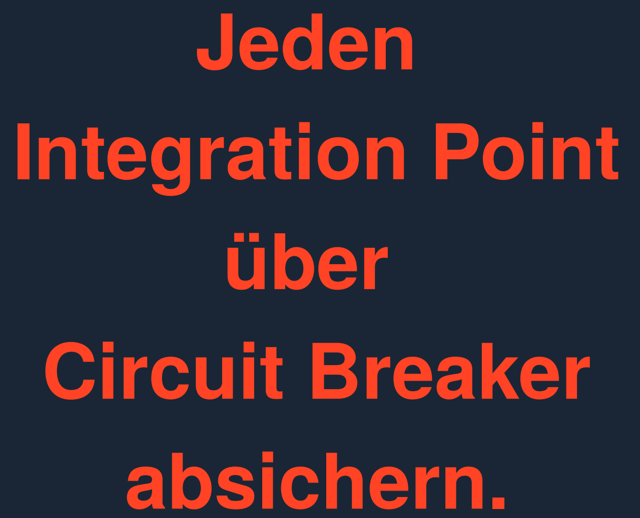
Beispiel

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

Vorsicht mit „Magic“ im Code!

Transparenter Einsatz!



Bulkhead

„Schotten“ sollen, wie in der Schi fahrt, verhindern, dass der Ausfall einer Komponente das gesamte System beeinträchtigt.

#Schadensminimierung

Bulkhead

- **Partitionierung des Systems**

  Redundanz von Systemen ist einfachste Möglichkeit

  Anwendung: Server bestimmten Aufgaben zuordnen

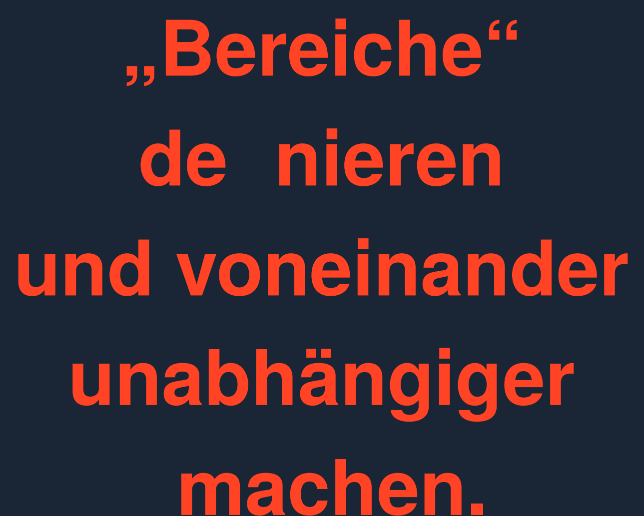
  Trennung innerhalb von Anwendungen

- **Abhängigkeiten zwischen Anwendungen über Drittanwendungen**

  Trennung innerhalb von Anwendung nötig

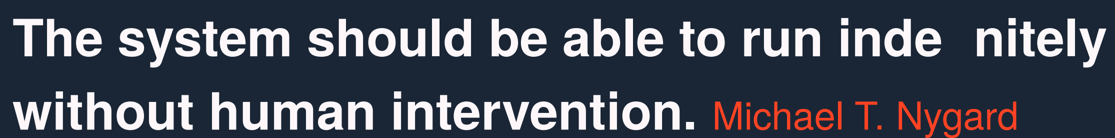
Bulkhead in Java

- **Separate Pools für Integration Points verwenden**
  Jeder Integration Point eigenen HTTP-Client
- **Trennung eingehender Verbindungen**
  Unterschiedliche Ports
- **Einsatz von Bibliotheken andenken**
  Zum Beispiel Resillience4J
- **Serverstart auch ohne Backend ermöglichen**
  Lazy-Init, Wiederholungen, Circuit-Breaker



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

Fail Fast

Wenn langsame Antworten schlimmer sind als gar keine Antworten, dann ist das Schlimmste sicher ein langsam gemeldeter Fehler. Michael T. Nygard

Fail Fast

- **Vor einer Transaktion prüfen ob Resourcen vorhanden**
  - Kombination mit Circuit Breaker Status
- **Schnell abbrechen!**
  - Nicht warten ob System doch noch reagiert (Bsp. Load-Balancer)
- **Frühe Validierung von Eingaben**
  - Prüfung z. B. in HTTP-Handler
  - Requests abweisen, die später Probleme machen können
- **Passende Fehlermeldungen liefern**
  - Unterscheidung Benutzer und Systemfehlern

Decoupling

Synchrone Aufrufe verleiten zu kaskadierenden Fehlern.

Messaging führt zur Entkopplung

- **Zentraler Broker übernimmt Nachrichten-Verwaltung**
  - Sender und Consumer senden bzw. empfangen Nachrichten
  - Persistenz der Nachrichten möglich
- **Messaging führt zur zeitlichen Entkopplung**
  - Sender muss nicht auf eine Antwort warten
  - Vermeidung von kaskadierten Fehlern
  - „Unterbrechung“ einer Transaktion
- **Komplexität der Anwendung steigt**
  - Antwort (auch Fehler) muss asynchron verarbeitet werden
  - Neue Infrastrukturkomponenten

Vielen Dank für Eure Aufmerksamkeit!