Eberhard Wolff

# Microservices Rezepte

## Technologien im Überblick

INNOQ

# Microservices Rezepte

Eberhard Wolff

Dieses Buch wird verkauft unter
http://leanpub.com/microservices-rezepte

Diese Version wurde veröffentlicht am 2020-09-30

# Ebenfalls von Eberhard Wolff

Microservices Primer

Microservices - Ein Überblick

Microservices - A Practical Guide

Microservices Recipes

Domain-Driven Design Referenz

Service Mesh Primer

# Inhaltsverzeichnis

# Einführung

Diese Broschüre führt die Begriffe Microservice und Self-contained System ein. Dann gibt es einen Überblick über verschiedene Konzepte und Rezepte zur Implementierung von Microservices. Die Rezept-Metapher drückt aus, dass der Text jeden Ansatz *praktisch* beschreibt und eine *Implementierung als Beispiel* zur Verfügung steht. Außerdem muss der Leser für sein Projekt mehrere Rezepte *kombinieren*, wie er es für ein Menü eines mehrgängigen Essens auch tun müsste. Und schließlich gibt es zu jedem Rezept *Variationen und Alternativen. Experimente* laden dazu ein, sich selbst mit den Beispielen zu beschäftigen.

Der Code für die Beispiele liegt auf GitHub. Es gibt eine Übersicht[1], die jede Demo in der Broschüre und noch einige andere kurz erläutert.

Eine ausführliche Darstellung der Rezepte und weiterer Konzepte rund um Microservices finden sich im Microservices-Praxisbuch[2].


## Danksagung

Ich möchte allen danken, mit denen ich über Microservices diskutiert habe, die mir Fragen gestellt oder mit mir zusammengearbeitet haben. Es sind viel zu viele, um sie alle zu nennen. Der Dialog hilft sehr und macht Spaß!

Viele der Ideen und auch die Umsetzungen sind ohne meine Kollegen bei der INNOQ nicht denkbar.

Danke an Jörn Hameister für das umfangreiche Feedback!


[1] http://ewolff.com/microservices-demos.html
[2] http://microservices-praxisbuch.de/

# Grundlagen: Microservices

Der Begriff Microservice ist leider nicht einheitlich definiert, sodass diese Broschüre zunächst Microservices und andere grundlegende Ideen darstellen muss.

## Independent-Systems-Architecture-Prinzipien (ISA)

ISA[3] (Independent Systems Architecture) ist eine Sammlung von grundlegenden Prinzipien für Microservices. Sie basiert auf Erfahrungen mit Microservices in vielen verschiedenen Projekten.

## Bedingungen

Bei den Prinzipien wird *"muss"* verwendet für Prinzipien, die unbedingt eingehalten werden müssen. *"Sollte"* beschreibt Prinzipien, die viele Vorteile haben, aber nicht unbedingt eingehalten werden müssen.

Die ISA-Prinzipien sprechen von einem *System*. Die IT-Landschaft eines Unternehmens besteht aus vielen Systemen. Jedes System kann mit einer anderen Architektur und daher mit anderen Prinzipien implementiert sein.

[3] http://isa-principles.org

# Prinzipien

1. Das System muss in *Module* unterteilt werden, die *Schnittstellen* bieten. Der Zugriff auf andere Module ist nur über diese Schnittstellen möglich. Module dürfen daher nicht direkt von den Implementierungsdetails eines anderen Moduls abhängen, wie zum Beispiel den Datenmodellen in der Datenbank.
2. Module müssen *separate Prozesse, Container oder virtuelle Maschinen* sein, um die Unabhängigkeit zu maximieren.
3. Das System muss zwei klar getrennte Ebenen von Architekturentscheidungen haben:
   - Die *Makro-Architektur* umfasst Entscheidungen, die alle Module betreffen.
   - Die *Mikro-Architektur* sind jene Entscheidungen, die für jedes Modul anders getroffen werden können.

   Alle weiteren Prinzipien ab Punkt 4 sind Teil der Makro-Architektur.
4. Die Wahl der *Integrations-Optionen* muss für das System begrenzt und standardisiert sein. Die Integration kann mit synchroner oder asynchroner Kommunikation stattfinden und / oder auf Frontend-Ebene.
5. *Kommunikation* muss auf einige Techniken wie REST oder Messaging begrenzt und standardisiert sein. Auch Metadaten, zum Beispiel zur Authentifizierung, müssen standardisiert sein.
6. Jedes Modul muss seine *eigene unabhängige Continuous-Delivery-Pipeline* haben. Tests sind Teil der Continuous-Delivery-Pipeline, folglich müssen die Tests der Module unabhängig sein.
7. *Betrieb* sollte standardisiert werden. Dies beinhaltet Konfiguration, Deployment, Log-Analyse, Tracing, Monitoring und Alarme. Es kann Ausnahmen vom Standard geben, wenn ein Modul sehr spezifische Anforderungen hat.

8. *Standards* für Betrieb, Integration oder Kommunikation sollten auf Schnittstellenebene definiert werden. Das Protokoll kann als REST standardisiert sein, und Datenstrukturen können standardisiert werden. Aber jedes Modul sollte frei sein, eine andere REST-Bibliothek zu verwenden.
9. Module müssen *resilient* sein. Sie dürfen nicht ausfallen, wenn andere Module nicht verfügbar sind oder Kommunikationsprobleme auftreten. Sie müssen in der Lage sein, heruntergefahren zu werden, ohne Daten oder Zustand zu verlieren. Es muss möglich sein, sie auf andere Umgebungen (Server, Netzwerke, Konfiguration usw.) zu verschieben.

## Begründung

ISA zeigt, dass Microservices eine Art der Modularisierung sind (Prinzip 1). Dadurch können Ideen wie Information Hiding[4] oder High Cohesion[5] / Low Coupling für Microservices übernommen werden. Neu ist bei Microservices die Umsetzung als separate Container (Prinzip 2). Das erlaubt mehr Freiheit bei der technischen Umsetzung der Module, die jedoch mindestens so weit eingeschränkt werden muss, dass man noch von einem Gesamtsystem sprechen kann (Prinzip 3). Also müssen Integration (Prinzip 4) und Kommunikation (Prinzip 5) standardisiert sein, weil das System sonst in mehrere Inseln zerfällt.

Unabhängiges Deployment ist ein Vorteil von Microservices und kann nur mit eigener Continuous-Delivery-Pipeline sichergestellt werden (Prinzip 6).

Die Standardisierung des Betriebs (Prinzip 7) erleichtert den Betrieb, gerade bei einer großen Anzahl an Microservices. Aber die Standards dürfen nicht die Technologie-Freiheit beschneiden und

[4] https://de.wikipedia.org/wiki/Datenkapselung_%28Programmierung%29
[5] https://de.wikipedia.org/wiki/Koh%C3%A4sion_%28Informatik%29

daher nur an der Schnittstelle ansetzen (Prinzip 8). Ein Microservices-System wird vielleicht nicht von Anfang an viele unterschiedliche Technologien nutzen, aber man sollte sich die Möglichkeit dazu offenhalten, um bei der Weiterentwicklung besser aufgestellt zu sein.

Microservices sind ein verteiltes System. Das erhöht die Wahrscheinlichkeit, dass ein Server, das Netzwerk oder ein Microservice ausfällt. Daher ist Resilience (Prinzip 9) unerlässlich. Außerdem können durch die Resilience Microservices auf eine andere Umgebung verschoben werden, was für den Betrieb im Cluster von Vorteil ist.

So ergeben die ISA-Prinzipien eine gute Grundlage für ein Microservices-System.

Weitere Informationen, Links und eine ausführlichere Begründung finden sich unter http://isa-principles.org/.

## Self-contained Systems

Die ISA-Prinzipien definieren wichtige Prinzipien für Microservices, aber lassen bewusst Fragen offen, wie beispielsweise die Organisation, die fachliche Aufteilung oder ob Microservices eine UI enthalten sollen oder nicht.

Self-contained Systems (SCSs) sind ein Ansatz für Microservices, der sich schon in vielen Projekten bewährt hat. Alle wesentlichen Informationen zu SCSs finden sich auf der Website http://scs-architecture.org/ . Hier ein Überblick über die Eigenschaften:

- Jedes SCS ist eine *autonome Webanwendung*. Der Code für die Darstellung der Weboberfläche ist in dem SCS enthalten. So kann ein SCS seine Funktionalitäten erbringen, ohne auf andere SCSs angewiesen zu sein.

- SCSs dürfen sich *keine UI teilen*. Schließlich soll ein SCS seine Features über seine eigene UI nutzbar machen.
- Ein SCS kann eine *optionale API* haben. Das ist aber nicht unbedingt notwendig, da das SCS bereits eine Weboberfläche für die Benutzer hat. Für mobile Clients oder andere SCSs kann der Zugriff über eine API aber nützlich sein.
- Das SCS enthält *Daten und Logik*. Ein Feature wird typischerweise Anpassungen in UI, Logik und Daten erfordern. Alle diese Änderungen können in einem SCS vorgenommen werden.
- Für ein SCS ist immer genau *ein Team* verantwortlich. Ein Team kann jedoch durchaus mehrere SCSs betreuen.
- Um eine enge Kopplung zu vermeiden, sollten SCSs sich *keinen fachlichen Code teilen*. Nur gemeinsamer technischer Code ist erlaubt. Als grobe Regel gilt: Nur Code, den man als Open Source veröffentlichen würde, darf zwischen SCS geteilt werden.
- Um die SCS weiter zu entkoppeln, sollten sich die SCS *keine Infrastruktur teilen*, also beispielsweise keine gemeinsame Datenbank nutzen. Aus Kostengründen können allerdings Kompromisse eingegangen werden.
- Die *Kommunikation* zwischen SCSs ist *priorisiert*:
  - Frontend-Integration hat die höchste Priorität.
  - Dann folgt asynchrone Kommunikation.
  - Und schließlich ist auch synchrone Kommunikation möglich.

    Dabei liegt der Fokus auf Entkopplung und Resilience. Die höher priorisierten Kommunikationsarten helfen beim Erreichen dieser Ziele.

Die SCS-Idee hat sich in vielen Projekten schon bewährt. Die Links der Website[6] geben einen Eindruck von der Nutzung. Sie ist in Reinform nur für Webanwendungen nutzbar, da zu jedem SCS eine

[6] http://scs-architecture.org/links.html

Weboberfläche gehört. Die konsequente Trennung in Systeme, die eine Fachlichkeit implementieren und von einem Team entwickelt werden, ist aber auch für andere Arten von Systemen sinnvoll.

## Fazit & Ausblick

Die Independent-Systems-Architecture (ISA) definiert die Prinzipien, denen alle Microservice-Systeme entsprechen sollten, während Self-contained Systems Best Practices definieren, die in vielen Projekten erfolgreich eingesetzt worden sind.

Die weiteren Kapitel beschreiben technische Rezepte für die Kommunikation zwischen Microservices, und zwar in der Reihenfolge der Prioritäten bei SCS: Frontend-Integration, asynchrone Kommunikation und schließlich synchrone Kommunikation.

# Konzept: Frontend-Integration

Microservices können ein Web-Frontend enthalten. Self-contained Systems müssen sogar ein Web-Frontend haben. Daher können Microservices am Frontend integriert werden.

## Warum Frontend-Integration?

Frontend-Integration erzeugt eine *lose Kopplung*. Wenn Links für die Integration verwendet werden, muss für eine Integration nur eine URL bekannt sein. Was sich hinter der URL verbirgt und wie die Informationen dargestellt werden, kann geändert werden, ohne dass es das System beeinflusst, das die URL in einem Link anzeigt.

Ein weiterer Vorteil von Frontend-Integration ist die *freie Wahl der Frontend-Technologien*. Gerade bei Frontend-Technologien gibt es sehr viele Innovationen. Ständig gibt es neue JavaScript-Frameworks und neue Möglichkeiten, attraktive Oberflächen zu gestalten. Ein wichtiger Vorteil von Microservices ist die Technologie-Freiheit. Jeder Microservice kann eigene Technologien wählen. Wenn Technologie-Freiheit auch für das Frontend gelten soll, dann muss jeder Microservice sein eigenes Frontend mitbringen, das eine eigene Technologie nutzen kann. Dazu müssen die Frontends geeignet integriert sein.

Dank Frontend-Integration ist für eine Funktionalität *alle Logik in einem Microservice* implementiert. Beispielsweise kann ein Microservice dafür verantwortlich sein, eingegangene Nachrichten anzuzeigen, selbst wenn die Nachrichten in der UI eines anderen

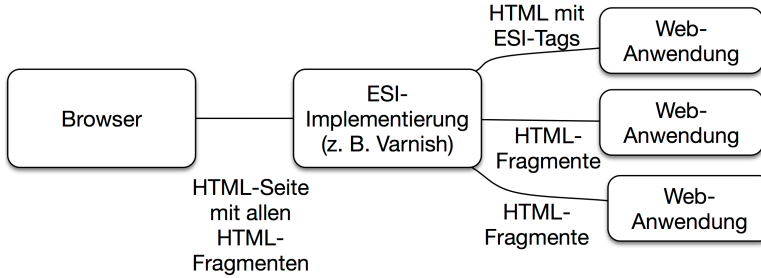
Microservice integriert dargestellt werden. Wenn weitere Informationen wie beispielsweise eine Priorität angezeigt werden sollen, kann die Logik, die Datenhaltung und auch die Darstellung durch Änderung von nur einem Microservice umgesetzt werden, selbst wenn ein anderer Microservice die Darstellung nutzt.

## Rezept: ESI (Edge Side Includes)

ESI[7] (Edge Side Includes) ermöglichen es einem Microservice, HTML-Fragmente anderer Microservices zu integrieren. Dazu erzeugt der Microservice HTML, das ESI-Tags enthält. Die ESI-Implementierung wertet die ESI-Tags aus und bindet dann an den richtigen Stellen HTML-Fragmente anderer Microservices ein.

ESI wird vor allem von Caches implementiert. Durch die Aufteilung der HTML-Seiten können statische Fragmente gecacht werden, selbst wenn sie in einer dynamischen Website integriert sind. CDNs (Content Delivery Networks) können ebenfalls ESI implementieren. CDNs dienen eigentlich dazu, statische HTML-Seiten und Images auszuliefern. Dazu betreiben CDNs Server an Knotenpunkten im Internet, sodass jeder Nutzer die Seiten und Images von einem Server in der Nähe laden kann und die Ladezeiten reduziert werden. Mit ESI können die CDNs zumindest Fragmente dynamischer Seiten ebenfalls cachen.

[7] https://www.w3.org/TR/esi-lang




**ESI-Konzept: HTML-Fragmente zu HTML-Seiten integrieren**

Also setzt ESI eine HTML-Seite aus mehreren HTML-Fragmenten zusammen, die von verschiedenen Microservices geliefert werden können.

Ein Beispiel für eine ESI-Integration steht unter https://github.com/ewolff/SCS-ESI bereit. Die Anleitung, um das Beispiel auszuprobieren, findet sich unter https://github.com/ewolff/SCS-ESI/blob/master/WIE-LAUFEN.md.

Im Beispiel dient die ESI-Integration dazu, Fragmente eines Common-Microservices in alle Microservices zu integrieren. Das Beispiel enthält nur einen konkret implementierten Microservice, nämlich den Order-Microservice. Der Order-Microservice ist eine Spring-Boot-Anwendung und in Java geschrieben, während der Common-Microservice in Go geschrieben ist. Das zeigt, dass auch sehr unterschiedlichen Technologien im Frontend integriert werden können. Der Varnish Cache ist auch dafür zuständig, die Requests an den richtigen Microservice weiterzuleiten. Damit nimmt er die Rolle eines Reverse Proxys ein.

**Listing 1: Vom Order-Microservice ausgegebenes HTML**

```
<html>
<head>
...
  <esi:include src="/common/header"></esi:include>
</head>

<body>
  <div class="container">
    <esi:include src="/common/navbar"></esi:include>
    ...
  </div>
  <esi:include src="/common/footer"></esi:include>
</body>
</html>
```

Der Order-Microservice gibt eine HTML-Seite wie in Listing 1 aus. Eine solche Seite steht unter http://localhost:8090/ zur Verfügung, wenn die Docker-Container auf dem lokalen Rechner laufen. Wenn man diese Seite im Browser ansieht, interpretiert der Browser die ESI-Tags nicht, sodass der Browser eine verstümmelte Webseite anzeigt.

Das Beispiel nutzt den Webcache Varnish[8] als ESI-Implementierung. Der Common-Microservice ergänzt die Inhalte für die ESI-Tags. Der Varnish steht unter http://localhost:8080/ bereit, wenn die Docker-Container auf dem lokalen Rechner laufen. Listing 2 zeigt das HTML, das der Varnish ausgibt.

[8] https://varnish-cache.org/

**Listing 2: Vom Varnish ausgegebenes HTML**

```
<html>
<head>
...
  <link rel="stylesheet"
   href="/common/bootstrap-3.3.7/bootstrap.css" />
  <link rel="stylesheet"
   href="/common/bootstrap-3.3.7/bootstrap-theme.css" />
</head>

<body>
  <div class="container">
    <a class="brand"
     href="https://github.com/ultraq/thymeleaf-layout-dia\
lect">
     Thymeleaf - Layout </a>
    Mon Sep 18 2017 17:52:01 </div></div>
    ...
  </div>
  <script
   src="/common/bootstrap-3.3.7/bootstrap.js" />
</body>
</html>
```

Wie man sieht, ergänzt der Common-Microservice Header und Footer sowie eine Navigationsleiste. Der Common-Microservice implementiert außerdem eine Art Asset-Server: Er stellt gemeinsam genutzte Bibliotheken wie Bootstrap zur Verfügung.

Bei einer neuen Version von Bootstrap muss nur das HTML-Fragment im Common-Microservice geändert und die neue Bootstrap-Version durch den Common-Microservice ausgeliefert werden. In einem produktiven System ist das jedoch kaum wünschenswert, weil die Oberfläche des Order-Microservices sicher mit der neuen Bootstrap-Version getestet werden muss.

### Caching und Resilience

Da das System einen Varnish-Cache nutzt, werden die HTML-Fragmente gecacht, und zwar 30 Sekunden lang. Das kann man an der eingeblendeten Zeit in der Navigationsleiste erkennen, die sich nur alle 30 Sekunden ändert. Wenn einer der Microservices ausfällt, verlängert sich die Zeit für das Caching sogar auf 15 Minuten. Die Konfiguration für Varnish findet sich in der Datei `default.vcl` im Verzeichnis `docker/varnish/` im Beispiel.

Der Varnish-Cache verbessert durch das Caching also nicht nur die Performance des Systems, sondern auch die Resilience.

Bei einer Server-seitigen Integration wird immer die gesamte HTML-Seite mit allen Fragmenten ausgeliefert. Im Beispiel müssen tatsächlich alle Fragmente der Seite vorhanden sein: Die Seite ohne Rahmen und Bootstrap ist eigentlich nicht benutzbar. Eine optionale Information wie die Anzahl der Waren im Warenkorb muss nicht zwangsläufig mit ESI integriert werden.

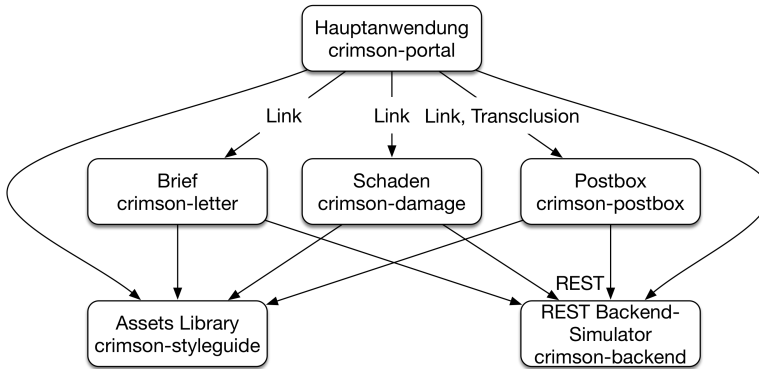
### Alternative: Server-Side Includes (SSI)

Eine andere Option für Server-seitige Frontend-Integration ist SSI[9] (Server-side Includes). Das ist ein Feature, das die meisten Webserver anbieten. In der Integration wird dann der Varnish-Cache durch einen Webserver ersetzt. Das hat den Vorteil, dass ein solcher Webserver beispielsweise für die TLS/SSL-Terminierung vielleicht schon vorhanden ist und daher der Aufwand für die Server-seitige Integration sinkt. Auf der anderen Seite geht der Performance- und Resilience-Vorteil des Caches verloren. https://scs-commerce.github.io/ ist ein Beispiel für ein System, das SSI mit nginx zur Integration der Frontends nutzt.

[9] https://de.wikipedia.org/wiki/Server_Side_Includes

# Alternative Rezepte: Links und JavaScript

Einen ganz anderen Ansatz für Frontend-Integration nutzt das Crimson-Assurance-Beispiel. Das Beispiel ist als Prototyp entstanden, der zeigt, wie eine Webanwendung mit Frontend-Integration umgesetzt werden kann. Die beiden INNOQ-Mitarbeiter Lucas Dohmen und Marc Jansing haben das Beispiel implementiert.




**Integration mit Links und JavaScript**

Es steht unter https://crimson-portal.herokuapp.com/ im Internet zur Verfügung und unter https://github.com/ewolff/crimson-assurance-demo als Docker-Container, um das Beispiel auf dem eigenen Laptop ablaufen zu lassen. https://github.com/ewolff/crimson-assurance-demo/blob/master/WIE-LAUFEN.md erläutert, wie man das Beispiel starten kann.

Dieses Beispiel implementiert eine Anwendung für einen Sachbearbeiter einer Versicherung. Die Hauptanwendung `crimson-portal` hat Links zu den Anwendungen zum Brief-Schreiben `crimson-letter`, für das Melden von Schäden `crimson-damage` und zum REST-Backend-Simulator `crimson-backend`. Diese Links übergeben als Parameter Informationen wie die Vertrags-ID.

Nur für die Integration der Postbox wird zusätzlich mithilfe von ca. 50 Zeilen JavaScript eine Client-seitige Integration implementiert, sodass die Postbox auch in das Portal eingeblendet werden kann. Alle diese Anwendungen haben ein einheitliches Look and Feel, das durch gemeinsam genutzte Assets im Projekt `crimson-styleguide` unterstützt wird. Die Assets werden als Library beim Build in die Projekte integriert.

Dieses Beispiel zeigt, wie weit man mit einer einfachen Integration mit Links kommt. Zudem zeigt auch dieses Beispiel die Integration sehr unterschiedlicher Systeme: Hauptanwendung, Brief und Schaden sind mit Node.js implementiert, während die Postbox mit Java und Spring Boot implementiert ist.

Ein weiteres Beispiel für die Client-seitige Integration ist https://github.com/ewolff/SCS-jQuery, das eine sehr einfache Client-seitige Integration implementiert. Das Beispiel ist identisch mit dem ESI-Beispiel.

Beide Projekte nutzen Links. Die verlinkten Seiten werden durch JavaScript eingeblendet. Selbst wenn das JavaScript aufgrund irgendwelcher Fehler nicht ausgeführt werden kann oder die verlinkte Seite nicht zur Verfügung steht, funktioniert das System dennoch: Es wird einfach ein Link angezeigt, anstatt die Postbox einzublenden.

Da die Client-seitige Integration bei beiden Beispielen in jQuery implementiert ist, muss jedes System diese JavaScript-Bibliothek integrieren, und zwar in einer Version, mit der dieser Integrationscode funktioniert. Das führt zu einer Beschränkung der Technologie-Freiheit. Eine Implementierung mit reinem JavaScript wäre diesbezüglich besser.

## Fazit

Frontend-Integration führt zu einer sehr losen Kopplung. In vielen Fällen reichen Links aus. Dann müssen die Systeme nur die URLs der verlinkten Seiten kennen. Wenn eine Webseite aus Fragmenten verschiedener Systeme zusammengesetzt werden soll, dann kann die dafür notwendige Integration auf dem Server erfolgen. Mit einem Cache kann ESI genutzt werden. Durch den Cache können HTML-Fragmente im Cache abgelegt werden. Das kommt der Performance und der Resilience zugute. Webserver können SSI implementieren. Wenn bereits ein Webserver im Einsatz ist, dann kann so die zusätzliche Infrastruktur eines Caches eingespart werden. Und schließlich kann eine Client-seitige Integration optionale Inhalte nachladen, wie beispielsweise den Überblick über die Postbox.

Hörenswert ist auch die Frontend-Integrations-Episode[10] des INNOQ-Podcasts.

## Experimente

- Starte das ESI-Beispiel. Siehe https://github.com/ewolff/SCS-ESI/blob/master/WIE-LAUFEN.md.
- Betrachte die Ausgabe des Varnish-Caches unter http://localhost:8080/ und vergleiche ihn mit der Ausgabe des Order-Microservices unter http://localhost:8090/. Wirf einen Blick auf den Quelltext der zurückgegebenen HTML-Seiten mit dem Browser. Wie kann man auf die HTML-Fragmente des Common-Microservices zugreifen?
- Probiere die Oberfläche aus. Stoppe die Microservices mit `docker-compose up --scale common=0` bzw. `docker-compose up --scale order=0`. Welche Teile der Microservice sind noch nutzbar?

[10] https://www.innoq.com/de/podcast/025-scs-frontend-integration/

# Konzept: Asynchrone Microservices

Microservices können Nachrichten austauschen. Asynchrone Kommunikation erlaubt eine lose Kopplung und eine gute Resilience.

## Definition

Asynchrone Microservices grenzen sich von synchronen Microservices ab. Das nächste Kapitel beschreibt synchrone Microservices detailliert. Der Begriff "synchrone Microservices" steht für Folgendes:

> Ein Microservice ist synchron, wenn er bei der Bearbeitung von Requests selbst einen Request an andere Microservices stellt und auf das Ergebnis wartet.

Asynchrone Microservices warten also nicht auf die Antworten anderer Systeme, wenn sie gerade selbst einen Request bearbeiten. Dazu gibt es zwei Möglichkeiten:

- Der Microservice kommuniziert während der Bearbeitung eines Requests gar nicht mit anderen Systemen. Dann wird der Microservice typischerweise zu einem anderen Zeitpunkt mit den anderen Systemen kommunizieren. Der Microservice kann beispielsweise Daten replizieren, die bei der Bearbeitung eines Requests genutzt werden. Beispielsweise können

Kundendaten repliziert werden, um dann bei der Bearbeitung einer Bestellung auf die lokal vorhandenen Kundendaten zuzugreifen.
- Der Microservice schickt einem anderen Microservice einen Request, wartet aber nicht auf eine Antwort. Ein Microservice für die Abwicklung einer Bestellung kann eine Nachricht an einen anderen Microservice schicken, der die Rechnung erstellt. Eine Antwort auf diese Nachricht ist nicht notwendig und muss daher auch nicht abgewartet werden.

## Warum asynchrone Microservices?

Asynchrone Microservices haben einige Vorteile:

- Beim Ausfall eines Kommunikationspartners wird die Nachricht später übertragen, wenn der Kommunikationspartner wieder verfügbar ist. So bietet asynchrone Kommunikation *Resilience*, also eine Absicherung gegen den Ausfall von Teilen des Systems.
- Die Übertragung und auch die Bearbeitung einer Nachricht können fast immer *garantiert* werden: Die Nachrichten werden langfristig gespeichert. Irgendwann werden sie bearbeitet. Dass sie bearbeitet werden, kann man beispielsweise absichern, indem die Empfänger die Nachricht quittieren (Acknowledge).
- Asynchrone Microservices können *Events* implementieren. Events bieten eine fachliche Entkopplung. Ein Event könnte beispielsweise "Bestellung eingegangen" sein. Jeder Microservice kann selbst entscheiden, wie er auf den Event reagiert. Beispielsweise kann ein Microservice eine Rechnung erstellen und ein anderer die Lieferung anstoßen. Wenn weitere Microservices beispielsweise für ein Bonusprogramm hinzukommen, müssen diese nur geeignet auf den bereits vorhandenen Event reagieren. So ist das System sehr leicht erweiterbar.

# Rezept: Messaging mit Kafka

Kafka ist ein Beispiel für eine Message-oriented Middleware (MOM). Eine MOM verschickt Nachrichten und stellt sicher, dass die Nachrichten beim Empfänger ankommen. MOMs sind asynchron. Sie implementieren also kein Request / Reply wie bei synchronen Kommunikationsprotokollen, sondern verschicken nur Nachrichten.

## Grundlegende Kafka-Konzepte

Kafka[11] unterscheidet sich von anderen MOMs vor allem dadurch, dass es die Nachrichten, die es überträgt, dauerhaft speichert, statt sie nach der Übertragung zu verwerfen.

Die wesentlichen Konzepte von Kafka sind:

- Es gibt drei *APIs*: die *Producer API* zum Senden von Daten, die *Consumer API* zum Empfangen von Daten und die *Streams API* zum Transformieren der Daten.
- Kafka organisiert Daten in *Records*. Sie enthalten den transportierten Wert als *Value*. Außerdem haben *Records* einen Schlüssel (*Key*) und einen Zeitstempel (*Timestamp*).
- *Topics* fassen Records zusammen. So können Events einer bestimmten Art in einem Topic verschickt werden.
- Topics sind in *Partitionen* unterteilt. Wenn ein Producer einen neuen Record erstellt, wird der Record an eine Partition des Topics angehängt. Die Aufteilung der Records auf die Partitionen erfolgt anhand des Keys des Records.
- Kafka speichert für jeden Consumer den *Offset* für jede Partition. Dieser Offset zeigt an, welchen Record in der Partition der Consumer zuletzt gelesen hat. Wenn ein Consumer einen Record bearbeitet hat, kann der Consumer einen neuen Offset committen. Für jeden Consumer muss nur der Offset in jeder Partition gespeichert werden, was relativ leichtgewichtig ist.

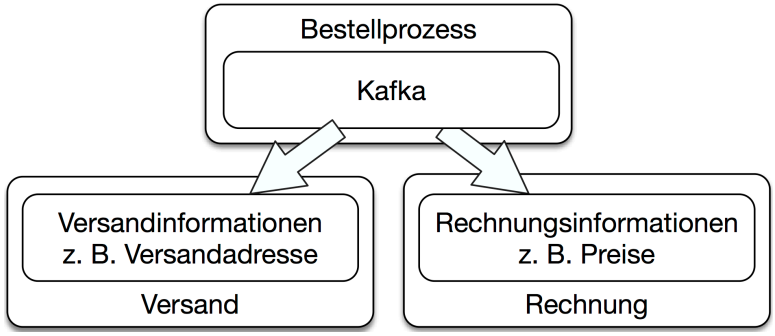
[11] https://kafka.apache.org/

- In einer *Consumer Group* gibt es für jede Partition genau einen Consumer. So kann sichergestellt werden, dass ein Record von einem Consumer bearbeitet wird: Der Record wird einer Partition zugeteilt, die dank der Consumer Group von einem Consumer bearbeitet wird.
- *Log Compaction* ist ein Mechanismus, mit dem alte Records gelöscht werden können: Wenn es mehrere Records mit derselben ID gibt, werden bei einer Log Compaction alle Records bis auf den letzten gelöscht.

Kafka kann als Cluster betrieben werden, um die Ausfallsicherheit und die Skalierung zu verbessern.

## Das Kafka-Beispiel

Das Beispiel findet sich unter https://github.com/ewolff/microservice-kafka. Unter https://github.com/ewolff/microservice-kafka/blob/master/WIE-LAUFEN.md steht eine umfangreiche Dokumentation bereit, die Schritt für Schritt die Installation und das Starten des Beispiels erläutert.




**Überblick über das Kafka-Beispiel**

Wenn die Docker-Container auf der lokalen Maschine laufen, steht unter http://localhost:8080/ eine Weboberfläche bereit. Die Oberfläche wird von einem Apache-httpd-Webserver angezeigt,

der als Reverse Proxy auch HTTP-Anfragen an die Microservices weitergibt.

### Aufteilung des Beispiels in Microservices

Das System besteht aus einem Microservice `order`, der eine Bestellung über die Weboberfläche entgegennimmt. Die Bestellung schickt der Bestellprozess dann als Record über Kafka an den Microservice für den Versand `shipping` und den Microservice für die Erstellung der Rechnung `invoicing`. Die Bestellung wird als JSON übertragen. So können der Rechnungs-Microservice und der Versand-Microservice aus der Datenstruktur jeweils die Daten auslesen, die für den jeweiligen Microservice relevant sind.

Alle Versand-Microservices und alle Rechnungs-Microservices sind jeweils in einer Consumer Group organisiert. Das bedeutet, dass die Records für die Bestellungen auf alle Consumer verteilt werden, aber jeder Record nur an genau einen Consumer geschickt wird. So kann sichergestellt werden, dass zu einer Bestellung nur ein Rechnungs-Microservice eine Rechnung schreibt und nur ein Versand-Microservice eine Lieferung veranlasst.

Der Versand-Microservice und der Rechnungs-Microservice speichern die Informationen aus den Records in ihren eigenen Datenbank-Schemata. Alle Microservices nutzen eine gemeinsame Postgres-Datenbank.

Jeder Kafka-Record enthält eine Bestellung. Der Key ist die ID der Bestellung mit dem Zusatz `created`, also beispielsweise `1created`.

### Avro: Ein alternatives Datenformat

Eine Alternative wäre Avro[12]. Das ist ein Datenformat, das ein binäres Protokoll anbietet, aber auch eine JSON-basierte Repräsentation. Avro hat ein Schema. Dabei ist es zum Beispiel mit Vorgabewerten auch möglich, Daten von einer alten Version des Schemas in

[12] http://avro.apache.org/

eine neue Version des Schemas zu konvertieren. Dadurch können alte Events selbst dann noch verarbeitet werden, wenn das Schema sich mittlerweile geändert hat.

## Alternatives Rezept: REST mit Atom

Asynchrone Microservices kann man auch mit REST umsetzen. So ist es zum Beispiel möglich, Bestellungen als Atom-Feed[13] anzubieten. Atom ist ein Datenformat, das ursprünglich entwickelt wurde, um Blogs für Leser verfügbar zu machen. So wie für jeden neuen Blog-Beitrag ein neuer Eintrag in einem Atom-Dokument erzeugt wird, ist dasselbe auch für jede neue Bestellung möglich. Ein Client muss dann regelmäßig das Atom-Dokument abholen und neue Einträge verarbeiten. Das ist nicht besonders effizient. Es kann aber durch HTTP-Caching optimiert werden. Dann werden nur Daten übertragen, wenn wirklich neue Einträge vorliegen. Eine Paginierung kann außerdem dafür sorgen, dass nur die neuesten Einträge übertragen werden und nicht etwa alle.

Ein Beispiel für eine asynchrone Integration von Microservices mit Atom findet sich unter https://github.com/ewolff/microservice-atom. https://github.com/ewolff/microservice-atom/blob/master/WIE-LAUFEN.md erläutert im Detail die einzelnen Schritte, um das Beispiel ablaufen zu lassen.

Atom hat den Vorteil, dass es auf REST und HTTP aufsetzt. Dadurch muss kein eigenes MOM betrieben werden. Meistens sind schon Erfahrungen mit HTTP und eine Umgebung mit Webservern vorhanden. So kann der Betrieb auch bei großen Datenmengen eher sichergestellt werden.

Leider kann diese Art der Kommunikation aber nicht dafür sorgen, dass eine Bestellung nur von einer Microservice-Instanz empfangen und bearbeitet wird. Wenn eine der Microservices-Instanzen im

[13] https://validator.w3.org/feed/docs/atom.html

Beispiel eine neue Bestellung aus dem Atom-Feed ausliest, dann überprüft sie zunächst in der Datenbank, ob es schon einen Eintrag für diese Bestellung gibt, und erzeugt nur dann selbst einen Eintrag, wenn das nicht der Fall ist. So wird für jede Bestellung nur ein Eintrag in der Datenbank erstellt.

Es ist übrigens nicht zwingend, das Atom-Format zu nutzen. Genauso gut kann man ein eigenes Format verwenden, das die Änderungen als Liste und dann Details als Links zur Verfügung stellt. Ebenso kann ein anderes Feed-Format wie RSS[14] oder JSON Feed[15] genutzt werden.

### Andere MOMs

Natürlich kann außer Kafka auch ein anderes MOM genutzt werden. So gibt es beispielsweise JMS-Implementierungen[16], die den Java-Messaging-Service-Standard ( JMS[17]) implementieren, oder Implementierungen[18] des AMQP[19] (Advanced Message Queuing Protocol). Dann müssen die Microservices allerdings damit umgehen, dass alte Events nach einiger Zeit nicht mehr zur Verfügung stehen.

## Fazit

Asynchrone Microservices bieten Vorteile bei der Resilience, aber auch bei der Entkopplung. Kafka ist eine interessante Alternative für asynchrone Microservices, weil die Historie der Events auch langfristig gespeichert wird. Außerdem kann dadurch auch eine Vielzahl von Clients unterstützt werden, ohne dass allzu viele Ressourcen verbraucht werden.

[14] http://web.resource.org/rss/1.0/spec
[15] http://jsonfeed.org/
[16] https://en.wikipedia.org/wiki/Java_Message_Service#Provider_implementations
[17] https://jcp.org/aboutJava/communityprocess/final/jsr914/index.html
[18] https://en.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol#Implementations
[19] https://www.amqp.org/

Ein HTTP- bzw. REST-basiertes System, das Änderungen als Atom-Feed oder in einem anderen Datenformat anbietet, hat gegenüber Kafka den Vorteil, dass es keinen zusätzlichen Server benötigt. Dafür ist es nicht ganz so einfach, eine Nachricht nur an einen Empfänger zu schicken, weil dafür das Protokoll keine eigene Unterstützung mitbringt.

## Experimente

- Starte das Kafka-Beispiel. Siehe dazu https://github.com/ewolff/microservice-kafka/blob/master/WIE-LAUFEN.md.
- Es ist möglich, mehrere Instanzen des Shipping- oder Invoicing-Microservices zu starten. Dazu kann `docker-compose up -d --scale shipping=2` oder `docker-compose up -d --scale invoicing=2` dienen. Mit `docker logs mskafka_invoicing_2` kann man die Logs betrachten. Dort gibt der Microservice dann auch aus, welche Kafka-Partitionen er bearbeitet.

# Konzept: Synchrone Microservices

Viele Microservice-Systeme setzen synchrone Kommunikation ein. Dieses Kapitel zeigt, wie synchrone Microservices mit verschiedenen Technologien umgesetzt werden können.

## Definition

Das letzte Kapitel hat synchrone Microservices bereits definiert:

> Ein Microservice ist synchron, wenn er bei der Bearbeitung von Requests selbst einen Request an andere Microservices stellt und auf das Ergebnis wartet.

Also kann ein synchroner Bestellungs-Microservice von einem anderen Microservice Kundendaten abholen, während er einen Request für eine Bestellung bearbeitet.

## Warum synchrone Microservices?

Zu den Gründen für den Einsatz synchroner Microservices zählen:

- Synchrone Microservices sind *einfach zu verstehen.* Statt eines lokalen Methoden-Aufrufs wird eine Funktionalität in

einem anderen Microservice aufgerufen. Das entspricht dem, was Programmierer gewohnt sind.
- Es kann eine bessere *Konsistenz* erreicht werden. Wenn bei jedem Aufruf die neuesten Informationen aus den anderen Services geholt werden, dann sind die Daten aktuell und entsprechen den Informationen der anderen Microservices, wenn nicht in letzter Sekunde noch eine Änderung eingetreten ist.

Dafür ist *Resilience* aufwändiger: Wenn der aufgerufene Microservice gerade nicht zur Verfügung steht, muss der Aufrufer mit dem Ausfall so umgehen, dass er nicht ebenfalls ausfällt. Dazu kann der Aufrufer Daten aus einem Cache nutzen oder auf einen vereinfachten Algorithmus zurückgreifen, der die Informationen aus dem anderen Microservice nicht benötigt.

## Herausforderungen

Für synchrone Kommunikation müssen einige Herausforderungen gelöst werden:

- Ein Microservice bietet seine Schnittstelle typischerweise per TCP/IP unter einer bestimmten IP-Adresse und einem bestimmten Port an. Der Aufrufer muss diese Informationen bekommen. Dazu dient *Service Discovery*.
- Von jedem Microservice können mehrere Instanzen laufen. *Load Balancing* muss die Aufrufe auf die Instanzen verteilen.
- Nach außen sollen alle Microservices als Teile eines Systems wahrgenommen werden und unter einer URL bereitstehen. *Routing* sorgt dafür, dass Aufrufe an den richtigen Microservice weitergeleitet werden.
- Wie erwähnt, stellt *Resilience* eine besondere Herausforderung dar, mit der ebenfalls umgegangen werden muss.

Eine Technologie für die Umsetzung synchroner Microservices muss für jede dieser Herausforderungen eine Lösung anbieten.

# Rezept: Kubernetes

Kubernetes[20] wird als Umgebung für die Entwicklung und den Betrieb von Microservices immer wichtiger.

## Docker

Kubernetes basiert auf Docker[21]. Docker erlaubt es, in einem Linux-System Prozesse stärker voneinander zu entkoppeln: *Docker-Container* bieten einem Betriebssystem-Prozess ein eigenes Dateisystem und ein eigenes Netzwerk-Interface mit einer eigenen IP-Adresse. Im Gegensatz zu einer virtuellen Maschine nutzen aber alle Docker-Container denselben Linux Kernel. So ist ein Docker-Container kaum aufwändiger als ein Linux-Prozess. Es ist ohne Weiteres möglich, Hunderte Docker-Container auf einem Laptop laufen zu lassen.

Die Dateisysteme der Docker-Container basieren auf *Docker-Images*. Die Images enthalten alle Dateien, die der Docker-Container benötigt. Dazu kann eine Linux-Distribution zählen oder auch eine Java-Laufzeitumgebung. Docker-Images haben Schichten. Die Linux-Distribution kann eine Schicht sein und die Java-Laufzeitumgebung eine weitere. Alle Java-Microservices können sich diese beiden Schichten teilen. Diese Schichten werden dann nur einmal auf dem Docker-Host gespeichert. Das reduziert den Speicherbedarf der Docker-Images erheblich.

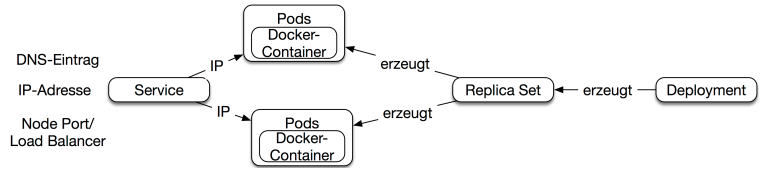
[20] https://kubernetes.io/
[21] https://www.docker.com/

## Kubernetes ist ein Docker Scheduler.

Docker-Container auf einem einzigen Docker-Host ablaufen zu lassen, ist nicht ausreichend. Wenn der Docker-Host ausfällt, dann fallen alle Docker-Container aus. Außerdem ist die Skalierbarkeit durch die Leistungsfähigkeit des Docker-Hosts begrenzt.

Um Docker-Container in einem Cluster von Rechnern laufen zu lassen, gibt es Scheduler wie *Kubernetes*. Kubernetes führt dazu einige Konzepte ein:

- *Nodes* sind die Server, auf denen Kubernetes läuft. Sie sind in einem Cluster organisiert.
- *Pods* sind mehrere Docker-Container, die zusammen einen Dienst erbringen. Das kann beispielsweise ein Container mit einem Microservice zusammen mit einer Container für die Log-Verarbeitung sein.
- Ein *Replica Set* sorgt dafür, dass immer von jedem Pod eine bestimmte Anzahl an Instanzen läuft.
- Ein *Deployment* erstellt ein Replica Set und stellt dafür die benötigten Docker-Images zur Verfügung.
- *Services* machen Pods zugreifbar. Die Services sind unter einem Namen im DNS zu finden und haben eine feste IP-Adresse, unter der sie im gesamten Cluster kontaktiert werden können. Außerdem ermöglicht der Service das Routing von Zugriffen von außen.




**Zusammenspiel der Kubernetes-Konzepte**

Die Grafik zeigt die Kubernetes-Konzepte im Überblick: Das Deployment erzeugt ein Replica Set. Das Replica Set erzeugt nicht

nur die Kubernetes-Pods, sondern startet auch neue, falls einige der Pods ausfallen. Die Pods umfassen einen oder mehrere Docker-Container.

Der Kubernetes-Service erstellt den DNS-Eintrag und macht den Microservice unter einer IP-Adresse verfügbar, die im gesamten Cluster eindeutig ist. Schließlich erstellt der Server einen *Node Port*. Unter diesem Port kann der Service auf allen Kubernetes-Nodes erreicht werden. Statt einem Node Port kann ein Service auch einen *Load Balancer* erstellen. Das ist ein Load Balancer, der von der Infrastruktur angeboten wird. Wenn Kubernetes in der Amazon Cloud läuft, würde Kubernetes einen Amazon Elastic Load Balancer erstellen.

### Synchrone Microservices mit Kubernetes

Die Herausforderungen synchroner Microservices löst Kubernetes folgendermaßen:

- Für *Service Discovery* nutzt Kubernetes DNS. Der Kubernetes-Service richtet den entsprechenden Eintrag ein. Andere Microservices können dann über den Hostnamen zugreifen.
- *Load Balancing* implementiert Kubernetes auf IP-Ebene. Der Kubernetes-Service hat eine IP-Adresse. Hinter den Kulissen wird der Verkehr zu der IP-Adresse auf eine der Service-Instanzen umgeleitet.
- Beim *Routing* kann der Kubernetes-Service entweder über den Node Port oder über einen Load Balancer erreicht werden. Das hängt davon ab, wie der Service konfiguriert ist und ob die Infrastruktur einen Load Balancer anbietet. Ein externer Nutzer kann entweder auf den Load Balancer oder auf den Node Port zugreifen und so den Microservice nutzen.
- Für *Resilience* hat Kubernetes keine Lösung parat. Natürlich kann Kubernetes-Pods neu starten, aber weitere Resilience Patterns wie Timeout oder Circuit Breaker implementiert Kubernetes nicht.

Die Lösung, die Kubernetes für die Herausforderungen synchroner Microservices bietet, führen zu keinen Code-Abhängigkeiten von Kubernetes. Wenn ein Microservice einen anderen aufruft, muss er den Namen aus dem DNS auslesen und kommuniziert mit der zurückgegebenen IP-Adresse. Das unterscheidet sich nicht von der Kommunikation mit einem beliebigen anderen Server. Beim Routing nutzt ein externes System einen Port auf einem Kubernetes-Host oder einen Load Balancer. Auch in diesem Fall ist transparent, dass hinter den Kulissen Kubernetes am Werk ist.

## Das Beispiel mit Kubernetes

Das Beispiel ist unter https://github.com/ewolff/microservice-kubernetes verfügbar. https://github.com/ewolff/microservice-kubernetes/blob/master/WIE-LAUFEN.md erläutert die Schritte detailliert, um die notwendige Software zu installieren und das Beispiel laufen zu lassen.

Das Beispiel besteht aus drei Microservices: Order, Customer und Catalog. Order nutzt Catalog und Customer mit der REST-Schnittstelle. Außerdem bietet jeder Microservice einige HTML-Seiten an.

Zusätzlich ist im Beispiel ein Apache-Webserver installiert, der dem Benutzer mit einer Webseite einen einfachen Einstieg in das System ermöglicht.

Ebenso steht ein Hystrix Dashboard als eigener Kubernetes-Pod zur Verfügung. Das Beispiel nutzt die Java-Library Hystrix[22], um Resilience zu erreichen. Diese Bibliothek führt Aufrufe in einem anderen Thread Pool aus und implementiert unter anderem einen Timeout für die Aufrufe.

Auf einem Laptop kann man das Beispiel mit Minikube[23] ausführen. Diese Kubernetes-Distribution ist sehr einfach installierbar. Sie bietet aber keinen Load Balancer, sodass die Services nur über einen Node Port bereitstehen.

[22] https://github.com/Netflix/Hystrix/
[23] https://github.com/kubernetes/minikube

Das Skript `docker-build.sh` erzeugt die Docker-Images für die Microservices und lädt sie in den öffentlichen Docker Hub hoch. Dieser Schritt ist optional, da die Images bereits dort zur Verfügung stehen.

Das Skript `kubernets-deploy.sh` deployt die Images aus dem öffentlichen Docker Hub. Dazu nutzt das Skript das Werkzeug `kubectl`. `kubectl run` dient dazu, das Image zu starten. Das Image wird heruntergeladen. Außerdem wird definiert, welchen Port der Docker-Container bereitstellen soll. `kubectl run` erzeugt das Deployment, welches das Replica Set und damit die Pods erzeugt. `kubectl expose` erzeugt den Service, der auf das Replica Set zugreift und so IP-Adresse, Node Port bzw. Load Balancer und DNS-Eintrag erstellt.

Dieser Ausschnitt aus `kubernetes-deploy.sh` zeigt die Nutzung der Werkzeuge am Beispiel des Catalog-Microservice:

```
#!/bin/sh
if [ -z "$DOCKER_ACCOUNT" ]; then
  DOCKER_ACCOUNT=ewolff
fi;
...
kubectl run catalog \\
 --image=docker.io/$DOCKER_ACCOUNT/microservice-kubernete\
s-demo-catalog:latest  \\
 --port=80
kubectl expose deployment/catalog --type="LoadBalancer" -\
-port 80
...
```

# Alternative Rezepte: Netflix, Consul, Cloud Foundry

Außer Kubernetes gibt es einige weitere Lösungen für synchrone Microservices:

- *Cloud Foundry* nutzt wie Kubernetes auch Docker. Allerdings ist Cloud Foundry ein PaaS (Platform as a Service). Es bietet Anwendungen eine vollständige Plattform. Daher ist es nicht notwendig, selbst Docker-Container zu erstellen. Es reicht aus, eine Java-Anwendung anzubieten.
  - *Service Discovery* setzt Cloud Foundry ebenfalls mit DNS um.
  - Die Plattform implementiert *Load Balancing* auf Netzwerk-Ebene.
  - Für das *Routing* von Zugriffen von außen reicht ebenfalls der DNS-Name des Microservices.
  - Wie Kubernetes auch, unterstützt Cloud Foundry *Resilience* nicht wirklich.

    Die Cloud-Foundry-Demo[24] implementiert ein Beispiel, das im Wesentlichen mit dem Kubernetes-Beispiel identisch ist. Es gibt eine ausführliche deutsche Anleitung zum Starten des Beispiels[25].
- *Consul* ist eigentlich eine Service-Discovery-Technologie. Sie lässt sich allerdings mit einigen anderen Technologien zu einer vollständigen Lösung für Microservices ausbauen.
  - Für *Service Discovery* bietet Consul ebenfalls eine DNS-Schnittstelle, aber auch eine eigene Schnittstelle, mit der Service-Discovery-Informationen eingetragen und ausgelesen werden können.

[24] https://github.com/ewolff/microservice-cloudfoundry
[25] https://github.com/ewolff/microservice-cloudfoundry/blob/master/WIE-LAUFEN.md

  - Für das *Routing* bietet Consul selbst keine Lösung. Aber Consul Template[26] kann mit einem Template eine Konfigurationsdatei für einen beliebigen Service mit Informationen über die Microservices ergänzen. So kann beispielsweise ein Webserver so konfiguriert werden, dass er die HTTP-Zugriffe von außen an die Microservices verteilt. Der Webserver liest lediglich die Konfigurationsdatei aus und muss keinerlei Schnittstelle zu Consul implementieren.
  - *Load Balancing* kann genauso wie Routing mit einem Webserver und Consul Template implementiert werden. Eine Alternative ist eine Java-Bibliothek wie Ribbon[27], die das Load Balancing im aufrufenden Microservice implementiert.
  - *Resilience* muss mit einer zusätzlichen Bibliothek implementiert werden.

    Das Consul-Beispiel[28] nutzt Spring Cloud, um die Microservices bei Consul zu registrieren, und Ribbon für das Load Balancing. Resilience deckt Hystrix ab. Ein Apache httpd setzt das Routing um. Consul Template konfiguriert den Apache httpd. Eine Alternative wäre Registrator[29], das Docker-Container in Consul registrieren kann. Zusammen mit einem Zugriff auf Consul über DNS könnte Consul genauso transparent genutzt werden wie Kubernetes oder Cloud Foundry. Das Consul-DNS-Beispiel[30] implementiert diesen Ansatz,
- Der *Netflix-Stack* stellt eine vollständige Lösung für synchrone Microservices dar.
  - Für *Service Discovery* steht Eureka bereit. Es bietet ein REST-Interface und ermöglicht auch einen Cache auf

[26] https://github.com/hashicorp/consul-template
[27] https://github.com/Netflix/ribbon/wiki
[28] https://github.com/ewolff/microservice-consul/
[29] https://github.com/gliderlabs/registrator
[30] https://github.com/ewolff/microservice-consul-dns/

dem Client, beispielsweise mit der Eureka-Java-Client-Bibliothek.
- *Load Balancing* setzt der Netflix-Stack mit Ribbon um. Das ist eine Java-Bibliothek, die aus den von Eureka übermittelten Service-Instanzen eine auswählt.
- Für das *Routing* steht Zuul bereit, ein in Java geschriebener Proxy. Zuul kann mit eigenen Filtern ergänzt werden, die in Java oder Groovy geschrieben sein können. Dadurch kann Zuul sehr flexibel erweitert werden.
* *Resilience* setzt der Netflix-Stack mit Hystrix um.

Das Netflix-Beispiel[31] nutzt Spring Cloud, um den Netflix-Stack in die Java-Anwendungen zu integrieren. Das Microservices-System implementiert dasselbe Szenario wie die anderen Beispiele für synchrone Microservices.

Die Beispiele Kubernetes und Cloud Foundry haben keine Code-Abhängigkeiten. So eine Lösung ist mit Consul ebenfalls implementierbar. Dadurch können in den Microservice-Systemen auch andere Technologien als Java genutzt werden. Das unterstützt die Technologie-Freiheit.

## Fazit

Kubernetes bietet eine sehr mächtige Lösung für die Implementierung von synchronen Microservices, die außerdem den Betrieb der Microservices abdeckt. PaaS wie Cloud Foundry bieten eine höhere Abstraktion, sodass der Nutzer sich nicht mit Docker auseinandersetzen muss. Aber sowohl Kubernetes als auch Cloud Foundry erzwingen eine andere Ablaufumgebung. Das ist bei Consul und Netflix nicht so: Beide Systeme können in Docker-Containern wie auch auf virtuellen Maschinen oder physischen Servern betrieben werden. Consul bietet dabei wesentlich mehr Features.

[31] https://github.com/ewolff/microservice

# Experimente

- Starte das Kubernetes-Beispiel wie unter https://github.com/ewolff/microservice-kubernetes/blob/master/WIE-LAUFEN.md beschrieben.
  - Öffne die Apache httpd Website mit `minikube service apache`.
  - Öffne das Kubernetes Dashboard mit `minikube dashboard`.
- Teste das Load Balancing im Beispiel:
  - `kubectl scale` ändert die Anzahl der Pods in einem Replica Set. `kubectl scale -h` zeigt an, welche Optionen es gibt. Skaliere beispielsweise das Replica Set `catalog`.
  - `kubectl get deployments` zeigt an, wie viele Pods im jeweiligen Deployment laufen.

# Wie weiter?

Die vorliegende Broschüre kann nur eine knappe Einführung in Microservices bieten.

Das Microservices Praxisbuch[32] enthält eine detaillierte Beschreibung der Beispiele aus dieser Broschüre und darüber hinaus eine Einführung in Microservices und eine Auswahl von Technologien für das Monitoring von Microservices. Das Buch gibt es auch als englische Übersetzung[33].

Motivation, Architektur und Konzepte für Microservices stehen im Mittelpunkt des Microservice-Buchs[34]. Dieses Buch gibt es auch als englische Übersetzung[35].

Schließlich bietet der Microservices-Überblick[36] oder die englische Übersetzung[37] einen kostenlosen Einblick in die grundlegende Motivation und Architektur von Microservices.

[32] http://microservices-praxisbuch.de/
[33] http://practical-microservices.com/
[34] http://microservices-buch.de/
[35] http://microservices-book.com/
[36] http://microservices-buch.de/ueberblick.html
[37] http://microservices-book.com/primer.html