CLOUD NATIVE COMPUTING

CLOUD-NATIVE

Unit:

Microservices

(6) Skalierung von Microservices

Architectural Safety + Scaling + Caching

PROF. DR.
NANE KRATZKE

creative commons

CLOUD NATIVE COMPUTING

Urheberrechtshinweise

Diese Folien werden zum Zwecke einer praktikablen und pragmatischen Nutzbarkeit im Rahmen der **CCo 1.0 Lizenz** bereitgestellt.

Sie dürfen die Inhalte also kopieren, verändern, verbreiten, mit eigenen Inhalten mixen, auch zu kommerziellen Zwecken, und ohne um weitere Erlaubnis bitten zu müssen.

Eine Nennung des Autors ist nicht erforderlich (aber natürlich gern gesehen, wenn problemlos möglich).

Diese Folien sind insb. für die Lehre an Hochschulen konzipiert und machen daher vom **§51 UrhG (Zitate)** Gebrauch.

Die CCo Lizenz überträgt sich nicht auf zitierte Quellen. Hier sind bei der Nutzung natürlich die Bedingungen der entsprechenden Quellen zu beachten.

Die Quellenangaben finden sich auf den entsprechenden Folien.

KAPITEL 12

Microservice Architekturen

12.1 Eigenschaften von Microservices

12.2 Integrationsmuster für Microservices

- Datenbank-basierte Integration
- gRPC, REST
- API Versioning
- Event-Driven Architectures

12.3 Architekturelle Sicherheit

- Circuit-Breaker, Bulkhead
- Idempotente API-Operationen

12.4 Skalierung von Microservices

- Load Balancing
- Messaging
- Scaling for Reads/Writes
- Command Query Responsibility Segregation (CQRS)
- Caching

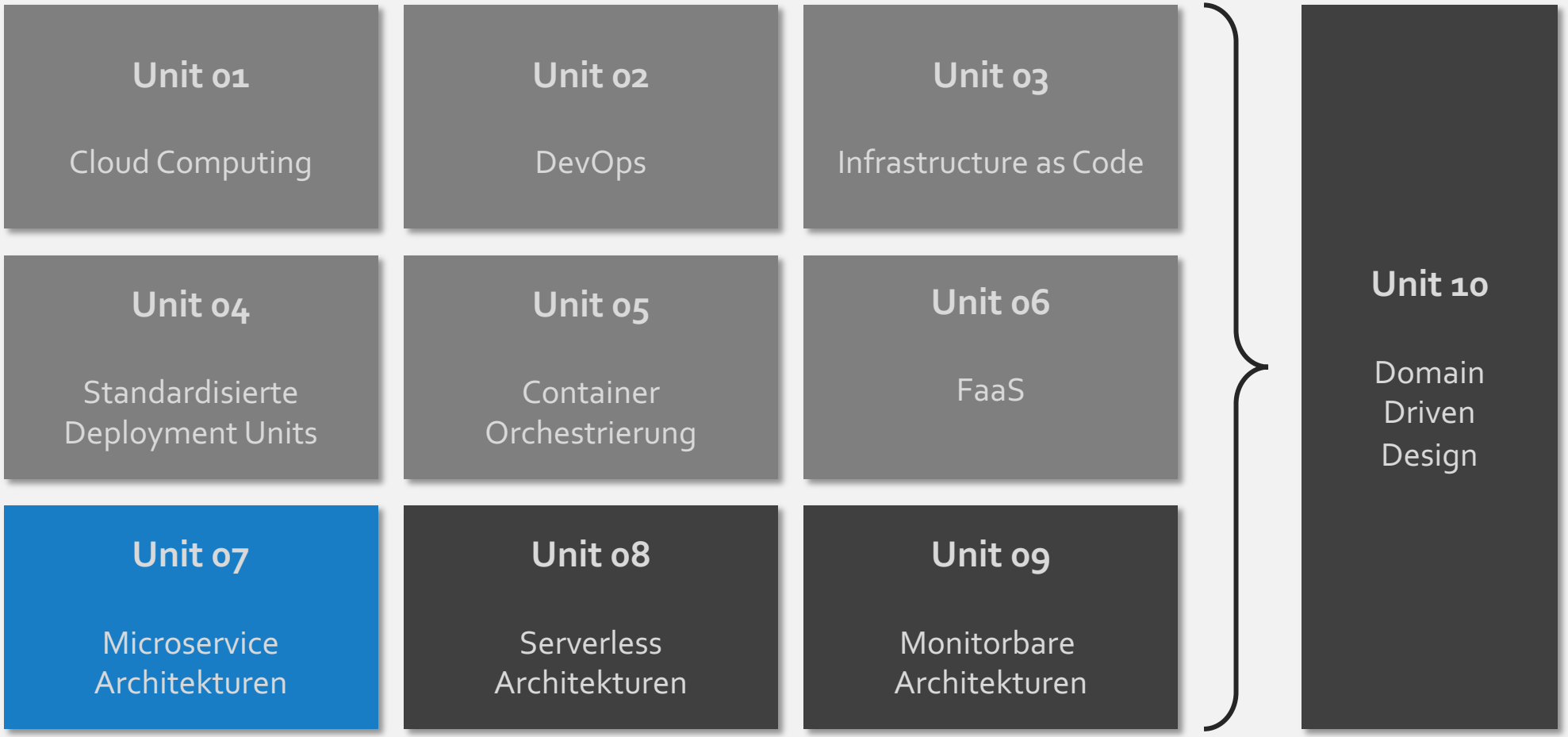
12.5 Prinzipien zur Entwicklung von Microservices

12.6 Serverless-Architekturen

- Architekturelle Konsequenzen von Serverless-Limitierungen
- Das API-Gateway Pattern
- Abgrenzung zu Microservices

INHALTSVERZEICHNIS

Überblick über Units und Themen dieses Moduls



INHALTE

CLOUD NATIVE COMPUTING

Microservices

- Was ist das für ein Architekturstil?
- Vor- und Nachteile von Microservices

Randbedingungen

- Lose Kopplung und hohe Kohäsion
- Bounded Context und Domain Driven Design
- Conways Law
- Service Ownership und Team-Strukturen

Integration

- Shared Databases, Sync vs. Async, RPC
- REST, Services as State Machines (HATEOS)
- Versioning

Skalierung von Microservices

- Architectural Safety Measures
- Scaling
- Caching

7 Prinzipien von Microservices

- Model Around Business Concepts
- Adopt a Culture of Automation
- Hide Internal Implementation Details
- Decentralize All the Things
- Independently Deployable
- Isolate Failure
- Highly Observable

ENTWURFSMUSTER FÜR MICROSERVICES

Architektonische Sicherheitsmaßnahmen

sind Maßnahmen, die üblicherweise in Gebäuden, Infrastrukturen oder Fahrzeugen implementiert werden, um die Sicherheit und den Schutz von Personen zu gewährleisten.

Es gibt allerdings auch analoge Maßnahmen für IT-Architekturen.

Skalierung

Unter Skalierung versteht man in der Regel die Anpassung der Größe eines Systems oder einer Struktur, z.B. durch Hinzufügen oder Entfernen von Hardware oder Software.

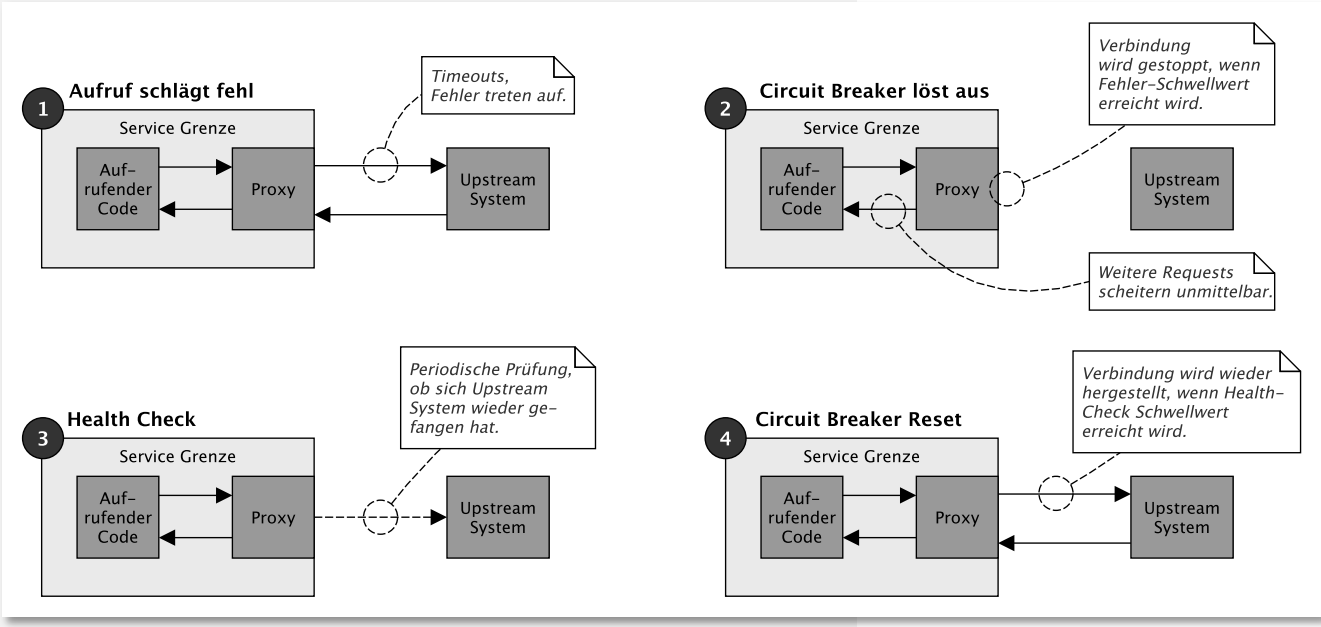
Caching

Unter Caching versteht man das Zwischenspeichern von Daten in einem schneller zugänglichen Speicher (z.B. RAM), um den Zugriff auf diese Daten zu beschleunigen. Dadurch werden Ladezeiten reduziert und die Leistung von Anwendungen verbessert.

ARCHITECTURAL SAFETY

Circuit Breakers

- Circuit Breaker sind ein Entwurfsmuster, um das **Risiko von Ausfällen** durch überlastete oder fehlerhafte Ressourcen zu reduzieren.
- Schutzmechanismus zwischen den Anwendungscode und externen Ressourcen wie APIs, Datenbanken oder anderen Netzwerkdiensten
- Der Circuit Breaker überwacht die **Anzahl der Fehler** und schaltet den Zugriff auf die Ressource ab, wenn die Fehlerquote einen Schwellenwert überschreitet (Open)
- Im Open Zustand werden keine Anfragen an die fehlerhafte Ressource gestellt
- Während die Ressource nicht verfügbar ist, prüft der Circuit Breaker **periodisch** ob die Ressource wieder verfügbar ist.
- Circuit Breaker **reduzieren Ausfallraten**, da sie Fehler im Zusammenhang mit Ressourcenüberlastung oder -ausfällen schnell und effektiv erkenn können.

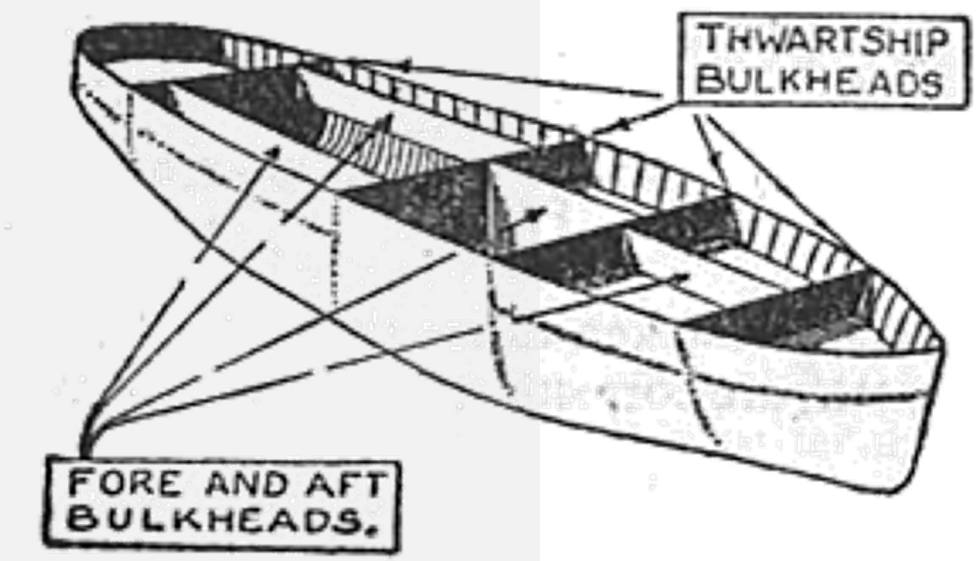


Quelle: Sam Newman, Microservices, O'Reilly 2014

ARCHITECTURAL SAFETY

Bulkheads (deutsch: Schottwand)

- Ein Bulkhead ist ein Konzept zur Aufteilung eines Systems in **separate Module**, die isoliert voneinander ausgeführt werden können.
- Sollte ein Teil des Systems ausfallen oder überlastet sein, sollte sich dies nicht auf andere Teile des Systems übertragen.
- Ähnlich wie bei einem Schiff, das durch **Schottwände** in verschiedene Bereiche unterteilt ist, um bei Leckagen den Wassereintritt zu begrenzen.
- In der Praxis lässt sich durch den Einsatz von **isolierten Prozessen**, **Containern** oder **virtuellen Maschinen** realisieren.
- Diese Technologien erlauben es, verschiedene Teile eines Systems voneinander zu trennen und unabhängig voneinander zu betreiben.
- Dadurch wird die Ausfallsicherheit und Skalierbarkeit des Systems erhöht.

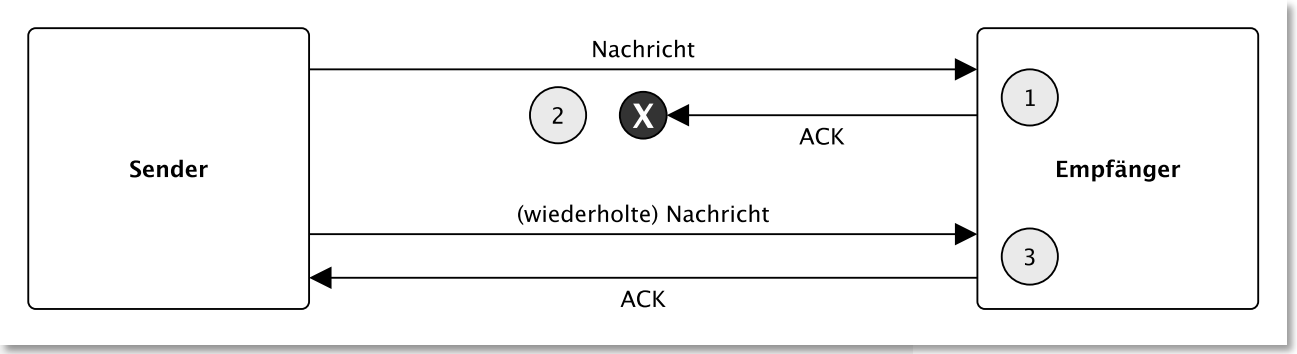


Quelle: Sam Newman, Microservices, O'Reilly 2014

ARCHITECTURAL SAFETY

Idempotenz

- Eine idempotente Operation ist eine Operation, bei der das Ergebnis unabhängig von der Anzahl der Anwendungen der Operation gleich bleibt.
- Zum Beispiel ist das Löschen einer Datei eine idempotente Operation, da das Ergebnis gleich bleibt, egal wie oft die Operation ausgeführt wird.



- Idempotente Operationen sind hilfreich bei Remote-APIs, da **wiederholte Anforderungen** das gleiche Ergebnis liefern, ohne unerwünschte Auswirkungen auf das System zu haben.
- Beim **REST-Paradigma** müssen bspw. die Methoden **GET, HEAD, PUT** und **DELETE** laut HTTP-Spezifikation idempotent sein.
- Idempotente Operationen haben bspw. nach einem **Restart eines Services** den Vorteil, das Resultate bereits erfolgreich verarbeiteter Operationen unverändert bleiben und Resultate nicht erfolgreicher Operationen wiederhergestellt werden.
- Diese Eigenschaft macht insb. **Recovery-Operationen** einfacher.

ENTWURFSMUSTER FÜR MICROSERVICES

Architektonische Sicherheitsmaßnahmen

sind Maßnahmen, die üblicherweise in Gebäuden, Infrastrukturen oder Fahrzeugen implementiert werden, um die Sicherheit und den Schutz von Personen zu gewährleisten.

Es gibt allerdings auch analoge Maßnahmen für IT-Architekturen.

Skalierung

Unter Skalierung versteht man in der Regel die Anpassung der Größe eines Systems oder einer Struktur, z.B. durch Hinzufügen oder Entfernen von Hardware oder Software.

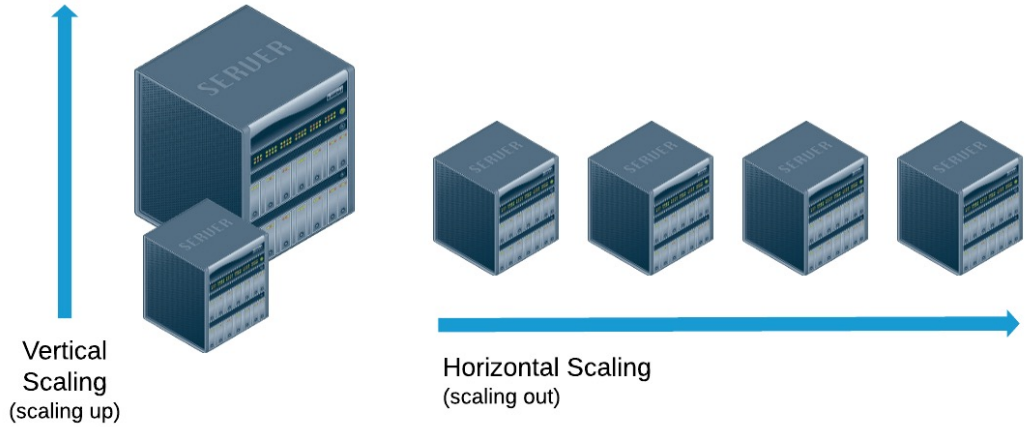
Caching

Unter Caching versteht man das Zwischenspeichern von Daten in einem schneller zugänglichen Speicher (z.B. RAM), um den Zugriff auf diese Daten zu beschleunigen. Dadurch werden Ladezeiten reduziert und die Leistung von Anwendungen verbessert.

SCALING

Vertical Scaling vs. Horizontal Scaling

- Wenn eine Anwendung zusätzliche Anforderungen nicht mehr effektiv verarbeiten kann, nennt man dies die **Grenze ihrer Skalierbarkeit**.
- Diese Grenze wird erreicht, wenn mind. eine kritische Hardwareressource aufgebraucht ist (CPU, Memory, Disk, Network, etc.)
- Diese Grenze kann grundsätzlich auf zwei Arten verschoben werden.
- Unter **Horizontaler Skalierung** versteht man das Hinzufügen weiterer Maschinen zu einem Ressourcenpool (*Scaling out*)
- Unter **Vertikale Skalierung** versteht man die Skalierung einer vorhandenen Maschine durch Hinzufügen von mehr Leistung (z. B. CPU, RAM) (*Scaling up*)



Einer der grundlegenden Unterschiede zwischen den beiden Skalierungsformen besteht darin, dass für die horizontale Skalierung das Programmlogik so angepasst werden muss, dass diese **auf mehreren Maschinen parallel** ausgeführt werden kann.

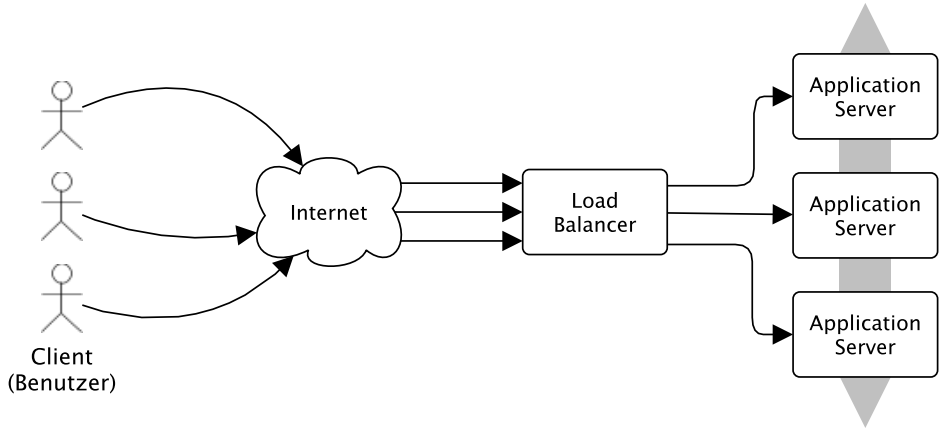
Die vertikale Skalierung ist einfacher, da sich die Logik nicht wirklich ändern muss. Man führt lediglich denselben Code auf Rechnern mit höherer Leistung aus. Vertikale Skalierung ist aber limitiert (insb. bei Big Data, Geographische Distribution, Vermeidung von Single Point of Failures, etc.).

Merke: Microservices setzen auf horizontale Skalierung

SCALING

Load Balancing

- Unter Load Balancing versteht man die effiziente **Verteilung** des eingehenden **Netzwerkverkehrs** auf eine Gruppe von Back-End-Servern.
- Load Balancer werden meist in **Request-Response** basierten Systemen eingesetzt
- Ein Load Balancer **verteilt** eingehende Requests an mehrere Server, um deren Geschwindigkeit und Auslastung zu optimieren.
- Fallen Server aus, leitet der Load Balancer den Datenverkehr an die verbleibenden Server um, und kann so **Ausfälle kompensieren**.
- Werden neue Server hinzugefügt, beginnt der Load Balancer automatisch, eingehende Requests auch an diese zu senden und **verteilt so den Traffic**.



Effekte von Load Balancern:

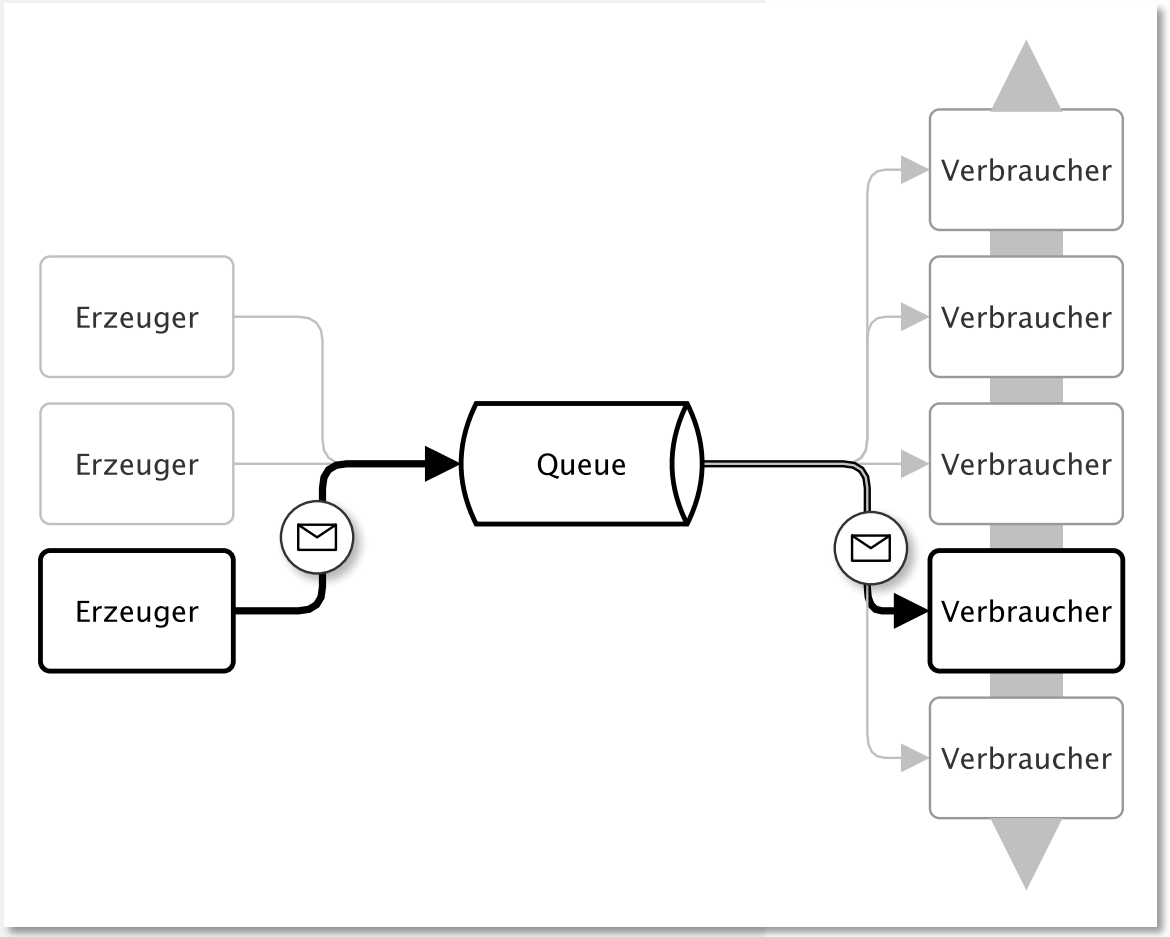
- Effiziente Verteilung von Requests (**Lastausgleich**)
- Optimierung der Gesamtverfügbarkeit und **Zuverlässigkeit**
- Ermöglichen einer **horizontalen Skalierung**

Quelle: NGINX, https://www.nginx.com/resources/glossary/load-balancing/

SCALING

Message Queuing

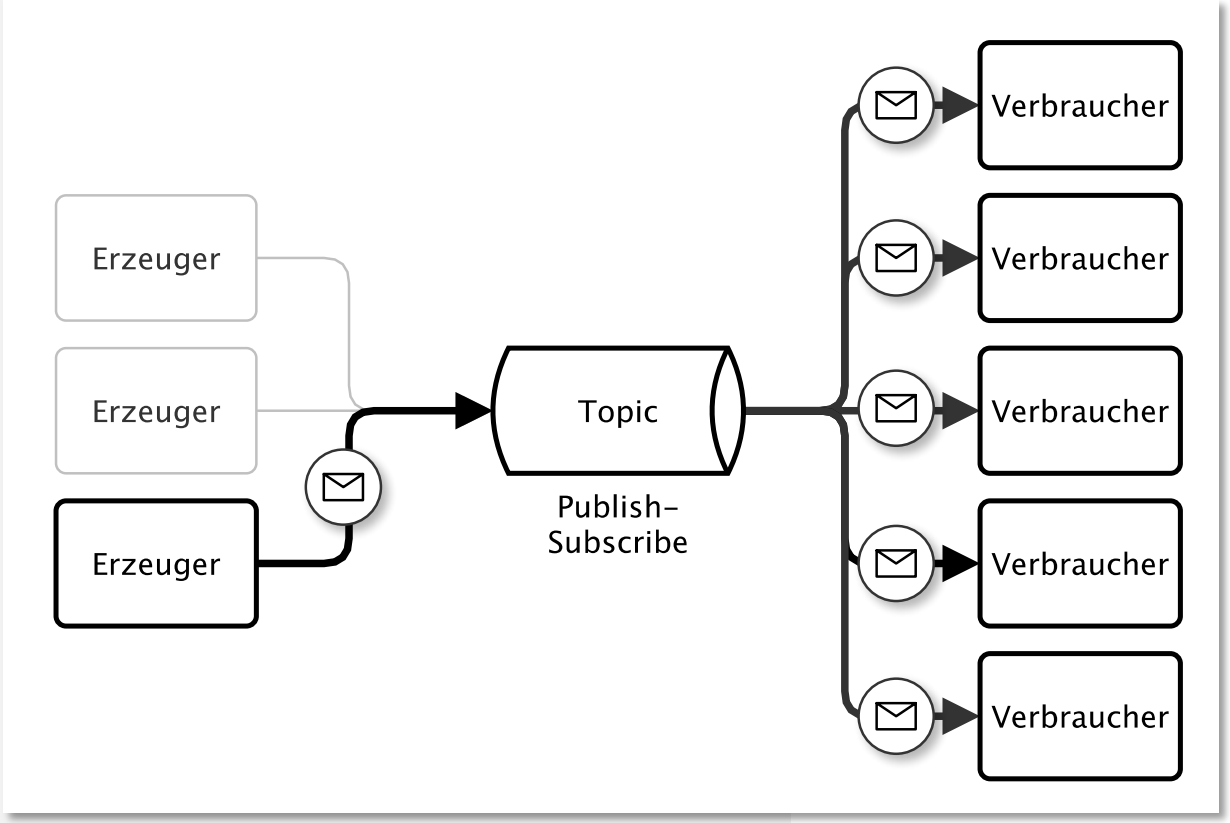
- In einer Nachrichtenwarteschlange werden Nachrichten von Erzeugern in der Warteschlange gegeben, um von einem Verbraucher bezogen und verarbeitet zu werden
- Jede Nachricht wird von einem einzelnen Verbraucher immer nur einmal verarbeitet. Dadurch sind Warteschlangen für die **Lastverteilung** geeignet
- Mithilfe von Nachrichtenwarteschlangen können umfangreiche Verarbeitungsprozesse entkoppelt, **Aufgaben gepuffert** und so hohe **Lastspitzen entschärft** werden
- Laufen Warteschlangen voll, können zusätzliche Ressourcen **zugeschaltet** werden, um die Warteschlangen wieder abzubauen
- Laufen Warteschlangen leer, können Ressourcen **abgeschaltet** werden, um Ressourcenverbrauch zu minimieren.
- Kenntnisbeziehungen zwischen Diensten nicht erforderlich, sondern nur zu einem zentralen Queueing-Service **(lose Kopplung)**



SCALING

Publish-Subscribe

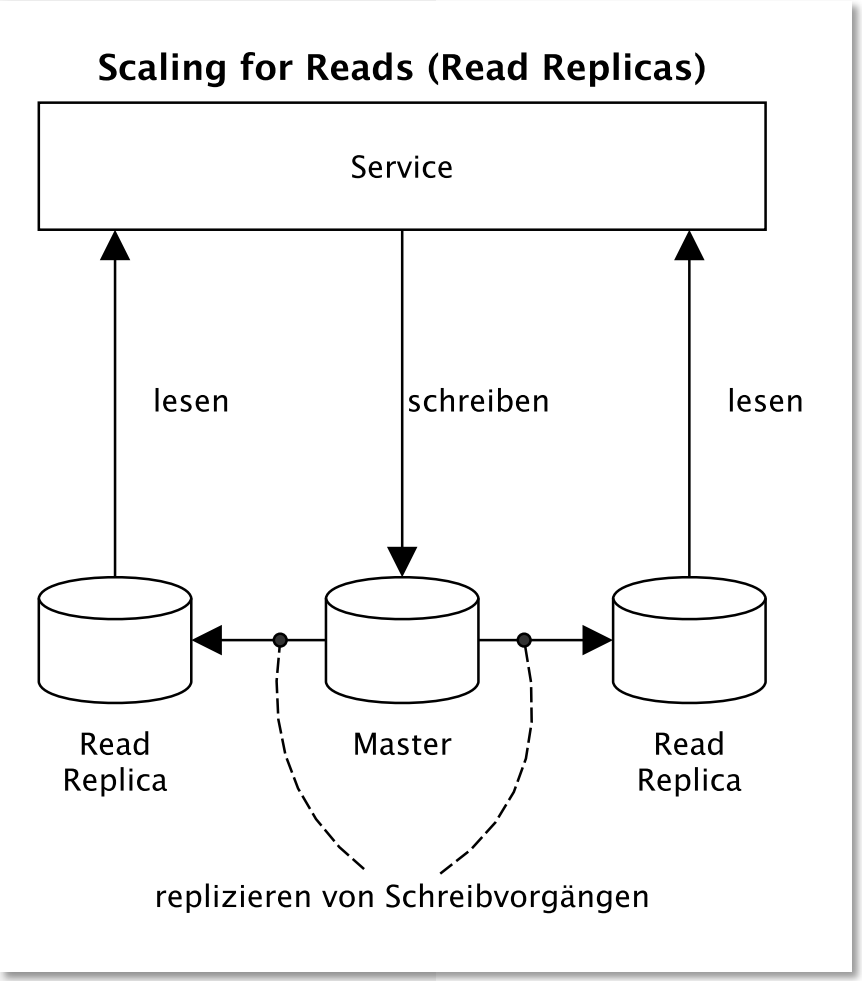
- Publish / Subscribe-Messaging (PubSub) ist eine Form der Service-to-Service **Broadcast**-Kommunikation.
- In einem Pub / Sub-Modell wird jede zu einem Topic veröffentlichte Nachricht sofort von allen **Abonnenten** des Topics empfangen.
- Auf diese Weise sind Kenntnisbeziehungen zwischen Diensten nicht erforderlich, sondern nur zu einem zentralen PubSub-Service **(lose Kopplung)**.
- Da alle Subscriber eines Topics dieselben Nachrichten verarbeiten, ist **kein Lastausgleich** über mehrere Service-Instanzen möglich.
- Das Pub/Sub-Modell eignet sich vielmehr, um Resultate von Publishern zu serialisieren und an zuständige Subscriber weiterzuleiten und diese **Liste** von Subscribern auch zu **erweitern**.



SCALING

Databases: Scaling for Reads

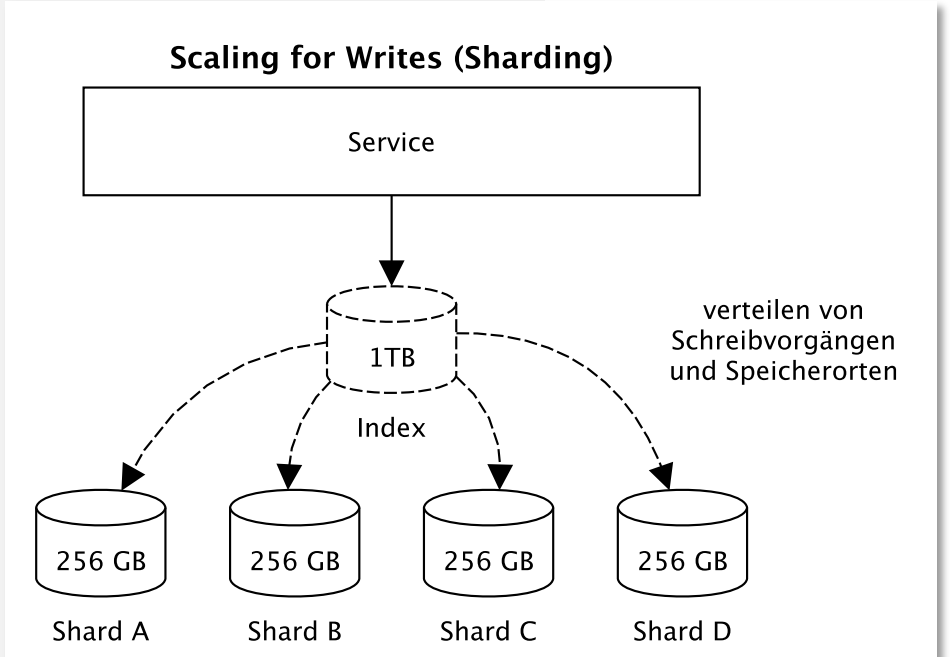
- Viele Dienste werden überwiegend gelesen.
- Um mehrere Endpunkte zum Lesen anbieten zu können, lassen sich in vielen Datenbanksystemen Daten daher von einem Primärknoten auf ein oder mehrere Replikate kopiert werden.
- Dies geschieht häufig, um sicherzustellen, dass eine Kopie der Daten sicher aufbewahrt wird. Man kann dies jedoch auch zum **Load Balancing von Lesevorgängen** verwenden.
- Die Replikation von der Primärdatenbank auf die Replikate erfolgt jedoch zeitversetzt nach dem Schreiben.
- Beim Lesen können manchmal **veraltete Daten** angezeigt werden, sofern eine Replikation noch nicht abgeschlossen wurde.
- Dieses Setting wird als eventuell konsistent bezeichnet **(Eventual Consistency)**.
- Wenn vorübergehende Inkonsistenzen in Anwendungsfällen folgenlos bleiben (und dies ist häufig der Fall), ist dies eine recht einfache und übliche Methode, um Stateful Services zu skalieren.



SCALING

Databases: Scaling for Writes (Sharding)

- Auch für die Schreibrichtung gibt es Skalierungsansätze.
- **Sharding** ist eine Methode der Datenbankpartitionierung für große Datensätze.
- Dabei wird ein Datenbestand in mehrere Teile **aufgeteilt** und jeweils von einer eigenen Serverinstanz verwaltet.
- Mittels Sharding können umfangreiche Datenmengen verwaltet werden, welche die Kapazitäten eines einzelnen Servers sprengen würden.
- Da die einzelnen Teile von einer eigenen Serverinstanz verwaltet werden, werden nicht nur die Daten selbst, sondern auch die dafür benötige Rechenleistung aufgeteilt.
- Die Methode der Aufteilung, die sog. **Partitionsstrategie**, spielt dabei eine entscheidende Rolle.



Der größte Nachteil des Shardings ist, dass ein Zugriff über andere Kriterien als das Aufteilungskriterium unverhältnismäßig aufwändig ist. Abfragen über andere Kriterien müssen auf mehrere Server aufgeteilt werden. Das Datenmodell muss so designed werden, dass derartige Zugriffe nur selten oder gar nicht vorkommen.

ENTWURFSMUSTER FÜR MICROSERVICES

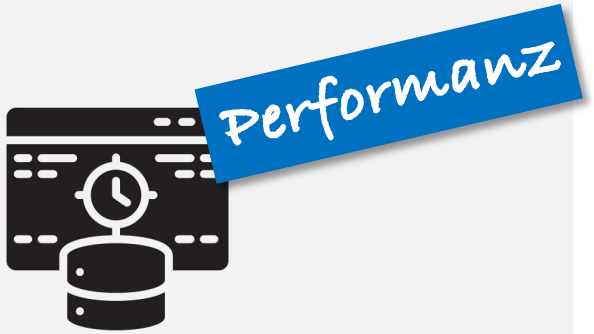
Architektonische Sicherheitsmaßnahmen

sind Maßnahmen, die üblicherweise in Gebäuden, Infrastrukturen oder Fahrzeugen implementiert werden, um die Sicherheit und den Schutz von Personen zu gewährleisten.

Es gibt allerdings auch analoge Maßnahmen für IT-Architekturen.

Skalierung

Unter Skalierung versteht man in der Regel die Anpassung der Größe eines Systems oder einer Struktur, z.B. durch Hinzufügen oder Entfernen von Hardware oder Software.



Caching

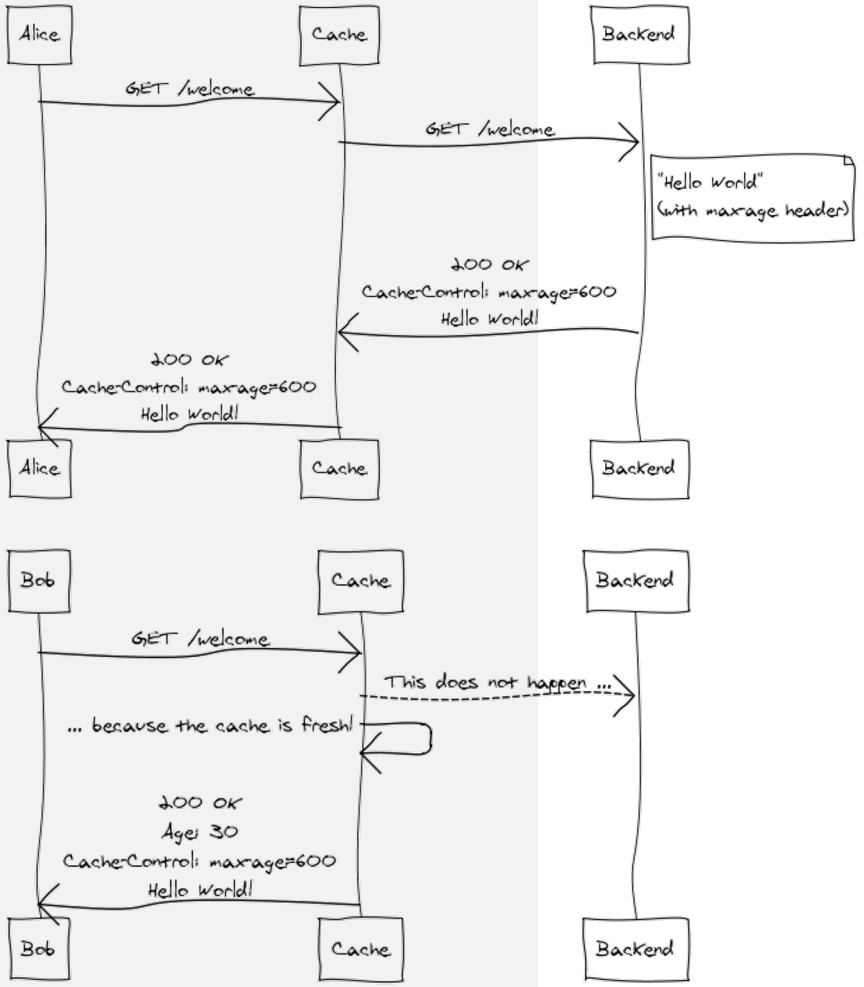
Unter Caching versteht man das Zwischenspeichern von Daten in einem schneller zugänglichen Speicher (z.B. RAM), um den Zugriff auf diese Daten zu beschleunigen. Dadurch werden Ladezeiten reduziert und die Leistung von Anwendungen verbessert.

CACHING

Performance Optimierung durch Vermeidung von unnötigen Wiederberechnungen

- Cache bezeichnet einen schnellen **Pufferspeicher**, der (wiederholte) Zugriffe auf ein langsames Hintergrundmedium oder aufwendige Neuberechnungen zu vermeiden hilft.
- Daten, die bereits einmal geladen oder generiert wurden, verbleiben im Cache, so dass sie bei späterem Bedarf schneller aus diesem abgerufen werden können.
- Auch können Daten, die vermutlich bald benötigt werden, vorab vom Hintergrundmedium abgerufen und vorerst im Cache bereitgestellt werden (**read-ahead**).
- Die Frage ist an welcher Stelle der Kommunikationsbeziehung man den Cache platziert (Client, Server, Proxy)?

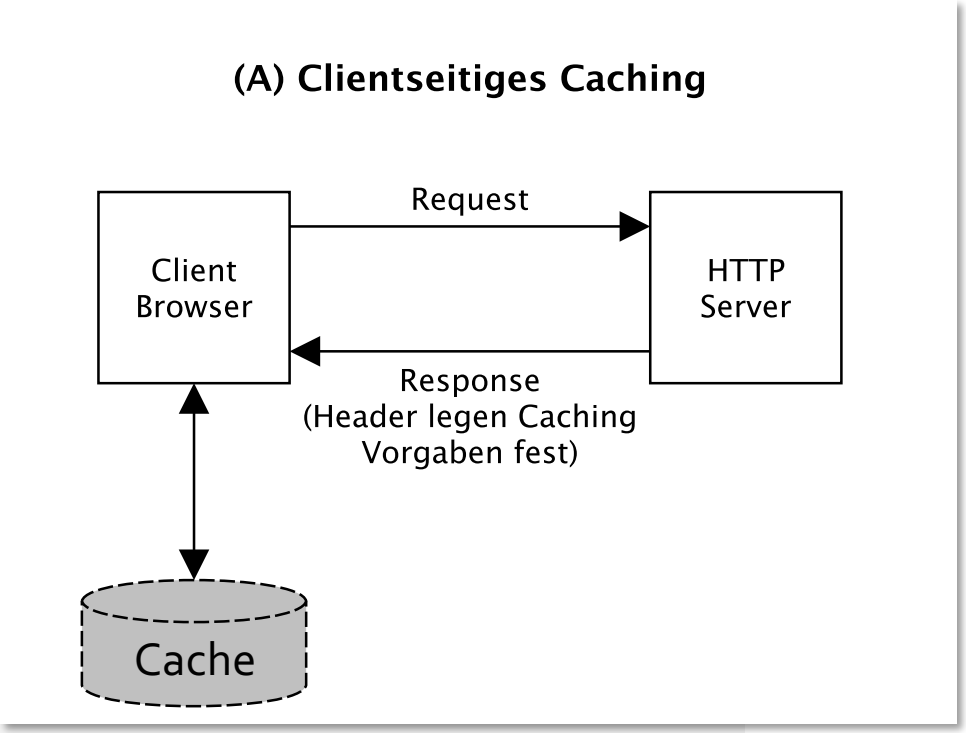
> *There are only two hard things in Computer Science: cache invalidation and naming things.*
>
> *-- Phil Karlton (Developer of the X Window System)*



CACHING

Möglichkeit 1: Client-side caching

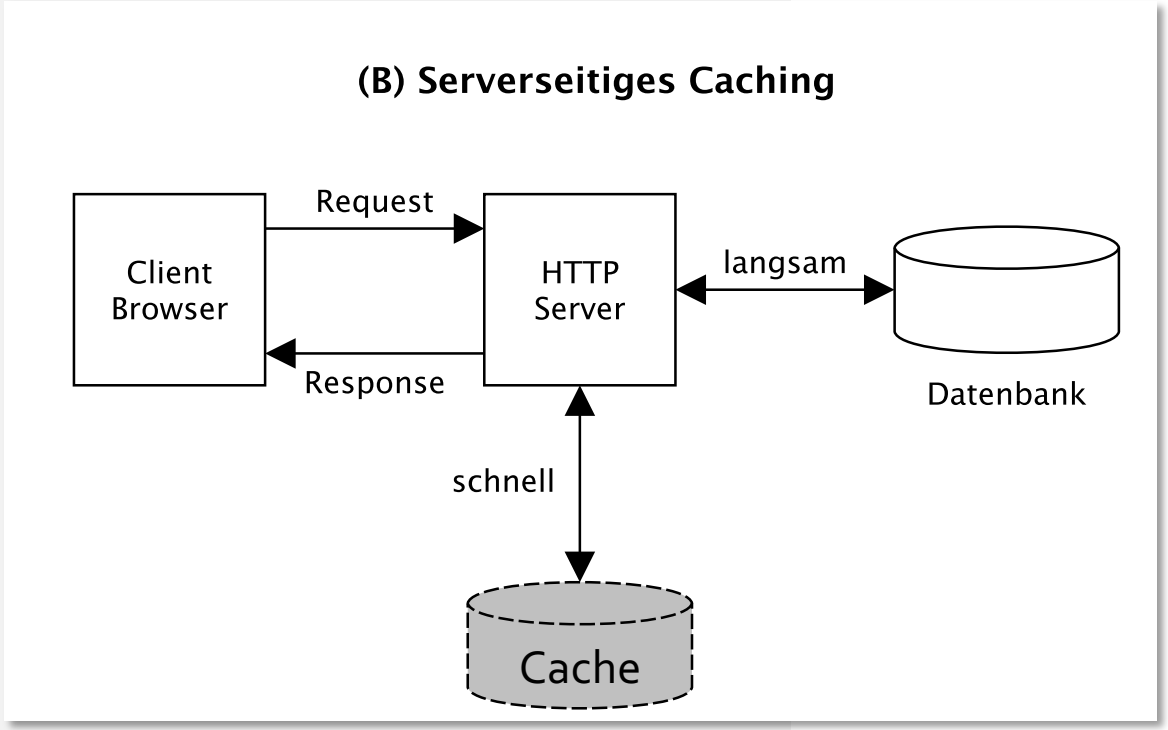
- Beim clientseitigen Caching speichert der Client das zwischengespeicherte Ergebnis.
- Der Client kann entscheiden, wann (und ob) er eine neue Kopie abruft.
- Im Idealfall bietet der Upstream-Service Hinweise, die dem Client helfen, zu verstehen, was mit der Antwort zu tun ist, wann und ob eine neuer Request gestellt werden muss.
- Das clientseitige Caching kann dazu beitragen, Netzwerkrequests drastisch zu reduzieren, und so wirkungsvoll die **Last auf Upstream-Services zu verringern**.



CACHING

Möglichkeit 2: Server-side caching

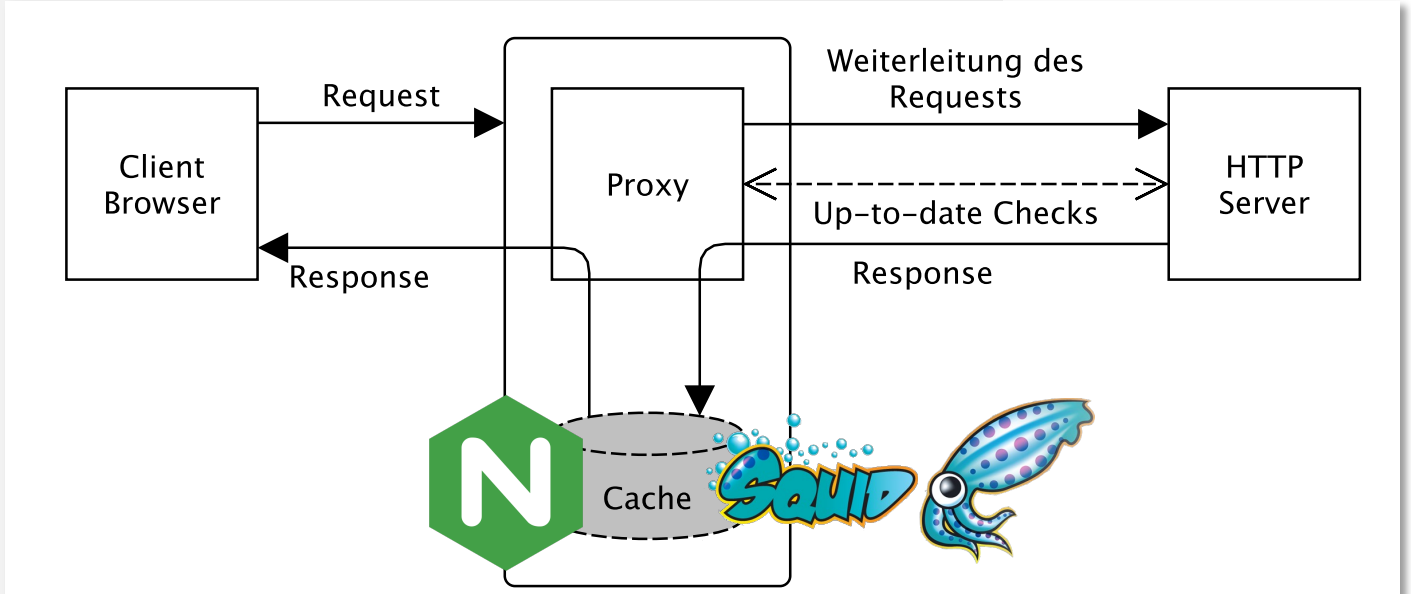
- Beim serverseitigen Caching übernimmt der Server die Caching-Verantwortung.
- Oft genutze Systeme sind Redis oder Memcache. Oft reichen sogar einfache In-Memory-Caches.
- Beim serverseitigen Caching ist für den Client das **Caching transparent**.
- Mit einem serverseitigen Cache innerhalb der Service-Grenze kann es einfacher sein, Dinge wie **Invalidierung** von Daten vorzunehmen oder Cache-Treffer zu verfolgen und zu optimieren.
- In einer Situation, in der mehrere Arten von Clients existieren, kann ein serverseitiger Cache der schnellste Weg sein, um die Leistung zu verbessern.



CACHING

Möglichkeit 3: Proxy caching

- Beim Proxy-Caching wird ein Proxy zwischen dem Client und dem Server platziert.
- Beim Proxy-Caching ist das Caching sowohl für Client als auch für den Server **transparent**.
- Proxy Caching ist oft eine sehr einfache Möglichkeit, einem vorhandenen System Caching hinzuzufügen.
- Wenn der Proxy so konzipiert ist, dass er generischen Datenverkehr zwischenspeichert, kann er auch mehr als einen Dienst bedienen (z.B. mittels Reverse-Proxies wie Squid oder NGINX).



Ein Proxy zwischen Client und Server führt erst einmal zu zusätzlichen Netzwerk-Hops und damit Latenzen. Meist führt dies jedoch selten zu Problemen, da die Leistungsoptimierungen, die sich aus dem Caching selbst ergeben, die zusätzlichen Netzwerklatenzen häufig überwiegen.

AUSBLICK

Microservices

- Was ist das für ein Architekturstil?
- Vor- und Nachteile von Microservices

Randbedingungen

- Lose Kopplung und hohe Kohäsion
- Bounded Context und Domain Driven Design
- Conways Law
- Service Ownership und Team-Strukturen

Integration

- Shared Databases, Sync vs. Async, RPC
- REST, Services as State Machines (HATEOS)
- Versioning

Skalierung von Microservices

- Architectural Safety Measures
- Scaling
- Caching

7 Prinzipien von Microservices

- Model Around Business Concepts
- Adopt a Culture of Automation
- Hide Internal Implementation Details
- Decentralize All the Things
- Independently Deployable
- Isolate Failure
- Highly Observable

KONTAKT

Disclaimer

Nane Kratzke

+49 451 300-5549

nane.kratzke@th-luebeck.de

kratzke.mylab.th-luebeck.de