CLOUD-NATIVE ARCHITEKTUREN

Unit 07:

Microservice Architekturen

Stand: 04.03.21

TECHNISCHE HOCHSCHULE LÜBECK

1

INHALTSVERZEICHNIS

Überblick über Units und Themen dieses Moduls

TECHNISCHE HOCHSCHULE LÜBECK

2

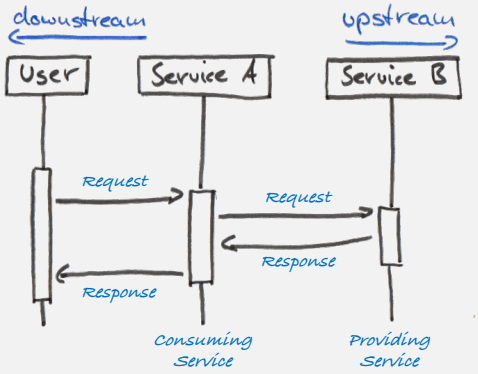
BEVOR WIR BEGINNEN

Klärung einer wesentlichen Begrifflichkeit

Cloud-native Systeme sind Service-of-Service Systeme und beruhen daher substantiell auf Service-Interaktionen. Bei der Bearbeitung eines Service Requests ist es daher nicht ungewöhnlich, dass ein Service weitere nachgelagerte Services abfragt. Service Requests wandern somit sinnbildlich erst einmal die interagierenden Services entlang „flussaufwärts" und die Antworten (Responses) fließen dann wieder „flussabwärts" zum Requestor (User).

Hierfür haben sich die Begrifflichkeiten Upstream und Downstream Services etabliert. Obwohl diese Terminologie oft genutzt wird, werden diese Begrifflichkeiten von Autor:in zu Autor:in unterschiedlich genutzt. Das hängt damit zusammen, was als Bezugspunkt gewählt wird. Der Requestor oder der einen Request beantwortende Service. Oft (leider nicht immer) kann man sich den Bezugspunkt aus dem Kontext erschließen.

Wir nutzen diese Begrifflichkeiten aus dem Blickwinkel des Requestors (User) aus.



Wir verwenden ferner die Rollen-Begrifflichkeiten **Consuming Service** und **Providing Service**. Consuming Services fragen Upstream-Services ab, die dann eine Providing Service Rolle haben.
Der Unterschied zwischen Consuming und Providing Service-Rolle bezieht sich jeweils nur auf zwei interagierende Services. Es kann durchaus sein, dass ein Consuming Service, für einen anderen Downstream Service die Rolle eines Providing Service einnimmt.

TECHNISCHE HOCHSCHULE LÜBECK

Merken:

Upstream Service
Downstream Service

Consuming Service
Providing Service

3

INHALTE

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

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

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

Skalierung von Microservices
- Failure is Everywhere
- 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

TECHNISCHE HOCHSCHULE LÜBECK

4

INHALTE

TECHNISCHE HOCHSCHULE LÜBECK

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

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

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

Skalierung von Microservices
- Failure is Everywhere
- 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

WAS SIND MICROSERVICES

TECHNISCHE HOCHSCHULE LÜBECK

*In short, the microservice architectural style is an approach to developing a single application as **a suite of small services**, each running in its **own process** and **communicating with lightweight mechanisms**, often an **HTTP resource API**. These services are built around business capabilities and **independently deployable** by fully **automated deployment machinery**. There is a bare **minimum of centralized management** of these services, which may be written in **different programming languages** and use **different data storage** technologies.*

Martin Fowler, 2014

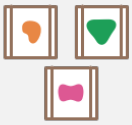
Quelle: https://martinfowler.com/articles/microservices.html

MONOLITHIC VS. MICROSERVICES

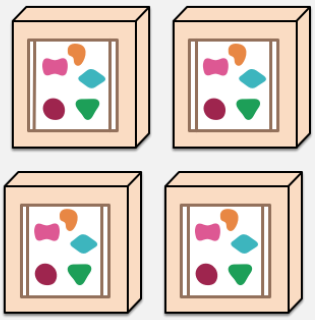
TECHNISCHE HOCHSCHULE LÜBECK

A monolithic application puts all its functionality into a single process...

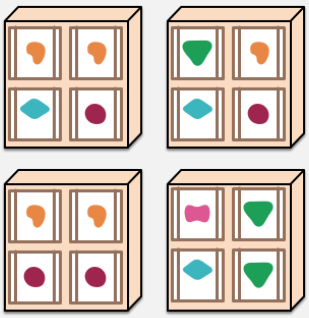
A microservices architecture puts each element of functionality into a separate service...



... and scales by replicating the monolith on multiple servers



... and scales by distributing these services across servers, replicating as needed.



Quelle: https://martinfowler.com/articles/microservices.html

DEKOMPOSITION MITTELS DIENSTEN

TECHNISCHE HOCHSCHULE LÜBECK

- *A component is a unit of software that is **independently replaceable** and **upgradeable**.*
- *Microservice architectures will use libraries, but their primary way of componentizing their own software is by **breaking down into services**.*
- ***Libraries** are components that **are linked into a program** and called using in-memory function calls*
- ***Services** are **out-of-process components** who communicate with a mechanism such as a web service request, or remote procedure call.*

Merke:

Komponenten:
unabhängig austauschbar und aktualisierbar.

Libraries:
In-Process Komponenten

Services:
Out-of-Process Komponenten

Quelle: https://martinfowler.com/articles/microservices.html

DEKOMPOSITION MITTELS DIENSTEN

Vor- und Nachteile

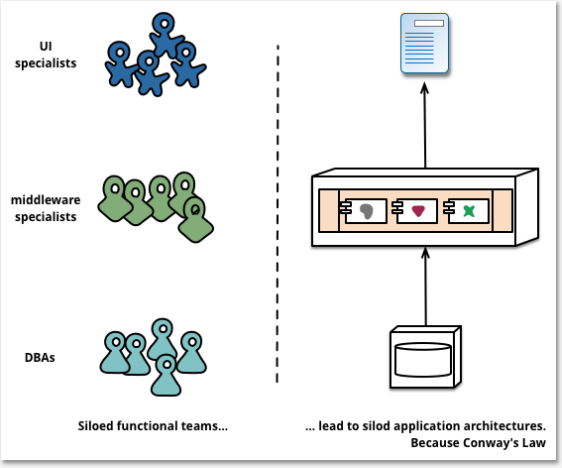
- *Another consequence of using services as components is a more explicit component interface.*
- *Most languages do not have a good mechanism for defining an explicit Published Interface.*
- *Often it's only documentation and discipline that prevents clients breaking a component's encapsulation, leading to overly-tight coupling between components.*
- *Services make it easier to avoid this by using explicit remote call mechanisms.*

- *Using services like this does have downsides.*
- *Remote calls are more expensive than in-process calls, and thus remote APIs need to be coarser-grained, which is often more awkward to use.*
- *If you need to change the allocation of responsibilities between components, such movements of behavior are harder to do when you're crossing process boundaries.*

Quelle: https://martinfowler.com/articles/microservices.html

DEKOMPOSITION MITTELS DIENSTEN

Organized around Business Capabilities (I)



Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.

Melvin Conway, 1968

Conways Law

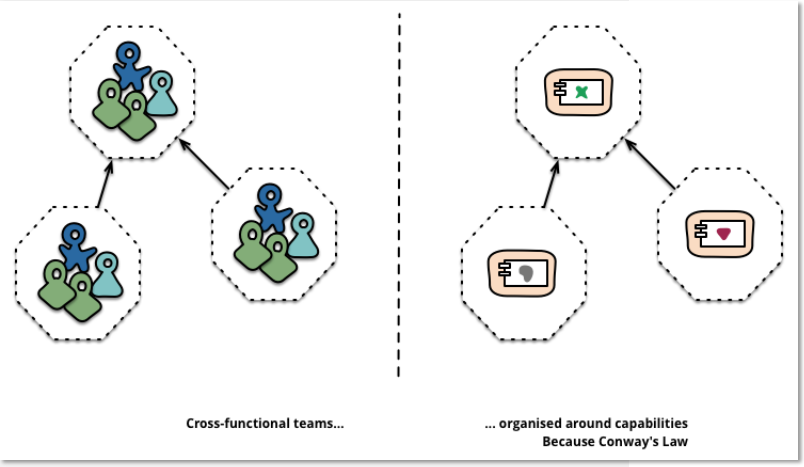
Quelle: https://martinfowler.com/articles/microservices.html

DEKOMPOSITION MITTELS DIENSTEN

Organized around Business Capabilities (II)

TECHNISCHE HOCHSCHULE LÜBECK

The microservice approach to division is different, splitting up into services organized around **business capability**. Such services take a broad-stack implementation of software for that business area, including user-interface, persistant storage, and any external collaborations. Consequently the teams are cross-functional, including the full range of skills required for the development: user-experience, database, and project management.



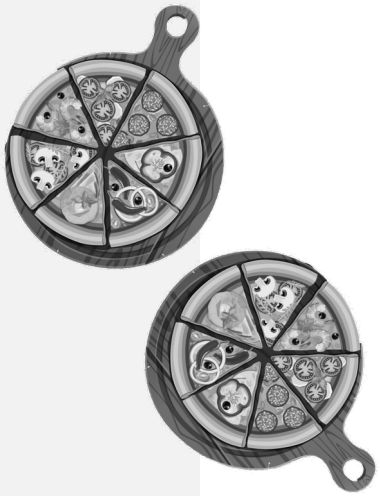
Quelle: https://martinfowler.com/articles/microservices.html

DEKOMPOSITION MITTELS DIENSTEN

How big is a microservice?

TECHNISCHE HOCHSCHULE LÜBECK

- Although "microservice" has become a popular name for this architectural style, its name does lead to an unfortunate focus on the size of service, and arguments about what constitutes "micro".
- In our conversations with microservice practitioners, we see a range of sizes of services. The largest sizes reported follow Amazon's notion of the Two Pizza Team (i.e. the whole team can be fed by two pizzas), meaning no more than a dozen people.
- On the smaller size scale we've seen setups where a team of half-a-dozen would support half-a-dozen services.
- This leads to the question of whether there are sufficiently large differences within this size range that the service-per-dozen-people and service-per-person sizes shouldn't be lumped under one microservices label.



Quelle: https://martinfowler.com/articles/microservices.html

DEKOMPOSITION MITTELS DIENSTEN

Products not Projects

- Most application development efforts use a project model: where the aim is to deliver some piece of software which is then considered to be completed. On completion the software is handed over to a maintenance organization and the project team that built it is disbanded.
- Microservice proponents prefer instead the notion that a team should own a product over its full lifetime. This brings developers into day-to-day contact with how their software behaves in production and increases contact with their users, as they have to take on at least some of the support burden.
- The product mentality, ties in with the linkage to business capabilities. Rather than looking at the software as a set of functionality to be completed, there is an on-going relationship where the question is how can software assist its users to enhance the business capability.

Quelle: https://martinfowler.com/articles/microservices.html

TECHNISCHE HOCHSCHULE LÜBECK

Software wird gerne in Projekten entwickelt (ähnlich einem Haus).

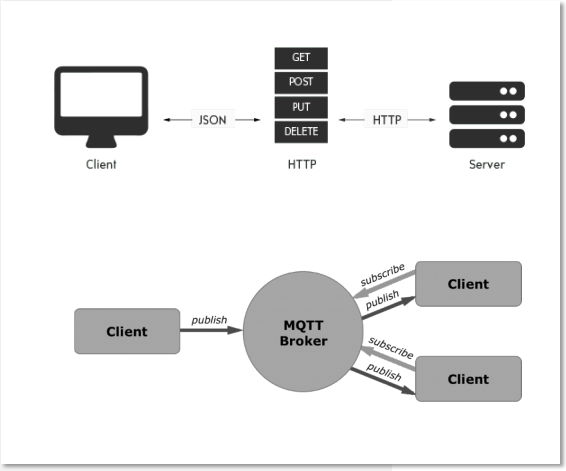
Verstehen Sie Services lieber als Produkte, die dauerhaft gewartet werden müssen (eher das Selbst-verständnis eines Kathedralen-Baumeisters).

DEKOMPOSITION MITTELS DIENSTEN

Smart endpoints and dumb pipes

TECHNISCHE HOCHSCHULE LÜBECK

- When building communication structures between different processes, we've seen many products and approaches that stress putting significant smarts into the communication mechanism itself. A good example of this is the Enterprise Service Bus (ESB), where ESB products often include sophisticated facilities for message routing, choreography, transformation, and applying business rules.
- Applications built from microservices aim to be as decoupled and as cohesive as possible - they own their own domain logic and act more as filters in the classical Unix sense - receiving a request, applying logic as appropriate and producing a response. These are choreographed using simple RESTish protocols rather than complex protocols such as WS-Choreography or BPEL or orchestration by a central tool.
- The two protocols used most commonly are HTTP request-response with resource API's and lightweight messaging.



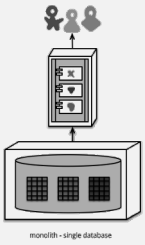
Quelle: https://martinfowler.com/articles/microservices.html

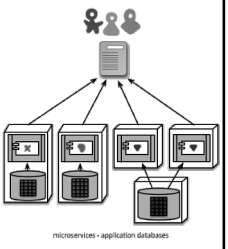
DEKOMPOSITION MITTELS DIENSTEN

Decentralized Data Storage

TECHNISCHE HOCHSCHULE LÜBECK

- A conceptual model of the world will differ between systems. This is a common issue when integrating across a large enterprise, the sales view of a customer will differ from the support view.
- This issue is common between applications, but can also occur within applications, particular when that application is divided into separate components.
- As well as decentralizing decisions about conceptual models, microservices also decentralize data storage decisions.
- While monolithic applications prefer a single logical database for persistant data, enterprises often prefer a single database across a range of applications.
- Microservices prefer letting each service manage its own database, either different instances of the same database technology, or entirely different database systems - an approach called Polyglot Persistence.
- This will come at a cost in complexity. Each data storage mechanism introduces a new interface to be learned. Furthermore data storage is usually a performance bottleneck, so you have to understand a lot about how the technology works to get decent speed. Using the right persistence technology will make this easier, but the challenge won't go away.



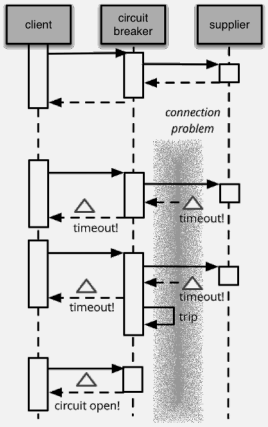


Quellen: https://martinfowler.com/articles/microservices.html, https://martinfowler.com/bliki/PolyglotPersistence.html

DEKOMPOSITION MITTELS DIENSTEN

Design for Failure

- A consequence of using services as components, is that applications need to be designed so that they can tolerate the failure of services.
- Any service call could fail due to unavailability of the supplier, the client has to respond to this as gracefully as possible.
- This is a disadvantage compared to a monolithic design as it introduces additional complexity to handle it.
- The consequence is that microservice teams constantly reflect on how service failures affect the user experience.
- Since services can fail at any time, it's important to be able to detect the failures quickly and, if possible, automatically restore service.
- Microservice applications put a lot of emphasis on real-time monitoring of the application, checking both architectural elements (how many requests per second is the database getting) and business relevant metrics (such as how many orders per minute are received).



Quellen: https://martinfowler.com/articles/microservices.html, https://martinfowler.com/bliki/CircuitBreaker.html

TECHNISCHE HOCHSCHULE LÜBECK

Die Grundidee hinter eines Circuit Breakers ist einen geschützten Funktionsaufruf in ein Proxy-Objekt zu verpacken, das auf Fehler überwacht. Sobald die Fehler einen bestimmten Schwellenwert erreichen, löst die „Sicherung" aus und alle weiteren Anrufe kehren mit einem Fehler zurück, ohne dass der Request an den eigentlichen Supplier geht (um diesen vor Überlast zu schützen).

DEKOMPOSITION MITTELS DIENSTEN

Evolutionary Design

- Microservice architcetures follow an evolutionary design philospohy.
- To break a software system into components is a decision of how to divide up the whole into components.
- The key property of a component is the notion of independent replacement and upgradeability.
- This implies to look for points where we can imagine rewriting a component without affecting its collaborators.
- This emphasis on replaceability is a special case of a more general principle of modular design, which is to drive modularity through the pattern of change.
- You want to keep things that change at the same time in the same module.
- Putting components into services adds an opportunity for more granular release planning.
- With a monolith any changes require a full build and deployment of the entire application.
- With microservices, you only need to redeploy the service(s) you modified.
- This can simplify and speed up the release process.
- The downside is that you have to worry about changes to one service breaking its consumers.
- The traditional integration approach is to try to deal with this problem using versioning, but the preference in the microservice world is to only use versioning as a last resort. We can avoid a lot of versioning by designing services to be as tolerant as possible to changes in their suppliers.

Quellen: https://martinfowler.com/articles/microservices.html

TECHNISCHE HOCHSCHULE LÜBECK

DEKOMPOSITION MITTELS DIENSTEN

Responsibilities of Evolutionary Architects

TECHNISCHE HOCHSCHULE LÜBECK

Quelle: Sam Newman, Building Microservices, O'Reilly 2015

INHALTE

TECHNISCHE HOCHSCHULE LÜBECK

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

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

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

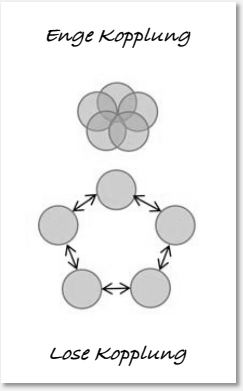
Skalierung von Microservices
- Failure is Everywhere
- Architectural Safety Measures
- Scaling
- Caching

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

WAS IST EIN „GUTER" (MICRO-)SERVICE?

Lose Kopplung

- Wenn Dienste lose gekoppelt sind, sollte eine Änderung an einem Dienst keine Änderung an einem anderen erfordern.
- Der springende Punkt des Microservice-Ansatzes besteht darin, Änderungen an einem Service vorzunehmen und ihn bereitzustellen zu können, ohne dass ein anderer Teil des Systems geändert werden muss.
- Welche Art von Dingen verursacht eine enge Kopplung? Ein klassischer Fehler besteht darin, einen Integrationsstil zu nutzen, der einen Dienst eng an einen anderen bindet und dazu führt, dass Änderungen innerhalb eines Services Änderungen für die aufrufenden Services erforderlich machen.
- Ein lose gekoppelter Service weiß daher idealerweise so wenig wie möglich über die Services, mit denen er zusammenarbeitet. Dies bedeutet auch, dass die Anzahl der verschiedenen Arten von Requests zwischen zwei Services begrenzt werden sollten, da eine „gesprächige Kommunikation" nicht nur zu potenziellen Leistungsproblemen sondern auch zu einer engen Kopplung führen kann.



TECHNISCHE HOCHSCHULE LÜBECK

*Die Begrifflichkeiten **lose Kopplung** und **hohe Kohäsion** werden insbesondere in der OOAD genutzt, um „gute" (also vor allem flexibel wiederverwandbare) Domnänenmodelle zu entwickeln.*

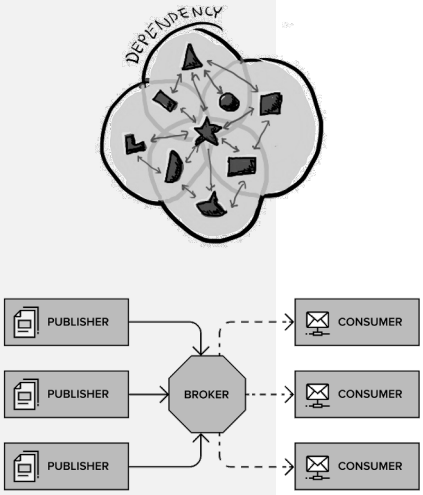
Doch diese Begriffe sind nicht dem OOAD vorbehalten und können auch für das Design von Services herangezogen werden.

Quelle: Sam Newman, Building Microservices, O'Reilly 2015

WAS IST EIN „GUTER" (MICRO-)SERVICE?

Methoden um Kopplungen zu reduzieren

- Die **lose Kopplung von Schnittstellen** kann durch Veröffentlichen von Daten in einem Standardformat (z. B. XML oder JSON) verbessert werden.
- Die **lose Kopplung zwischen Programmkomponenten** kann durch Verwendung von Standarddatentypen in Parametern verbessert werden. Für die Übergabe benutzerdefinierter Datentypen oder Objekte müssten nämlich ansonsten beide Komponenten über Kenntnisse der benutzerdefinierten Datendefinition verfügen.
- Die **lose Kopplung von Diensten** kann verbessert werden, indem nur Schlüsseldaten in der Service-to-Service Kommunikation verwendet werden (bspw. kann ein Dienst, der einen Brief sendet, am besten wiederverwendet werden, wenn nur die Kundenkennung übergeben wird und die Kundenadresse der aufgerufene Dienst selber ermittelt). So können Services entkoppelt werden, da Services nicht in einer bestimmten Reihenfolge aufgerufen werden müssen (z. B. GetCustomerAddress, SendLetter).



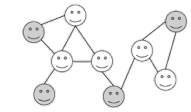
TECHNISCHE HOCHSCHULE LÜBECK

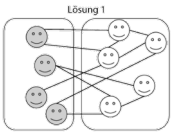
Quelle: Sam Newman, Building Microservices, O'Reilly 2015

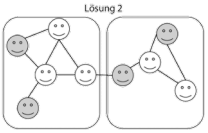
WAS IST EIN „GUTER" (MICRO-)SERVICE?

Hohe Kohäsion

- Insbesondere in der objektorientierten Programmierung beschreibt Kohäsion, wie gut eine Programmeinheit eine logische Aufgabe oder Einheit abbildet.
- In einem System mit starker Kohäsion ist jede Komponente (Service) idealerweise für nur genau eine wohldefinierte Aufgabe zuständig **(Single-Responsibility-Prinzip)**.
- Nur zusammengehörige Aufgaben sollten auch zusammen deployt werden müssen. Davon unabhängige Aufgaben sollten hingegen unabhängig deployt werden können. Um dies zu erreichen muss man allerdings zusammengehörige Aufgaben in Deployment Units voneinander isolieren.
- Warum? Wenn wir das Verhalten zusammengehöriger Aufgaben ändern möchten, möchten wir es an einem Ort ändern und diese Änderung so schnell wie möglich freigeben können.
- Wenn wir hierfür erforderliche Änderungen an vielen verschiedenen Orten in den Quelltext einbringen müssen, müssen wir viele verschiedene Dienste (möglicherweise gleichzeitig) freigeben, um diese Änderung deployen zu können.
- Das Vornehmen von Änderungen an vielen verschiedenen Orten ist langsamer und das gleichzeitige Bereitstellen vieler Dienste ist riskant. Das Bestreben von DevOps (siehe Unit 02) sind allerdings schnelle und unriskante Deployments.
- Architekturen mit einer hohen Kohäsion und looser Kopplung sind also eine Voraussetzung für agile DevOps-Ansätze (viele kleine, schnelle, unriskante Updates).





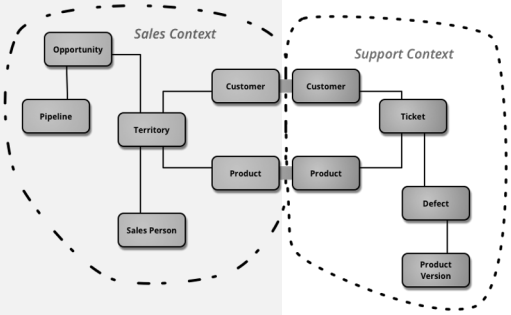


Quelle: Sam Newman, Building Microservices, O'Reilly 2015

WAS IST EIN „GUTER" (MICRO-)SERVICE?

Bounded Context

- Bounded Context ist ein zentraler Begriff aus dem Domain Driven Design (DDD, vgl. Unit 09).
- DDD fokussiert den Umgang mit großen Modellen und Teams. DDD ist eine Methodik um große Modelle in verschiedene begrenzte Kontexte (Bounded Contexts) zu unterteilen und ihre Wechselbeziehungen explizit angibt.
- Jeder Bounded Context kann dann allerdings für sich genommen betrachtet werden (um bspw. im Rahmen eines Microservice realisiert zu werden).
- Bounded Contexts haben sowohl nicht verwandte Konzepte (z. B. ein Support-Ticket, das nur in einem Kunden-Support-Kontext vorhanden ist) als auch gemeinsame Konzepte (z. B. Produkte und Kunden).
- Unterschiedliche Kontexte können (aufgrund unterschiedlicher Blickwinkel) völlig unterschiedliche Modelle gemeinsamer Konzepte aufweisen.
- Ein Verkäufer in der Retail-Abteilung eines Warenhauses hat vermutlich ein anderes „mentales Modell" des allgemeinen Konzepts „Kunde" als für eine Rechnungsprüferin in der Buchhaltung.
- Selbst innerhalb eines Domänenkontexts finden sich mehrere Kontexte. So ist Ihnen bspw. die Trennung zwischen speicherinternen und relationalen Datenbankmodellen in einer einzigen Anwendung durchaus bekannt.

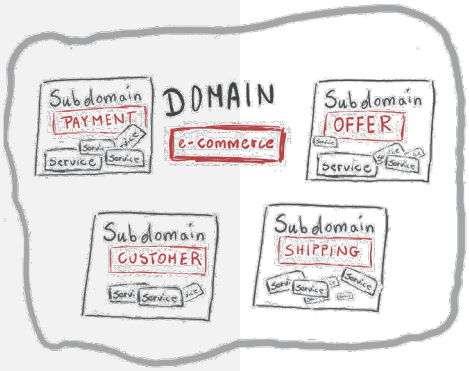


Quelle: Sam Newman, Building Microservices, O'Reilly 2015, `https://martinfowler.com/bliki/BoundedContext.html`

DOMAIN DRIVEN DESIGN (DDD)

Wie kommt man zu Bounded Contexten?

- Bei DDD geht es darum, Software basierend auf Modellen der zugrunde liegenden Domäne zu entwerfen.
- Ein Modell unterstützt die Kommunikation zwischen Softwareentwicklern und Domain-Experten und dient auch als konzeptionelle Grundlage für das Design der Software selbst.
- Um effektiv zu sein, muss ein Modell vereinheitlicht werden - das heißt, es muss intern konsistent sein, damit es keine Widersprüche enthält.
- Die Erfahrung zeigt allerdings, dass eine vollständige Vereinheitlichung eines Domänenmodells für ein großes System weder durchführbar noch kostengünstig ist.
- Daher gliedert DDD ein großes System in begrenzte Kontexte, von denen jeder ein einheitliches Modell haben kann (MultipleCanonicalModels).
- Wenn man nämlich versucht, eine größere Domäne zu modellieren, wird es zunehmend schwieriger, ein einzelnes einheitliches Modell zu erstellen.
- Verschiedene Personengruppen verwenden in verschiedenen Teilen einer großen Organisation subtil unterschiedliche Vokabeln. Die Präzision der Modellierung legt dies offen und führt so häufig zu Verwirrung. Typischerweise konzentriert sich diese Verwirrung auf die zentralen Konzepte der Domäne.



Mehr zu DDD in Unit 09

Quelle: Sam Newman, Building Microservices, O'Reilly 2015, https://martinfowler.com/bliki/BoundedContext.html

CONWAY'S LAW

Und System Design

„Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure."

— Melvin E. Conway, 1968 (How do committees invent)

„ If you have four groups working on a compiler, you'll get a 4-pass compiler."

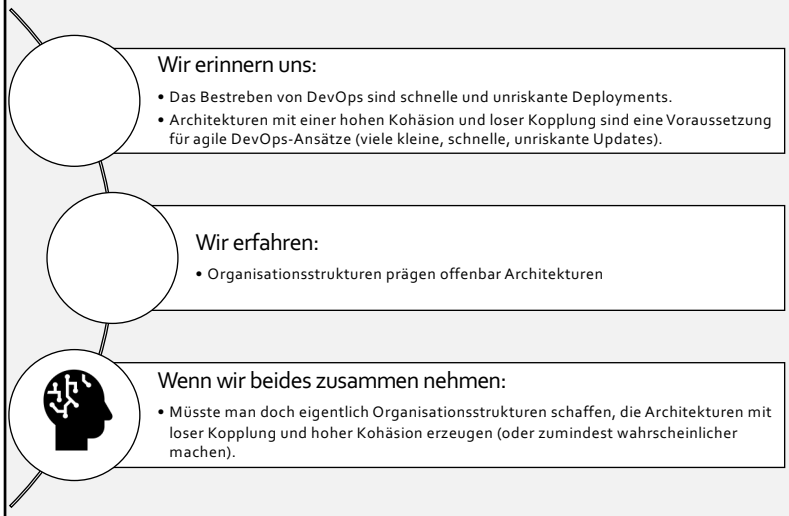
„ If a group of N persons implements a COBOL compiler, there will be N-1 passes. Someone in the group has to be the manager."

— Eric S. Raymond (Open Source Evangelist, 1996)

Quelle: Sam Newman, Building Microservices, O'Reilly 2015, https://martinfowler.com/bliki/BoundedContext.html

CONWAY'S LAW

Und System Design



Quelle: Sam Newman, Building Microservices, O'Reilly 2015

TECHNISCHE HOCHSCHULE LÜBECK

Kann das sein? Ist das nicht ein wenig extrem?

CONWAY'S LAW

DevOps + You build it, you run it => Service Ownership

- Ein Beispiel für die Auswirkungen des Conway-Gesetzes ist das Design einiger Websites von Organisationen. Nigel Bevan erklärte 1997 in einem Paper zu Usability-Problemen auf Websites: *"Unternehmen erstellen häufig Websites mit Inhalten und Strukturen, die eher die internen Anliegen der Organisation als die Bedürfnisse der Benutzer der Website widerspiegeln."*
- Beweise zur Unterstützung des Conway-Gesetzes wurden auch vom MIT und der Harvard Business School veröffentlicht. U.a. wurde festgestellt, das Software-Produkte von lose gekoppelten Organisationen (z.B. Open Source Communities) wesentlich modularer waren als Produkte von eng gekoppelten Organisation (z.B. straff geführten Unternehmen wie Microsoft).
- Interne Studien von Microsoft zu Windows Vista (hatte massive Qualitätsprobleme) führten zu ähnlichen Erkenntnissen.
- Unternehmen wie Netflix und Amazon haben tatsächlich ihre internen Strukturen an den gewünschten Qualitätseigenschaften von Microservices ausgerichtet.
  - You build it, you run it.
  - Two pizza principle: Kein Team soll größer sein, als das es nicht von zwei Pizzas satt würde!

Quelle: Sam Newman, Building Microservices, O'Reilly 2015

TECHNISCHE HOCHSCHULE LÜBECK

Prüfen Sie diese These mal an der Website der THL ;-)

Amerikanische Pizzen natürlich ;-)

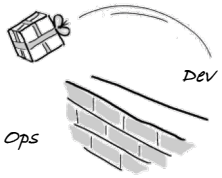
CONWAY'S LAW

Service Ownership

- Im Allgemeinen bedeutet Service Ownership, dass das Team, das einen Service initial entwickelt hat auch für Änderungen an diesem Service und dessen Betrieb verantwortlich ist. Das Team sollte sich frei fühlen, den Code nach Belieben umzustrukturieren, solange diese Änderung die konsumierenden Services nicht beeinträchtigt.
- Für viele Teams erstreckt sich Service Ownership auf alle Aspekte des Service. D.h. von der Anforderungserhebung über das Erstellen, Bereitstellen und Warten der Anwendung.
- Dieses Ownership-Modell führt zu einer erhöhten Autonomie von Teams und Liefergeschwindigkeit. Wenn ein Team für die Bereitstellung und Wartung der Anwendung verantwortlich ist, besteht ein Anreiz für die Erstellung von Services, die einfach bereitzustellen sind.
- Es ist also für ein Team nicht möglich "etwas über die Mauer zu werfen" und dann zu verschwinden. Es gibt ja niemanden, dem man es zuwerfen kann! Bzw. man wirft es sich selber zu.
- Dieses Modell überträgt die Entscheidungen an den Personenkreis, der am besten in der Lage ist, technische Entscheidungen fundiert zu treffen. Dadurch erhält das Team mehr Macht und Autonomie, ist aber auch für seine Arbeit verantwortlich.

Quelle: Sam Newman, Building Microservices, O'Reilly 2015

TECHNISCHE HOCHSCHULE LÜBECK

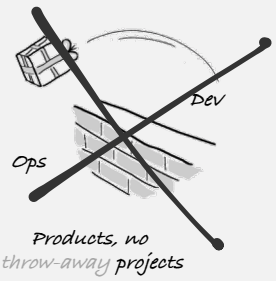


Service Ownership vermeidet den „throw-over-wall" Effekt oder die „It works on my machine"-Ausrede.

CONWAY'S LAW

Merkregeln für das Microservice Team-Building

Two Pizza Teams



Products, no throw-away projects

TECHNISCHE HOCHSCHULE LÜBECK

Wenn Conways Law gilt, fördern Sie mit diesen Regeln gute Microservice Architekturen.

INHALTE

TECHNISCHE HOCHSCHULE LÜBECK

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

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

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

Skalierung von Microservices
- Failure is Everywhere
- 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

INTEGRATION

Richtlinien für gute integrierbare und in einem DevOps-Kontext betreibbare Microservices (I)

TECHNISCHE HOCHSCHULE LÜBECK

Vermeidung von Breaking Changes:

- Ab und an sind Änderungen in einem Service erforderlich, die auch Änderungen in Consuming Services erforderlich machen.
- Wir sollten also grundsätzlich Technologien auswählen, die sicherstellen, dass dies so selten wie möglich geschieht.
- Wenn ein Microservice beispielsweise einem gesendeten Datenelement neue Felder hinzufügt, sollten bestehende Verbraucher nicht betroffen sein (dies schließt bspw. Ansätze die von Client- wie Serverseitig identisch generierten Stubs abhängig sind (z.B. Java RMI), häufig aus.

Technologie-agnostische APIs:

- Die einzige Gewissheit in der IT ist der Wandel. Es kommen ständig neue Tools, Frameworks und Sprachen heraus.
- Dies gilt insbesondere für die Cloud-native Domäne, die noch einen Konsolidierungsprozess in einer Vielzahl von Bereichen durchlaufen muss (z.B. einer Standardisierung von FaaS).
- Es ist also nicht unwahrscheinlich, dass ein Microservice in naher Zukunft im Kontext eines alternativen Technologie-Stacks funktionieren muss.
- Für die Kommunikation zwischen Microservices bedeutet dies, dass APIs möglichst Technologie-unabhängig oder auf Basis von in der Vergangenheit bewährten Protokollen wie bspw. HTTP gestaltet werden sollten.
- Integrationstechnologien sollten vermieden werden, die bestimmen, mit welchem Technologie-Stack Microservices implementiert werden müssen (z.B. Legacy Enterprise Service-Busses).

Es gibt eine ganze Reihe von technischen Optionen, wie Microservices untereinander kommunizieren können: SOAP, XML-RPC, REST, Protokollpuffer (gRPC), uvm.

Lassen Sie uns daher erst einmal darüber nachdenken, was wir von einer Technologie erwarten sollten, die wir im Kontext von Microservices und DevOps auswählen.

INTEGRATION

Richtlinien für gute integrierbare und in einem DevOps-Kontext betreibbare Microservices (II)

TECHNISCHE HOCHSCHULE LÜBECK

Einfache Integration für Consuming Services:

- Wir möchten es Upstream-Services einfach machen, einen Downstream-Services zu nutzen.
- Ein noch so „schöner" Microservice bringt nicht viel, wenn die Integrations-Kosten (z.B. Einarbeitung, Anbindung, innere Komplexität) für Upstream-Services zu hoch sind.
- Was macht es Consuming Services nun leichter andere Downstream Services zu nutzen?
- Im Idealfall sollten Clients Freiheit bei der Auswahl ihrer Integrations-Technologie haben. Andererseits kann auch eine Client-Library für den Microservice die Akzeptanz durch eine schneller Lernkurve und Entwicklungsgeschwindigkeit bei der Integration in einem Upstream-Service erhöhen.
- Man sollte jedoch bedenken, dass solche Client-Libraries möglicherweise nicht immer mit anderen Zielen deckungsgleich sind, die man ebenfalls erreichen möchte.
- Komfortable Client-Bibliotheken können zwar die Integration leicht machen, jedoch führen diese meist auch zu einer höheren Kopplung (z.B. ist der Microservice dann nicht mehr so ohne weiteres durch andere Microservices ersetzbar, wenn diese nicht mit derselben Client-Library integriert werden können).

Kapselung von Implementierungsdetails:

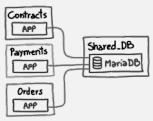
- Consuming Services sollten nicht an interne Implementierung von Downstream Services gebunden sein. Dies führt meist zu einer erhöhten Kopplung.
- Ein Microservice sollte also so designt sein, dass interne Änderungen an Microservices nicht erfordern, dass auch Consuming Services geändert werden müssen.
- Dies würde die Kosten für Änderungen erhöhen - das Gegenteil der Intention des Microservice-Ansatzes.
- Solche Kopplungen können auch zur Folge haben, dass erforderliche Änderungen in einem Microservice verschoben werden, aus Angst Änderungen bei Consuming Services zu erzeugen. Dies kann zu einer erhöhten technischen Verschuldung (Technical Debt) innerhalb des Services führen.
- Daher sollten in Microservices grundsätzlich Technologien vermieden werden, die es erforderlich machen, interne Repräsentation freizulegen (um bspw. Performance zu gewinnen).
- In diesen Fällen sollten die Vor- und Nachteile sorgsam abgewogen werden.

INTEGRATION

Integrationsstile

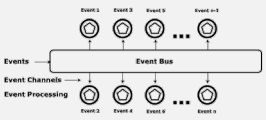
TECHNISCHE HOCHSCHULE LÜBECK

- Daten-basierte Integration
  - The shared database



- Request/Response-basierte Integration (synchron)
  - Remote Procedure Calls
  - REST

- Ereignis-basierte Integration (asynchron)
  - Asynchrone Architekturen
  - Reactive Programming

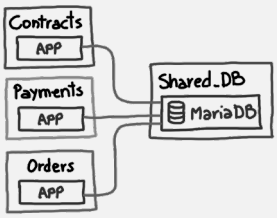


- API-Versioning

INTEGRATION

Daten-basierte Integration (Shared Database-Pattern)

Die häufigste Form der Integration „in freier Wildbahn" ist vermutlich die Datenbankintegration. Wenn Services Informationen von einem Stateful Service benötigen, müssen sie nur eine Datenbank abfragen. Und wenn sie Daten ändern wollen, aktualisieren oder erstellen sie einer Datensätze in einer Datenbank. Dies ist wahrscheinlich die schnellste und beliebteste Form der Integration. Obwohl dies ein weit verbreitetes Muster ist, ist es mit Schwierigkeiten behaftet.



Erstens zwingt man Consuming Services dazu, sich an interne Implementierungsdetails zu binden. Die Datenstrukturen der Datenbank werden vollständig mit allen anderen Parteien geteilt, die Zugriff auf die Datenbank haben. Das Ändern des Datenbankschemas kann Änderungen in Consuming Services nach sich ziehen. Dies erhöht die Kopplung und das entsprechende Abstimmung von Änderungen zwischen Diensten, die idealerweise isoliert sein und autonom weiterentwickelt werden sollen.

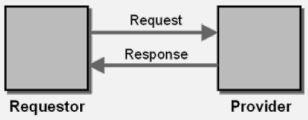
Zweitens sind Consuming Services an eine bestimmte Datenbanktechnologie gebunden. Infolgedessen könnten Consuming Services bspw. datenbankspezifische Treiber verwenden, die im Falle einer Datenbankänderung problematisch werden können (z. B. MariaDB anstelle von Postgres). Implementierungsdetails sollten vor Consuming Services grundsätzlich verborgen werden, um ein gewisses Maß an Autonomie in Bezug darauf zu ermöglichen, wie Stateful Services seine Interna im Laufe der Zeit ändern können.

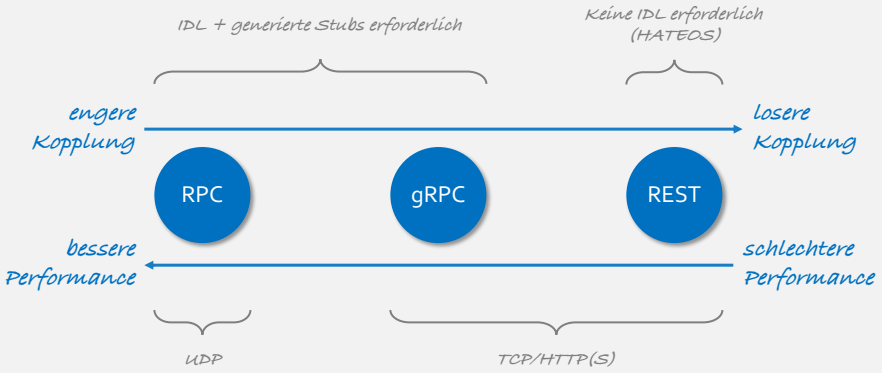
TECHNISCHE HOCHSCHULE LÜBECK

Daher haben sich im Cloud-native Umfeld weitere Integrationsmuster entwickelt, die die Autonomie zwischen Consuming und Providing Services besser sicherstellen können.

INTEGRATION

Request/Response-basierte Integration (synchron)





TECHNISCHE HOCHSCHULE LÜBECK

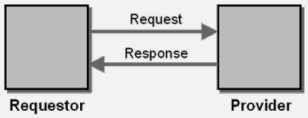
IDL: Interface definition language

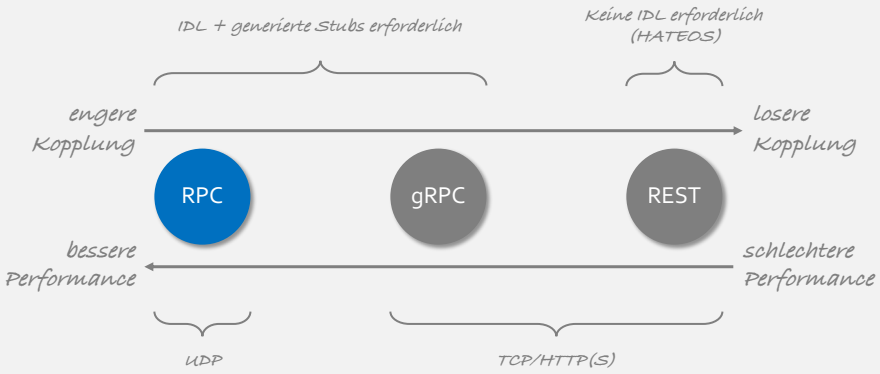
Häufig muss man also zwischen Performance und loser Kopplung abwägen.

Achtung:
Es gilt auch die „ewige" Empfehlung erst Performanceprobleme anzugehen, wenn es einen aus dem Betrieb ersichtlichen Grund dazu gibt!

INTEGRATION

Request/Response-basierte Integration (synchron)





TECHNISCHE HOCHSCHULE LÜBECK

IDL: Interface definition language

Häufig muss man also zwischen Performance und loser Kopplung abwägen.

Achtung:
Es gilt auch die „ewige" Empfehlung erst Performanceprobleme anzugehen, wenn es einen aus dem Betrieb ersichtlichen Grund dazu gibt!

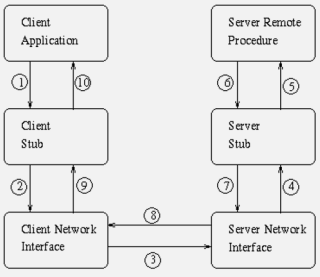
INTEGRATION

Request/Response-basierte Integration: Remote Procedure Calls (RPC)

Remote Procedure Calls (RPC) sind eine Technik zur Realisierung von Interprozesskommunikation, die den den Aufruf von Funktionen in anderen Adressräumen ermöglicht. Im Normalfall werden die aufgerufenen Funktionen auf einem anderen Computer als das aufrufende Programm ausgeführt. Es existieren mehrere Implementierungen dieser Technik, die in der Regel untereinander nicht kompatibel sind.

Die am weitesten verbreitete Variante ist das ONC RPC (Open Network Computing Remote Procedure Call), das vielfach auch als Sun RPC bezeichnet wird. ONC RPC wurde ursprünglich durch Sun Microsystems für das Network File System (NFS) entwickelt.

RPC basiert auf dem Client-Server-Modell. Die Kommunikation beginnt, indem der Client eine Anfrage (Call) an einen ihm bekannten Server schickt und auf die Antwort wartet. In der Anfrage gibt der Client an, welche Funktion mit welchen Parametern ausgeführt werden soll. Der Server bearbeitet die Anfrage und schickt die Antwort an den Client zurück. Nach Empfang der Nachricht kann der Client seine Verarbeitung fortführen.



TECHNISCHE HOCHSCHULE LÜBECK

Aus Sicht eines Client-seitigen Entwicklers sehen Remote Procedure Calls aus, wie normale Local Procedure Calls im eigenen Prozessraum.

Da RPCs aber Netzwerkkommunikation beinhalten, können diese (anders als LPC im eigenen Prozessraum) natürlich scheitern.

Weitere diesem RPC-Modell folgende Technologien sind bspw.:
- CORBA
- XML-RPC
- JSON-RPC

INTEGRATION

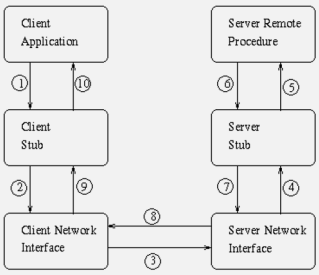
Request/Response-basierte Integration: Remote Procedure Calls (RPC)

RPC setzt meist auf UDP auf. Dies hat den Vorteil, dass kein Traffic-Overhead durch ACK-Pakete entsteht. Jedoch ist die UDP-Paketgröße nicht ausreichend für große Antworten. Daher muss hier von zwei Arten von RPC-Protokollen ausgegangen werden: Request-Reply und Request-Reply-ACK.

Um eine entfernte Funktion aufzurufen, muss eine Nachricht vom Client-Prozess zum Server-Prozess versendet werden. In dieser müssen der Name der Funktion (oder eine ID) und die zugehörigen Parameterwerte enthalten sein.

Die Suche nach einem entsprechenden Server kann durch Broadcast (in einem lokalen Netz) realisiert werden oder durch Inanspruchnahme eines Verzeichnisdienstes (Registry).

Die erforderlichen serverseitigen (register) und clientseitigen (lookup + binding + error recovery) Vorgänge werden normalerweise in sogenannten Client-/Server Stubs gekapselt, die aus Interface Definitions generiert werden.



TECHNISCHE HOCHSCHULE LÜBECK

INTEGRATION

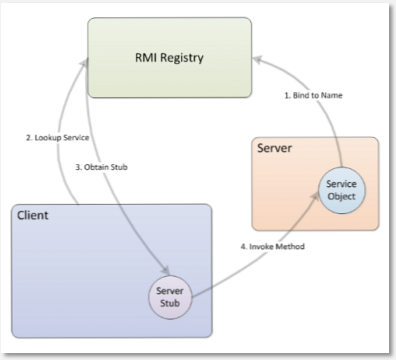
Request/Response-basierte Integration: RPC am Beispiel von Java (RMI)

RPC ist in Java als **Remote Method Invocation (RMI)** bekannt.

Dabei sieht der Aufruf für das aufrufende Objekt genauso aus wie ein lokaler Aufruf. Jedoch müssen besondere Ausnahmen (Exceptions) abgefangen werden, die zum Beispiel einen Verbindungsabbruch signalisieren können.

Auf der Client-Seite kümmert sich ein Stub um den Netzwerktransport. Das Erstellen des Stubs wird seit Java 1.5 von der Java Virtual Machine übernommen.

Für die erste Verbindungsaufnahme werden die Adresse des Servers und ein Bezeichner (ein RMI-URL) benötigt. Für den Bezeichner liefert ein (RMI Registry) auf dem Server eine Referenz auf das entfernte Objekt zurück. Damit dies funktioniert, muss sich das entfernte Objekt im Server zuvor unter diesem Namen beim Namensdienst registriert haben. Der Namensdienst ist in Java als eigenständiges Programm implementiert und wird RMI Registry genannt.



Da dies alles eine sehr Technologie-spezifische Lösung darstellt und damit eine hohe Tendenz zur engen Kopplung beinhaltet (insbesondere durch eine Stubgenerierung) hat sich dieses reine RPC-Modell im Cloud-native Computing nicht wirklich durchgesetzt.

Auch Technologie-agnostischere Ansätze wie bspw. CORBA haben die Kernprobleme der engen Kopplung nicht wirklich minimiert.

TECHNISCHE HOCHSCHULE LÜBECK

INTEGRATION

Request/Response-basierte Integration: RPC am Beispiel von Java (RMI)

TECHNISCHE HOCHSCHULE LÜBECK

Interface (IDL -> Stub - Generation)

```
import java.rmi.*;

public interface RMIInterface extends Remote {
  public String helloTo(String name) throws RemoteException;
}
```

Es muss ein `Remote`-Schnittstelle definiert werden, die sowohl Server als auch Client implementieren müssen. Diese Schnittstelle spielt dieselbe Rolle wie eine IDL in anderen RPC-Ansätzen.

Client

```
import java.net.MalformedURLException;
import java.rmi.*;

public class ClientOperation {
  private static RMIInterface remote;

  public static void main(String[] args) throws Exception {
    remote = (RMIInterface)Naming.lookup("//localhost/hello-svc");
    String response = remote.helloTo(txt);  // !!! Network call !!!
  }
}
```

Um den Server zu "finden", verwendet der Client die RMI-Registry (`Naming`), um anhand eines registrierten Namens eine Referenz für das entfernte Objekt zu erhalten. Mit diesem `RMIInterface`-Objekt kann der Client jetzt mit auf dem Server einen Remote Call absetzen und die Antwort der entfernten Methode erhalten.

Auch wenn Java, die Stub Generierung seit Version 1.5 transparent im Hintergrund durch die JVM erledigt, ist Client- wie Server-seitig weiterhin eine gemeinsame Schnittstelle erforderlich (wie bei allen RPC-Ansätzen).

Problematisch ist dies deswegen:
Wenn man auf Server-Seite die Schnittstelle nur um neue Funktionen erweitert (also nicht einmal bestehende Funktionen ändert), muss dennoch auch alle Clients ändern!
=> zu enge Kopplung für Microservice-Architekturen.

INTEGRATION

Request/Response-basierte Integration: RPC am Beispiel von Java (RMI)

TECHNISCHE HOCHSCHULE LÜBECK

Server

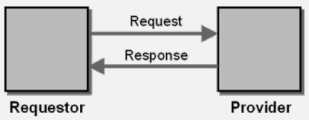
```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class ServerOperation extends UnicastRemoteObject implements RMIInterface {

  private static final long serialVersionUID = 1L;

  protected ServerOperation() throws RemoteException { super(); }

  @Override public String helloTo(String name) throws RemoteException {
    System.out.println(name + " is trying to contact!");
    return "Server says hello to " + name;
  }

  public static void main(String[] args) throws Exception {
    Naming.rebind("//localhost/hello-svc", new ServerOperation());
    System.out.println("Hello service ready");
  }
}
```

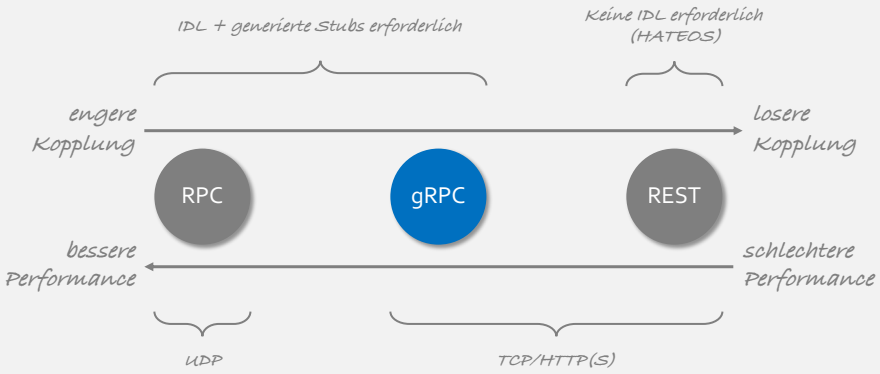
Ein Java RMI-Server erweitert immer das `UnicastRemoteObject` und implementiert die zuvor erstellteRMI-Schnittstelle (IDL).

In der `main()`-Methode binden wir den Server auf localhost an den Namen "hello-svc".

INTEGRATION

Request/Response-basierte Integration (synchron)





TECHNISCHE HOCHSCHULE LÜBECK

IDL: Interface definition language

Häufig muss man also zwischen Performance und loser Kopplung abwägen.

Achtung:
Es gilt auch die „ewige" Empfehlung erst Performanceprobleme anzugehen, wenn es einen aus dem Betrieb ersichtlichen Grund dazu gibt!

INTEGRATION

Request/Response-basierte Integration: gRPC Remote Procedure Calls

gRPC

gRPC (gRPC Remote Procedure Calls) ist ein universelles RPC Protokoll zum Aufruf von Remote Procedures in verteilten Systemen. Es basiert auf HTTP/2 und Protocol Buffers und ist ein Projekt der Cloud Native Computing Foundation.

Mittels einer IDL wird die Schnittstelle unabhängig von einer konkreten Programmiersprache spezifiziert.

Mittels des Protocol-Buffer-Compilers 'protoc' (und eines Plugins für gRPC) kann aus dieser Beschreibung Server- und Client-Code, sogenannte Stubs, generiert werden.

Language	OS	Compilers / SDK
C/C++	Linux, Mac	GCC 4.9+, Clang 3.4+
C/C++	Windows 7+	Visual Studio 2015+
C#	Linux, Mac	.NET Core, Mono 4+
C#	Windows 7+	.NET Core, NET 4.5+
Dart	Windows, Linux, Mac	Dart 2.2+
Go	Windows, Linux, Mac	Go 1.13+
Java	Windows, Linux, Mac	JDK 8 recommended
Kotlin/JVM	Windows, Linux, Mac	Kotlin 1.3+
Node.js	Windows, Linux, Mac	Node v8+
Objective-C	macOS 10.10+, iOS 9.0+	Xcode 7.2+
PHP	Linux, Mac	PHP 7.0+
Python	Windows, Linux, Mac	Python 3.5+
Ruby	Windows, Linux, Mac	Ruby 2.3+

Quelle: https://www.grpc.io

TECHNISCHE HOCHSCHULE LÜBECK

Dadurch das gRPC auf HTTP (und somit auf Anwendungsebene definiert ist) und nicht UDP (wie RPC) beruht, ist es einfacher zu handhaben (bspw. mittels Kubernetes Ingress Rules).

Die Kopplung zwischen Services ist im Vergleich zu RPC geringer (Requestor und Service müssen bspw. nicht in derselben Sprache geschrieben sein, wie bspw. bei Java RMI).

INTEGRATION

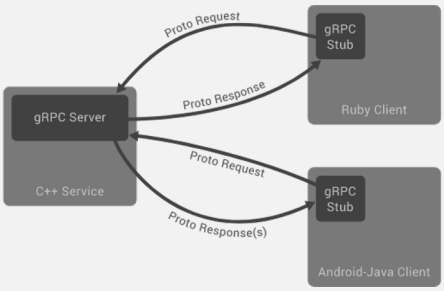
Request/Response-basierte Integration: gRPC Remote Procedure Calls

TECHNISCHE HOCHSCHULE LÜBECK

Wie in vielen RPC-Systemen basiert gRPC auf der Idee, einen Service zu definieren und die Methoden anzugeben, die mit ihren Parametern und Rückgabetypen remote aufgerufen werden können.

Auf der Serverseite implementiert ein Service diese Schnittstelle und führt einen gRPC-Server Prozess aus, um Client-Requests zu verarbeiten. Auf der Clientseite verfügt der Client über einen Stub, der dieselben Methoden wie der Server Prozess bereitstellt.

gRPC-Clients und -Server können in einer Vielzahl von Laufzeit-Umgebungen ausgeführt werden und miteinander kommunizieren.



Quelle: https://www.grpc.io

INTEGRATION

Request/Response-basierte Integration: gRPC Protocol Buffers

TECHNISCHE HOCHSCHULE LÜBECK

Standardmäßig verwendet gRPC Protocol Buffers zum Serialisieren strukturierter Daten (obwohl es mit anderen Datenformaten wie bspw. JSON verwendet werden kann).

Hierzu müssen die zu serialisierenden Daten definiert werden. Protocol Buffer Daten sind als Nachrichten strukturiert, wobei jede Nachricht aus einer Liste von Feldern (Key-Value Paare) besteht, die den Namen und den Datentyp und die Serialisierungsposition innerhalb der Nachricht angeben.

Mittels des Protocol Buffer Compiler `protoc` lassen sich Protokolldefinitionen in Datenzugriffsklassen unterstützter Programmiersprachen generieren. Diese bieten einfache Zugriffsmethoden für jedes Feld wie `name()` und `set_name()` sowie Methoden zum Serialisieren / Parsen der gesamten Struktur.

Nachricht

```
message Person {
  string name = 1;
  int32 id = 2;
  bool has_ponycopter = 3;
}
```

Service Definition + Nachrichten

```
// The greeter service definition.

service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

Quelle: https://www.grpc.io

INTEGRATION

Request/Response-basierte Integration: gRPC Service Methods

TECHNISCHE HOCHSCHULE LÜBECK

Mit gRPC lassen sich vier Arten von Servicemethoden definieren.

1 **Unary RPCs**, bei denen der Client einen einzelnen Request an einen Service sendet und eine einzelne Response zurückerhält, genau wie bei einem normalen synchronen Funktionsaufruf.

```
rpc SayHello(HelloRequest)
returns (HelloResponse);
```

2 Bei **Server-Streaming-RPCs** sendet der Client einen Request an einen Service sendet und erhält als einen Stream von Nachrichten. Der Client liest aus dem Rückgabe-Stream, bis keine Nachrichten mehr vorhanden sind. gRPC garantiert die Reihenfolge der Nachrichten innerhalb eines einzelnen RPC-Requests.

```
rpc LotsOfReplies(HelloRequest)
returns (stream HelloResponse);
```

3 Bei **Client-Streaming-RPCs** sendet der Client einen Stream von Nachrichten an den Service. Sobald der Client die Nachrichten fertig geschrieben hat, wartet er darauf, dass der Server sie liest und seine Response zurückgibt. gRPC garantiert die Reihenfolge der Nachrichten innerhalb eines einzelnen RPC-Requests.

```
rpc LotsOfGreetings(stream HelloRequest)
returns (HelloResponse);
```

4 Bei **Bidirectional Streaming-RPCs** senden beide Seiten Streams von Nachrichten. Die beiden Streams arbeiten unabhängig voneinander, sodass Clients und Server in beliebiger Reihenfolge lesen und schreiben können. Die Reihenfolge der Nachrichten in jedem Stream bleibt erhalten.

```
rpc BidiHello (stream HelloRequest)
returns (stream HelloResponse);
```

Quelle: https://www.grpc.io

INTEGRATION

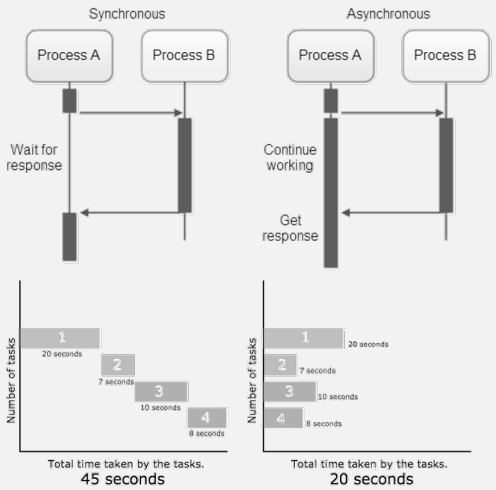
Request/Response-basierte Integration: gRPC Asynchron (Non-Blocking) oder Synchron (Blocking)

TECHNISCHE HOCHSCHULE LÜBECK

Synchrone – also blockierende – RPC-Aufrufe sind die engste Annäherung an die Abstraktion eines lokalen Prozeduraufrufs, den das RPC-Modell ja grundsätzlich anstrebt.

Andererseits sind Netzwerkzugriffe von Natur aus I/O-Prozesse und damit um Größenordnungen langsamer als lokale Prozeduraufrufe. Daher ist es in vielen Szenarien üblich, RPCs asynchron zu starten, ohne den aktuellen Thread zu blockieren.

Die gRPC-Programmier-API ist in den meisten Sprachen ist sowohl synchron als auch asynchron (d.h. mittels Futures und await/async) nutzbar.



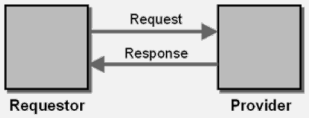
gRPC

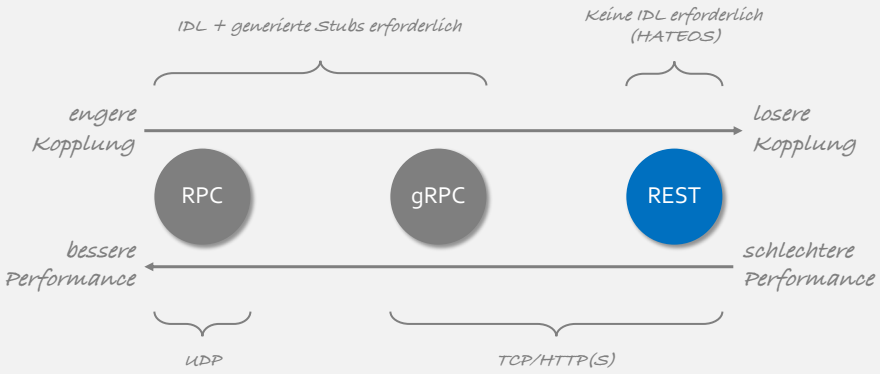
Bei mehreren asynchron gestarteten Tasks braucht man so nicht die Summe aller Tasks, sondern nur die Dauer des längsten Tasks.

Quelle: https://www.grpc.io

INTEGRATION

Request/Response-basierte Integration (synchron)





TECHNISCHE HOCHSCHULE LÜBECK

IDL: Interface definition language

Häufig muss man also zwischen Performance und loser Kopplung abwägen.

Achtung:
Es gilt auch die „ewige" Empfehlung erst Performanceprobleme anzugehen, wenn es einen aus dem Betrieb ersichtlichen Grund dazu gibt!

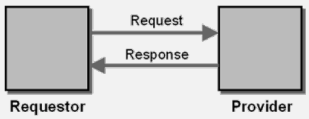
INTEGRATION

Request/Response: Representational State Transfer (REST)

REST ist ein Paradigma für die Softwarearchitektur von verteilten Systemen, insbesondere für Webservices. REST hat das Ziel, einen Architekturstil zu schaffen, der die Anforderungen des modernen Web besser darstellt. REST fordert eine einheitliche Schnittstelle zu Ressourcen im Netz.

Der Zweck von REST liegt schwerpunktmäßig auf der Maschine-zu-Maschine-Kommunikation. REST stellt eine einfache Alternative zu ähnlichen Verfahren wie SOAP und WSDL und dem verwandten Verfahren RPC dar. Anders als bei vielen verwandten Architekturen kodiert REST keine Methodeninformation in Uniform Resource Identifiern (URI), da der URI Ort und Namen der Ressource angibt, nicht aber die Funktionalität, die der Web-Dienst zu der Ressource anbietet.

Der Vorteil von REST liegt darin, dass im WWW bereits ein Großteil der für REST nötigen Infrastruktur (z. B. Web- und Application-Server, HTTP-fähige Clients, usw.) vorhanden ist, und viele Web-Dienste per se REST-konform sind. Eine Ressource kann dabei über verschiedene Medientypen repräsentiert werden.



TECHNISCHE HOCHSCHULE LÜBECK

INTEGRATION

Request/Response: REST – 5 Prinzipien

TECHNISCHE HOCHSCHULE LÜBECK

Client-Server (1)

Dabei stellt der Server (Provider) einen Dienst bereit, der bei Bedarf vom Client (Requestor) angefragt werden kann. Der Hauptvorteil den diese Anforderung bringt ist die einfache Skalierbarkeit der Server, da diese unabhängig vom Client agieren. Dies ermöglicht u.a. eine Unterschiedlich schnelle Entwicklung der beiden Komponenten.

Caching (2)

HTTP Caching kann zur Performance-Optimierung genutzt werden. Mittels Caching kann es allerdings passieren, dass Clients auf veraltete Cache-Daten zurückgreifen.

Zustandslosigkeit (Stateless) (3)

Jede REST-Nachricht enthält alle Informationen, die für den Server bzw. Client notwendig sind, um die Nachricht zu verstehen. Weder der Server noch die Anwendung soll Zustandsinformationen zwischen zwei Nachrichten speichern. Jeder Request eines Clients an den Server ist insofern in sich geschlossen, als dass sie sämtliche Informationen über den Anwendungszustand beinhaltet, die vom Server für die Verarbeitung der Anfrage benötigt werden.

Dies begünstigt die horizontale Skalierbarkeit eines Services.

Mehrschichtige Systeme (4)

Die Systeme sollen mehrschichtig aufgebaut sein. Dadurch reicht es, dem Anwender lediglich eine Schnittstelle anzubieten.

Dahinterliegende Ebenen können verborgen bleiben und somit die Architektur insgesamt vereinfacht werden. Dies ermöglicht eine bessere horizontale Skalierbarkeit der Server, sowie eine mögliche Abkapselung durch Firewalls. Durch Cache-Speicher an den Grenzen (z. B. vom Server zum Web) kann die Effizienz der Anfragen erhöht werden; siehe Caching.

INTEGRATION

Request/Response: REST – Einheitliche Schnittelle (5)

TECHNISCHE HOCHSCHULE LÜBECK

Selbstbeschreibende Nachrichten (5.1)

REST-Nachrichten sollen selbstbeschreibend sein. Dazu zählt u. a. die Verwendung von Standardmethoden (bspw. HTTP-Verben). Über diese Standardmethoden lassen sich Ressourcen manipulieren.

Adressierbarkeit von Ressourcen (5.2)

Jede Ressource eines Service wird über einen Uniform Resource Identifier (URI) identifiziert. Jeder REST-konforme Service ist über einen Uniform Resource Locator (URL) adressierbar. Dieser Locator standardisiert den Zugriffsweg zum Angebot eines Webservices für eine Vielzahl von Anwendungen (Clients). Eine konsistente Adressierbarkeit erleichtert es, einen Webservice im Rahmen eines Service-of-Service Ansatzes zu nutzen.

Repräsentationen zur Veränderung von Ressourcen (5.3)

Ressourcen können unterschiedliche Darstellungsformen (Repräsentationen) haben. Ein REST-konformer Service kann verschiedene Repräsentationen einer Ressource ausliefern (häufig HTML, JSON oder XML) oder auch die Beschreibung oder Dokumentation des Dienstes. Die Veränderung einer Ressource (also deren Status) soll nur über eine Repräsentation erfolgen.

Hypermedia as the Engine of Application State (HATEOAS) (5.4)

Bei HATEOAS navigiert der Client einer REST-Schnittstelle ausschließlich über URLs, die vom Service bereitgestellt werden. Abhängig von der gewählten Repräsentation geschieht die Bereitstellung der URIs über Hypermedia,

- also z. B. in Form von „href"- und „src"-Attributen bei HTML-Dokumenten, oder
- in für die jeweilige Schnittstelle definierten und dokumentierten JSON- bzw. XML-Attributen/-Elementen.

HATEOAS-konforme REST-Services sind formal ein endlicher Automat, dessen Zustandsveränderungen durch die Navigation mittels der bereitgestellten URIs erfolgt.

HATEOAS ermöglicht dadurch lose Kopplung von Services und gewährleistet, dass die Schnittstelle verändert werden kann. Im Gegensatz dazu kommuniziert ein SOAP-basierter Webservice über ein fixiertes Interface. Für eine Änderung des Service muss hier eine neue Schnittstelle bereitgestellt werden und in der Schnittstellenbeschreibung (ein WSDL-Dokument) definiert werden.

REST hat u.a. zum Ziel die Einheitlichkeit und einfache Nutzbarkeit der Schnittstelle mittels der folgenden Prinzipien zu gewährleisten.

INTEGRATION

Request/Response: REST – HATEOS Beispiel

TECHNISCHE HOCHSCHULE LÜBECK

Beispiel für einen GET-Request, der Konto-Informationen im JSON-Format abruft.

```
GET /accounts/123abc
HTTP/1.1 Host: bank.example.com
Accept: application/json
...
```

Examplarische Antwort inkl. HATEOS-konformer Folgeoperationen (Links)

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: ...

{ "account":
  { "account_id": "123abc",
    "balance": {
      "currency": "EUR",
      "value": 100.0
    },
    "links": {
      "deposit": "/accounts/123abc/deposit",
      "withdraw": "/accounts/123abc/withdraw",
      "transfer": "/accounts/123abc/transfer",
      "close": "/accounts/123abc/close"
    }
  }
}
```

HATEOS

D.h. die Links zu Folgeoperationen stehen in der Antwort und müssen nicht durch den Client gewusst werden.

Die Links können daher auch in einer neue Service-Version geändert werden ohne den Client anpassen zu müssen.

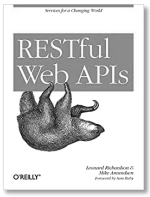
INTEGRATION

Request/Response: REST – Maturity Model

TECHNISCHE HOCHSCHULE LÜBECK

Level	Eigenschaften
0	• verwendet **XML-RPC** oder **SOAP** • der Service wird über einen einzelnen URI adressiert • verwendet eine einzelne HTTP-Methode (oft POST)
1	• verwendet verschiedene URIs und Ressourcen • verwendet eine einzelne HTTP-Methode (oft POST)
2	• verwendet verschiedene URIs und Ressourcen • verwendet mehrere HTTP-Methoden
3	• basiert auf HATEOAS und verwendet daher **Hypermedia** für Navigation • verwendet verschiedene URIs und Ressourcen • verwendet mehrere HTTP-Methoden

Das Richardson Maturity Model (RMM) ist ein von Leonard Richardson entwickelter Maßstab, der angibt, wie REST-konform ein Service implementiert wurde.



CRUD	SQL-92	HTTP (REST)
Create	INSERT	PUT oder POST
Read (Retrieve)	SELECT	GET
Update	UPDATE	PATCH oder PUT
Delete (Destroy)	DELETE	DELETE

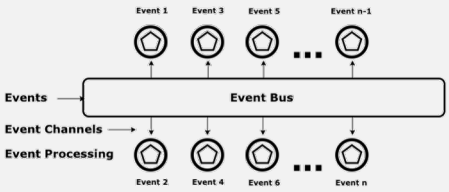
CRUD umfasst die vier grundlegenden atomaren Operationen persistenter Speicher und wird häufig wie hier gezeigt in REST-konforme Service-APIs übersetzt. Erläuternd sind die korrespondierenden SQL-92 Statements relationaler Datenbanken angegeben.

INTEGRATION

Ereignis-basierte Integration (asynchron)

Anstelle einer auf Request-Response basierenden Integration kehrt die ereignisbasierte Integration die Dinge um. Consuming Services initiieren keine Requests mehr, um dann auf Responses zu warten. Consuming Services werden vielmehr zu Event-Emitting Services, die Events erzeugen um mitzuteilen, dass etwas geschehen ist und erwarten, dass andere Services wissen, wie diese Events zu verarbeiten sind. Event Emitting Services sagen niemandem, was zu tun ist.

Daher sind ereignisbasierte Systeme von Natur aus asynchronAsynchrone Architekturen sindmeist auch gleichmäßiger verteilt. Die Geschäftslogik ist also nicht zentralisiert, sondern wird gleichmäßiger durch mehrere Services realisiert.



Die ereignisbasierte Integration resultiert somit meist in starker Entkoppelung von Emitting und Consuming Services. Genau dies strebt man in Microservice-Architekturen an.

Da ein Event erzeugender Service nicht weiß, welche anderen Services darauf reagieren, bedeutet dies auch, dass Ereignissen problemlos neue Abonnenten hinzugefügt werden können, was vor allem die horizontale Skalierung enorm vereinfacht.

TECHNISCHE HOCHSCHULE LÜBECK

NATS

APACHE kafka

redis

Daher erfreuen sich Messaging Systeme wie Kafka oder NATS (aber auch DB wie Redis) steigender Beliebtheit im Cloud-native Computing, da diese die Grundlage für asynchrone und entkoppelte Architekturen legen.

INTEGRATION

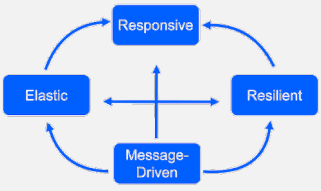
Ereignis-basierte Integration => Das Reaktive Manifest (I)

Das Reaktive Manifest

*„Wir glauben, dass die Anforderungen, die heute an Computersysteme gestellt werden, nur zu erfüllen sind durch die gleichzeitige Ausrichtung an vier Qualitäten, deren Wert bislang nur einzeln betrachtet wurde: Systeme müssen stets **antwortbereit**, **widerstandsfähig**, **elastisch** und **nachrichtenorientiert** sein. Dann nennen wir sie reaktive Systeme.*

*Computersysteme, die nach diesen Anforderungen entwickelt werden, erweisen sich als anpassungsfähiger, mit **weniger starr gekoppelten Komponenten** und in jeder Hinsicht **skalierbarer**. Sie sind **einfacher weiterzuentwickeln und zu verändern**. Sie **reagieren zuverlässiger** und eleganter **auf Fehler** und vermeiden so desaströse Ausfälle. Reaktive Systeme bereiten dem Benutzer durch ihre fortwährende Antwortfreudigkeit eine interaktive und höchst befriedigende Erfahrung.“*

Quelle: https://www.reactivemanifesto.org/de



Große Anwendungen bestehen stets aus mehreren Komponenten und sind daher abhängig von deren Reaktivität. Deshalb müssen die genannten vier reaktiven Qualitäten in der Architektur einer jeden Ebene des Gesamtsystems berücksichtigt werden, wodurch reaktive Systeme miteinander komponierbar sind.

TECHNISCHE HOCHSCHULE LÜBECK

„Die größten Computersysteme der Welt basieren bereits auf diesen Prinzipien und dienen Milliarden von Menschen in deren täglichen Leben. Es ist an der Zeit, diesen Ansatz bewusst zu Grunde zu legen anstatt ihn in Teilen für jedes Projekt neu zu entdecken.“

INTEGRATION

Ereignis-basierte Integration => Das Reaktive Manifest (II)

Antwortbereit (Englisch: **responsive**): Das System antwortet **zeitgerecht** […]. Antwortbereitschaft ist die Grundlage für Funktion, Benutzbarkeit und Zuverlässigkeit eines Systems, da Fehler in verteilten Systemen nur durch die Abwesenheit einer Antwort sicher festgestellt werden können. Ohne vereinbarte Timeouts ist die Erkennung und Behandlung von Fehlern nicht möglich.

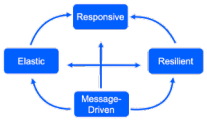
Elastisch (Englisch: **elastic**): Das System bleibt auch unter sich **ändernden Lastbedingungen** antwortbereit. Bei Laständerungen werden automatisch die Replizierungsfaktoren und damit die genutzten Ressourcen angepasst. Dazu darf das System keine Engpässe aufweisen, die den Gesamtdurchsatz vor Erreichen der geplanten Maximalauslegung einschränken. Ideal ist eine Architektur, die keine fixen Engpässe aufweist. In diesem Fall kann das System in unabhängige Komponenten zerlegt und diese unabhängig auf beliebig viele Ressourcen verteilt werden. Reaktive Systeme unterstützen die Erfassung ihrer Auslastung zur Laufzeit, um automatisch regelnd eingreifen zu können. Dank ihrer Elastizität können sie auf Speziallösungen verzichten und mit handelsüblichen Komponenten implementiert werden.

Widerstandsfähig (English: **resilient**): Widerstandsfähigkeit ist erreichbar durch **Replizieren** der Funktionalität, **Isolation** von Komponenten sowie **Delegieren** von Verantwortung. Der Ausfall eines Teilsystems soll auf dieses begrenzt bleiben, um andere Teilsysteme nicht in ihrer Funktion zu beeinträchtigen. Die Wiederherstellung des Normalzustandes wird einer übergeordneten Komponente übertragen, die die geforderte Verfügbarkeit sicherstellt.

Nachrichtenorientiert (Englisch: **message driven**): Das System verwendet **asynchrone Nachrichtenübermittlung** zwischen seinen Komponenten zur Sicherstellung von deren Entkopplung und Isolation. Die explizite Verwendung von Nachrichtenübermittlung führt zu einer **ortsunabhängigen** Komponenten und erlaubt die transparente Skalierung. Die Überwachung von Nachrichtenpuffern ermöglicht kontinuierlichen Einblick in das Laufzeitverhalten des Systems. Nicht-blockierende nachrichtenorientierte Systeme erlauben eine effiziente Verwendung von Ressourcen, da Komponenten beim Ausbleiben von Nachrichten vollständig inaktiv bleiben können.

Quelle: https://www.reactivemanifesto.org/de

TECHNISCHE HOCHSCHULE LÜBECK



Ortsunabhängigkeit bedeutet, dass Code und Semantik des Programms nicht davon abhängen, ob dessen Teile auf demselben Computer oder verteilt über ein Netzwerk ausgeführt werden

INTEGRATION

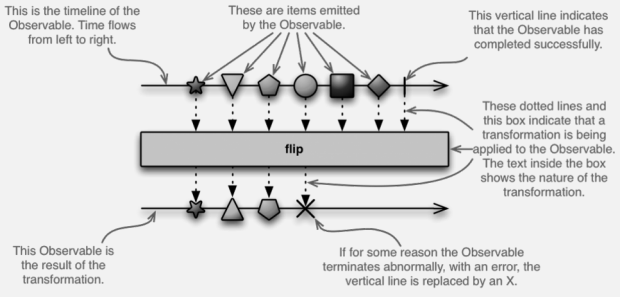
Ereignis-basierte Integration => Reaktive Erweiterungen von Programmiersprachen (Rx X)

Reaktive Erweiterungen von Programmiersprachen (Rx) sind ein Mechanismus, mit dem die Ergebnisse mehrerer Requests zusammengesetzt und Operationen auf diesem Event-Stream ausgeführt werden können. Die Requests können dabei sowohl blockierend als auch nicht blockierend sein. Im Kern kehrt Rx traditionelle Kontrollefllüsse um.

Anstatt nach Daten zu fragen (und auf die Antwort zu warten), beobachtet man das Ergebnis einer Operation und reagiert, wenn sich etwas ändert. Bei einigen Rx-Implementierungen können Funktionen (wie filter(), map() oder reduce()) auf solchen Observables ausgeführt werden.

Verschiedenste Rx-Implementierungen erfahren im Rahmen asynchroner Architekturen steigende Beliebtheit, da es diese Bibliotheken ermöglichen, die Details der asynchronen Requestabwicklung zu vereinfachen.

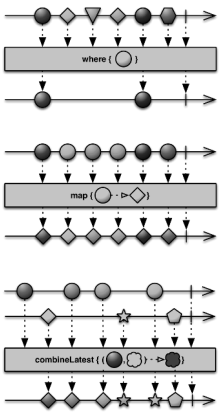
So lassen sich bspw. mehrere (blocking oder non-blocking) Requests asynchron starten. Anschließend beobachtet man das Eintreffen von Antwortevents der Downstream Services der in einem Observable auftaucht und reagiert einfach nur noch mittels einer Operator-Chain darauf.



Quelle der Diagramme: http://rxpy.readthedocs.io

TECHNISCHE HOCHSCHULE LÜBECK

Beispiele reaktiver Operatoren in RxPY *(Marble Diagrams)*

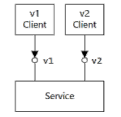


INTEGRATION

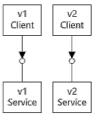
API-Versioning (Abwärts- und Vorwärtskompatibilität)

- Die API eines Services sollte möglichst stabil sein.
- Im Unternehmensumfeld möchte man Änderungen insb. an Legacy-Client-Anwendungen vermeiden.
- Betreiber einer öffentlichen Web-API müssen bei Updates ebenfalls vorsichtig vorgehen. Wenn z.B. Twitter seine REST-API ändern würde, wären Millionen von Client-Installationen zu aktualisieren.
- Ganz allgemein sind zwei Services S1 und S2 zueinander kompatibel, falls sie sich gegeneinander austauschen lassen, ohne dass Consuming Services dies bemerken würden.
- Kompatibilität kann jedoch auch als ein historischer Zusammenhang verstanden werden. Wenn z.B. der Dienst S2 durch Änderung von S1 entstand, spricht man von **Abwärts- und Vorwärtskompatibilität**.
- Angenommen es wurde ein Consuming Service C1 für S1 und später ein weiterer Consuming Service C2 für S2 entwickelt. Wenn C2 auch mit S1 funktioniert, dann sind C2 abwärts- und S1 vorwärtskompatibel.
- Wenn umgekehrt C1 mit S2 funktioniert, dann ist C1 aufwärts- und S2 abwärtskompatibel.
- Nachrichten mit Datenformaten wie XML, JSON oder Protocol Buffers lassen sich zum Beispiel um neue optionale Felder erweitern. Weil die Felder optional sind, muss ein älterer Consuming Service sie nicht nutzen, wenn er eine Nachricht an einen Server schickt.
- Die API des Servers ist in diesem Fall **abwärtskompatibel**.
- Eventuell antwortet der Service dem Consuming Service mit einer Nachricht und verwendet dabei Felder, die der Consuming Service nicht kennt. Falls Letzterer diese Nachricht trotzdem akzeptiert, ist er **vorwärtskompatibel**. Der Client ignoriert die unbekannten Felder, sollte sie aber nicht aus dem erhaltenen Dokument entfernen, falls weitere dokumentenzentrierte Interaktionen mit dem Server folgen.

TECHNISCHE HOCHSCHULE LÜBECK



Service supports two versions

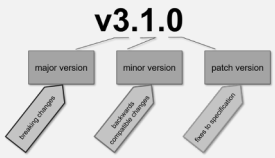


Side-by-side deployment

INTEGRATION

API-Versioning (Semantic Versioning)

Eine API sollte grundsätzlich stabil sein. Falls trotzdem Änderungen zum Umsetzen neuer Anforderungen notwendig sind, ist es besser, die Änderungen auf abwärts- und vorwärtskompatible Art und Weise durchzuführen. Ist eine inkompatible Änderung unausweichlich, wird meist zur Veranschaulichung einen neuer Versionsidentifikator eingeführt. Bekannt ist hier beispielsweise das Semantic Versioning mit dem Versionerungsschema MAJOR.MINOR.PATCH:



- Die **MAJOR-Version** wird bei **inkompatiblen API-Änderungen** inkrementiert.
- Die **MINOR-Version** kommt zum Einsatz, wenn die API um zusätzliche **Funktionen erweitert** wird, aber insgesamt **abwärtskompatibel** bleibt.
- Die **PATCH-Version** wird ausschließlich für **abwärtskompatible Fehlerkorrekturen** verwendet.

Selbst bei sehr kleinen inkompatiblen Änderungen, ist also immer die MAJOR-Version hochzuzählen. Semantic Versioning dient generell nicht dazu anzuzeigen, wie umfangreich eine Änderung ist oder wie hoch der Migrationsaufwand für einen Client bei einem Umstieg wäre.

Man kann auch nicht voraussetzen, dass ein Consuming Service, der für Version 1.2.3 entwickelt wurde, auch mit der Version 1.1.9 läuft, da die Versionen 1.1.x ja dennoch weniger Funktionsumfang bieten, als die Version 1.2.x.

TECHNISCHE HOCHSCHULE LÜBECK

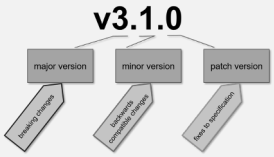
INTEGRATION

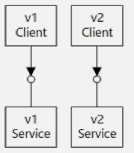
API-Versioning (Wo platziert man die Versionsnummer?)

- Der Version-in-der-URL-Ansatz ist vermutlich am weitesten verbreitet und leicht nachvollziehbar. Dieser Ansatz ist aber streng genommen nicht REST-konform.

```
https://my.service.io/v1.0.1/ressource
https://my.service.io/v1.2.0/ressource
```

- Der Version-in-der-URL-Ansatz passt jedoch zu APIs mit Remote Procedure Calls (RPC).
- Eine versionierte API kann beispielsweise bei mehreren Backend-Servern umgesetzt werden, die jeweils eine API-Version unterstützen.
- Ein API-Gateway, das zwischen Client und Server steht, könnte anhand der URLs die Requests zum jeweiligen Backend-Server weiterleiten.
- Der Versionsidentifikator lässt sich auch als Query-Parameter, Urlform-encoded-Parameter oder applikationsspezifischer Header übergeben. API-Gateways unterstützen in der Regel alle genannten Alternativen.
- Ein Versionsidentifikator in der URL macht für eine REST-API wenig Sinn. Zweck einer URL beziehungsweise einer URI ist die eindeutige Identifikation einer Ressource.
- Wer mit dem Versionsidentifikator tatsächlich eine andere Entität identifizieren möchte, kann das so durchführen. Was dann allerdings gem. REST versioniert wird, ist die Ressource und nicht die API!





Side-by-side deployment

TECHNISCHE HOCHSCHULE LÜBECK

INTEGRATION

API-Versioning (Vorsicht bei rein semantischen Änderungen der API)

Probleme bei semantischen Änderungen der API:

- Problematisch können semantische Veränderungen sein, die nicht an eine Syntaxänderung gekoppelt sind.
- Angenommen, das Feld "Amount" wird zur Angabe von Beträgen in Euro verwendet.
- Nach einer Änderung der API fügt man ein neues Currency-Feld hinzu, sodass die API flexibler wird, denn nun ließen sich auch andere Währungen verwenden.
- Ältere Clients, die das neue Currency-Feld nicht kennen, gehen weiterhin davon aus, dass im Amount-Feld ausschließlich Beträge in Euro angegeben sind.
- Eventuelle Integrationsfehler könnten unbemerkt bleiben, weil ältere Clients noch immer das Amount-Feld vorfinden.

Lösungsansätze:

- Man könnte es deswegen umbenennen, um die semantische Veränderung auch syntaktisch zu signalisieren.
- Alternativ ließe sich ein zweites Amount-Feld einführen.
- Das ursprüngliche Feld mit Euro-Beträgen bleibt für die älteren Clients erhalten und wird in der Dokumentation als "deprecated" (= veraltet) markiert.

TECHNISCHE HOCHSCHULE LÜBECK

INTEGRATION

API-Versioning (Fazit)

- Wenn Consuming Service und Downstream-Service-API vom selben Team entwickelt und betrieben werden, besteht kein wirklicher Grund, die API zu versionieren. Die Komplexität, die Versionierung mit sich bringt, kann man in diesem Fall vermeiden.
- Schwierig wird es, falls Consuming Services (oder sonstige Clients) existieren, deren Aktualisierung man nicht beeinflussen kann oder möchte: Der Betreiber einer öffentlichen API kennt häufig gar nicht alle Consuming Services (insb. nicht bei Public APIs) oder kann zumindest nicht erwarten, dass sie zu einem bestimmten Zeitpunkt aktualisiert werden.
- Typischerweise werden daher APIs in der Produktentwicklung versioniert, weil die Entwicklung über einen längeren Zeitraum erfolgt und weil je nach Art des Produktes viele Installationen existieren können.
- Dennoch sollte eine API möglichst stabil sein. Falls Änderungen trotzdem notwendig sind, dann sollten sie abwärts- und vorwärtskompatibel umgesetzt werden.
- Die Einführung eines neues Versionsidentifikators bei einer Service-API sollte eine Ausnahme sein, die aber manchmal notwendig ist, um Consuming Services auf inkompatible Änderungen hinzuweisen.

INHALTE

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

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

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

Skalierung von Microservices
- Failure is Everywhere
- 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

FAILURE IS EVERYWHERE

und die „fallacies of distributed computing"

TECHNISCHE HOCHSCHULE LÜBECK

Die 5 Gebote der Cloud

1. Anything fails all the time
2. Focus on MTTR and not on MTTF
3. Respect the Eight Fallacies of Distributed Computing
4. Scale out, not up
5. Treat ressources as cattle, not pets

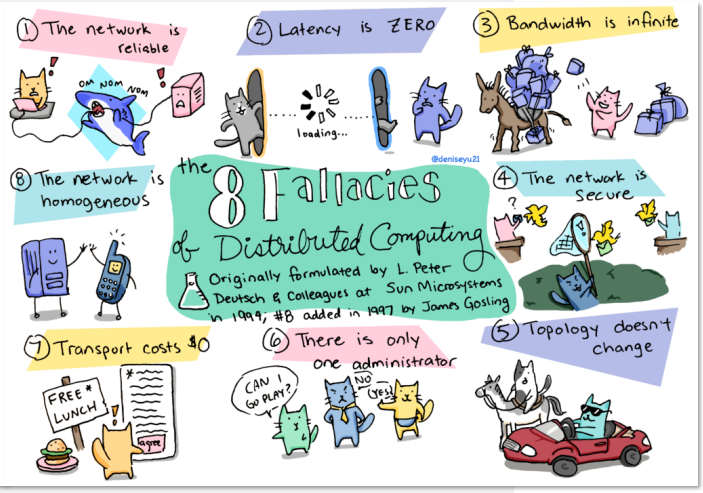


Bild: Denise Yu, https://www.linkedin.com/in/deniseyu/

FAILURE IS EVERYWHERE

Bill Joy + Tom Lyon (1 – 4) , *Peter Deutsch (5- 7)*, *James Gosling (8)*

TECHNISCHE HOCHSCHULE LÜBECK

Diese Liste der sogenannten **Irrtümer der verteilten Datenverarbeitung** sind eine Sammlung eigentlich trivialer, aber doch häufiger fehlerhafter Annahmen, die Entwickler (häufig unbewusst) voraussetzen, wenn sie insbesondere das erste Mal eine verteilte Anwendung entwickeln.

Die Liste der Fallacies of Distributed Computing kommt ursprünglich von **Sun Microsystems** und wurde dort von Bill Joy und Tom Lyon mit vier Trugschlüssen eröffnet. Weite Bekanntheit erlangten sie 1994 durch Peter Deutsch, der sie zu sieben Trugschlüsse erweiterte und als "The Seven Fallacies of Distributed Computing" veröffentlichte. James Gosling, ebenfalls von Sun, setzte ungefähr 1997 noch den achten Punkt dazu.

(1) Das Netzwerk ist ausfallsicher
(2) Die Latenzzeit ist gleich null
(3) Der Datendurchsatz ist unbegrenzt
(4) Das Netzwerk ist sicher
(5) Die Netzwerktopologie wird sich nicht ändern
(6) Es gibt immer nur einen Netzwerkadministrator
(7) Die Kosten des Datentransports können mit null angesetzt werden
(8) Das Netzwerk ist homogen

Fallacies of distributed computing

SUN Microsystems

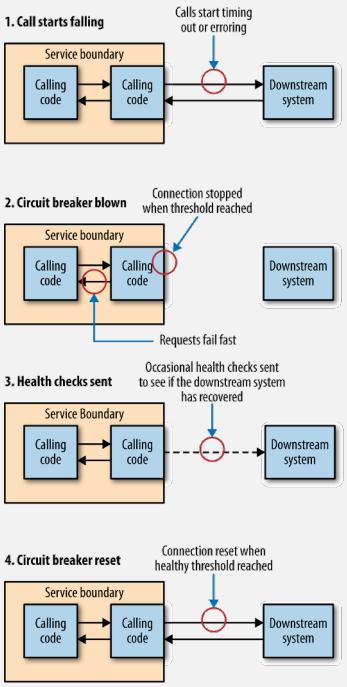
*ist die Firma, die **Java** (1996, V1.0) entwickelt hat und 2009/2010 von **Oracle** übernommen wurde.*

PROF. DR. NANE KRATZKE 66

ARCHITECTURAL SAFETY

Circuit Breakers

- Ein *Circuit Breaker* ist ein Verhaltensmuster in der Softwareentwicklung.
- Es dient dazu, wiederkehrende Verbindungsfehler zu einer Ressource, wie z. B. einer Datenbank oder einem Webservice, zu entdecken und den Zugriff zu der Ressource für eine vorgegebene Zeit zu blockieren.
- Circuit Breaker können ferner dazu verwendet werden, Funktionalität der Anwendung zeitweise zu deaktivieren.
- Das Konzept leitet sich von der elektrischen Sicherung ab und soll insb. Downstream Systems vor Überlastsituationen schützen.



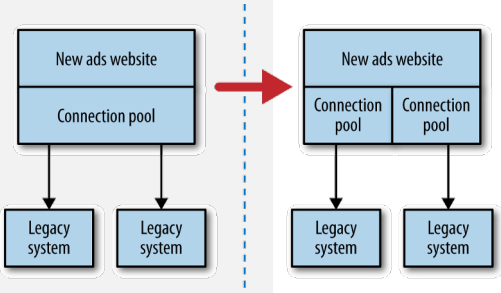
TECHNISCHE HOCHSCHULE LÜBECK

Quelle: Sam Newman, Microservices, O'Reilly 2014

ARCHITECTURAL SAFETY

Bulkheads

- Ein Bulkhead ist ein Stabilitätsmuster in der Softwareentwicklung.
- Der Name leitet sich vom Schott (englisch: bulkhead) eines Schiffes ab.
- Beim Bulkhead wird ein Softwaresystem in mehrere Teilsysteme unterteilt.
- Falls eines der Teilsysteme von einem Service überlastet wird, schlägt dies nicht auf die anderen Teilsysteme durch, so dass diese weiterhin zur Verfügung steht.
- Nebenstehendes Beispiel zeigt, dass man pro Downstream Service jeweils ein Connection Pool verwenden kann, um Connection Pools nicht über mhrere Downstream Services zu „poolen".
- Fällt ein Downstream Service aus, könnte nämlich ein gemeinsamer Connection Pool ggf. von einem fehlerhaften Service gebunden werden und so auch Verbindungen zu eigentlich voll funktionsfähigen Downstream Services verhindern.



TECHNISCHE HOCHSCHULE LÜBECK

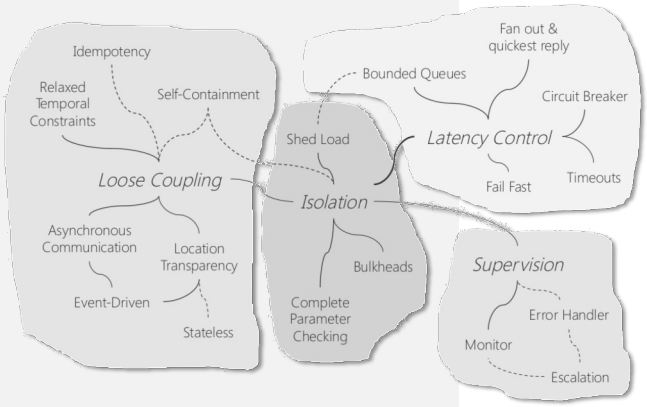
Quelle: Sam Newman, Microservices, O'Reilly 2014

ARCHITECTURAL SAFETY

Isolation

TECHNISCHE HOCHSCHULE LÜBECK

- Je mehr ein Dienst davon abhängt, dass ein anderer aktiv ist, desto stärker wirkt sich die Service Health des einen auf die Fähigkeit des anderen aus, seine Arbeit zu erledigen.
- Unter den Resilience Patterns ist Isolation ein zentrales Leitmotiv.
- Häufig es möglich mittels einer **losen und gepufferten** Integration Services so zu koppeln, dass ein Downstream-Service durchaus offline geschaltet werden kann, ohne das ein Upstream-Service von geplanten oder ungeplanten Ausfällen betroffen wird.



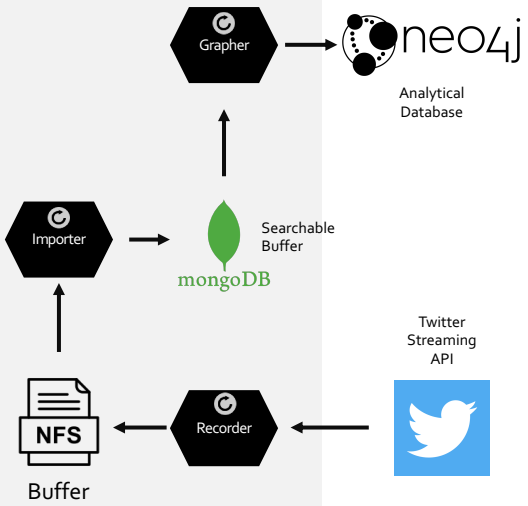
Quelle: Resilience Pattern nach Uwe Friedrichsen
https://www.slideshare.net/ufried/patterns-of-resilience

ARCHITECTURAL SAFETY

Beispiel: Service Isolation via Buffering

TECHNISCHE HOCHSCHULE LÜBECK

- Rechts stehendes Beispiel zeigt, wie Twista (Tool zum Mitschneiden von Large-Scale Twitter-Traffic) einen zusammenhanglosen Stream von Twitter Events in eine Graph-DB (Neo4j) überführt, um graphbasierte Analysen von Twitter-Interaktionen vornehmen zu können.
- Hierzu schneidet ein **Recorder**-Service einfach alle Twitter JSON-Events mit und speichert diese als Dateien auf einem NFS-Server ein. Dieser Server fungiert als Recording Buffer.
- Ein davon unabhängiger **Importer**-Service liest alle n Sekunden alle aufgelaufenen Tweets ein und importiert diese in einen Document Store (MongoDB), um die Tweets dort unter anderem nach Creation Timestamps zu indizieren.
- Ein wiederum davon unabhängiger **Grapher**-Importing Service kann aus diesem Searchable Buffer Tweets eines spezifischen Zeitfenster von Interesse einlesen und für Analysezwecke in eine Graph-DB überführen.
- Sowohl der Recorder-, Importer- als auch der Grapher-Service können dabei ausfallen (z.B. im Rahmen von Aktualisierungen), ohne dass dabei die anderen Dienste in Mitleidenschaft gezogen werden.
- Bei Ausfällen laufen die Buffer zwar voll, werden aber mit Restart der Services dann kontinuierlich wieder abgebaut, so dass sich das Gesamtsystem wieder „fängt".



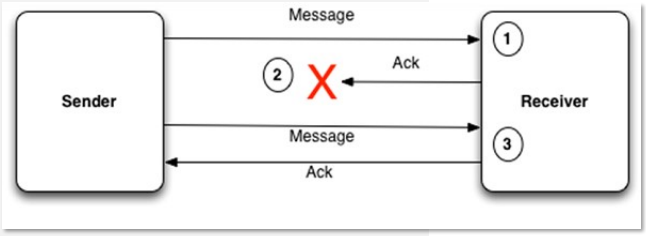
Hinweis: Twista wird u.a. dazu genutzt, einen Large-Scale Datensatz des deutschsprachigen Twitter-Traffics aufzuzeichnen und als Monthly Public Sample Datensatz der Social Network Forschung zur Verfügung zu stellen. https://doi.org/10.5281/zenodo.2783954

ARCHITECTURAL SAFETY

TECHNISCHE HOCHSCHULE LÜBECK

Idempotenz

- Man bezeichnet eine Funktion f als idempotent, wenn für alle x gilt: f(f(x)) = f(x)
- Bspw. ist die Betragsfunktion |x| idempotent, denn ||x|| = |x| für alle x in R.
- Beim REST-Paradigma müssen bspw. die Methoden GET, HEAD, PUT und DELETE laut HTTP-Spezifikation idempotent sein. Das bedeutet, dass das mehrfache Absenden der gleichen Anforderung sich nicht anders auswirkt als ein einzelner Aufruf.
- Idempotente Operationen haben den Vorteil, dass bei einer Wiederholung von Requests/Messages – bspw. nach einem Restart eines Services – Resultate erfolgreich verarbeiteter Operationen unverändert bleiben und Resultate nicht erfolgreicher Operationen wiederhergestellt werden.
- Diese Eigenschaft macht es Recovery-Operationen natürlich deutlich einfacher.
- Eine idempotente Operation stellt also im besten Fall verlorene Daten wieder her und hat im schlimmsten Fall einfach nur keinen Effekt.

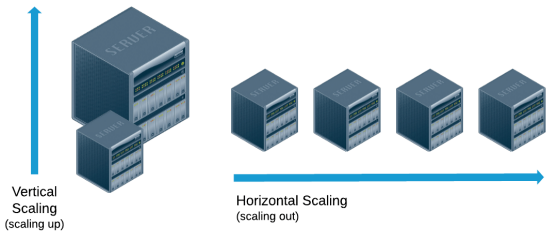


SCALING

TECHNISCHE HOCHSCHULE LÜBECK

Vertical Scaling vs. Horizontal Scaling

- Die Skalierbarkeit einer Anwendung kann an der Anzahl der Requests gemessen werden, die gleichzeitig verarbeitet werden können.
- Der Punkt, an dem eine Anwendung zusätzliche Anforderungen nicht mehr effektiv verarbeiten kann, bezeichnet man als die **Grenze ihrer Skalierbarkeit**.
- Diese Grenze wird erreicht, wenn mindestens eine kritische Hardwareressource aufgebraucht ist und daher mehr Ressourcen für die Verarbeitung benötigt werden (CPU, Memory, Disk, Network, etc.).
- Dies kann grundsätzlich auf zwei Arten erfolgen.
- Unter **Horizontaler Skalierung** versteht man Skalierung durch Hinzufügen weiterer Maschinen zu einem Ressourcenpool (*Scaling out*), während **Vertikale Skalierung** die Skalierung durch Hinzufügen von mehr Leistung (z. B. CPU, RAM) zu einer vorhandenen Maschine bezeichnet (*Scaling up*).



Einer der grundlegenden Unterschiede zwischen den beiden Skalierungsformen besteht darin, dass für die horizontale Skalierung ein sequentielles Stück **Logik so in kleinere Teile zerlegt werden** muss, damit sie **auf mehreren Maschinen parallel** ausgeführt werden kann.

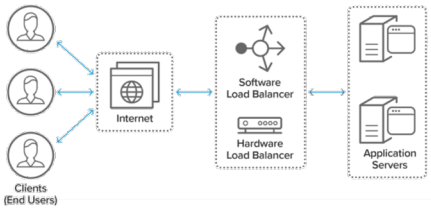
In vielerlei Hinsicht ist die vertikale Skalierung einfacher, da sich die Logik nicht wirklich ändern muss. Man führt lediglich denselben Code auf Rechnern mit höherer Leistung aus. Vertikale Skalierung ist aber limitiert (insb. bei Big Data, Geographische Distribution, Vermeidung von Single Point of 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.
- Ein Load Balancer fungiert als „Verkehrspolizist", der vor Servern sitzt und Requests an mehrere Server verteilt, um deren Geschwindigkeit und Auslastung zu optimieren.
- Fallen Server aus, leitet der Load Balancer den Datenverkehr an die verbleibenden Server um, und kann so einzelne Ausfälle kompensieren.
- Werden neue Server hinzugefügt wird, beginnt der Load Balancer automatisch, eingehende Requests auch an diese zu senden und verteilt so den Traffic.



Aufgaben eines Load Balancers:

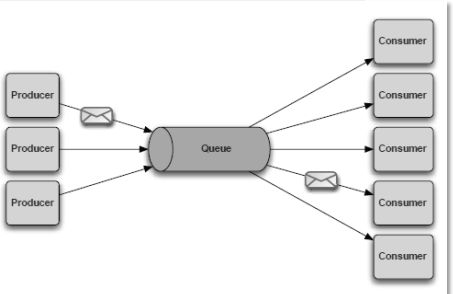
- Effiziente Verteilung von Requests an mehrere Server mit dem Ziel die Last über alle Server auszugleichen (**Lastausgleich**)
- Optimierung der Gesamtverfügbarkeit und Zuverlässigkeit, indem Anforderungen nur an Server gesendet werden, die verfügbar sind.
- Ermöglichen einer **horizontalen Skalierung**, indem Server nach Bedarf hinzugefügt oder entfernt werden können.

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

SCALING

Message Queuing

- Eine Nachrichtenwarteschlange ist eine Art der Kommunikation zwischen Services, die gerne in serverlosen Architekturen und Architekturen mit Microservices verwendet wird.
- Die Nachrichten werden von Erzeugern (Producer) in der Warteschlange gespeichert, bis sie verarbeitet und gelöscht werden.
- Jede Nachricht wird von einem einzelnen Verbraucher (Consumer) immer nur einmal verarbeitet.
- Mithilfe von Nachrichtenwarteschlangen können umfangreiche Verarbeitungsprozesse entkoppelt, Aufgaben gepuffert oder im Stapelbetrieb verarbeitet und hohe Workloads 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.



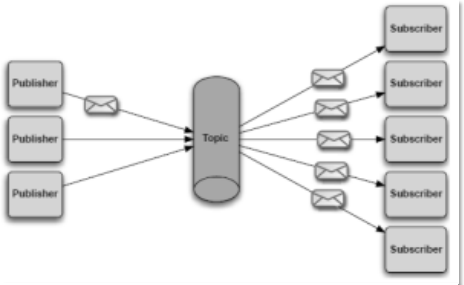
Nachrichtenwarteschlangen ermöglichen unterschiedlichen Teilen eines Systems, asynchron zu kommunizieren und Vorgänge gepuffert zu verarbeiten.
Viele Erzeuger und Verbraucher können die Warteschlange nutzen, aber jede Nachricht wird nur einmal von einem einzelnen Verbraucher verarbeitet. Aus diesem Grund wird dieses Messaging-Muster häufig als 1-zu-1- oder Punkt-zu-Punkt-Kommunikation bezeichnet.

SCALING

TECHNISCHE HOCHSCHULE LÜBECK

Publish-Subscribe

- Publish / Subscribe-Messaging (PubSub) ist eine Form der asynchronen Service-to-Service-Kommunikation, und wird ebenfalls gerne in Serverless oder Microservice-Architekturen eingesetzt.
- In einem Pub / Sub-Modell wird jede zu einem Topic veröffentlichte Nachricht sofort von allen Abonnenten des Topics empfangen.
- Pub / Sub-Messaging kann verwendet werden, um ereignisgesteuerte Architekturen zu ermöglichen und so Dienste zu entkoppeln.
- Auf diese Weise sind Kenntnisbeziehungen zwischen Diensten nicht erforderlich, sondern nur zu einem zentralen PubSub-Service.



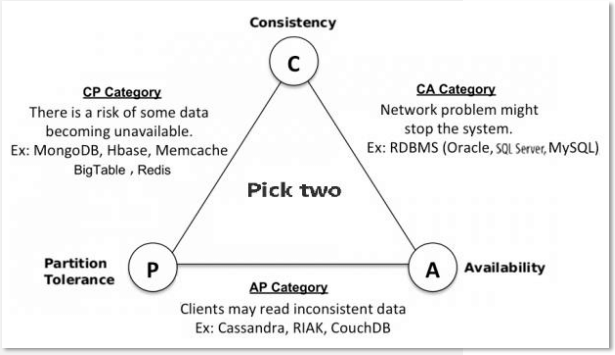
Da alle Subscriber eines Topics dieselben Nachrichten verarbeiten, ist ein Lastausgleich über mehrere Service-Instanzen nur eingeschränkt möglich. Das PubSub-Modell eignet sich vielmehr, um Resultate von Publishern zu serialisieren und an zuständige Subscriber weiterzuleiten. Sollen serialisierten Resultate wieder einem Lastausgleich unterworfen werden, wäre eine Option alle Nachrichten eines Topics von einem Subscriber in eine Queue weiterzuleiten. Queue und PubSub können also durchaus miteinander kombiniert werden.

SCALING

TECHNISCHE HOCHSCHULE LÜBECK

Databases

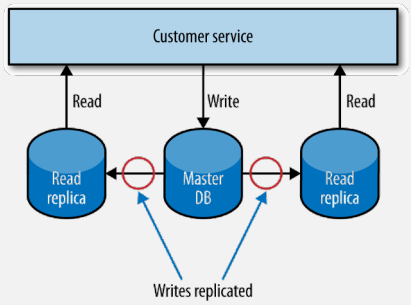
- Das Skalieren Stateless Services ist einfach, da kein Zustand zwischen den Service Instanzen synchronisiert werden muss.
- Jedoch müssen auch Statefull Services wie bspw. Datenbank skaliert werden können.
- Verschiedene Arten von Datenbanken bieten unterschiedliche Formen der Skalierung. Insbesondere NoSQL-Datenbanken haben hier die Möglichkeiten enorm erweitert.
- All diesen Überlegungen liegt das CAP-Theorem zugrunde, d.h. das man von den Eigenschaften Strict Consistency (Konsistent), Availability (Verfügbarkeit) und Partion Tolerance (Partitionstoleranz), nur zwei zweitgleich sicherstellen kann.
- Häufig ist Verfügbarkeit gesetzt und man muss sich dann entweder die strikte Konsistenz (BASE statt ACID) oder die Verfügbarkeit aufgeben.
- Ferner spielt eine Rolle, ob es mehr Read- oder mehr Write-Requests gibt. Meist überwiegen Read-intensive Anwendungsfälle und Writes können besser an Einzelknoten kanalisiert werden.



SCALING

Databases: Scaling for Reads

- Viele Dienste werden überwiegend gelesen.
- In einem relationalen Datenbanksystem (z.B. MySQL, Postgres) können Daten von einem Primärknoten auf ein oder mehrere Replikate kopiert werden.
- Dies geschieht häufig, um sicherzustellen, dass eine Kopie der Daten sicher aufbewahrt wird. Man können dies jedoch auch zum Load Balancing von Lesevorgängen verwenden.
- Alle Schreibvorgänge können hierfür an einen einzelnen Primärknoten gehen, aber Lesevorgänge kann man über mehrere Lesereplikate verteilen.
- Die Replikation von der Primärdatenbank auf die Replikate erfolgt irgendwann nach dem Schreiben. Dies bedeutet, dass bei dieser Technik beim Lesen manchmal veraltete Daten angezeigt werden, bis die Replikation abgeschlossen ist. Dieses Setting wird als eventuell konsistent bezeichnet.
- Wenn solchen vorübergehenden Inkonsistenzen in Anwendungsfällen folgenlos bleiben (und dies ist häufig der Fall), ist dies eine recht einfache und übliche Methode, um Stateful Services zu skalieren.

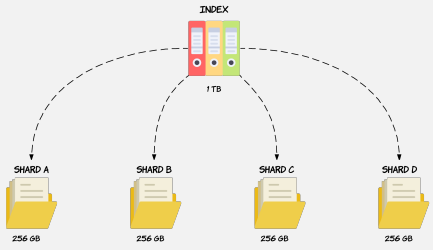


TECHNISCHE HOCHSCHULE LÜBECK

SCALING

Databases: Scaling for Writes (Sharding)

- Sharding ist eine Methode der Datenbankpartitionierung. 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 natürlich, dass ein Zugriff über andere Kriterien als das Aufteilungskriterium unverhältnismäßig aufwändig ist. Abfragen über andere Kriterien oder Joins müssen auf mehrere Server aufgeteilt werden. Das Datenmodell und die Zugriffspfade müssen so designed werden, dass derartige Zugriffe nur selten oder gar nicht vorkommen, sonst werden die Vorteile des Shardings zunichte gemacht.

Diese Notwendigkeit macht Sharding zu einer attraktiven Methode vor allem für NoSQL-Systeme, bei denen typischerweise Joins ohnehin nicht unterstützt werden, und auch Zugriffe über Sekundärschlüssel eher eine Ausnahme sind. Somit lässt sich Sharding für diese Systeme vergleichsweise einfach implementieren. Man kann sogar sagen, dass das relativ problemlose Sharding einer der Hauptgründe für den rasch steigenden Einsatz von NoSQL-Systemen ist.

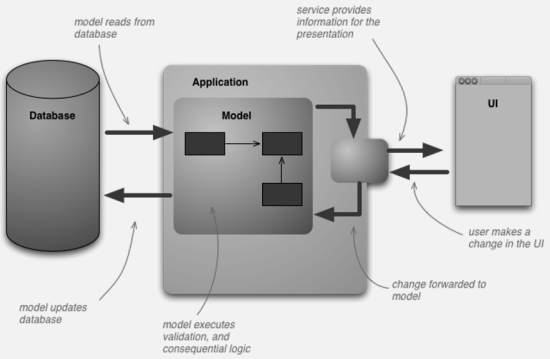
TECHNISCHE HOCHSCHULE LÜBECK

Bei mehreren NoSQL-Systemen wird Sharding und Replikation verknüpft, etwa bei Cassandra und Riak. Dabei wird z.B. in einer ringförmigen Anordnung der Server jeder Shard auch gleichzeitig auf einen oder mehrere nachfolgende Server im Ring repliziert. Somit hält jeder Server jeweils eine Kopie mehrerer Shards. Durch diese Anordnung und Synchronisierungs-mechanismen entstehen sehr flexible und fehlertolerante Systeme, welche auch mit sehr groben Datenmengen befüllt werden können.

SCALING

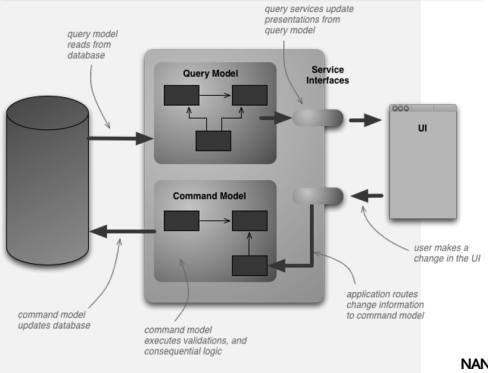
TECHNISCHE HOCHSCHULE LÜBECK

Databases: CQRS (Command-Query Responsibility Segregation pattern)



Der gängige Ansatz für die Interaktion mit Datenbanken besteht darin, diese als CRUD-Datenspeicher zu behandeln. Man hat also nur Datenmodell, in dem man neue Datensätze erstellen, Datensätze lesen, vorhandene Datensätze aktualisieren und Datensätze löschen kann.

Die von CQRS eingeführte Änderung besteht darin, dieses Datenmodell zur Aktualisierung und Anzeige in separate Modelle aufzuteilen, die als Command bzw. Query bezeichnet werden. Für viele komplexe Probleme, eine Separierung in Modelle für Command und Query dazu führen, das man Commands und Query unabhängig von einander skalieren kann.



Achtung: *Dieses Pattern gilt auch für erfahrene Software-Architekten als schwer zu beherrschen. Wenn es keine substantiell unterschiedlichen Skalierungs-erfordernisse bei Queries und Commands an der Schnittstelle zur Datenbank gibt, bleiben Sie besser beim klassischen CRUD und einem integrierten Datenmodell!*

Quelle: https://martinfowler.com/bliki/CQRS.html

CACHING

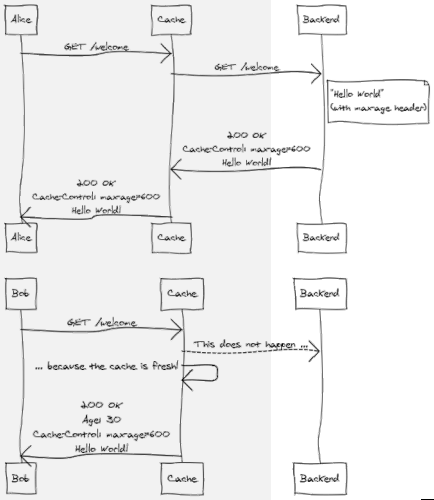
TECHNISCHE HOCHSCHULE LÜBECK

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).

There are only two hard things in Computer Science: cache invalidation and naming things.

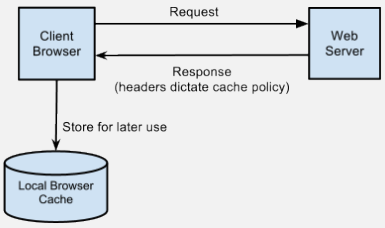
-- Phil Karlton (Developer of the X Window System)



CACHING

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 Downstream-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 Downstream-Services zu verringern.

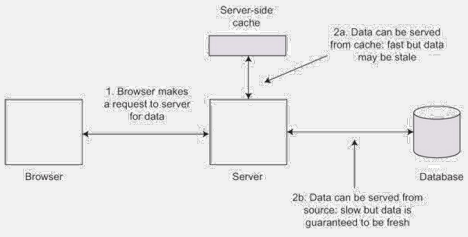


TECHNISCHE HOCHSCHULE LÜBECK

CACHING

Server-side caching

- Beim serverseitigen Caching übernimmt der Server die Caching-Verantwortung und verwendet möglicherweise ein System wie Redis oder Memcache oder sogar einen einfachen In-Memory-Cache.
- Beim serverseitigen Caching ist für Client das Caching transparent.
- Mit einem Cache in der Nähe oder innerhalb einer Service-Grenze kann es einfacher sein, über Dinge wie eine Ungültigmachung von Daten nachzudenken 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.

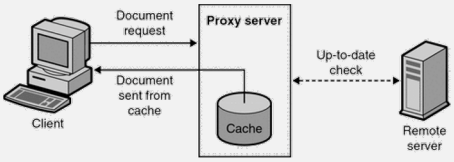


TECHNISCHE HOCHSCHULE LÜBECK

CACHING

Proxy caching

- Beim Proxy-Caching wird ein Proxy zwischen dem Client und dem Server platziert.
- Ein gutes Beispiel hierfür ist die Verwendung eines Reverse-Proxy- oder Content-Delivery-Netzwerks (CDN).
- Beim Proxy-Caching ist das Caching sowohl für Client als auch für den Server transparent.
- Proxy Caching 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 zwischenspeichern (z.B. mittels Reverse-Proxies wie Squid).
- 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.



TECHNISCHE HOCHSCHULE LÜBECK

INHALTE

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

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

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

Skalierung von Microservices
- Failure is Everywhere
- 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

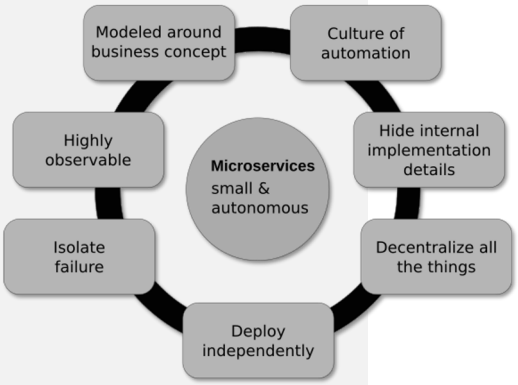
TECHNISCHE HOCHSCHULE LÜBECK

7 PRINZIPIEN VON MICROSERVICES

Zusammenfassung dieser (und vorhergehender) Units

- Bilde Modelle um Geschäftskonzepte
- Erschaffe eine Kultur der Automatisierung
- Blende interne Implementierungsdetails aus
- Dezentralisiere
- Definiere unabhängig aktualisierbare Einheiten
- Isoliere Fehler
- Baue gut beobachtbare Services

- Wann man keine Microservices nutzen sollte



Quelle: Sam Newman, Building Microservices, O'Reilly, 2014

7 PRINZIPIEN VON MICROSERVICES

Bilde Modelle um Geschäftskonzepte

- Die Erfahrung hat gezeigt, dass Schnittstellen, die aus geschäftsgebundenen Kontexten (d.h. Domain-driven) abgeleitet werden, stabiler sind als solche, die um technische Konzepte herum gebaut werden.
- Durch die Modellierung der System-Domäne bildet man nicht nur stabilere Schnittstellen, sondern stellt auch sicher, dass Änderungen in Geschäftsprozessen besser abbildbar sind.
- Die Methodik der Bounded Contexts (DDD) bietet sich an, um potenzielle Domänengrenzen zu definieren.

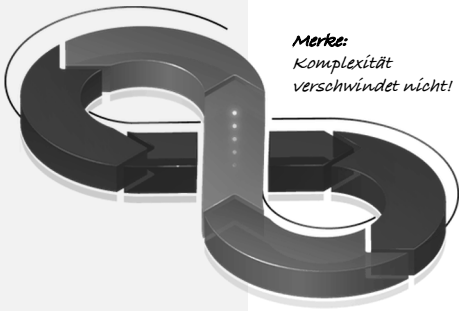
Die Methodik des Domain-Driven-Designs werden wir in Unit 09 behandeln.

Quelle: Sam Newman, Building Microservices, O'Reilly, 2014

7 PRINZIPIEN VON MICROSERVICES

Erschaffe eine Kultur der Automatisierung

- Einzelne Microservices sind einfach. Eine Architektur von Microservices ist im Allgemeinen komplex. Ein wesentlicher Grund ist die im Resultat steigende Anzahl von Wechselwirkungen eines Gesamtsystems.
- Die Einführung einer Kultur der Automatisierung ist ein wichtiger Weg, um diese Komplexitätsverschiebung beherrschen zu können.
- Automatisierte Tests stellen sicher, dass Services auch nach Updates in komplexeren Wechselwirkungsketten stets wie erwartet funktionieren.
- Automatisierte Deployments ermöglichen schnelles Feedback zur Produktionsqualität jedes Check-ins.
- Die Definition von ***Enviroments*** (z.B. Test, Staging, Development) ermöglichen unterschiedliche Ausbaustufen eines Systems mit einer einheitlichen Bereitstellungsmethode bereitstellen und testen (bzw. in Produktion bringen) zu können.
- ***Container Images*** können die Bereitstellung zu beschleunigen, und vollautomatisch unveränderliche Server zu erstellen, um die Erstellung von Deployment Units, deren Konfiguration und deren Betrieb zu standardisieren und mittels ***Orchestrierungsplattformen*** zu automatisieren.



Merke: Komplexität verschwindet nicht!

Deploy Environments können Sie bspw. auch in Gitlab-CI Pipelines definieren (vgl. Unit 02, DevOps).

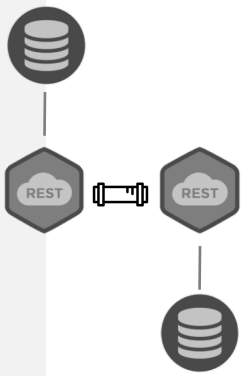
Container (Images) haben wir in Unit 04+05 behandelt.

Quelle: Sam Newman, Building Microservices, O'Reilly, 2014

7 PRINZIPIEN VON MICROSERVICES

Blende interne Implementierungsdetails aus

- Damit ein Service unabhängig von anderen Diensten entwickelt werden kann, ist es wichtig, dass dessen Implementierungsdetails verborgen werden, um implizite Abhängigkeiten zu vermeiden.
- ***Bounded Contexts*** können hilfreich sein, Modelle zu identifizieren, die gemeinsam an Schnittstellen genutzt werden sollen, und die Modelle zu identifizieren, die ausgeblendet werden können.
- Services sollten auch ihre Datenbanken verbergen, um zu vermeiden, dass eine der häufigsten Arten der Kopplung (Datenkopplung) vermieden werden.
- Stattdessen sollten Konzepte wie Data Pumps oder Event Data Pumps genutzt verwenden, um über mehrere Services hinweg zu konsolidieren.
- Technologieunabhängige APIs ermöglichen Flexibilität hinsichtlich des Einsatzes verschiedener Technologie-Stacks.
- Insbesondere mittels ***REST*** kann die Trennung von internen und externen Implementierungsdetails methodisch sichergestellt werden.
- Selbst wenn mittels Remote Procedure Calls (RPCs) kommuniziert wird, kann die REST-Philosophie genutzt werden.



Bounded Contexts und Domain Driven Design werden wir in Unit 09 behandeln.

Quelle: Sam Newman, Building Microservices, O'Reilly, 2014

7 PRINZIPIEN VON MICROSERVICES

Dezentralisiere

- Um die erforderliche Autonomie der Mikroserviceteams zu maximieren, sind Self-Service Lösungen zu nutzen, um Software bei Bedarf bereitzustellen und die Entwicklung und das Testen für Teams so reibungsarm wie möglich zu halten.
- Teams sollten Servicebesitzer sein.
- Die Teams sollten für die vorgenommenen Änderungen verantwortlich sein.
- Teams sollten an der Organisation ausgerichtet werden, um sicherzustellen, dass das ***Gesetz von Conway*** funktioniert.
- Wenn übergreifende Richtlinien (z.B. im Rahmen von Architekturen) erforderlich sind, sollten diese in einem Shared-Governance-Modell entwickelt werden.
- Da Ansätze wie Enterprise Service Bus oder Orchestrierungssysteme leicht zur Zentralisierung von Geschäftslogik und dummen Diensten führen können, sollten diese vermieden werden.
- Vielmehr sollten ***Prefer-Choreography-over-Orchestration*** und ***Dumb-Middleware-with-Smart-Endpoints*** Ansätze genutzt werden, um die zugehörige Logik und Daten innerhalb von Service Grenzen zu halten, um eine hohe Service-Kohäsion zu gewährleisten.

Quelle: Sam Newman, Building Microservices, O'Reilly, 2014

TECHNISCHE HOCHSCHULE LÜBECK

7 PRINZIPIEN VON MICROSERVICES

Definiere unabhängig aktualisierbare Einheiten

- Bei Breaking-Changes sollten versionierte Endpunkte nebeneinander verwendet werden, damit konsumierende Services erst im Laufe der Zeit angepasst werden können.
- Dies ermöglicht es die Geschwindigkeit und die Veröffentlichung neuer Funktionen zu optimieren.
- Bei Verwendung der RPC-basierten Integration sollte eine eng gebundene Client/Server-Stub-Generierung vermieden werden (wie sie bspw. Java RMI fördert).
- Die Verwendung eines ***One-Service-per-Host/Pod/Container-Modells*** reduziert Auswirkungen, die die Aktualisierung eines Dienstes auf einen anderen nicht verwandten Dienst haben könnte.
- Erwägen Sie die Verwendung von ***Blue/Green*** oder ***Canary-Release-Techniken***, um Deployments von Releases zu trennen.
- Verwenden Sie Consumer-Driven Kontrakte, um wichtige Änderungen zu erkennen, bevor sie eintreten.
- Es sollte die Norm sein, Änderungen an einem einzelnen Service vorzunehmen und ihn für die Produktion freizugeben, ohne dass andere Services hierfür gesperrt werden werden müssen.
- Konsumierende Services sollten entscheiden, wann sie sich selbst aktualisieren, nicht die bereitstellenden Services.

Quelle: Sam Newman, Building Microservices, O'Reilly, 2014

TECHNISCHE HOCHSCHULE LÜBECK

Release Pattern hatten wir bereits in Unit 02 (DevOps) kennen gelernt und werden auch noch mal in Unit 08 (Service Meshes behandelt werden).

7 PRINZIPIEN VON MICROSERVICES

Isoliere Fehler

- Eine Microservice-Architektur kann widerstandsfähiger sein als ein monolithisches System, aber nur, wenn Fehler als Teil des System-Designs von Grund auf berücksichtigt werden.
- Behandeln Sie bei Verwendung von RPC Remote Calls nicht wie Local Calls, da dadurch andere Arten von Fehlermodi (u.a. Netzwerkfehler) ausgeblendet werden.
- Timeouts sollten entsprechend eingestellt werden.
- Verwenden Sie ***Circuit Breaker***, um die Auswirkungen einer fehlerhaften Komponente zu begrenzen.
- Verstehen Sie, welche Auswirkungen sich für den Nutzer ergeben, wenn sich nur ein Teil des Systems fehlerhaft verhält.
- Berücksichtigen Sie, welche Auswirkungen eine Netzwerkpartition haben kann. Stellen Sie sich die Frage, ob Sie in dieser Situation die **Verfügbarkeit** oder **Konsistenz** opfern können.

TECHNISCHE HOCHSCHULE LÜBECK

Circuit Breaker werden noch detaillierter in Unit 08 (Service Meshes) behandelt.

Quelle: Sam Newman, Building Microservices, O'Reilly, 2014

7 PRINZIPIEN VON MICROSERVICES

Baue gut beobachtbare Services

- Bei Microservices reicht es nicht, das Verhalten einer einzelnen Service-Instanz oder den Status einer einzelnen Maschine zu beobachten, um festzustellen, ob das System als Ganzes ordnungsgemäß funktioniert.
- Stattdessen ist eine integrierte Sicht auf das System erforderlich.
- Mittels Semantic Monitoring lässt sich feststellen, ob sich ein System wunschgemäß verhält, indem man synthetische Transaktionen in Ihr System einspielt, um so das Verhalten realer Benutzer zu simulieren.
- Protokolle und Statistiken sollten über alle Service Instanzen und Ressourcen aggregiert werden, damit sich bei Auftreten eines Problems einen Drilldown zur Quelle durchführen lässt.
- Um zu inspezieren, wie die einzelnen Systemteile miteinander in der Produktion interagieren, verwenden Sie Korrelations-Ids, damit Sie Anrufe über das System mittels ***Tracing*** verfolgen können.

TECHNISCHE HOCHSCHULE LÜBECK

Tracing und Observability werden wir in Unit 08 behandeln.

Quelle: Sam Newman, Building Microservices, O'Reilly, 2014

WANN SOLLTEN MICROSERVICES NICHT VERWENDET WERDEN?

Microservices are not a free lunch

Irgendwie hat es (vermutlich) Netflix geschafft Microservices für die Fachwelt zu "romantisieren". Angeblich sollen über 700 Microservices für die Plattform ausgeführt werden. Aufgrund des Erfolgs von Netflix waren folglich viele Unternehmen bereit diesen Architekturansatz zu adaptieren.

Was häufig übersehen wurde und wird, ist allerdings die Tatsache, dass Microservice-Erfolgsgeschichten hauptsächlich von ***produktbasierten Unternehmen*** stammen, die aufgrund dieses Fokus Unmengen an ***Iterationen*** und Modellen durchlaufen können, bevor ein erfolgreiches und marktreifes Produkt entstanden ist.

Microservice Architekturen ermöglichen somit solche agilen Entwicklungen erfolgreicher Produkte. Aber Microservices sind kein Garant für Erfolg.

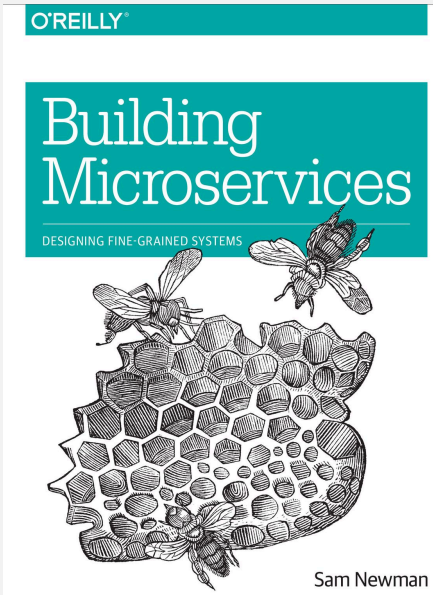
Insbesondere in der Cloud-basierten Geschäftswelt sind erfolgreiche Produkte oft Microservice-basiert. Aber nicht alle Microservice-basierten Produkte sind auch erfolgreich.

- Microservices sind Lösungen für komplexe Probleme. *Ohne komplexes Problem, gibt es keinen wirklichen Grund für Microservices.*
- Die Teamgröße ist für erfolgreiche Microservice-Ansätze entscheidend. Wenn Sie nicht genügend (große) Teams haben, vermeiden Sie besser Microservices. *(Netflix braucht gem. dem One-Service-Per-Team Ansatz somit etwa 700 Teams!)*
- Wenn Ihre Anwendung nicht in Microservices unterteilt werden muss, *machen Sie es auch nicht.*
- Wenn Sie nicht skalieren müssen, bringen Microservices häufig wenig Mehrwert. *Die THL hat bspw. etwa 5.000 Studierende, das werden nicht über Nacht plötzlich mehr.*

TECHNISCHE HOCHSCHULE LÜBECK

NETFLIX

ZUM NACHLESEN



1. Microservices
2. The Evolutionary Architect
3. How to Model Services
4. Integration

5. *Splitting the Monolith*
6. *Deployment*
7. *Testing*
8. *Monitoring*
9. *Security*

10. Conway's Law and System Design
11. Microservices at Scale
12. Bringing It All Together

TECHNISCHE HOCHSCHULE LÜBECK

ZUM NACHLESEN

Paper

- Baldini et al. **Serverless Computing: Current Trends and Open Problems**. In *Research Advances in Cloud Computing*; Springer, 2017
  https://arxiv.org/pdf/1706.03178.pdf
- Nane Kratzke: **A Brief History of Cloud Application Architectures**. In *Applied Sciences*; MDPI, 2018
  https://doi.org/10.3390/app8081368
- Jonas et al., **A Berkley View on Serverless Computing**, 2019 (Technical Report, UC Berkley)
  https://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.pdf
- Roy Fielding, **Architectural Styles and the Design of Network-based Software Architectures**, 2000 (Dissertation, PhD, UC Irvine)
  https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

TECHNISCHE HOCHSCHULE LÜBECK

KONTAKT

Disclaimer

Nane Kratzke
+49 451 300-5549
nane.kratzke@th-luebeck.de
kratzke.mylab.th-luebeck.de