Applikationsentwicklung

Zusammenfassung

Stephan Stofer

4. Juni 2021

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Software Architektur

Eine Architektur ist eine **Abstraktion**, es wird etwas zusammenfassend und vereinfachend dargestellt. Vergleich mit Modellierung und Design, es ist ein fliessender Übergang. Sie beschreibt ein **System** durch dessen Strutur und Aufbau (Sub-/ Teilsysteme, Schichtung, Zwiebel, Verteilung) und der darin enthaltenen Softwareteile (Komponenten) und deren Beziehungen. Abhängigkeiten, Schnittstellen, Daten- und Kontrollflüsse, usw.

1.1 Definition von Architektur

> Softwarearchitektur definiert sich durch die Kernelemente eines Systems, welche als Basis für alle weiteren Teile nur schwer und aufwändig verändert werden können. Martin Fowler
>
> Die Architektur repräsentiert die signifikanten Designentscheidungen die ein System festhalten, wobei die Signifikanz an den Kosten von Änderungen bemessen wird. Grady Booch

1.1.1 Welche Architektur besprechen wir

Wir konzentrieren uns hauptsächlich auf die **Softwarearchitektur** und streifen die Betriebsarchitektur.

1.1.1.1 Softwarearchitektur

Besteht unteranderem aus:

- Applikationsarchitektur
- Systemarchitektur
- Lösungsarchitektur

1.1.2 Aspekte und Herausforderungen der Softwarearchitektur

Grobe Übersicht unter Einhaltung angemessener Qualitätsaspekte

- Grundlende Struktur
  - Schichten, Client/Server, n-tiers, Services usw.
  - Art der Applikation und Deployment
  - Architekturmuster und Prinzipien
- Kommunikation und Verarbeitung
  - Verteilbarkeit, Parallelität, Performance, Robustheit, Resilienz (Robustheit bei Störungen oder Teilausfällen von verteilten Systeme)
  - Kommunikationsmuster (synchron/async)
  - Transaktionalität
- Eingesetzte Technologie

  - Userinterface (Fat-, Richt, Thin)
  - Persistenz der Daten
  - Referenzarchitekturen

1.2 Arten von Applikationen

Es gibt eine grosse Bandbreite von Applikationen mit sehr unterschiedlichen Anforderungen

- Einzelbenutzerapplikatione
  - eher klein, sogar nur eine Mobile-App
  - lokale Persistenz
- Mehrbenutzer
  - Enterprise Software (Unternehmen)
  - Zentrale Services und Daten, beliebige Komplexität
  - Typisch in mehre (Teil-)Systeme aufgebrochen
- Internetanwendung (Public)
  - Typisch mit Web-GUI
  - beliebig viele Benutzer
  - verteilte Datenspeicherung, beliebige Komplexität

1.2.1 Architekturstile in der Zeit

Mit der Zeit entwickelten sich sehr viele Architekturstile. Je nach Applikationsart erwiesen sich diese als schlecht, aber auch als brauchbar. Erkenntnisse daraus, die Kunst ist die richtige Architektur am richtigen Ort.

1.2.2 Fliessender Wechsel zw. Design und Architektur

Ist die grosse Herausforderung Muster, Prinzipien und Techniken auf den unterschiedlichen Abstraktionsebenen angemessen zu befolgen und zu beurteilen. Schlussendlich haben wir viele Klassen und Schnittstellen - jede weitere Strukturierung ergbit sich weitgehend von der Organisation und Konvention.

1.3 Systeme und Komponenten

Softwaresysteme werden in einzelne Subsysteme zerlegt. Die Zerlegung führt zu mehrstufigen hierarchischen Subsysteme. Diese haben gewisse Unabhängigkeit und interagieren über **wohldefinierte Schnittstellen**. Das Ziel ist einfachere, verständlichere und schnellere Architektur.

- lose Kopplung - via möglichst lose Schnittstellen
- hohe Kohäsion - mit Datenkapselung und Information Hidding

Vorteile dieser hierarchischen Strukturierung sind:

- Präzise Schätz- und Planbarkeit
- Unabhängige Entwicklung möglich
- Einfachere Testbarkeit
- Unabhängiges Deployment
- Potential für Wiederverwendung höher

1.3.1 Komponenten und (Sub-)Systeme

Komponenten sind softwaretechnische Einheiten welche durch den Einsatz ein System realisiert werden kann. Ein System muss aber nicht zwingend eine Komponente enthalten. Sie sind damit orthogonal

```
System --> Komponente
```

1.4 Monolithische Architektur

Zu unrecht in Verruf gekommen. Man muss unterscheiden zwischen *monolithisches Design* (ohne Modularisierung) oder *monolithisches Deployment*.

1.4.1 Monolithisches Design

Keine gute Wahl, Codebasis weist schlechte Struktur und Design auf. Grosse Datenkapselung, Kohäsion und Kopplung sind die Folge. Schwierig Wart- und Erweiterbar. Vorsicht vor Coderosion.

1.4.2 Monolithisches Deployment

Modularisierte innere Struktur wird vorausgesetzt. Dadurch Freiheitsgrade im Deployment wie gesamte App kann als Packet verteilt werden. Dadurch aber auch Nachteile wie möglich

- Grösse des Deployment
- neuer Deploy auch bei kleinen Änderungen nötig
- eher häufigere und vollständige Nicht-Verfügbarkeit
- parallelisierte und entkoppelte Entwicklungsprozess wird durch Nadelöhr Deployment wieder synchronisiert

1.5 Verteilte Applikation

Einzelne Teile einer verteile Applikationen sind auf mehrere Host verteilt und laufen in einzelnen unabhängigen und parallelen Prozessen. Dadurch ergeben sich neue Anforderungen:

- Teile müssen Kommunizieren können
- Teile müssen sich finden und kennen
- Deployment wird aufwändiger (mehr und häufiger). Abhängigkeiten müssen beachtet und koordiniert werden
- mit Teilausfälle muss umgegangen werden können

1.6 Modularisierung

Fundamental für SW-Architektur. Identifiziert sinnvolle und funktional zusammengehörige «Kleinteile» einer Software. Sind in sich abgeschlossen und dadurch austauschbar. Kommunikation über Schnittstellen.

1.6.1 Einfluss auf Architektur

Die Modularisierung hat Einfluss auf *Gruppierung* (Klassen mit gleiche Eigenschaften (z.B. Export)) , *Hierarchie* oder *Geschichtet* - Module in logischer Kette.

1.6.2 Zentrale Eigenschaften

Besitzt explitzite Schnittstelle, starke Kohäsion / SoC und SRP, ergibt Information Hiding und lose Kopplung

1.6.3 Kriterien für Entwurf eines Modules

- Zerlegbarkeit / Dekomposition: Module voneinander möglichst unabhängig
- Kombinierbarkeit: Module sind flexibel kombinierbar
- Verständlichkeit: Module sind einzeln und mit ohne/wenig Kontext verständlich
- Stetigkeit / Kontinuität: Haben gewisse Beständigkeit

1.6.3.1 Prinzipien und Regeln für Modulkohäsion

Folgende Prinzipien arbeiten gegeneinander, es ist ein Trade-Off.

REP - Reuse Release Equivalence Prinzip

Granularität Der Wiederverwendung ist die Grannnularität des Release. Man fügt zusammen, was gemeinsam veröffentlicht wird. Mehr best-practice, als Prinzip. REP ist inkludierend (macht Dinge grösser).

Fasst zusammen, sodass Wiederverwendung einfacher wird.

CCP - Common Closure Prinzip

Klassen die man aus *denselben* Gründen und Zeit modifiziert fasst man in die gleiche Komponente zusammen. Klassen die aus unterschiedlichen Gründen anpasst, separiert man. Entspricht SRP und OCP auf Architekturebene. CCP ist inkludierend.

Fasst zusammen, was gemeinsam einfacher Warbar ist.

CRP - Common Reuse Prinzip

Zwinge Nutzer einer Komponente nicht in eine Abhängigkeit von Elementen, die er nicht benötigt. CRP ist exkludierend.

Teilt auf, damit weniger Abhängigkeiten vorhanden sind.

1.6.3.2 Prinzipien und Regeln für Modulkopplung

ADP - Acyclic Dependencies Prinzip

Zwischen Komponenten sind keine Zyklen erlaubt.

SDP - Stable Dependencies Prinzip

Die Abhängigkeiten zwischen Komponenten sollten in Richtung der stabileren Komponenten verlaufen.

SAP - Stable Abstractions Prinzip

Stabile Komponenten sollten im gleichen Masse auch abstrakt sein.

2 Anforderungsdiagramme und Modelle

Diagramme sind klarer Verständlich als Worte, weshalb sie häufig eingesetzt werden. Es gibt zahlreiche unterschiedliche Arten von Diagramme. Die Methoden/Techniken unterscheiden sich, um Anforderungen zu erheben, festzuhalten und zu analysieren.

2.1 Domain Modell

Soll die Fachdomäne/Anwendungsdomäne. Enthält Daten und Entitäten und deren Beziehungen. Das Naming sollte aussagekräftig und eindeutig sein. Ein Domain-Model ist ein konzeptionelles Modell der Anwendungsdomäne und beinhaltet sowohl Verhalten als auch Daten. Es wird typischerweise als UML-Diagramm erstellt.

2.1.1 Ist-/Hat Beziehung

Komposition ist eine Ist-Beziehung -> Ein Raum ist in einem Gebäude. Ein Gebäude besteht aus 1 bis n Räumen.

Eine Aggregation ist eine Hat-Beziehung -> Ein Student hat Vorlesungen. Eine Vorlesung enthält 3 bis n Studenten. Die Komposistion wird mit einem ausgefüllten schwarzen Diamond gekennzeichnet. Die Aggregation mit einem weissen. Ein weisser Pfeil heisst, eine Entität *hat* die Entität worauf gezeigt wird.

2.2 Kontext Diagramm

In einem Kontext Diagramm sehen wir das System mit seinen Grenzen mit allen Akteuren und Nachrichten (Datenfluss) zwischen Akteur und System. Das Diagramm dient der Systemabgrenzungen.

2.3 Use Case / Anwendungsfall Diagramm

Beschreibt einen Anwendungsfall, welcher durch unterschiedliche Szenarien eintreten können. Darin werden alle Use Cases und die Verbindungen zu den Akteuren aufgezeigt. Die Pfeile sollten gerichtet werden um zu erkennen, wer den Use Case anstösst.

2.4 Geschäftsprozess Modell

Die Struktur bzw. der Prozess wird modelliert, was passiert wann genau. In welcher Struktur und Reihenfolge. Entspricht BPNM.

2.5 Feature Liste

Zeigt auf welche Features es überhaupt gibt und welche Kundengruppen diese verwenden können. Kann auch über einen Status Auskunft geben (Fertig, in Arbeit, geplant).

3 Methodik / Entwurf und Bau eines Microservice

Wie identifiziere oder entwerfe ich Microservices in einem komplexen System.

3.1 Strukturierung nach technischen Aspekten

Eine Möglichekiten nach Layer/Schichten zu gliedern. Vorteile keine Zyklen bei der Anpassungen eines Layers.

3.2 Strukturierung nach fachlichen Aspekten

Ziel der strukturierung von Microservices. Wird oft von Business (Fach) definiert und grenzt sich technisch ab. Die Features werden anhand Silos gegliedert. Zum Beispiel ein Feature Warenkorb.

3.2.1 Merkmale einer Microservice-Architektur

Es ist **kein** konsistentes Datenmodell über die ganze Applikation notwendig (Eventual Consistency - braucht Zeit bis Persistenz überall sichergestellt ist). Jeder MS hat eigene fachlichen Begrifflichkeiten, Datenattribute udn Datentypen. Konzepte wie *Bounded-Context*, welches aus DDD statmmt. Allfällige Kommunikation zwischen MS müssen Daten-Anpassungen gemacht werden. Die einzelnen MS kümmern sich selber über ihre Persistierung. Beim richtigen Vorgehen sind MS weitgehend undabhängig voneinander, fachliche Erweiterung und Wartung werden einfacher. Fachlicher Aufwand kann minimiert werden um Detail einzelner Teil-Domäne abzugleichen.

3.2.2 Nachteile einer Mircoservice-Architektur

Grosse technischer Overhead bezüglich technischer Komplexität. Kommunikation generel aufwändig, muss stabil sein. Sind die MS zu *klein* steigt Kommunikationsbedarf zwischen MS. Dies führt zu mehr Kopplung, Mehraufwand und kann Performance mindern. Führt zu *Accidental Complexity*.

3.3 Wie indetifiziere ich ein Microservice

Ideal sind Kontextdiagramme. Man erkennt die Abgrenzung zum System (Grenzen). Es enthält auch alle Akteure und alle Nachrichten zwischen Akteuren und System. Möchlich ist auch ein *Domain-Modell* und *Bounded-Context*. Sie beinhaltet alle Geschäftsobjekte (Business Entities) und ihre Beziehungen (Daten und Aktivitäten) unserer Applikation (es fokussiert also auf Daten-Objekte).

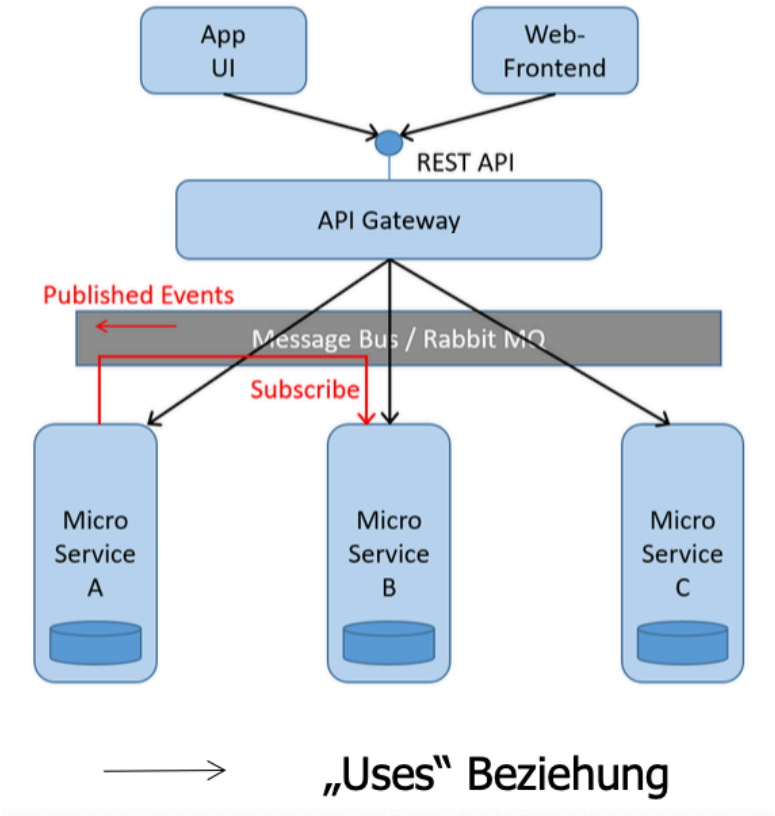
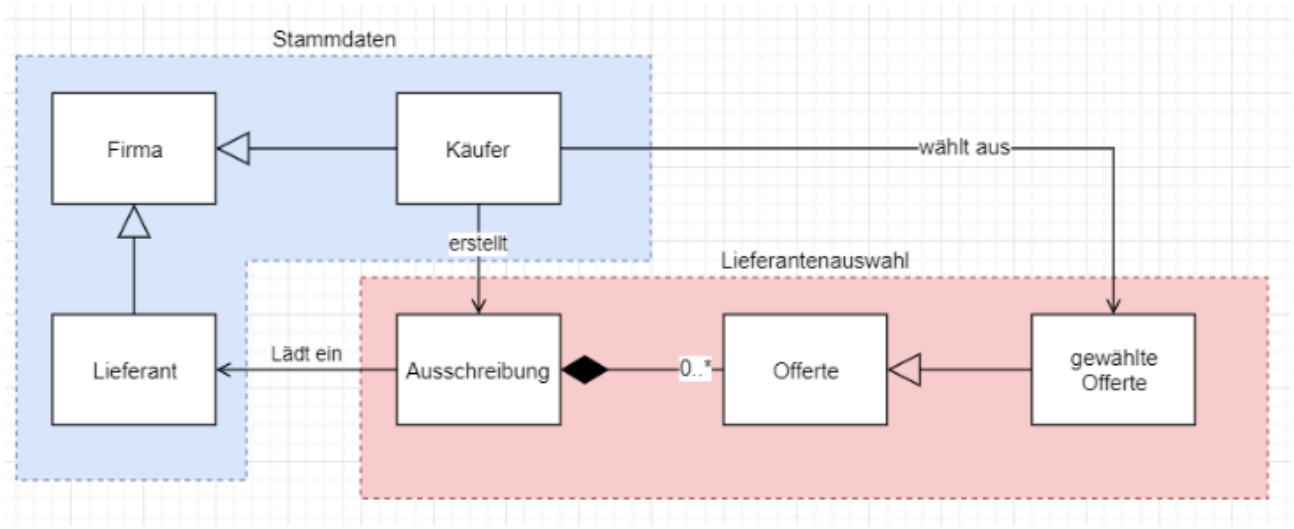


Abbildung 3.1: Ein pragmatisches Microservice Modell



{width=60%}

3.4 Bounded-Context

Angelehnt an die Methode DDD ist ein abgegrenzter Bereich des Domain-Models, welcher Gültigkeit für ein bestimmtes **fachliches** Modell hat. Die Domänen bestehen aus mehreren Bounded-Contexts.

3.5 Prozessanalyse

Erst werden alle wesentlichen Prozesse identifiziert und diese in ihre Teilschritte zerlegt. Neben Prozessschritten werden auch Akteure aufgeführt. Es bewusst einfach gehalten. Es wird meist nur der Hauptpfad und keine alternativen Pfade erfasst, sonst würde sich BPMN-Notation aufdrängen.

3.6 Event Storming

Ist eine Workshop-basierte Methode um schnell herauszufinden, was in der Domäne eines SW-Programms passiert. Grundidee ist, SW-Entwickler und Domänenexperten zusammenzubringen und voneinander zu lernen. Event Storming soll Spass machen.

1. Es wird eruiert, welche Nachrichten oder welche Events auftreten
2. Welcher Befehl führt zum Event
3. Welche Akteure sind daran beteiligt
4. Aggregate aus den Events

Die Elemente werden auf farbige Sticky Notes aufgeschrieben und auf Whiteboard oder Wand geklebt. Jede Farbe hat eigene Bedeutung.

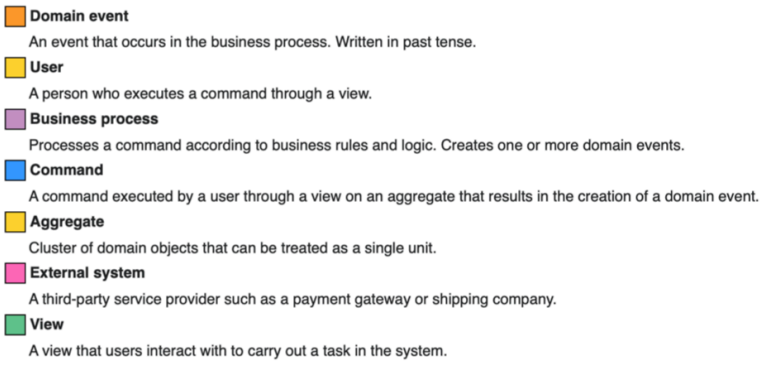


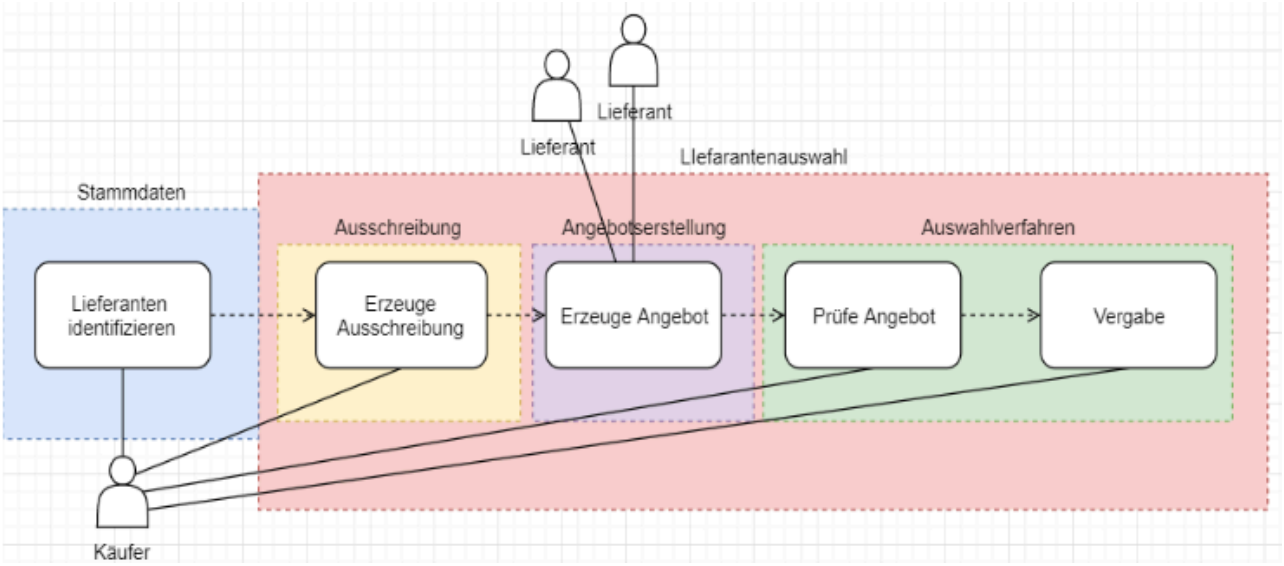
Abbildung 3.2: Farbenlegende zu Event Storming

Das Ergebnis aus dem Event Storming ist eine farbige Wand :)

3.7 Prozessanalyse

Dank der Prozessanalyse erkennt man zeitliche Zusammenhänge der einzenlen Aktivitäten und Beziehungen. Ein Ablauf einer Abfolge wird dokumentiert. Bei der Analyse der Prozessschritte wird klar, dass sich eine Aufteilung in *Bounded-Contexts* aufdrängt. Unterscheiden sich die Zugriffe? oder die Akteure der einzelnen Aktivitäten.

Abbildung 3.3: Ergebnis des Event Stormings



{width=70%}

Man erkennt auf der Abbildung **??** mögliche *Bounded-Contexts* wie Stammdaten, Ausschreibung, Angebotserstellung und Auswahlverfahren.

3.8 Diagramm der MS Architektur in APPE

Wir versuchen mit dem Kontext-Diagramm ein Syste mit Inputs und Outputs zu beschreiben, ohne uns um die Innereien des Systems zu kümmern. Wir nennen dies **Black-Box-View**.

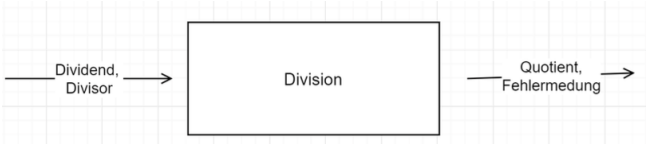


Abbildung 3.4: Black-Box-View

Daraus kann man die **Komponenten** oder Teilssysteme abbilden.

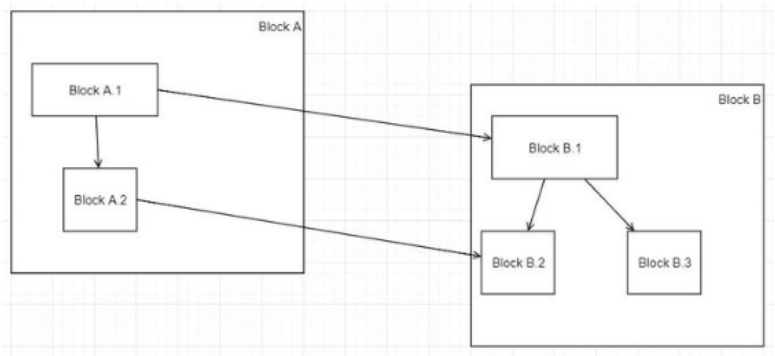


Abbildung 3.5: Komponenten in APPE

3.8.1 Beispiel einer Microservice Architektur

Nachfolgend eine Beispiel eine Modells einer Microservice Architektur.

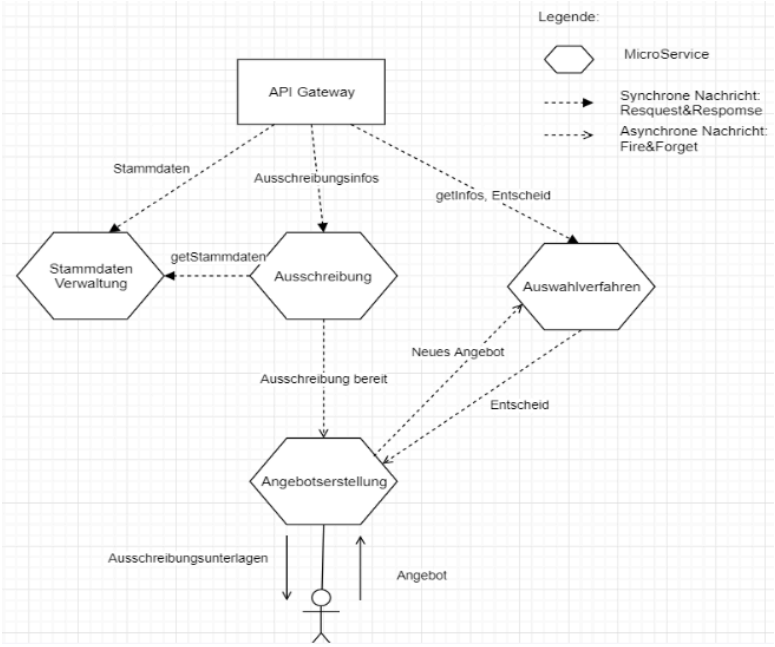


Abbildung 3.6: Beispiel einer Microservice Architektur

3.8.2 Datenfluss

Wichtig ist das man den Datenfluss und die Beziehung untereinander korrekt modelliert.

3.9 Legende

Falls verwendete Symbole für Blöcke, Beziehung oder Interaktion nicht klar ist, muss eine entsprechende Legende vorhanden sein.

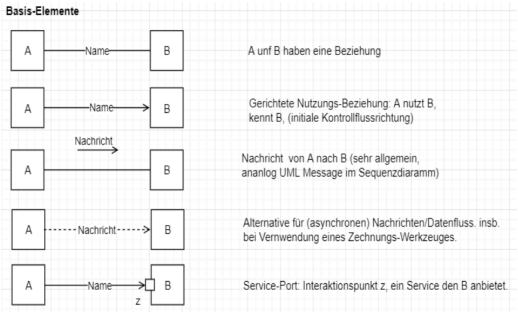


Abbildung 3.7: Basiselemente zur Interaktionsmodellierung

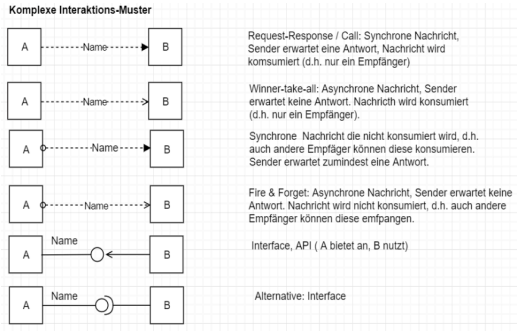


Abbildung 3.8: Komplexe Interaktionsmuster

4 Architekturmuster

Beschreiben als Konzept den Grundaufbau eines ganzen Systems (Vergleich Enturfsmuster). Es gibt versch. Muster welche auf ihre besonderen Anforderungeen angepasst sind (gross/komplex, stark verteilt, inter(re)aktiv, hochverfügbar). Diese sind aber nicht standardisiert wie GoF-Patterns. Siehe auch Catalog of Patterns of Enterprise Application Architecture

4.1 Schichtenbildung

Gliederung eines Systems in aufeinander aufbauende, funktional getrennte Schichten. Die Kommunikation findet über wohldefinierte Schnittstellen statt. Abhängigkeiten nur in Richtung der tieferliegenden Schicht. Kein Überspringen und keine Zyklen. Die Strukturierung sowohl innerhalb eines System als auch Systemübergreifend. Bei physischer Verteilung spricht man von *Tiers*.

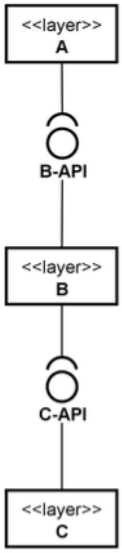


Abbildung 4.1: Layers in UML

Die Bildung einer Schicht kann nach versch. Kriterien erfolgen:

- Logische/Funktional
- Technik
- Abstraktionsebene
- Hierarchisch
  - Funktionale
  - Abstraktionslevel
  - Technologie/Sprache/Framework

In Java manifestieeren sich die versch. Schichten häufig in der Packagestruktur (ch.domain.sysmte.client.*). Allerdings schlechter Ansatz, da sie Modularisierung verunmöglicht. Seit JAVA9 kann Package nicht mehr in mehrere JAR-Dateien zerteilt werden..

4.2 Schichten vs. Features

Alternativen Ansatz der Schichtung als Packages, *package by feature*. Fachliche Sicht zusammenführen. Sehr vereinfacht; statt horizontal wird vertikal gruppiert. Problematik die Modularisierung von JAVA9, verunmöglicht *package by feature*, Konsequenz feingranulare Packages/JARs $\longrightarrow$ wieder positiv!

Ziel ist es *bounded-Contexts* zu identifizieren.

4.3 Verteilte Schichten auf Tiers

Die Schichtenbildung ist eine fundamentale Grundlage um verteilte Anwedungen bzw. ARchitekturen realisieren. Schichtengrenzen eignen sich sehr gut zur Auftrannung, um Teile auf versch. Systeme zu deployen. Zur Abstraktion der Aufteilung bzw. der Kommunikation verwendet man idealerweise austauschbare Schichten.

4.4 Logische 3-Schicht-Architektur

Enthält drei fundamentale Schichten

- Präsentation; GUI-Layer, Maske, WebUI
- Geschäftslogik; Geschäftsprozesse, -Modelle
- Datenhaltung; Datenspeicherung, Datenlogik

Diese Aufteilung lohnt sich praktisch immer, unabhängig der physischen Verteilung: *SoC* $\rightarrow$ *SRP* $\rightarrow$ *Modularisierung*

4.4.1 Verfeinerung der Präsentationssicht

Die reine Präsentation und Präsentationlogik weiter trennen.

- durch MVC
- bei Thin-Clients ist es sogar notwendig - HTML/CSS und Javascript

4.4.2 Verfeinerung der Geschäftslogikschicht

Sinnvoll auch hier weiter zu trennen.

- Business Objects/Domain Objects sind reine objektorientiertes Modell, enthält Daten und Methoden
- Business Funktionen als Services/Businessmethoden

4.4.3 Verfeinerung der Datenhaltungsschicht

Trennung und Abstraktion der reinen Datenlogik.

- Unabhängig von physischen Datenmodell
- O/R Mapping -> Persistenz-Framework (EF)
- Transparentes Einbinden von Legacy-Systeme
- Abstraktion mehrerr Backend (DBMS-)Systeme
- Transaktionshandling (nicht Steuerung!)

4.4.4 Resultat der Verfeinerung

Aus der Verfeinerungen der 3-Schicht-Architektur kann bald eine n-Schicht-Architektur. SRP und SLA wird somit noch stärker realisiert, der Aufwand steigt auch der Aufwand.

4.4.5 n-Schichten - Allgemeine Punkte

Je mehr Schichten umso

- Bessere Strukturierung, einzelne Schichten kleiner und einfacher
- Grössere Chance auf Wiederverwendung (einzelner Layers)
- Höhere Flexibilität, z.B. Austausch einzelner Schichten
- Bessere Skalierbarkeit (primär vertikal)
- Einfachere und präzisere Planung/Schätzbarkeit
- Parallele und getrennte Entwicklung möglich

aber

- Komplexität des ganzen Systems wird grösser
- Mehr Schnittstellen, mehr Aufwand, mehr Planung
- Schichten sind technisch – wo bleibt die Fachlichkeit

4.4.6 Probleme 2-Schicht Architektur

Klassische Client/Server. Typisch mit GUI-Clients mit zentraler Datenhaltung in DBMS

Die Anwedungsschicht (Rich-Client) enthält GUI-, Anwendungs- *und* Business-Logik. Datenschicht erreicht Integrität mit Datenmodell bzw. Contstraints. Logik über Triggers oder StoredProcedures.

Hauptproblem ist, dass das Datenmodel *meist* kanonisch ist und laufend grösser wird. Es entsteht starke Kopplung.

4.4.7 Dikussion 3-Schichten

Man erreicht «natürliche» Aufteilung gemäss logischen Schichten

- Präsenstation
- Domänen- / Businessschicht
- Datenschicht mit Logik und Haltung

Dadurch Vorteil von (manchmal) weniger Redundanzen in der Logik (UI, Backend, DB!?). Hat aber auch potential indem mehrere und verschiedene Datenquellen abstrahiert werden können und mehrere versch. Clients realisierbar sind. Echte Zentralisierung von Businesslogik möglich.

4.4.8 Bilanz

Für eine Schichten-Architektur kommt es auf die Grösse an. Im kleinen Funktinioniert es sehr gut zentrales Problem ist aber die Breite der Schicht, irgendwann unübersichtlich.

Die *Domäne* eine Applikation wird schlicht zu gross. Als Konsequenz werden die Schichten zu breit, worin sich Dinge zusammengelegt sind, die nichts miteinander zu tun haben.

4.5 Service Oriented Architektur

Services sind fachlich orientiert und technologieneutral. Services kapseln abgegrenzte Sub-Domänen ind eigenständige, verteilte Dienste, die dann von übergeordneten Applikationen zur Realisierung eines Businessprozesses genutzt werden. Ein Service verfügt über

- eine wohldefinierte Schnittstelle
- Services sind in einem Verzeichnisdienst eingebunden und werden dynamisch gesucht und gebunden (binding)
- Kommunikation kann über fast beliebgie Protokolle erfolgen (RPC, RMI, CORBA, HTTP, SOAP, REST, usw.)

Führt zur fachlichen Entkoppelung und die Services werden kleiner führt aber auch zu verteilten Applikationen.

5 Microservices

Was sind Microservices und die Herausforderungen.

5.1 Was ist ein Microservice

Die Idee liegt im Wort. Es sind kleine Services. Man unterteilt eine Domain in kleine, voneinander unabhängige Teildomänen *bounded context*. Microservices nähern sich Modulen, sorgen aber für *harte* Modularität. Durch getrenntes Deployment wird eine Architektur durch unerlaubte Kopplung stark erschwert (o(bzw. sichtbar.)) Eine Einheit wird somit kleiner und schlanker. Das hat aber Konsequenzen bzw. Preis.

5.2 Definition Microservice

Eine Applikation wird aufgeteilt in mehrere, kleine Services welche in einem *eigenen* Prozess/Plattform laufen. Leichtgewichtige Kommunikation meistvia RESTfull,/http/JSON. sie sind unabhängig voneinander deploybar (auch Release). Automatisiertes Deployment (DevOps, PaaS, IaaS).

5.2.1 Aufteilung einer Applikation in MS

- Applikation primär vertikal in mehrere, möglichst eigenständige Teile auf. Sollen autark arbeiten und auf eigene Daten zurückgreifen
- erfordert Aufbrechen des Domänenmodelles in versch. *bounded context*, ist aber schwierig. Datenhaltung ist getrennt.
- Einzelne Teile sollten **nicht** direkt miteinander kommunizieren. Einsatz von GUI oder Gateway orchestriert.
- Die Grösse eines MS sehr unterschiedlich gross, typisch: Applikation » MS >= Modul

5.2.2 Jeder Service hat eigenen Prozess / Plattform

Die Services laufen als eigenständige Prozesse (unterschiedliche Plattformen, OS, Programmiersprachen) in typisch virtualisierten Container. Sie laufen in echter Parallellität als verteilte Applikation. Birgt aber ganzes Potential und alle Herausforderungen von verteilten Systemen (Kommunikation über Netz, Latenz, Skalierung, Ausfall). Nutzt aber ein Client mehrere Service asynchron, erhöht sich natürlich die Performance. Insgesamt erhöht sich die Komplexität.

5.2.3 Leichtgewichtige Kommunikation

JSON basierte REST-Schnittstellen sind wegen ihrer «Einfachheit» sehr populär (nach XML). Sind den die JSON/REST-basierenden Schnittstellen leichtegwichtig? Im Vergleich mit effizienten binären Protokollen (RMI, RPC etc.) nicht unbedingt. Vorteile sind aber ohne Zweifel auf eine http(s) basierende Kommunikation relativ leicht zu implementieren und automatisiert testbar. Authentifizierung und Verschlüsselung ist damit ebenfalls abgedeckt. Auch gute Akzeptanz im Operating weil bestehende bekannte Protokolle.

5.2.4 Unabhängig Deploy- und Releasebar

Klar, sind eigenständige Projekte und Releaseeinheiten. Werden erst durch gemeinsame «Orchestrierung» zur Applikation.

- Kleinere Einheiten einfacher und flexibler entwickelbar (OCP - Open/Closed Prinzip)

Weil unabhängig Deploybar müssen Applikationen damit umgehen können, dass einzelne Teile/MS ausfallen oder nicht verfügbar sind. Ausfälle werden genrell häufiger. Haben fünf Services eine Verfügbarkeit von 95%, ergeben sich 0.95^5 = 77% Gesamtverfügbarkeit. Die *Resilienz* wird wichtig!

5.2.5 Deployment automatisiert

Im Gegensatz zu monolithischen Deployments, wo manuelles Deployment häufig, ist dieses Szenario mit nur 10 MS nicht haltbar. Alle Abläufe würden um Faktor 10 verlängern. Automatisierung ist für MS ein absolutes *Muss*-Feature.

5.3 Herausforderungen und Potential von MS

Schichten (horizontal) und Microservices (vertikal) sind orthogonal (gelbe Spalten). Idealerweise befände sich bei einem Schnittpunkt ein Modul (Organes Quadrat). Entscheidend ist die Grösse und Schnitt der Einheit. Abhängigkeiten zwischen den MS sollen unbedingt minimal sein.

Auf der Grafik sieht man links bei der DB weiteres Optimierungspotential (siehe Farben n-Tier und monolith DB)

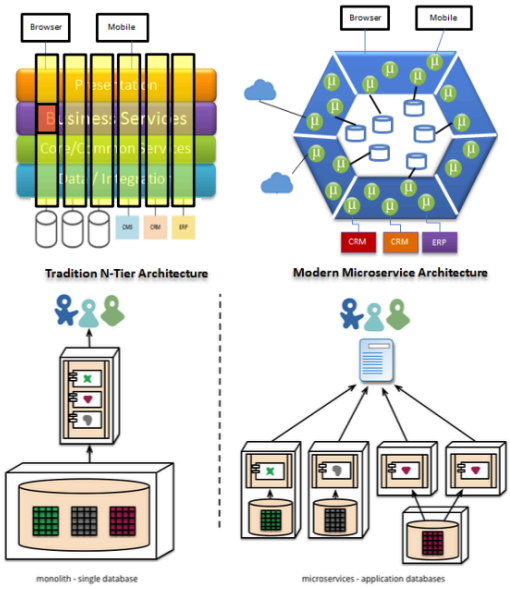


Abbildung 5.1: Vergleich von Microservices cs. Klassisch

5.4 Schnittstellen und Kommunikation zwischen MS

Zentrale Herausforderung sind die *Abhängigkeiten* welche bei MS primär durch Kommunikation. Ziel ist also möglichst wenig, minimale wenn nicht keine Kopplung. Microservices forcieren die Modularisierung, verlagern aber die Herausforderungen ins Deployment und *Operating*.

5.4.1 Kommunikation zwischen Client und [Port-]Micorservice

Mittels leichtgewichtigen Kommunikationsmittel, häufig REST im JSON oder XML-Format. Mehrheitlich *synchrone* Kommunikation (für unmittelbares Feedback). Damit nicht jeder Client jeden MS kennen muss, nutzt man *Gateway-Pattern* welches zusammengefasste, einheitliche Schnittstelle.

Gateway-Pattern

Entspricht dem Konzept des Fassaden-Patterns (GoF) auf Ebene Architektur. Kann Authetifizierung und Verschlüsselung oder einfache Mappings/Tranformationen von Daten vornehmen. Wird explizit als eigenen Service implementiert, es gibt aber auch generische Gateways.

5.4.2 Inter-Kommunikation zwischen MS

Direkte Kommunikation zwischen MS vermeiden, führt zu starke Kopplung. Dazu sollten *Queues* aus der Message-Oriented Architektur genutzt werden.

- Wenn immer möglich asynchron
- Events/Messages und Commands
- synchrone Kommunikation vermeiden

Neue Services können elegant und lose gekoppelt registrieren und so neue Funktionalitäten dynamisch ergänzen.

5.5 Anforderungen an Resilienz

Resilienz beschreibt die Widerstandsfähigkeit bei Teilausfällen um wesentliche Systemdienstleistungen aufrechtzuerhalten. Bezogen auf MS, ein Service soll weiter funktionieren, falls andere nicht verfügbar sind.

Für Ausfall eines Services gibt es zwei Szenarien.

1. Komplett weg, Verbindung kann nicht aufgebaut werden. Fehler wird schnell erkannt, da Abbruch stattfindet.
2. Service arbeitet langsamer, Antwortzeiten grösser, Aufrufer ist länger blockiert (synchron), läuft aber nicht in Timeout

5.5.1 MS Pattern: Circuit-Breaker

Transparenter *Circuit-Breaker*-Proxy wird zwischen synchroner Kommunikation zwischen zwei Services API-Gateway zu Port-MS. Dieser misst die Zeit zwischen Requests, wenn Threshold überschritten bricht er alle neuen Request ab. Nach definiertem Timeout öffnet sich Proxy wieder und prüft erneut. Vorteil Aufrufer brechen schnell ab, es werden weniger Ressourcen verschwendet.

5.5.2 Patterns

Einige Patterns nachforschen. ARC30 -> Folie 25

5.6 Umgang mit Transaktionen

MS und globale Transaktionen vertragen sich nicht, weil sie enge semantische Kopplung verursachen. Das möchte mit MS vermieden werden. Als Lösungsansatz dienen:

- das *SAGA*-Pattern
- getrennte Services zusammenfassen
- fachliche Umgestaltung und Transaktionen minimieren

5.6.1 Transaktionen fachliche Betrachtung

Durch schwierigen Umgang mit Transaktionen bei MS überlegt man genauer, welche temporären, nicht kritischen Inkonsistenzen man zulassen kann. Zum Beispiel Bestätigungsmail - unkritisch. Teile die bisher synchron laufen, werden herausgebrochen und neu asynchron und damit unabhängig verarbeitet.

5.7 Deployment von Microservices

Technologien wie Docker-Images und -Container, gar *server-less* sind zwingend → DevOps lässt grüssen.

5.7.1 Deployment von Java-basierten Services

Klassische Applikations-Container sind auf grosse Applikationen die lange und stabil laufen konzipiert. Entsprechend konsumieren diese viele Ressourcen und haben lange Start- und Initialisierungszeiten. Damit keine Option für MS, falsches Konzept. Daher die Idee von *Microprofile* aber war nicht wirklich nutzbar. Seit 2018 mit Java 9 ist Java fit für MS

5.8 Logging, Metrics und Tracing

Durch kleine, verteilte Services wird Logging, die durchgängige Verfolgbarkeit von (Geschäfts-) Prozessen und Schritte und die Überwachung von Performance und Ressourcen herausfordernd. Auch Debuggen über mehrere MS ist schwieriger als bei einer klassischen Architektur.

5.8.1 Logging in OWASP

Ungenügendes Logging und Monitoring ist auf Rang 10 der OWASP-Liste. Hauptfehler ist, dass essentieelle Ereignisse gar nicht oder nur mangelhaft geloggt werden. Seriöses Loggen wird immer wichtiger. Ein Loggingkonzept ist festzulegen mit *was* geloggt wird und *welches* Format verwendet wird. Technik ist dabei sekundär. Einhaltung der Konventionen verlangt *hohe* Eigendisziplin aller Entwickler. Maschinell auswertbare Logformate verwenden.

5.8.2 Metriken

Laufzeitmetriken sind explizit programmierte *Messpunkte* um Leistungsdaten zu meessen.

- Zähler (Anzahl Ereignisse)
- Messwert (Anzahl Elemente in Queue)
- Meter (Messwert/Zeiteinheit)
- Histogramme (statistische Verteilung von Messwerten wie Mittelwert, Min, Max)

Erste Herausforderungen die Auswahl sinnvoller Messpunkte. Vorhandene Technologien und Werkzeuge einsetzen (oft als Infrastrukturdienste). Vorsicht, Metriken kosten auch Laufzeit. Auswertung an Dritt-Tools delegieren und auch hier auf offene Formate setzen. Metriken sind *kein* Ersatz für detailliertes Profiling.

5.8.3 Tracing

Aufruf von mehreren Folgeaktivitäten (über Services) verfolgen. Jedem Request wird ID (als Fingerabdruck) hinzugefügt und reicht diese jeweils weiter. Somit unabhängig vom zeitlichen und örtlichen Kontext, aber ganzer Requeest einfach nachvollziehbar. Die Correation-ID wenn möglich *nicht* auf Ebene fachlichen Schnittsteellen (SoC) einfügen und nur wenn kein eindeutiges fachliches Attribut (eindeutige Bestellnummer, Transaktionsnummer). Remote-API in Kontext verscchieben (http-Header).

5.8.3.1 Tracing Server

Vergleichbar zu zentralen Logsenken. Erlaubt fachliche Prozesse über das ganze System zu verfolgen.

5.9 Service Discovery

Mit MS herrscht grosse Dynamik und es gibt viel mehr Umgebungen. Wie finden sich Services im Umfeld von Docker, Port-Mapping und Netzwerke?

5.9.1 Service Discovery Dienste

Schlanke, spezielle Dienste vergleichbar mit «lokalen» DNS oder RMI-Registry die logischen Namen einem oder mehreren Services (IP und Port) zuweisen. Können teilweise Load-Balancing übernehmen indem sie registrierte Services selbstständig monitoren (Service Mesh).

5.10 Technologien und Frameworks (Beispiele)

Jeweils mit Java

- Jetty, http-Server
- Dropwizard, für Metriken - relativ unbekannt trotz Alter
- Jakarta EE, Microprofiling im Enterprise-Umfeld

Mächtige Frameworks

- Spring, Spring Boot spezifisch für MS, Schwäche ist Laufzeit-Annotations während Laufzeit (verzögert Startzeiten)

- Micronaut, spezialisiert für MS und Serverless Applikationen, Compile-Time Annotation, dynamische Entwicklung, starke und flexible Integration von Infrastrukturdienste

Nativ kompilierte Java-Anwendungen

- GraalVM, alternative Runtime-VM, mehrsprachig (Java, Groovy, Kotlin, …), kann App nativ (plattformspezifisch) erzeugen, Startzeit von s auf μs reduziert
- Quarkus, Kubernetes Stack für OpenJDK oder GraalVM, native Ausführung, optimiert für Kubernetes - small footprint

Top-Level Framework

- Jhipster, orchestriert nicht nur Runtime, sondern ganzen Entwicklungsprozess. Boilerplate-Codegenerieerung, Konfig, Dependencies, Buildprozess, Testing, Dev, Security

5.11 Übung - Potential von Microservices

Was sind die Vorteile / das Versprechen von Microservices?

- Services können einfach ersetzt werden
- Ein Service ein Bounded Context
- sind gegen andere Services isoliert
- Kommunikation zwischen Teams reduziert sich, weil nur Schnittstellen vorhanden
- sind klein und übersichtlich und dadurch langfristig wartbar
- erlaubt Experimente (zB. Programmiersprache) in kleinem Rahmen
- Services unabhängig voneinander skalierbar
- überschaubar, unabhängig und gut wartbar
- können agil angepasst werden

Was sind die Nachteile von Microservices?

- verteilte Architektur erzeugt Komplexität besonders im Bereich Netzwerklatenz, Lastverteilung, Fehlertoleranz (Resilienz und Asynchronität)
- Logging & Monitoring wird komplex und herausfordernd
- Zeitsynchronisation
- Datenkonsistenz / Transaktionssicherheit muss gewährleistet sein
- DevOps ein muss! Anforderungen an Operating

5.12 Übung - Microservice Patterns

Studieren und Diskutieren Sie die folgenden, ausgewählten Patterns.

5.12.1 **Reliability** - *Circuit Breaker*

When one service synchronously invokes another there is always the possibility that the other service is unavailable or is exhibiting such high latency it is essentially unusable. Precious resources such as threads might be consumed in the caller while waiting for the other service to respond. This might lead to resource exhaustion, which would make the calling service unable to handle other requests. The failure of one service can potentially cascade to other services throughout the application.

5.12.1.1 Problem

How to prevent a network or service failure from cascading to other services?

5.12.1.2 Solution

A service client should invoke a remote service via a proxy that functions in a similar fashion to an electrical circuit breaker. When the number of consecutive failures crosses a threshold, the circuit breaker trips, and for the duration of a timeout period all attempts to invoke the remote service will fail immediately. After the timeout expires the circuit breaker allows a limited number of test requests to pass through. If those requests succeed the circuit breaker resumes normal operation. Otherwise, if there is a failure the timeout period begins again.

5.12.1.3 Resulting Context

This pattern has the following benefits:

- Services handle the failure of the services that they invoke This pattern has the following issues:
- It is challenging to choose timeout values without creating false positives or introducing excessive latency.

5.12.2 Data-Management - *Saga-Pattern*

You have applied the Database per Service pattern. Each service has its own database. Some business transactions, however, span multiple service so you need a mechanism to implement transactions that span services. For example, let's imagine that you are building an e-commerce store where customers have a credit limit. The application must ensure that a new order will not exceed the customer's credit limit. Since Orders and Customers are in different databases owned by different services the application cannot simply use a local ACID transaction.

5.12.2.1 Problem

How to implement transactions that span services?

5.12.2.2 Solution

Implement each business transaction that spans multiple services is a saga. A saga is a sequence of local transactions. Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga. If a local transaction fails because it violates a business rule then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions.

There are two ways of coordination sagas:

- Choreography - each local transaction publishes domain events that trigger local transactions in other services
- Orchestration - an orchestrator (object) tells the participants what local transactions to execute

5.12.2.3 Resulting context

This pattern has the following benefits:

- It enables an application to maintain data consistency across multiple services without using distributed transactions

This solution has the following drawbacks:

- The programming model is more complex. For example, a developer must design compensating transactions that explicitly undo changes made earlier in a saga.

There are also the following issues to address:

- In order to be reliable, a service must atomically update its database and publish a message/event. It cannot use the traditional mechanism of a distributed transaction that spans the database and the message broker. Instead, it must use one of the patterns listed below.

5.12.3 External API - *API-Gateway*

Umgesetzt im Projekt

5.12.4 Cross cutting concerns - *Microservice chassis*

When you start the development of an application you often spend a significant amount of time putting in place the mechanisms to handle cross-cutting concerns. Examples of cross-cutting concern include:

Externalized configuration - includes credentials, and network locations of external services such as databases and message brokers Logging - configuring of a logging framework such as log4j or logback Health checks - a url that a monitoring service can "ping" to determine the health of the application Metrics - measurements that provide insight into what the application is doing and how it is performing Distributed tracing - instrument services with code that assigns each external request an unique identifier that is passed between services. As well as these generic cross-cutting concerns, there are also cross-cutting concerns that are specific to the technologies that an application uses. Applications that use infrastructure services such as databases or a message brokers require boilerplate configuration in order to do that. For example, applications that use a relational database must be configured with a connection pool. Web applications that process HTTP requests also need boilerplate configuration.

It is common to spend one or two days, sometimes even longer, setting up these mechanisms. If you going to spend months or years developing a monolithic application then the upfront investment in handling cross-cutting concerns is insignificant. The situation is very different, however, if you are developing an application that has the microservice architecture. There are tens or hundreds of services. You will frequently create new services, each of which will only take days or weeks to develop. You cannot afford to spend a few days configuring the mechanisms to handle cross-cutting concerns. What is even worse is that in a microservice architecture there are additional cross-cutting concerns that you have to deal with including service registration and discovery, and circuit breakers for reliably handling partial failure.

5.12.4.1 Forces

Creating a new microservice should be fast and easy. A service must implement cross-cutting concerns such as externalized configuration, logging, health checks, metrics, service registration and discovery, circuit breakers. There are also cross-cutting concerns that are specific to the technologies that the microservices uses.

5.12.4.2 Solution

Build your microservices using a microservice chassis framework, which handles cross-cutting concerns

5.12.4.3 Example

Examples of microservice chassis frameworks:

- Java
  - Spring Boot and Spring Cloud
  - Dropwizard
- Go
  - Gizmo
  - Micro
  - Go kit

5.12.4.4 Resulting context

The major benefit of a microservice chassis is that you can quickly and easy get started with developing a microservice.

You need a microservice chassis for each programming language/framework that you want to use. This can be an obstacle to adopting a new programming language or framework.

6 Messaging

Von einem Monolith zu einem Service-Oriented System.

Wie soll ein Service in einem service-oriented System aussehen? Ein Service

- hat Interfaces
- hat Business Logik
- hat und verarbeitet eigene Daten

6.1 Serviceaufruf

Wie wird ein Service aufgerufen? Folgendes Beispiel ist Total falsch. Man sollte keine Methoden auf einem Service aufrufen. `double MyAwesomeService(double paramA, int paramB)`

- ist nicht erweiterbar, was passiert wenn dritter Parameter dazukommt?
- Hohe Kopplung zwischen Aufrufer und Service
- dies entspricht RPC und nicht service orientiert
- In einem service-oriented Umfeld gibt es keine Methodenaufrufe

In einem service-oriented Umfeld werden Nachrichten verschickt

```
Response MyAwesomeService(Request request)
```

- kann als JSON oder was auch immer enthalten
- Akzeptiert die Request nachricht
- die Nachricht kann serialisiert werden
- Service schickt Antwort zurück -wieder serialisiert

Noch nicht ganz ideal

6.1.1 Service-Orientierte Methode

```
void MyAwesomeService(Request request)
```

- Aufruf ist asynchron
- Anfrage wird in Queue gespeichert
- Service hört auf dieser Queue
- Falls er antworten muss, schickt er die Antwort auch in eine Queue

Dadurch asynchrone Kommunikation mit Queues. Vorteile von asynchron sind klar:

- Verfügbarkeit; ein Service kann ausfallen und der Aufrufer muss Fehler und Recovery handhaben können
- Fehlerresistenz; wenn Service down, wird Caller in Timeout fahren
- Erweiterbarkeit; synchrone Kommunikation führt zu grösserer Kopplung, Kosten für Modifikationen sind höher

- Skalierbarkeit; wie kann Service skalieren, wenn grosse Last?

6.2 Skalierbarkeit

6.2.1 Skalieren in einem synchronen Umfeld

Skalieren durch mehrere Instanzen des gleichen Services in einem synchronen Umfeld, braucht man dazu aber einen *Load Balancer*. Dieser kostet, benötigt Knowhow über die Technologie und führt neue *Point-of-Failure* ein.

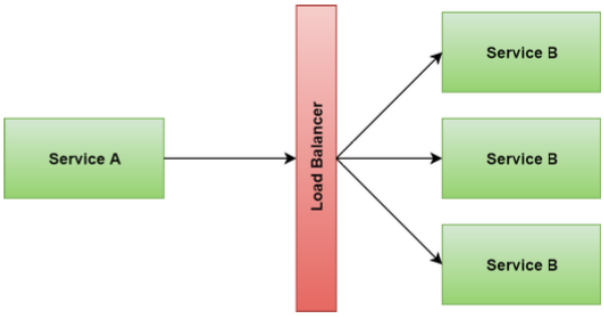


Abbildung 6.1: Skalierung in synchronem Umfeld

6.2.2 Skalieren in einem asynchronen Umfeld

Da bereits alle Services in Queues lesen führ die Skalierung durch mehrere Instanzen nur zu wenig mehr Kosten (ein paar Container), fügt aber keinen neuen *Point-of-Failure* hinzu. Die neuen Instanzen von *B* (deren Worker) hören einfach auf die gleiche Queue.

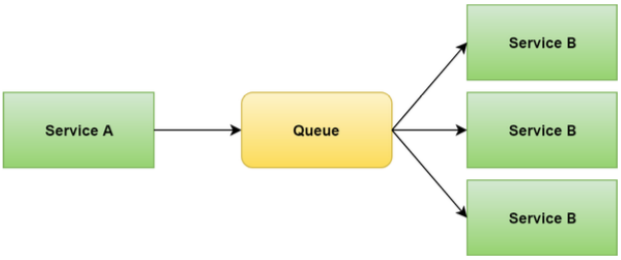


Abbildung 6.2: Skalierung in asynchronem Umfeld

6.3 Messaging Basics

MS müssen miteinander kommunizieren. Es gibt verschiedene Verfahren, welche alle ihre Vor- und Nachteile haben.

- Advanced Message Queuing Protocol (AMQP)
- REST
- REST mit HTTP/2 push
- SOAP
- RPC

Eine Nachricht triggert evtl. eine Antwort, welche auch eine Nachricht ist. Eine Message kann auch von einem oder mehreren Empfänger verarbeitet werden.

- **Message** ist eine Informationeinheit, welche durch System gesandt wird. Sie hat einen Header mit Metadaten und einen Body, welcher meist Daten in binärer Form enthält.
- **Producer** (Akteur) erstellt und schickt Nachrichten
- **Consumer** (Akteur) erhält und verarbeitet Nachriten
- **Queue** Eine Sammlung die Nachrichten in einer Warteschlange stellt und durch Consumer abgeholt werden

Nachrichten basierende Systeme nutzen typischerweise ein Messaging-System, auch *message-broker software* oder *message-oriented middleware* genannt.

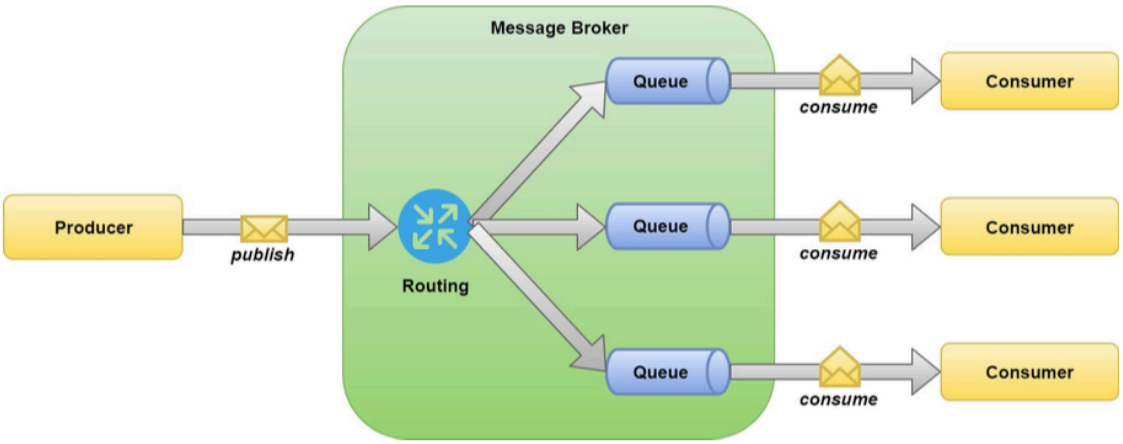


Abbildung 6.3: Überblick wie das so abläuft

Das Routing schickt Nachrichten weiter, an die Queues, die Nachricht von Producer abonniert haben. Wenn Consumer Zeit haben, holen sie die Nachrichten ab. Der grüne Teil wird von eben diesen *Message-Broker*-Systemen übernommen.

6.4 Vorteile durch Messages

- **Resilienz** Nachrichten gehen nicht im Netz verloren wenn Netzwerk down ist. Sie werden Zwischengespeichert und ausgeliefert wenn Netz wieder da ist
- **Fehlertoleranz** Die Nachricht kann einfach wieder gesandt werden, wenn Fehler auftritt oder Service down
- **Asynchronität** die Services können andere Aufgaben wahrnehmen sobald Nachricht verschickt worden ist, egal ob diese noch unterwegs oder schon verarbeitet wird
- **Entkoppelung** Die kommunizierenden Services kennen einander nicht und sind so voneinander unabhängig/entkoppelt (klar das Nachrichtenformat ist bekannt)
- **Effizient und Skalierbar** Asynchrone Nachrichten sind der Schlüssel für Effizienz und Skalierbarkeit

6.5 Message-orientiert Denken

Man sollte in Nachrichten anstatt im System denken. Den Datenfluss sollte nicht modelliert werden. Die Services sollen sich nicht kennen. Diese senden einfach Nachrichten ans Universum. Verteilte Systeme sind schwierig, daran ändern auch Microservices und Messages nichts.

6.5.1 Message-first approach

Kann helfen das System zu designen.

- Die Anforderungen enthalten den Status der Aktivitäten die im System ablaufen sollen, was geht rein - was raus
- Die Nachricht drücken eine Intension aus, was passieren soll, wenn die Nachricht verschickt werden soll
- die Anfoderungen können also in Nachrichten transformiert können
- Nachrichten können konzeptionelle Ebene zwischen Business Requirements und technischer System Spezifikation sein

6.5.1.1 Nachrichten Modellierung

Beispiel Webshop. Kunde wählt Produkte aus und fügt sie in den Warenkorb. Später bestellt er sie und erhält eine Bestätigungsmail und der Versand wir duasgelöst.

Activity description	Message name	Message data
Checking out	*checkout*	Cart items and prices
Recording the purchase	*record-purchase*	Cart items and prices; sales tax and total; customer details
Sending an email confirming the purchase	*checkout-email*	Recipient; cart summary; template identifier
Delivering the goods	*deliver-cart*	Cart items; customer address

Abbildung 6.4: Nachrichtenmodellierung

6.6 Synchron und Asnychron

Kommunikation kann auf technischem Level synchron oder asynchron erfolgen. Die Kommunikation via HTTP erfolgt synchron udn ist ein **request/response** Modell. Ein Szenario mit Nachrichten und Queues ist asynchron auf dem technischen Leve.. Auf einem konzeptuellen Level kann die Kommunikation synchron oder asynchron erfolgen, egal wie es technisch umgesetzt wurde. In einem synchronen Szenario wenn der Sender auf eine Antwort wartet ist er blockiert. Wenn er aber darauf vorbereitet ist, kann er darauf reagieren, bis das Resultat zurück ist. Dies ist dann asynchrone Kommunikation.

Wenn immer möglich asnychrone Kommunkation einsetzen, denn synchrone führt zu höherer Kopplung. Asynchrone systeme sind eher Event*event-driven*.

6.7 Routing

Services wissen nicht genau, wer ihre Nachrichten liest. Sie schicken ihre Nachrichten einfach an einen Message-Broker. Diese werden anhand den Headerinformationen vom Broker weitergeleitet.

6.7.1 Patteren Matching

Ein Service gelangt mittels *Pattern Matching* an die Nachrichten an die er interessiert ist. Damit können Nachrichten vonvom gleichen Typ gruppiert werden. Damit diese dynamisch weitergeleitet werden können, implementieren wir einen Namespace für diese Nachrichten.

6.7.1.1 Labels

Labels sind im Message Header und können beliebige *key-value-pairs* enthalten. Mit einem Label-Key im Header kann das routing nun wie folgt gemacht werden.

Message Name	Message Header
create-customer	label: customer
delete-customer	label: customer
create-product	label: product
delete-product	label: product

Abbildung 6.5: Pattern Matching mit Labels

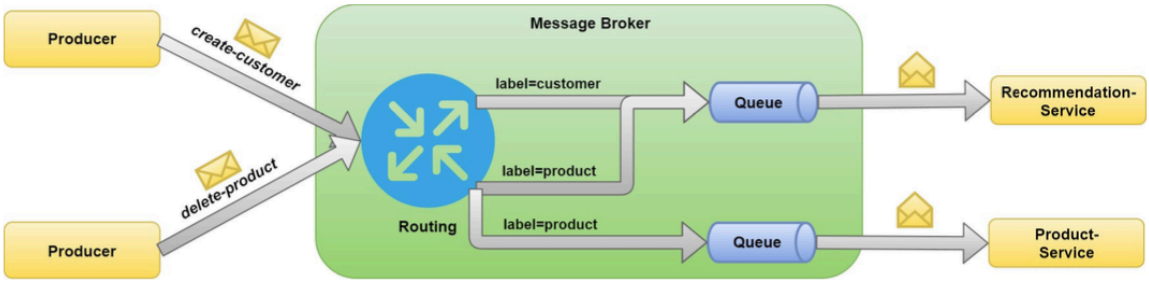


Abbildung 6.6: Routing mit Labels

6.7.1.2 Hierarchical Naming

Die Nachrichten werden anders benannt, ähnlich wie Packages.

Message Name	dynamic routable Name
create-customer	customer.create
delete-customer	customer.delete
create-product	product.create
delete-product	product.delete

Abbildung 6.7: Pattern Matching mit Hierarchie Namen

6.7.2 Response Message routing

Der Sender fügt im Header der Nachricht das optionale *ReplyTo*-property hinzu, welches die Adresse der Queue beinhaltet wohin die Response gehen soll.

6.8 Reliable Delivery

Wie kann sichergestellt werden dass alle Nachrichten eintreffen? Grundsätzlich muss Transportmedium als *unzuverlässig* betrachtet werden (Netzwerkausfälle, HW failure, Serviceausfall). Der Message-Broker kann Nachrichten im Fehlerfall neu zustellen. Aber auch er muss wissen, ob die Nachricht zugestellt wurde und von der Queue

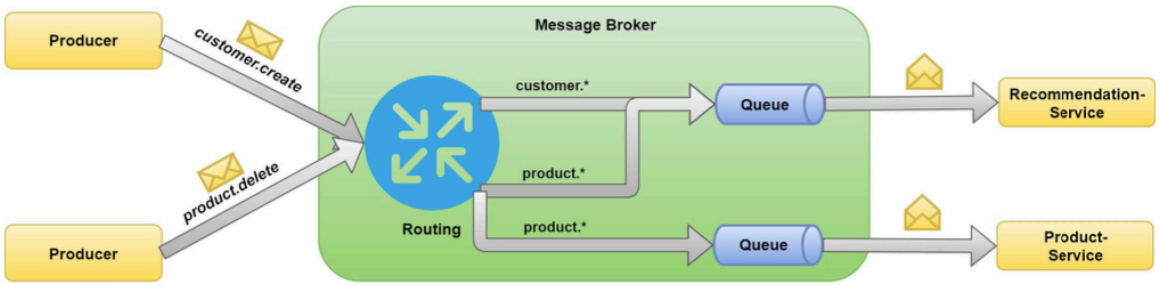


Abbildung 6.8: Routing mit Hierarchie Namen

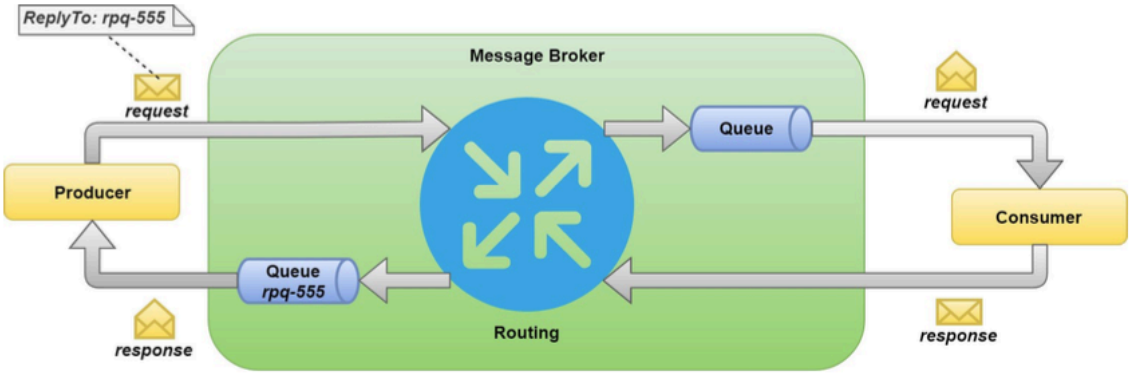


Abbildung 6.9: Wie gelangen Reponses zurück

gelöscht werden kann. Dazu muss der Empfänger den Empfang *oder* die Verarbeitung bestätigen *acknowledgments (ACK)*.

Acknowledgments gibt es in beide Richtungen und bestätigen, dass die Nachricht **mindestens einmal** zugestellt wurden.

6.9 Messaging Systems - RabbitMQ

Ein *message-broker* ist ein zentrales System mit Warteschlagen. RabbitMQ wurde entwickelt um AMQP und weitere Protokolle und viele Programmieresprachen zu überstützten. Es ist OpenSoruce und leichtgewichtig.

6.9.1 Concepts

Bestehend aus drei grundlegenden Konzepten

- **Exchange** erhält Nachrichten und verteilt diese an Queues auf Basis von Routing-Rules.
- **Queue** eine Sammlung an Nachrichten diese später vom Consumer abgeholt werden
- **Bindings** eine Konfiguration, die Exchange zu einer Queue mappt und das Routing definiert (mapping)

6.9.2 Types of Exchange

Es gibt bei RabbitMQ die drei Typen

- **Direct** nutzt den Queuenamen als binding pattren
- **Fanout** verteilt Nachrichten an all angeschlossnen Queues ohne explizites Routing

- **Topic** nutzt Routing-Schlüssel und Pattern-Matching um Nachrichten zu verteilen, bietet grösste Flexibilität

6.9.3 Connect to RabbitMQ

Dazu müssen zwei Komponenten bekannt sein.

- **Connection** ist eine TCP Verbindung zu RabbitMQ. Es ist eine Verbindung pro Prozess zu verwenden und diese offen lassen. Öffnen und schliessen sind «teuer».
- **Channel** ist eine virtuelle Verbindung in einer Connection. Sind aber leichtgewichtiger. Auch hier offen lassen. Pro Thread ein Channel (Channel sind nicht threadsafe).

6.10 Messaging Patterns

- **Synchron/Asynchron** Sender erwartet eine/keine Antwort
- **Observe/Consume** die Nachricht kann nur vom ersten Verarbeitet werden oder dürfen noch weitere die Nachricht lesen

6.10.1 Fire and Forget

Purste Form (in MS) der Kommunikation ist asynchron und werden nicht *consumed*, Nachricht existiert weiter und darf von allen gelesen werden. Entspricht dem *publish-subscribe* pattern

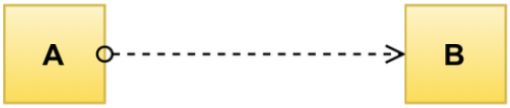


Abbildung 6.10: Fire and Forget modelliert

6.10.2 Winner take all

Kommunikation ist asynchron und werden aber *consumed*, Nachricht kann nur von eiem gelesen werden. Typischerweise in einem Szenario wo mehrere Worker parallel arbeiten.

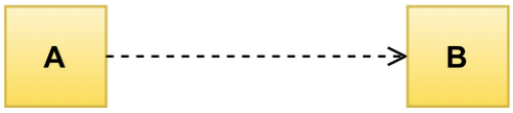


Abbildung 6.11: Winner take all modelliert

6.10.3 Request/Response

Ist wie Klassische HTTP oder REST Kommunikation. Ist synchron und Nachrichten werden *consumed*. Der Sender erwartet eine Antwort auf den Request.

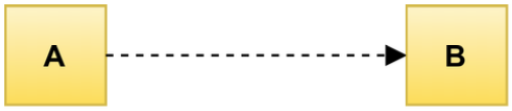


Abbildung 6.12: Request/Response modelliert

6.10.4 Synchronous-Observed

Synchrone Kommunikation aber Nachrichten werden nicht *consumed*. Der Sender erwartet eine Antwort. Weitere Services können mithören, aber der Sender ist nur an einer Antwort interessiert.

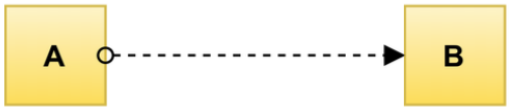


Abbildung 6.13: Synchronous-Observed modelliert

6.10.5 Dead Letter Queue

Darin werden alle Nachrichten gesammelt die aus diversen Gründen nicht zugestellt werden konnten. Damit können potentielle Probleme im System aufgedeckt werden.

6.11 Scaling and Back Pressure

Asynchrone Kommunikation hat viele Vorteile, aber es gibt auch einige Herausforderungen wenn das System stark ausgelastet ist.

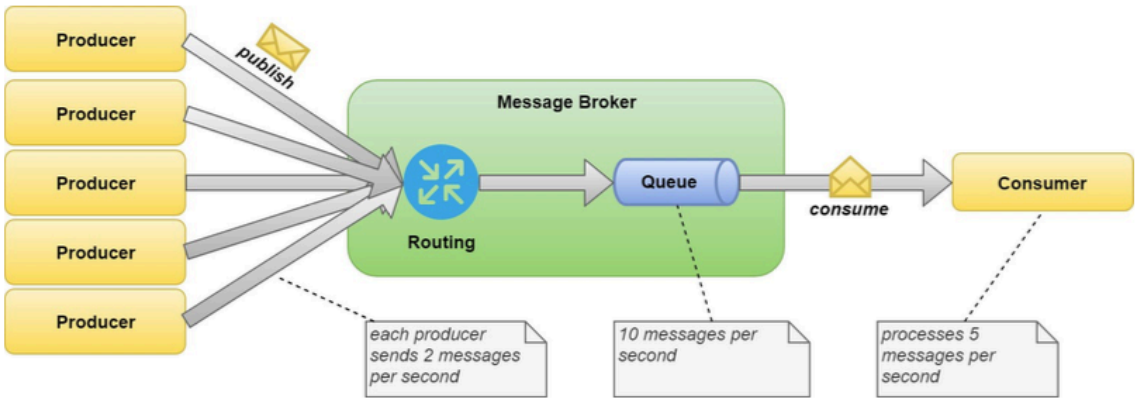


Abbildung 6.14: Überlastung eines Consumers / Queue durch Heay-Load

Dauert dieser Zustand an, wächst die Queue ins unendliche. Eine Möglichkeit ist Skalierung. Die andere Möglichkeit ist das Anwenden von *backpressure* -Strategien auf die Produzenten. Diese werden so verlangsamt.

- synchrones Ausbremsen
- Load Shedding, Queue ist limitiert - Pakete werden verworfen wenn Threshold erreicht
- Flow Control - Bandbreite wird limitiert, damit Producer ausgebremst werden -> RabbitMQ macht dies per default (kann anderes konfiguriert werden)

7 Testen von Applikationen

Testaspekte für verschiedene Architekturen.

7.1 Ziele für gute Tests

- Testausführung weit möglichst automatisieren
- Qualität von Testcode wie produktiven Code behandeln
- Tests lohnen sich
- UseCases und Funktionen möglichst einzeln testen, sonst instabiler auf Veränderungen
- Kurze einfache Testfälle, (Selektivität der Testfälle)
- Teste der normalen, erwarteten Ablaufs
- Ausnahmen und provozierte Fehler testen

Nur so komplex wie nötig und so einfach wie möglich.

7.2 Testaspekte

Unterschiedliche Testarten wie *Funktionale*, *Performance-/Stress* und *Sicherheits*-Tests. Je nach Architektur und Applikationsart konzetrieren sich die Tests stärker auf die Server.

7.2.1 Funktionale Applikationstests

- Testen vollständige App
- kombiniertes Testen von Business- und Applikationslogik
- können sehr aufwendig werden
- Erfordern wohldefinierte Startbedingungen (Daten)
- Zeitaufwendig
- Aufwändige Umgebung (Systemanforderung)

7.2.2 Performance- und Stress-Tests

- Belastungstests durch viele parallele Clients
- Verhalten testen wie Queueing, zurückweisen, abweisen der Anfragen
- Wiederanlaufverhalten zur Robustheit und Stabilität und Resilienz

7.2.3 Sicherheits-Test

- vorallem Web-Anwendungen
- siehe OWASP

7.3 Tests in Schichtenarchitektur

Gefahr besteht, dass man immer alle unteren Schichten *mittestet*. Daraus negative Konsequenzen für die Integrationstests, weil Ausführungszeit steigt, Selektivität sinkt. Werden immer mehr zu Integrationstests statt Unit-Tests. Einsatz von Test Doubles um die Sichten einzeln und entkoppelt voneinander testen.

7.4 Testen von Rich-GUI-Applikationen

GUI ist Schnittstelle selber. Interaktivität muss für automatisierten Tests «simuliert» und aufgezeichnet werden. Verifikation muss «optisch» erfolgen. Stolpersteine wie Bildschirmauflösung, Fensterpositionen, Zeitverhalten etc. Relativ hoher Aufwand für Testprogrammierung. Starke Kopplung an die interne GUI-Struktur.

7.5 Testen von Applikationen mit DB

Herausforderung ist die Datenmanipulation durch Testfälle (Reproduzierbarkeit). Anforderungen sind

- Zustanz der DB muss vor und nach Ausführung definiert sein
- Effiziente Verifikation des DB-Inhaltes
- Parallele Testausführung

Grundlage dazu ist Testdatenmanagement. Das Bereitstellen und aktive Wartung von Testdaten.

7.5.1 Lösungsansätze

- Frameworks für Daten, Fälle und Verifikation, z.B. DBUnit
- Verwendung mehrere Schemas auf zentralem DBMS
- Lokale DB pro Entwickler - Aufwand/Kosten für Pflege
- In-Memory-DB für jeden Test. Ist der Repräsentativ?
- Prod DB als Docker-Container

7.6 Testen von Web-Apps und -Services

Sind einfacher zu Testen weil auf Http/s Request-Response Modell basieren. Ausnahme AJAX-calls sind schwieriger. Stress- und Security Tests ebenfalls gut machbar. Je besser HTML-Code desto effizienter und einfacher wird Testen (positive Mitkoppelung). Alle Elemente zur Identifikation und Selektion mit eindeutiger ID kennzeichnen. Schwierigkeiten durch grössere Dynamik (reactive programming, Clientscripts, Asynchronität). Grundlage dazu ist Browser-Automation (Versionsproblematik, IE, Chrome, Safari, Mobile etc.) Bekannte Frameworks sind SeleniumHQ und Arquillian

7.7 Testen von REST-(Micro)-Services

Weil REST-Services sprachunabhängig gibt es grosse Zahl von Frameworks. Für simple Fälle reichen Bordmittel oder auch curl. Jüngere Libraries für REST-Clients sind so abstrakt und kompakt (OkHttp, Unirest) dass man quasi mittesten kann. MS-Frameworks (SpringBoot, Micronaut) bieten integrierte Ergänzungen für Testing an. Es gibt auch *Testcontainer*

7.8 Bilanz

Pragmatisch, selektiv und effizient testen.

8 Docker

Vereinfacht das bereitstellen von Anwendungen via Container, weil diese alle nötigen Pakete beinhalten und sich so als Dateien transportieren und installieren lassen.

8.1 Container

Isolierter Prozess auf einer Maschine, mit seinem eigenen *Namespace*. Er hat/kann

- Kann keine anderen Prozesse sehen.
- Isolierter Netzwerkstack, kann andere Prozesse nicht beeinflussen.
- eigene cgroup (beschränkter Zugriff auf Memory, CPU, Disks)
- eigenes root-file System (kann keine anderen Files anlangen)

8.2 Image

Ein Template um Docker Container zu erstellen. Wenn ein Container ein Objektinstanz ist, ist das Image die Klasse

8.3 Dockerfile

Textfile um ein Docker-Image zu erstellen. Es ist der Sourcecode um das Image zu erstellen.

8.4 Using Backbone-Stack

1. Start the services: `docker-compose -f docker-compose.local.yml up`
2. Ensure all services (containers) are running: `docker-compose -f docker-compose.local.yml ps -a`
3. Display the logs of all services or of a single service: `docker-compose -f docker-compose.local.yml logs -f`

8.4.1 Local Version

Enthält essentielle Services für unsere Microservice-System.

The local version can be used to run the system locally on you machine for development. It is defined in the file docker-compose.local.yml. This version contains the following services:

- bus: RabbitMQ as messaging system. Our microservices communicate over this bus.
- portainer: A web-based tool to monitor all your docker artefacts such as containers, networks….
- mongodb: A nosql database to store data. No actively used, just for your convenience.

- mongo-viewer: A webbased DBMS for the mongo-db to view and modify the database content. Here we use port-mapping to connect these docker container to our host network. In the productive version, there will be no port-mapping, but only isolated virtual networks!

8.4.1.1 Network View local Version

Die Netzwerke sind voneinander isoliert. Für den Zugriff werden die Container mittels Port-Mapping ins lokale Netz exponiert. Definiert werden beide im docker-compose.local.yml (*software-defiend* by Docker, keyword *network* und *ports*).

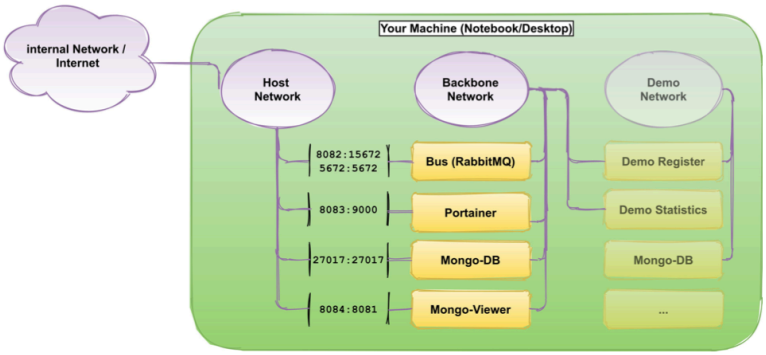


Abbildung 8.1: Netzwerkübersicht Lokale Version

8.5 Service Templates

kurz beschreiben

8.6 Tool Chain - automated CI/CD

Die zur Verfügung gestellte Toolchain macht folgende Dinge:

- Builted Service
- Testing
- Qualitätschecks
- publiziert Service als Docker Image
- Deployt der neue Service als Docker Container

9 Architektur - API-Design

Eine API (Application Programming Interface) ist eine Schnittstelle die darauf ausgelegt ist, eine Softwareeinheit nutzbar zu machen. Besteht aus mehreren Klassen/Interfaces und definiert gemeinsame Funktionen und Datentypen. Eine API soll

- einfache Nutzung ermöglichen
- Aufwand für Nutzer minimieren
- Unabhängig von konkreter Implementierung sein
- implizites Mass an Öffentlichkeit

9.1 Von der Schnittstelle zur API

Ist eine Frage des Scopes. Technisch betrachtet kann eine Schnittstelle

- im OO-Design zu Implementation existieren
- zu einem Framework
- zu einem Teilsystem
- zu einem System

darstellen. Definition des Scopes wird auf Ebene Architektur. Vorsicht vor nicht als API gedachten Schnittstellen

9.1.1 Motivation

Gute APIs haben einen grossen Wert. Fordern die Modularität und Entkoppelung und machen komplizierte einfach nutzbar oder auch austauschbar. Machen es ein Produkt attraktiv. **Schlechte** APIs verursachen hingegen Ärger und Kosten. Fehlerhafte Nutzung, aufwändige Fehlersuche und Support. Verführen dazu eine vorhandene Lösung nicht zu nutzen. Verführen zu Workarounds und eher unsaubere Arbeit.

9.1.2 Herausforderungen

Öffentlich verfügbare APIs leben fast ewig und somit auch allfällige Designfehler. Änderungen sind nur mit grossen Aufwand/Konsequenzen änderbar. Meist nur **eine** Chance. Das **entwerfen** einer guten API ist herausfordern. Die grosse Kunst dabei die API so zu entfernen, dass sie entwicklungsfähig ist.

9.1.3 Hauptziele

Hauptziele einer guten API sind maximale Entkoppelung (lose Kopplung) der Implementiation. Maximales Information Hidding und Kohäsion. Prinzipien wiederholen sich, jedoch ist die Gefahr von Fehler viel Grösser.

9.1.4 Qualitätsanforderungen

- Konsistente Namengebung
- einfache, treffende Namen
- Verhalten so erfüllen, wie es erwartet wird
- Erweiterbarkeit für Weitentwicklungen sicherstellen
- einfache und hilfreiche Dokumentation
- Perspektive auf den Nutzer wenden - für ihn soll es einfacher werden
- KISS-Prinzip - möglichst einfach
- Sicherheit - Nutzung sicher gestalten

9.1.5 API-Patterns

Werden genutzt um Namensgebung und Verhalten vorhersehbar und durchgängig zu gestalten.

- Repetition - gleiche Namen für gleiche Dinge
- Periodisch - gleiche Bezeichnungen für gleiches Verhalten
- Symmetrie - symmetrische Methoden anbieten - open/close oder inuse/free

9.1.6 SPI - Die «API» für den Anbieter

Die Service Provider Interface ist eine Teilmenge oder Ergänzung zu einer API. Ist für die Anbieter von Implementationen einer API gedacht.

9.2 Class-based API

Werden über ein/mehrere Interfaces und Klassen realisiert. Typisch in Package bzw. Modul abgeschlossen mit vollständiger JavaDoc. Klassen die formale Parameter oder als Return genutzt werden sind automatisch **Bestandteil** der Schnittstelle! In geschichteter entsteht häufig ein API-Layer (Service-Layer). Oft brechen Datentypen zwischen Schichten - Pro Layer eigene Datentypen die dann oft reine *Value Objects* (VO) oder *Data Transfer Objects* (DTO) sind.

DTO/VO sind verkleinerte/vergrösserte Entity-Objekte (ViewModels). Diese brechen einzelne Schichten. Aufwand kann sich vorteilig oder nachteilig auswirken (je nach Auswirkung auf verschiedene Ebene).

9.2.1 Empfehlungen

Es gelten erwähnte Ziele und Qualitätsanforderungen (Namensgebung, Verhalten, Doku). Möglichst kurze und einfache Parameterlisten oder Overload verwenden. Für Rückgabetypen Interfaces (`IList`) nutzen. Ab drei Parameter *Builder Pattern* nutzen. API-Stil an Bedürfnisse/Sprache/Situation an Nutzer anpassen.

9.2.2 Fluent API - Stil

Ist das *flüssige* aneinanderreihen von Methodenaufrufen. Als Beipspiel `LINQ` `list.Where(e => e.Name == "Stofer").Select(e => e.Id);`

9.2.3 Fehlende Daten

Wie signalisiert eine Schnittstelle fehlende Daten? Bei Get wird etwas nicht gefunden? Bei `GetById` kann man `null` oder eine Exception zurückgeben, da man davon ausgeht, dass die Id vorhanden ist. Bei einer Liste leere Liste zurückgeben, kein `null`oder Exception.

9.3 REST API

REST heisst *RE*presentational *S*tate *T* ransfer und ist ein *Architekturstil* welcher einen Satz von Prinzipien definiert. Diese sollen konforme Architektur gewährleisten. Wesentliche sind:

- Statuslose Kommunikation (zustandslos)
- Verwendung von HTTP-Standardmethoden
- Ressourcen mit eindeutiger Identifikation (URI)
- Unterschiedliche Repräsentation von Ressourcen
- Verknüpfungen/Hypermedia (HATEOAS)

Es sollen auch unterschliedliche Protokolle genutzt werden können.

9.3.1 Zustandslosigkeit

Die Kommunikation **muss** bei REST zustandslos sein. Der Zustand muss entweder vom Client gehalten werden oder vom Server in einer Ressource abgebildet werden. Der Status der Applikation ergibt sich *ausschliesslich* aus der Ressourcenpräsentation (URI) und dessen Status. Daher keine Sitzungsinformationen und keine Sessioninformationen nötig. Vorteile sind Lesezeichen und bessere Skalierbarkeit der Infrastruktur (Load Balancing - Session auf gleichem Host halten).

9.3.2 Standardmethoden

REST-konforme Applikationen verwendet die http-Methoden `GET`, `PUT`, `POST`, `DELETE`, usw. verbindlich. Diese sind als **idempotente** Methoden (ausser Post) zu implementieren. Können beliebig oft aufgerufen werden ohne Seiteneffekte.

9.3.3 Identifizierung von Ressourcen

Eine Ressource kann über URI eindeutig angesprochen, identifiziert und als beliebige Datenformate (XML, JSON, Bild, PDF) genutzt werden. Die Räpresentation ist über `Accept: MIME-Type` wählbar. DIE URI enthält **keine** Namen von Operationen.

9.3.4 Schnittstellendesign

Die Bezeichner werden typisch in *Mehrzahl* geschrieben. Die ID's sind **immer** Bestandteil des Pfades. Nur für Suchargumente werden zusätzliche Attribute übermittelt. Das heisst, dass über die gleiche URI verschiedene Aktionen ausgeführt werden. Unterscheidung lediglich durch die verwendete *http*-Methode (`GET`, `PUT`, `POST`, `DELETE`, usw.)

9.3.5 Rückgabewerte über Body und http-Statuscode

Mit RESTfull-Schnittstellen über http haben wir mehrere Rückgabemöglichkeiten. Über Body, Statuscode, Header. Als Konsequenz keine Fehlermeldungen sonder Statuscode zurückgeben.

9.3.6 Hypermedia-Prinzip

Ist das Prinzip von Verknüpfungen von unterschiedlichen Ressourcen über Links. Wenn Client Antwort bekommt, kann er enthaltene Verknüpfungen verwenden bzw. folgen.

HATEOAS: Hypermedia As The Engine Of Application State

9.3.7 RESTfull - Richardson Maturity Model (RMM)

Begriff um zu definieren, dass die wesentlichen REST-Prinzipien im REST-Stack eingehalten werden. Zur Beurteilung wird RMM herangezogen

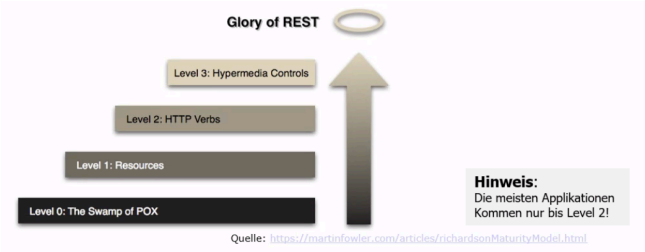


Abbildung 9.1: RESTfull - Richardson Maturity Model (RMM)

Es werden auch keine festen REssourcennamen oder Hierarchien vorausgesetzt. Server kann URI-Struktur jederzeit ändern, ohne den Client anzupassen. Client braucht nur Einstiegs-URI zu kennen.

9.3.8 Versionierung

REST-Schnittstellen können über URI-Pfadanteil relativ einfach versioniert werden. Dadurch paralleler Betrieb versch. Endpunkte möglich.