Software Development & Architecture

`I.BA_SWDA_MM.H25` — Zusammenfassung

Author(s) Dominic, Elias, Hannah, Ivo, Laura, Lukas, Manuel
Date 16. Mar. 2026
Pages 53

Inhaltsverzeichnis

1. Software Architektur 🏗️

1.1. Wichtige Begriffe

Begriff	Definition
Client/Server	• Zentraler Server bedient Anfragen von mehreren Clients • Request/Response
Message Orientierte Architektur	• Asynchrone Kommunikation über Nachrichten-Warteschlangen • Keine direkten Aufrufe zwischen Komponenten (Lose Kopplung)
Enterprise Service Bus (ESB)	• Zentraler Kommunikationsknotenpunkt (Bus) • Agiert als Vermittler von Services und leitet Nachrichten weiter • Konkrete Variante von MOA verstehen • Publish/Subscribe
Monolithische Architektur	• Gesamte Anwendung als **eine einzige, geschlossene Einheit** • Interne Module sind fest miteinander verbunden • „Weiche“ Modularisierung durch Code-Strukturen
Hexagonale Architektur	• Strikte Trennung von Anwendungskern und Aussenwelt (UI, DB) • Kommunikation über definierte „Ports & Adapter“
Service Oriented Architecture (SOA)	• Anwendung besteht aus einer Sammlung von wiederverwendbaren Diensten • Dienste sind lose gekoppelt und kommunizieren über ein Netzwerk

Mehrschicht-Architektur	• Gliederung in logische Schichten
	  ▸ **Geschlossen**: Nur Zugriff auf die Schicht direkt darunter.
	  ▸ **Offen**: Zugriff auch auf tiefere Schichten möglich.

Microservices	• Anwendung besteht aus vielen **kleinen, unabhängigen** Services • „Harte“ Modularisierung über das Netzwerk
Datenkapselung	• Bündelt Daten und zugehörige Methoden in einer Einheit (z.B. Klasse) • Schützt den internen Zustand vor direktem Zugriff (z.B. `private`) • Stärkt die **hohe Kohäsion**
Information Hiding	• Verbirgt die interne Implementierung eines Moduls („Wie“) • Nur die öffentliche Schnittstelle („Was“) ist von aussen sichtbar • Ermöglicht **lose Kopplung**
Modularisierung	• Aufteilung eines Systems in unabhängige, austauschbare Bausteine • Ziel ist eine **hohe Kohäsion** innerhalb der Module
Schnittstellen	• Definieren einen festen „Vertrag“ für die Kommunikation • Bilden einen schmalen, kontrollierten Zugangspunkt zu einem Modul • Erzwingen **lose Kopplung**, da nur der Vertrag bekannt sein muss

1.2. Erkenntnisse aus der Architekturübersicht

- **Schichtenmodell**: Client – Remote – Business – Data – Model
- Lose Kopplung durch DTOs & Interfaces
- Factories für Instanzierung & Abstraktion
- Domain Model als fachlicher Kern
  - Entities werden von mehreren Layern verwendet
  - Gefahr: starke Kopplung, besser über Schnittstellen abstrahieren
- UI und Backend unabhängig erweiterbar durch `remote`

- Persistenz via JPA, austauschbar durch klare Schnittstellen

1.3. Was ist Software Architektur?

- **Eine Abstraktion:** Sie stellt ein System zusammenfassend und vereinfacht dar, um dessen Komplexität zu bewältigen.
- **Definiert die Grundstruktur:** Sie beschreibt die fundamentalen Bausteine eines Systems (Subsysteme, Komponenten, Schichten) und deren Beziehungen zueinander.
- **Die schwer änderbaren Teile (nach Martin Fowler):** Architektur befasst sich mit den Kernelementen, die als Fundament dienen und später nur mit hohem Aufwand geändert werden können.
- **Signifikante Designentscheidungen (nach Grady Booch):** Eine Entscheidung ist dann architekturrelevant, wenn die Kosten einer späteren Änderung hoch sind.
- **Bauplan, nicht das Produkt selbst:** Die Architektur ist der Plan, nach dem ein Produkt gebaut wird – nicht das Produkt selbst.

1.4. Aspekte der Software Architektur

1.4.1. Grundlegende Struktur

- Aufbau durch Schichten, Systeme & Muster
- Äussere Struktur: Art der App & Deployment
- Innere Struktur: Prinzipien & Architekturmuster

1.4.2. Kommunikation & Verarbeitung

- Verteilbarkeit & Parallelität
- Synchrone oder asynchrone Kommunikation
- Transaktionen & Interaktion zwischen Systemen

1.4.3. Eingesetzte Technologien

- **UI-Varianten**: Fat-, Rich- oder Thin-Client
  - **Fat**: viel Logik im Client, kaum Serverlast
  - **Rich**: verteilt, UI + Teile der Logik im Client
  - **Thin**: fast alles auf dem Server, Client nur Anzeige
- Datenpersistenz mit Frameworks oder NoSQL
- Referenzarchitekturen als Vorlage

⟶ Immer Qualität im Blick

1.4.4. Applikationsarten (Beispiele)

- **Einzelbenutzer**: klein, lokal, simpel
- **Mehrbenutzer**: zentral, komplex
- **Internet**: verteilt, skalierbar

⟶ Architektur hängt von der Art der Nutzung ab

1.4.5. Entwicklung über Zeit

- **Früher**: Client/Server mit «FAT-Client» (Geschäftslogik im Client) ⟶ DB überlastet
  - **Vorteil**: zentrale Datenhaltung, hohe Konsistenz (RI, Trigger, SP)
  - **Nachteil**: schwierige Transaktionen & Ressourcen
- **Heute**: kleine lokale Apps sehr passend

⟶ Die Kunst: richtige Architektur am richtigen Ort

1.4.6. Architektur & Design

- Muster wie Schichten oder MVC (Model View Control) strukturieren
- Logische Trennung & klare Schnittstellen
- Herausforderung: passende Abstraktion + Konventionen

1.5. Systeme und Komponenten

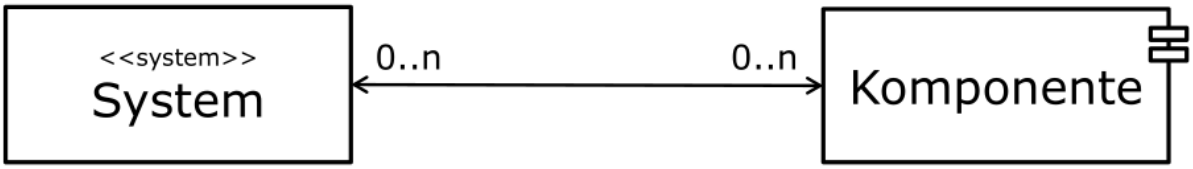
1.5.1. Softwaresysteme & Subsysteme

- Zerlegung in `Subsysteme` für Übersicht & Handhabbarkeit
- Prinzipien: **lose Kopplung** + **hohe Kohäsion**
- **Vorteile**: bessere Planung, unabhängige Entwicklung, Tests, Deployment, Wiederverwendung

Ziel: einfacher, verständlicher, schneller

1.5.2. Komponenten

- **Komponente** = `softwaretechnische Einheit`
- Systeme bestehen aus Komponenten (z.B. `Klassen`, `Interfaces`)
- Wiederverwendung in mehreren Systemen möglich
- **Orthogonalität**: `System` und `Komponente` sind unabhängig, aber kombinierbar



1.6. Monolithische Architektur

- Monolith = alles in **einer Anwendung** gebündelt
- Zu Unrecht oft im Verruf: wichtig ist Unterscheidung zwischen
  - **Monolithisches Design**: ohne Modularisierung ⟶ schlecht
  - **Monolithisches Deployment**: modularer Code, aber als Ganzes ausgeliefert

1.6.1. Monolithisches Design

- Codebasis hat sehr schlechte Struktur und Design -> hohe Koppelung, schlechte Struktur
- Wartung & Erweiterung stark erschwert
- auch gut modularisierter und strukturierter Code erodiert mit der Zeit
  - hohe Disziplin und Kontrolle notwendig

1.6.2. Monolithisches Deployment

- Auch modulare Systeme können monolithisch deployed werden
- **Vorteil**: einfaches Ausliefern („alles in einem Paket“)
- **Beispiel**: Mobile-App, grosse Web-App (WAR/EAR-File)
- **Nachteil**: Klumpenrisiko – kleine Änderung ⟶ komplette App neu deployen
- **Folge**: häufige Nichtverfügbarkeit, aufwändiges Konfigurationsmanagement

1.6.3. Fazit

- Monolith ≠ automatisch schlecht
- Schlecht = **fehlende Modularisierung**
- Deployment-Monolith kann sinnvoll sein, hat aber klare Grenzen
- Lösung bei Problemen: Anwendung **schrittweise teilen**

1.7. Verteilte Applikationen

Verteilte Systeme zerlegen Anwendungen in Teile (`Subsysteme`, `Komponenten`, `Module`, `Schichten`), die auf mehreren Rechnern laufen.

- Teile laufen in unabhängigen Prozessen
- Echte Parallelität

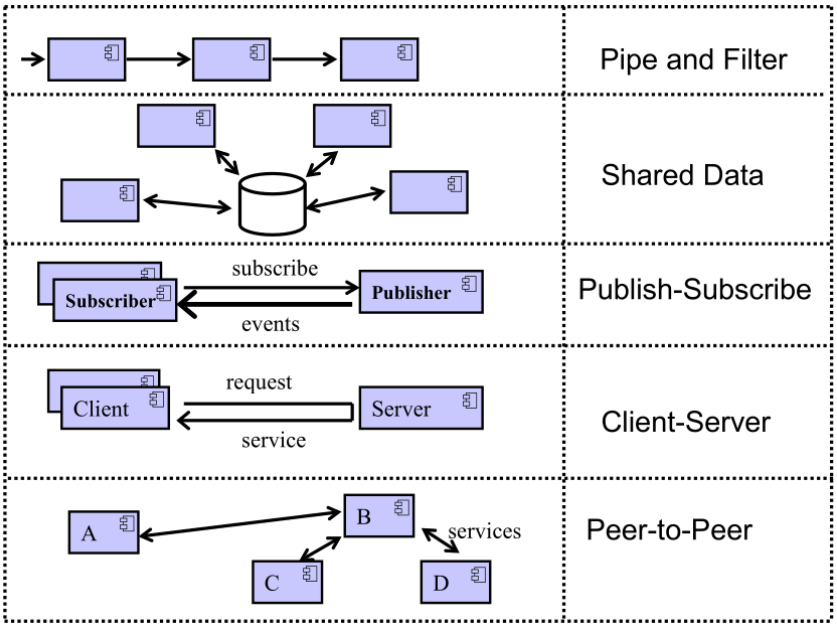
1.7.1. Anforderungen

- Kommunikation über geeignete Technologien
- **Binding**: Teile müssen sich finden und kennen
- Aufwändigeres Deployment & Koordination

- Umgang mit Ausfällen → **Resilienz**

1.7.2. Muster für verteilte Applikationen

- **Pipe and Filter** – Verarbeitungsketten
- **Shared Data** – gemeinsame Datennutzung
- **Publish-Subscribe** – lose Kopplung über Events
- **Client-Server** – klassische Aufteilung
- **Peer-to-Peer** – gleichberechtigte Teilnehmer



1.8. Modularisierung

Modularisierung strukturiert Softwaresysteme in klar abgegrenzte Einheiten, von Klassen bis hin zu ganzen Applikationen.

- Verfügen über wohldefinierte Schnittstellen
- Module sind einfach verständlich

Sie ist **fundamental für die SW-Architektur**.

1.8.1. Entwicklung

- 1972: Konzept von David Parnas
- 1978: Niklaus Wirth, Sprache **Modula**
- 90er: **Komponenten**-Begriff etabliert
- Beispiele: **EJB, Servlets, OSGi**

1.8.2. Kriterien

- **Kopplung**
  - Mass für die Abhängigkeit
  - **Möglichst gering** → weniger Abhängigkeiten
- **Kohäsion**
  - Mass für den internen Zusammenhalt
  - **Möglichst hoch** → starker interner Zusammenhalt
- Beides wird auf **allen Abstraktionsebenen** beurteilt, die Anforderungen werden dabei **zunehmend strenger**

1.8.3. Einfluss auf Architektur

- **Gruppierung**: Module mit gleichen Eigenschaften (z.B. Für Datenexport)
- **Hierarchie**: Rekursive Struktur, mehrere SubModule (z.B. Persistenzmodul)
- **Geschichtet**: Module können eine logische Kette bilden, Vertikale Ketten/Schichten (z.B. OSI-Modell)

1.8.4. Eigenschaften von Modulen

- **Explizite Schnittstelle**: klar und schmal
- **Starke Kohäsion**:
  - SoC (geschlossene Aufgabe)
  - SRP (Starken inneren Zusammenhalt)

daraus ergibt sich:

- **Information Hiding**: Implementierung verborgen, kann sich so auch ändern
- **Lose Kopplung**: Nutzung nur über Schnittstelle

1.8.5. Kriterien für den Entwurf

- **Verständlichkeit**: Module einzeln nachvollziehbar
- **Stetigkeit**: Beständig, wenig Anpassung nötig
- **Zerlegbarkeit**: Kleine, unabhängige Teile
- **Kombinierbarkeit**: Sinnvoll neu zusammensetzbar

1.9. Prinzipien für Modulentwurf

1.9.1. Modulkohäsion

1.9.1.1. `REP` – Reuse-Release-Equivalence

- **Granularität der Wiederverwendung = Granularität des Release**
- best practices (eigentlich kein Prinzip)
- Zusammenfassen, was gemeinsam veröffentlicht wird
- Beispiel: Maven-Modul als Release-Einheit
- Inkludierendes Prinzip (macht Dinge grösser)

1.9.1.2. `CCP` – Common-Closure

- **Klassen mit gleichen Änderungsgründen in eine Komponente**
- Unterschiedliche Änderungsgründe trennen
- Entspricht `SRP` (Single Responsibility Prinzip) und `OCP` (Open Closed Prinzip) auf Architekturebene
- Inkludierendes Prinzip

1.9.1.3. `CRP` – Common-Reuse

- **Keine Abhängigkeiten zu unnötigen Elementen erzwingen**
- Modul soll als Ganzes nutzbar sein
- Einzelteile ggf. ausgliedern
- Wichtig bei Libraries & Frameworks (transitive Abhängigkeiten)
- Exkludierendes Prinzip (macht Dinge kleiner)

1.9.2. Spannungsfeld REP / CCP / CRP

- `REP` erleichtert **Wiederverwendung**
- `CCP` erleichtert **Wartung**
- `CRP` reduziert **Abhängigkeiten**
- Prinzipien stehen teilweise im Widerspruch
- Klassischer **Trade-off**: Balance zwischen Wiederverwendung, Wartbarkeit und Abhängigkeiten

1.9.3. Modulkopplung

1.9.3.1. `ADP` – Acyclic Dependencies

- **Zwischen Komponenten keine Zyklen erlaubt.**

1.9.3.2. `SDP` – Stable Dependencies

- **Abhängigkeiten sollen in Richtung stabilerer Komponenten verlaufen.**

1.9.3.3. `SAP` – Stable Abstractions

- **Stabile Komponenten sollen im gleichen Masse abstrakt sein.**

1.9.4. Erkenntnis

- Ziel: hohe **Kohäsion**, niedrige **Kopplung**
- Module sind die **Grundbausteine der Architektur**
- Anforderungen ändern sich ⟶ Balance zwischen Wiederverwendung, Wartung und Abhängigkeiten verschiebt sich
- Gute Architektur bedeutet, diese Prinzipien sinnvoll auszubalancieren

1.10. Zwei einfache Architekturbeispiele

1.10.1. Beispiel 1: SLF4J – Simple Logging Facade

- Architektur nutzt Komponenten, Schnittstellen, Schichtenbildung und gezieltes Deployment
- Trennung zwischen API (Schnittstelle), Adapter und Implementierung

1.10.1.1. Erkenntnisse

- Einheitliche API für Logging (`Client/Server` -Struktur)
- Logging-Framework bleibt abstrakt ⟶ nur Laufzeitabhängigkeit
- Framework ist einfach austauschbar ⟶ Wechsel rein über Deployment möglich
- Beispiel für saubere Trennung von Schnittstelle und Implementierung

1.10.2. Beispiel 2: Schichtenarchitektur

1.10.2.1. Verantwortungen je Schicht

- `client` : UI/Controller, keine Geschäftslogik
- `remote` : API/Fassade, `DTO` -Grenze entkoppelt UI ↔ Backend
- `business` : Fachlogik über Interfaces (z. B. `CustomerManager`)
- `data` : Persistenzzugriff (z. B. `CustomerPersistor` , `JPA/Repository`)
- `model` : zentrale Domain-Objekte (`Customer` , `Address`)

1.10.2.2. Regeln

- Abhängigkeiten **nur nach unten**; nach oben via Interfaces umkehren
- Hohe **Kohäsion** je Schicht, **lose Kopplung** über klar definierte Schnittstellen
- Domain Model wird von mehreren Schichten genutzt → über Schnittstellen anbieten; Mapping zu `DTO` / `Repository` beachten

1.10.2.3. Deployment-Varianten

- **Variante 1**: Alle Schichten in einem Prozess (monolithisches Deployment) – klare Trennung bleibt intern erhalten
- **Variante 2**: `client` separat deployt; `remote` bildet Netzwerkgrenze (z. B. REST/RPC); `business` + `data` serverseitig – erleichtert Skalierung und Teamtrennung

1.10.2.4. Nutzen

- Austauschbare UI oder Persistenz, bessere Testbarkeit (Mocks an Interfaces), Wiederverwendung des Domain Models, klare APIs

1.10.2.5. Risiken

- Zu starke Kopplung ans Domain Model ohne Schnittstellen/DTOs
- Reine Durchleitungs-Fassaden erzeugen Overhead ohne Mehrwert

2. Architekturen und -muster 🏛️

2.1. Architekturmuster

DEFINITION: Beschreiben als Konzept den Grundaufbau eines ganzen Systems

Architekturmuster existieren für verschiedene zentrale Aspekte in der Softwarearchitektur:
- Strukturierung von (grossen, komplexen) Systemen
- für (stark) verteilte Systeme
- für int(er)aktive Systeme
- für hochverfügbare Systeme

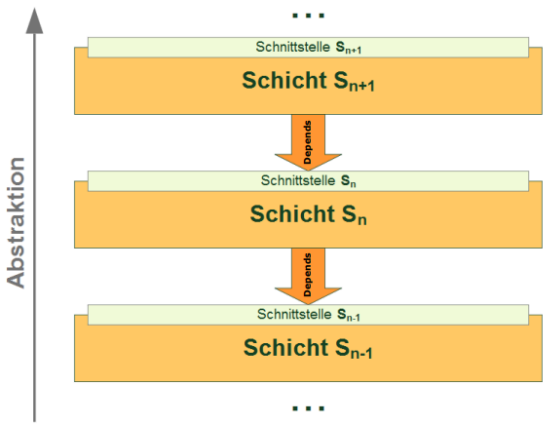
2.1.1. Beispiele von Architekturmustern

- Domain Logic Pattern: Domain Model, Service Layer
- Data Source Architektural Patterns: Data Mapper
- Object-Relational Behavioral Patterns: Unit of Work
- Object-Relational Metadata Mapping Patterns: Repository
- Web Presentation Patterns: Model View Controller,. Page Controller, Front Controller
- Distribution Patterns: Remote Facade, Data Transfer Object
- Base Patterns: Gateway, Mapper, Registry, Value Object, Special Case, Plugin, Service Stub

2.2. Schichtenbildung (Layering)

Bei der Schichtenbildung wird ein komplexes System in aufeinander aufbauende, funktional getrennte Schichten aufgeteilt.

Die **Kommunikation** dieser Schichten erfolgt über wohldefinierte **Interfaces / Schnittstellen**. Abhängigkeiten dieser Schichten ist nur in Richtung der tieferliegen Schicht (**kein Überspringen** von Schichten).



- Zur Strukturierung innerhalb eines Systems sowie auch Systemübergreifend
- Physich getrennte Schichten werden **Tiers** genannt.

2.2.1. Bildung von Schichten

Schichten können nach verschienden Kriterien gebildet werden:

- Logisch / Funktional
- Technik
- Abstraktionsebene

WICHTIG: Packages sollten nicht dazu verwendet werden um ein Unterteilung in Schichten zu implementieren. Packages dienen der logischen Gliederung und Kapselung

2.2.1.1. Package by Feature

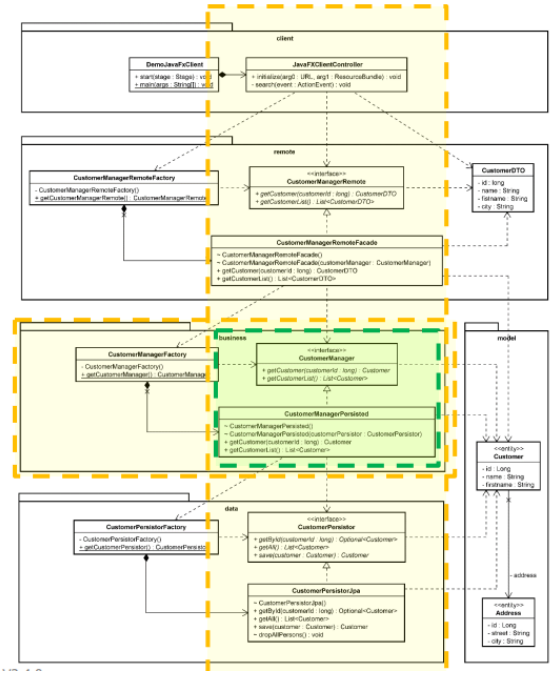
Als Alternative zur Schichtenbildung mittels Packages existiert Package-By-Feature.

Hier wird alle zu ein einem Feature gehörigen Code in ein Package gegliedert. -> Das Package enthält nun alle Schichten. –>**Vertikale Gruppierung (Slicing)**

Problematik Durch die Modularisierung in Java 9 darf ein Java Package nicht in mehreren JARs aufgebrochen werden

Lösung: Feingranulare Packages welche zuerst **vertikal** geschnitten und dann **horizontal** geschnitten werden.

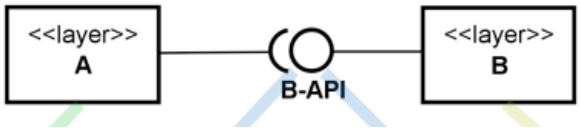
- ch.domain.system.customer.[view|business|customer]
- ch.domain.system.order.[view|busines|customer]



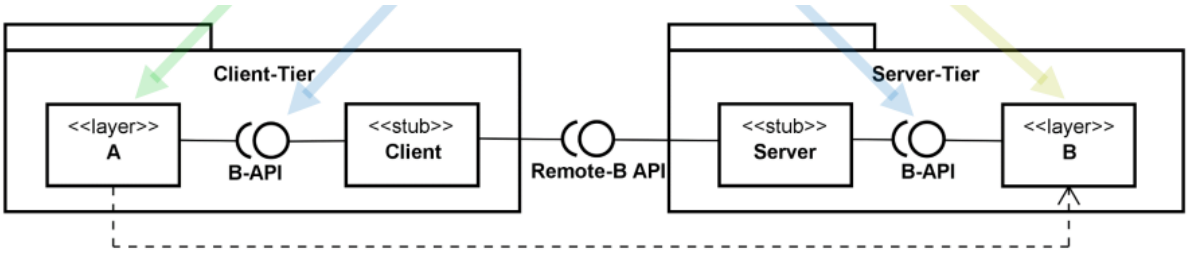
2.2.2. Tiers

Layering ist eine **fundamentale Grundlage** um eine verteilte Anwendung zu realisieren.
Schichtengrenzen eigenen sich sehr gut zur **physischen** Auftrennung und Verteilung.

Beispiel:



Man nehme die beiden Layers, separiert diese Physisch und setzt eine **Kommunikationsschicht** auf beiden Seiten **dazwischen**:



Durch diese Abstraktion ergeben sich einige **Vorteile**:

- Austauschbarkeit
- Flexibler
  - A & B könnte später wieder lokal kommunizieren
- einfacher zum Testen
  - Kommunikations-Stubs können einfach durch Fakes / Mocks ersetzt werden.
- A & B, sowie die Schnittstelle selbst bleibt gleich

2.3. 3-Schicht Architektur

Bei der 3-Schicht Architektur gibt es eine Aufteilung in drei **fundamentale** Schichten:

- **Präsentation**
  - Visualisierung, User Interface, UI-Logik
  - Bestellmaske für Artikel
- **Geschäftslogik**
  - Implementaiton der Geschäftsprozesse und -modelle
  - z.B. Ablauf einer Bestellung, Modell eines Artikels
- **Datenhaltung**
  - Persitente Datenspeicherung, Datenlogik
  - z.B. Speicherung der Artikeldaten in einem RDBMS

-> Diese Aufteilung lohnt sich **(praktisch) immer**, unabhängig von der physischen Verteilung (**SoC** -> **SRP** -> **Modularisierung**)

2.3.1. Verfeinerung Presentation-Layer

Bei einem Rich-GUI ist es möglich die reine Präsentation von der Präsentationslogik und den Daten noch stärker zu trennen.

- z.B. druch Model View Controller
  - Spezielle Modelle für Präsentation
  - Wiederverwendbare Views
  - Präsentationslogik im Controller
- Bei Thing-clients (Web/HTML) ist diese Trennung sogar zwinged notwendig.
  - View durch HTML/CSS
  - Präsentationslogik und Modelle
    - Im Client (z.B. JavaScript, Json, XML)
    - Im Server (z.B. Servlet, Pojo's, Json , XML)

2.3.2. Verfeinerung der Geschäftslogik

Weitere logische Trennung auch hier sinnvoll.

- **Businessfunktionen** -> Services
  - enthält Klassen für Geschäftsprozesse
  - arbeitet mit Business Model
  - z.B. **OrderService**
- Business Objects, Domain Objects -> **Domänen Model**
  - Reines objektorientiertes Modell, unabhähngig sowohl von Präsentation als auch Persistenz.
  - enthalten Daten und Methoden
  - Artikel, Kunde, Adresse

2.3.3. Verfeinerung der Datenhaltungsschicht

- Trennung / Abstraktion der reinen Datenlogik
  - Unabhängig vom physischen Datenmodell
  - Unabhängig vom verwendeten (R)DBMS
- O/R Mapping
- Abstracktion mehrere Backend (DBMS)-Systeme
- Transparente Einbindung von Legacy-Systemen
- Transaktionshandling

2.3.4. N-Schicht Architektur

Wenn man alle drei Schichten konsequent verfeinert erhält man etwa folgendes:

(Zufall dass alle 2 Schichten haben, könnte auch mehr oder weniger sein)

→ Bei **mehr als drei Schichten** spricht man von **n-Schicht-Architektur**
→ Aufwand und Komplexität steigt mit zuhnehmender Anzahl an Schichten.

2.3.5. Vor- und Nachteile

2.3.5.1. Vorteile

- Bessere Strukturierung -> besserers und schnellers Verständnis
- Grössere Chance auf Wiederverwendung
- Höhere Flexibilität
- Bessere Skalierbarkeit (primär vertikal)
- Einfachere und präzisere Planung/Schätzbarkeit
- Parallele und getrennte Entwicklung möglich
- Echte Zentralisierung der Businesslogik (Nur eine Stelle für Änderung)
- Skalierung durch Loadbalancing und Clustering
- Verschiedene Clients realisierbar

2.3.5.2. Nachteile

- Komplexität des ganzen Systems wird grösser
- Mehr Schnittstellen, mehr Aufwand, mehr Planung
- Schichten sind technisch -> Wo bleibt Fachlichkeit?
- Teilweise Redundanz auf Schichten (z.B. Validierung)
- Achtung vor **Dolchstoss** -> Datentypen bewusst an Schichtgrenzen brechen.

2.3.5.3. Bilanz

- Es kommt auf die Grösse an. 😉
- Im Kleinen funktionieren Schichten hervorragend
  - Herausforderung: Ab wann ist Interface sinnvoll / notwendig
- Im Grossen: Schnittstellen Pflicht
- **Zentrales Problem**: Nicht die Anzahl (oder die Höhe) der Schichten ist die Herausforderung, sondern zu erkennen, wann eine Schicht zu **breit** ist.
  - **Domäne** einer Applikation wird oft schlicht zu **gross** -> Schichten werden zu breit.

2.4. Service Oriented Architectur (SOA)

2.4.1. Services

Services kapseln **klar abgegrenzte Sub-Domänen** in eingenständige **verteilte** Dienste (Services), welche von übergeordnetetn Applikationen zur Realisierung eines Businessprozess genutzt werden.
-> Services sind **fachlich** und **technologieneutral**.

- Ein Service kommuniziert über wohldefinierte Schnittstellen

- Services sind in Verzeichnisdienste eingebunden und werden so dynamisch gesucht und gebunden (binding)
- Kommunikation kann über beliebige Protokolle erfolgen.

2.4.2. Potenzial

SOA erhöht die Abstraktion und Granularität, indem eine grosse Applikation in eigenständige Teile (Services) aufgebrochen werden.

-> Einzelne Services werden wieder kleiner und besser managebar.

Beispiel
Ein zu grosses System welches die komplette Warenbewirtschaftung mit Bestellung, Kunden, Artikeln und Lager verwaltet.
-> Kunden und Artikel haben wenig miteinander zu tun und bilden zwei lose Sub-Domänen.
-> Ein eigenere Service wird für Stammdaten erstellt.

SOA führt implizit auch zu **fachlicher Entkopplung**:
- Zwei Services benötigen teilweise dieselben Daten und beiden Wollen den Master Stand haben:

```text
|---------------| -----> |---------------|
| Applikation A |        | Applikation B |
|---------------| <----- |---------------|
```

-> **inakzeptale, zyklische** Abhängigkeit

Lösung: Der gemeinsame Teil wird wierderum in einen neuen Service ausgelagert

```text
|---------------|        |---------------|        |---------------|
| Applikation A | -----> | Service X     | <----- | Applikation B |
|---------------|        |---------------|        |---------------|
```

2.4.3. Gefahren durch SOA

Mit SAO wird eine Architekturebene errreicht, welche viele Interpretationen, Fehler und Missverständnisse erlaubt.

- SOA ist nur ein **Konzept** und losgelöst von konkreter Technologie.

2.5. Message- oder Event-Driven Architekturen

Mit Message oder Event-Driven Architekturen soll eine **möglichst lose Kopplung** erreicht werden. (Siehe Event/Listener Pattern)
Dabei muss man sich nur über die Semantik der Events einig sein.

- Bei SOA ist diese Abhängigket zwischen Services meistens über eine synchrone Kommunikation erreicht.
  - Jedoch existieren viele Szenarien in denen keine explizit synchrone Kommunikation erforderlich ist. **Hier können Messages oder Events in Queues oder Topics eingesetzt werden.**

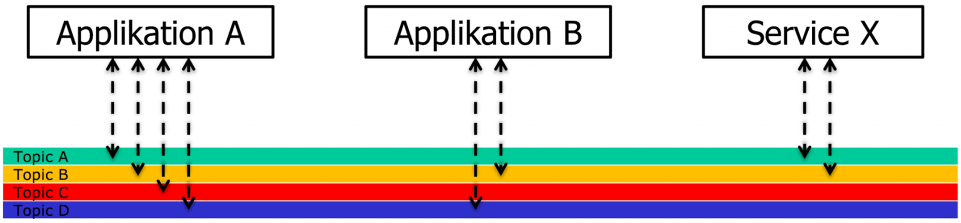
Persistente, klassische Datenbanken welche für „Informationsaustausch“ verwendet werden können **Message-Systeme** diese mindestens unterstützen (RabbitMQ, Kafka, MQSeries)

2.6. Enterprise Service Bus

Man kombiniert die Idee der **Services** mit **Messaging** und schliesst so verschiedene Systeme über einen **Message Bus** zusammen.

- Services registrieren sich für Topics
- Services publizieren Messages

-> Sehr lose Kopplung
-> Reaktion kann asynchron erfolgen



Messages können ein System wie folgt verbessern:

- Meistens asynchrone Kommuniation -> **schneller**
- Fire& Forget -> **einfacher**
- Queuing bei Ausfall -> **robuster**
- Neue Applikation können sich frei registrieren -> **flexibler**
- System kann elegant skalieren -> **leistungsfähiger**

Herausforderungen:

- Verfügbarkeit des Messagebus
- Welche Topics existieren
- Welche Messages / Events existieren
- Granularität der Messages

3. Messaging 📢

3.1. Monolith <> Microservice

- Single Tiered
- Komponenten sind zusammen paketiert
- Unabhängig von anderen Anwendungen

Vorteile
- Einfach zu deployen
- einfach zu betreiben
- Eine Codebase
- Cross-cutting-concerns (Logging, Security-Context, ...) können einfacher adressiert werden
- Gute Performance

Nachteile
- Fehlende Agilität
- Fehlende horizontale- und vertikale Skalierbarkeit
- Risiken in puncto Wartbarkeit
- Fehlertoleranz nicht optimal, ein Bug kann gesamtes System offline nehmen
- Stark an eine Technologie gebunden, wenig Flexibilität

3.2. Service

- In monolithischer Architektur haben wir unterschiedliche Methoden, welche einander im selben Memory auf der Maschine aufrufen.
- In einer MSA[1] haben wir nun eigenständige *Services*, welche
  - ein Interface haben
  - über eigene Business-Logik verfügen
  - Ihre eigenen Daten besitzen
- Bei einer MSA werden zudem keine Methodenaufrufe vorgenommen (kein RPC), sondern es werden Nachrichten zwischen unterschiedlichen Services versendet
  - Statt primitiven Datentypen wie `double` werden Message-Objects wie `Request` und `Response` verwendet, welche in unterschiedlichen Technologien serialisiert und deserialisiert werden können.
- In einer MSA versuchen wir, *synchrone Aufrufe* zu vermeiden und Nachrichten asynchron in einer Queue zur Verarbeitung zu platzieren
  - Antworten werden über eine entsprechende Response-Message verpackt und dem Requester in seiner Task-Queue zur Verfügung gestellt

3.2.1. Nachteile der synchronen Kommunikation

Verfügbarkeit Der Service muss für einen erfolgreichen Request zwingend verfügbar sein

Fault Tolerance Wenn der Service crasht, läuft der Aufrufer in ein Timeout und muss einen Umgang damit finden

Extensibility Point to point Kommunikation hat höhere Kopplung zum Effekt und macht demnach Systemänderungen teurer

Scalability Skalierung wird erschwert
- Einführung eines Load Balancers notwendig
- Wenn eine einfache Queue verwendet wird, so kann die Anzahl Konsumenten ohne Einführung eines Loadbalancers erhöht werden

3.3. Messaging Basics

- Advanced Message Queuing Protocol (AMQP) wird in SWDA als Protokoll verwendet, andere sind
  - REST / REST with HTTP/2 push
  - SOAP
  - RPC
- Begrifflichkeiten

[1] Micro Service Architektur

Message Informationseinheit welche über das System gesendet wird. Beinhaltet Header mit Metainformationen sowie Body in Binärformat

Producer Erstellt und versendet Nachrichten

Consumer Empfängt und liest Nachrichten

Queue Zwischenspeicher für Nachrichten, welche von einem Producer erstellt worden sind und auf eine Verarbeitung des Consumers warten

3.3.1. Vorteile von Messaging

Resilienz Nachrichten gehen bei Netzwerkausfällen nicht verloren und werden zwischengespeichert

Fault Tolerance Nachrichten können bei Serviceausfällen erneut zugestellt werden

Asynchronität Services können weiterarbeiten, auch wenn eine Antwort noch nicht erhalten wurde oder das Netzwerk überlastet ist

Entkopplung Services kennen einander nicht und sind entkoppelt

Effizienz und Skalierbarkeit Asynchrone Kommunikation ist eine Kerneigenschaft für die Fähigkeit zu skalieren. Durch Queues werden Producer und Consumer entkoppelt.

3.3.2. Nachrichten zentrisches Denken

- Grundprinzipien
  - Denk in Nachrichten statt in Services
  - Versuche nicht, den Datenfluss zwischen den Services zu definieren
  - Services sollten nichts voneinander wissen
  - Verteiltes Rechnen ist schwierig – Microservices und Messaging ändern daran nichts
- Systementwurf mit „Message First“ Ansatz
  - Businessrequirements definieren die Aktivitäten, welche in einem System passieren sollen
  - Nachrichten drücken Vorhaben und Aktionen aus
  - Anforderungen in Aktivitäten zu brechen helfen, Nachrichten in einem System zu finden

BEISPIEL: Wir haben einen Webshop, in welchem ein Kunde Produkte in seinen Warenkorb legen und kaufen kann. Nach dem Kauf wird dieser gespeichert, eine Bestätigungsmail wird ausgelöst und der Versand wird angestossen.

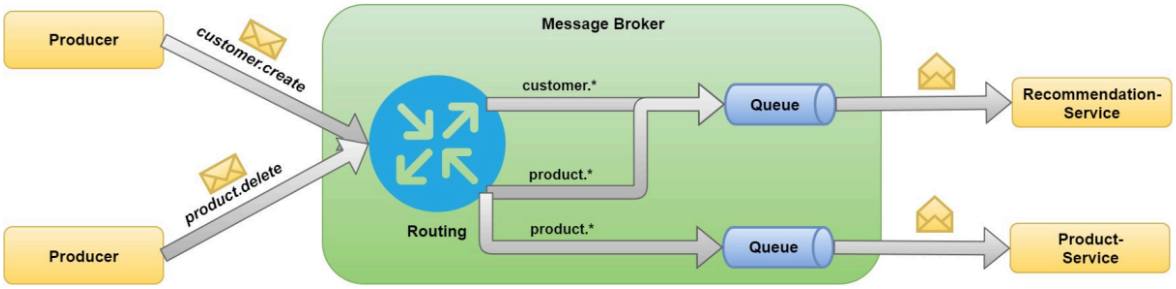
System-Messages könnten dabei z. B. folgende Sein

Activity description	Message Name	Message Data
Kaufabschluss	`checkout`	Warenkorbinhalt und Preise
Speichern des Kaufes	`record-purchase`	Warenkorb, Preise, Mehrwertsteuer, Total, Kundeninfos
E-Mail-Bestätigung	`checkout-email`	Recipient; cart summary; tempalte identifier
Auslieferung	`deliver-cart`	Produkte aus Warenkorb, Adresse

3.3.3. Sync vs Async

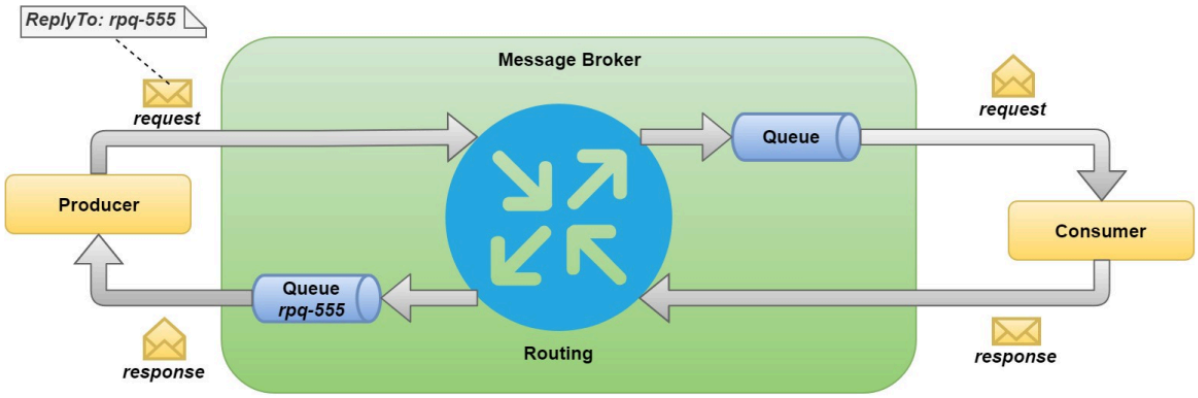
- Wir dann verwendet, wenn eine Bestätigung erforderlich ist
- Viele synchrone Szenarien können in asynchrone Szenarien umgewandelt werden

3.4. Routing



- Nachrichten werden über einen Message-Broker verteilt
- Routing geschieht basierend auf Attributen aus dem Nachrichten-Header
- Services kennen sich untereinander nicht
- Namespacing von Nachrichten erlaubt es, mit Pattern-Matching auf bestimmte Nachrichtentypen zu hören
  - Wird durch hierarchische Namensgebung vereinfacht, z. B. `customer.create` für die `create-customer` Message und analog `customer.delete` fürs Löschen.

3.4.1. Response



Response-Messages werden an eine in der ursprünglichen Nachricht vermerkten Return-Address gesendet

3.4.2. Zuverlässige Zustellung

- Unterschiedliche Gründe für das fehlschlagen der Zustellung einer Nachricht wie z. B. Netzwerkunterbruch, Hardwarefehler, …
- Messaging-Systeme können in solchen Fällen erneut zustellen, dazu müssen diese jedoch wissen, wann eine Nachricht empfangen worden ist und diese aus der Queue gelöscht werden kann
- Empfänger kann durch Senden eines **Acknowledgements** bestätigen, dass er die Nachricht erhalten hat
- Da auch ein Ack' verloren gehen kann, kann es zu mehrfacher Zustellung kommen und es wird die Auslieferungsgarantie **At least once** erreicht
- Die Funktion, welche auf der verarbeitenden Seite ausgeführt wird, muss dabei idempotent sein, damit bei doppelter Zustellung keine Seiteneffekte auftreten
- Ohne Acknowledgements wird eine Zustellung nach **At most Once** gewährleistet

3.5. Rabbit MQ

- Es gibt unterschiedliche Produkte, welche das Messaging zwischen Systemen ermöglichen und vereinfachen
- In SWDA wird hierzu RabbitMQ eingesetzt
- Unterstützt **AMQP** sowie andere Protokolle
- Wird von vielen Sprachen unterstützt, darunter Java, Python, Go, Rust, Scala, …
- Leichtgewichtige Open-Source-Software

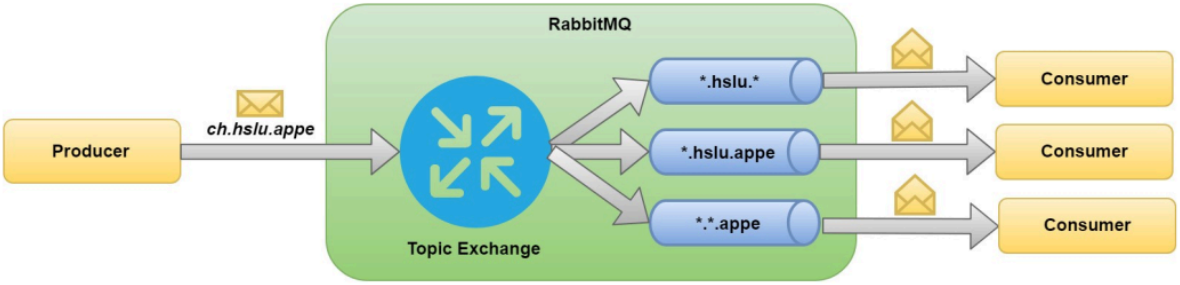
3.5.1. Bestandteile

Exchange Nimmt und verteilt Messages in Queues basierend auf Routing-Regeln
- **Direct** Queue-Name wird als Binding-Pattern verwendet
- **Fanout** Nachricht wird an alle angebundene Queues ohne Routing zugestellt
- **Topic** Routing Key in Kombination mit Pattern-Matching zur Verteilung von Messages

Queue Sammlung von Nachrichten, welche später von den Konsumenten empfangen werden

Bindings Konfiguration, welche Exchanges mit Queues ins Verhältnis setzt und Routingregeln enthält

In SWDA werden wir uns auf den **Topic-Exchange** fokussieren, da dieser die grösste Flexibilität bietet.



3.5.2. Verbindung

Um sich mit RabbitMQ verbinden zu können, werden zwei Komponenten benötigt.

Connection TCP-Verbindung, eine Verbindung pro Prozess, welche offengehalten werden sollte

Channel Virtuelle Verbindung innerhalb einer Connection, etwas leichtgewichtiger, sollten ebenfalls offengehalten werden

3.6. Patterns

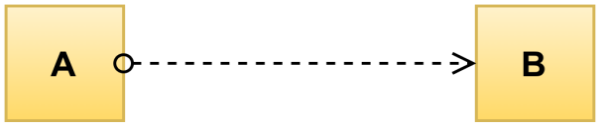
Routing mit Patternmatching erlaubt unterschiedliche Methoden zum Routen von Nachrichten.

Es gibt unterschiedliche Aspekte, welche ein Pattern charakterisieren, hier wird jedoch der Fokus auf die folgenden Eigenschaften gelegt:

Sync/Async Eine Antwort wird oder wird nicht erwartet

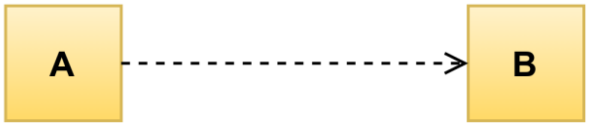
Observe/Consume Nachrichten werden nur beobachtet und andere können dieselbe Nachricht auch sehen, oder Nachrichten werden „konsumiert“ und stehen danach nicht mehr zur Verfügung.

3.6.1. Fire and Forget



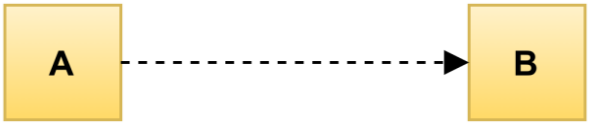
- asynchrone Kommunikation
- Nachrichten werden **nicht** konsumiert
- Sender erwartet keine Antwort
- Jeder Interessierte kann die Nachricht lesen
- Verwandt mit *Publish-Subscribe*

3.6.2. Winner take all



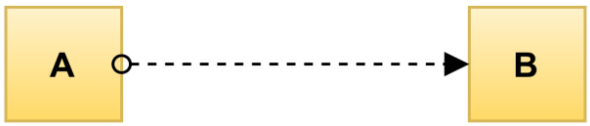
- asynchrone Kommunikation
- Nachrichten werden konsumiert
- Mehrere Zuhörer möglich, aber nur einer erhält eine Nachricht
- z. B. mehrere gleiche Worker, welche Arbeiten ausführen

3.6.3. Request / Response



- Synchrone Kommunikation
- Nachrichten werden konsumiert
- Sender erwartet eine Antwort
- z. B. HTTP / REST

3.6.4. Synchronous-Observed



- Synchrone Kommunikation
- Nachrichten werden nicht konsumiert
- Sender erwartet eine Antwort
- Andere können mithören, Sender ist aber nur an *einer* Antwort interessiert
- z. B. eine Nachricht über eine neue Bestellung, welche vom Versandservice beantwortet wird, wobei der Empfehlungsservice die Bestellung ebenfalls mitbekommt und Anpassungen bei ihm vornimmt

3.6.5. Dead Letter Queue

- Separate Warteschlange, in welcher Nachrichten platziert werden, welche nicht zustellbar sind

3.7. Skalierung

Wenn das Messaging-System starken Load erfährt, so kann es passieren, dass die Menge an produzierten Nachrichten von der verarbeitenden Seite nicht verarbeitet werden kann. Sofern in diesem Fall die Verarbeitungskapazitäten nicht erhöht werden, wächst die Menge an angestauten Nachrichten unendlich an.

Eine mögliche Lösung ist, auf der Konsumentenseite weitere Instanzen hinzuzufügen.

Eine andere Strategie ist, durch „Backpressure-Strategien“ die aufkommenden Nachrichten zu reduzieren:

Synchronous Backpressure Sender wartet, bis Empfänger bereit ist, weitere Daten zu verarbeiten. System kann signalisieren, dass Queues aktuell voll sind.

Load Shedding Nachrichten werden bei zu hoher Last gezielt verworfen oder keine neuen angenommen.

Flow Control Datenflüsse werden zwischen Sender und Empfänger automatisch reguliert.

4. Microservices 🌐

4.1. Grundlagen und Technologien

4.1.1. Definition

- kleinere Services
- Domäne wird unterteilt in kleine Teildomänen
- harte Modularisierung
  - unerlaubte Kopplung wird sofort sichtbar
- **Einheit ist kleiner und schlanker**

Definition Martin Fowler:
1. Eine Applikation wird aufgeteilt in mehrere kleine Services
2. Jeder Service läuft in einem eigenem Prozess
3. Leichtgewichtige Kommunikation
4. Microservices sind unabhängig voneinander deploybar
5. Deployment ist automatisiert

4.1.1.1. Aufteilung in mehrere kleine Services

Die verschiedenen Teile sollen möglichst autark arbeiten können und auf eigenen Datenmodell zurückgreifen -> daher (primäre) **vertikale** Aufteilung

-> Aufbrechen des Domänenmodelles in „bounded context“

Teile sollten **nicht** direkt miteinander kommunizieren:
- Ansteuerung über GUI oder vorgelagerten Gateway

Grösse: Applikation >> Microservice >= Modul

4.1.1.2. Service hat eigenen Prozess/Plattform

Unterschiedliche Plattformen (OS, Programmiersprache) sind möglich, sie müssen nicht über die gesamte Applikation identisch sein

-> **Container**

Services laufen parallel:
- Herausforderung von verteilten System (Netzwerk, Latenz, Skalierung, Ausfall)
- Höhere Performance durch Asynchronität oder nebenläufige Synchronität beim Client

Komplexität ist höher

4.1.1.3. Leichtgewichtige Kommunikation

JSON basierte REST Schnittstellen sind populär, aufgrund ihrer Einfachheit

Effiziente, Binäre Protokolle wären leichtgewichtiger

https basierte Kommunikation ist einfach zu implementieren und automatisch testbar
- Deckt authentifizierung und Verschlüsselung ab
- Protokolle sind bewährt und bekannt

4.1.1.4. Unabhängiges Deployment und unabhängige Releases

Microservices sind eigenständige Pakete und Releaseeinheiten

- durch gemeinsame Orchestrierung werden sie zu einer Applikation
- OCP -> Flexible Entwicklung von kleinen Einheiten, keineres Risiko bei Änderungen und Erweiterungen

Microservices können einzeln redeployt werden

Häufigere Ausfälle

- Applikation muss mit nicht verfügbaren Services umgehen können
- Resilienz ist wichtig

4.1.1.5. Automatisiertes Deployment

Monolithische Applikation mussten oft (halb-)manuell deployed werden

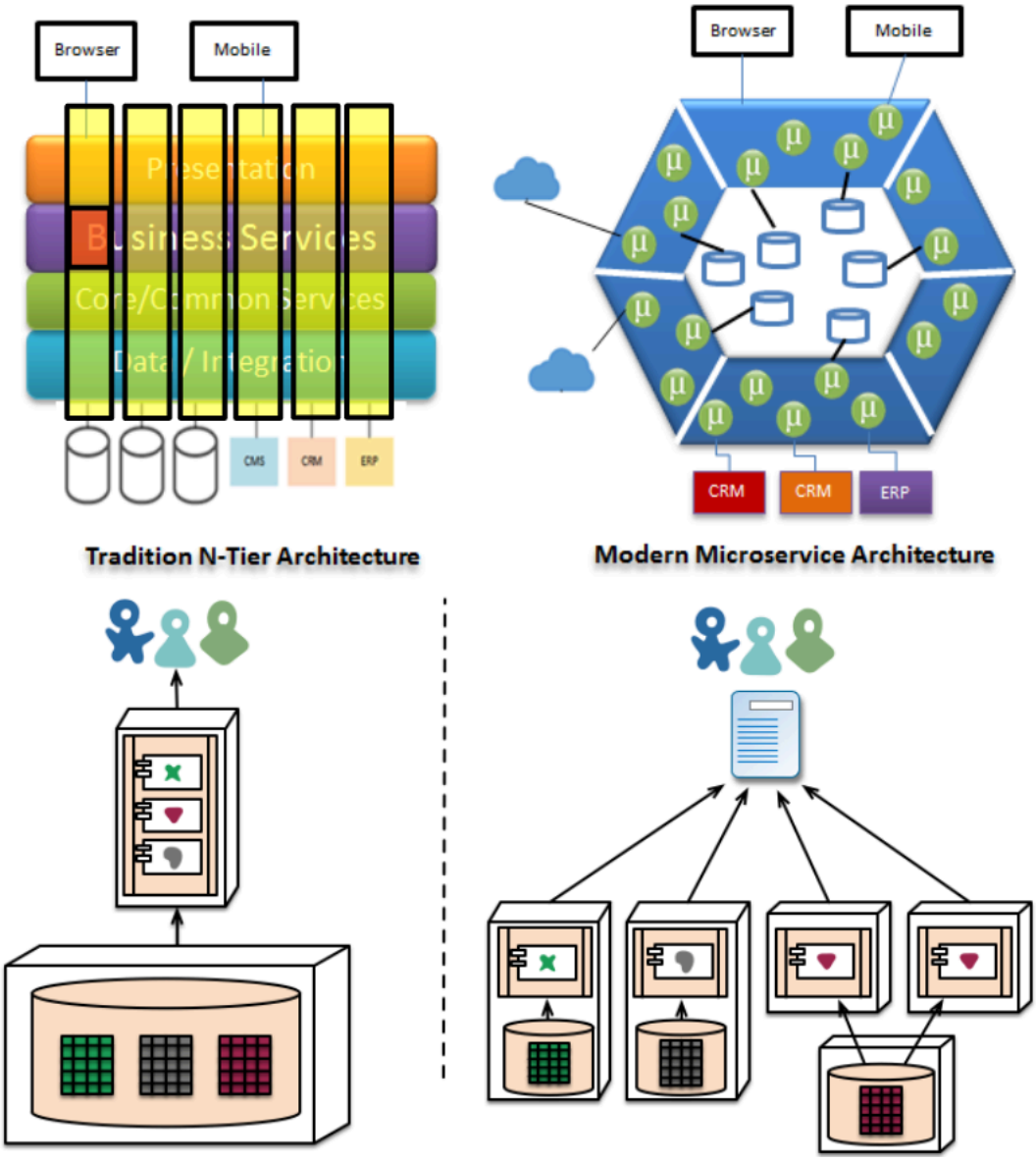
- fehleranfällig, dauert lange

Manuelles Vorgehen ist nicht praktikabel für Microservices

- Dauer und Komplexität des Deployments

4.1.2. Herausforderungen und Potential

- Schichten -> horizontal
- Microservices -> vertikal



Entscheidend ist die Grösse der Deployment Einheiten -> Abhängigkeiten zwischen den Microservices sollten minimal sien

4.1.3. Schnittstellen und Kommunikation

Herausforderung: Abhängigkeiten durch Kommunikation

Ziel: Möglichst minimale Kopplung

Microservices forcieren die Modularisierung, Herausforderungen werden ins Deployment verlagert.

4.1.3.1. Kommunikationsarten:

Client zu Service z.B UI mit Microservice

- leichtgewichtige Kommunikation
- Mehrheitlich synchrone Kommunikation (Feedback wird erwartet)
- Nutzung von Gateway-Patterns welche eine einheitliche Schnittstelle anbieten

Gateway Pattern:

- Entspricht dem Konzept der Fassade
- Übernimmt Authentifizierung und Verschlüsselung
- Kann einfache Mappings/Transformation von Daten vornehmen
- Wird explizit als Service Implementiert

Service zu Serivce Kommunikation zwichen Services

- Möglich, sollte aber vermieden werden (starke Kopplung)
- Queues und Topics nutzen
  - Ansynchron wenn möglich
  - Events/ Messages und Commands
- Neue Services können Funktionalität dynamisch ergänzen

4.1.4. Resilienz

DEFINITION: Fähigkeit von technischen Systemen bei Störungen/Teilausfällen nicht völlig zu versagen, sondern wesentliche Systemdienstleistungen aufrechtzuerhalten

-> Eine Applikation sollte damit umgehen können, wenn einzelne Microservices temporär nicht verfügbar sind

Ausfallszenarien:

Szenario 1 Service ist nicht erreichbar

- Fehlersituation wird schnell erkannt -> Abbruch

Szenarion 2 Service arbeitet langsam

- Verbindung kann aufgebaut werden, Aufrufer ist länger blockiert, kein Timeout
- Problem schaukelt sich hoch

Um Szenario 2 zu verhindern kann man einen **Circuit Breaker** einsetzen

Circuit Breaker

- Transparenter Proxy
- Überprüft wie lange Requests im Schnitt dauern
- Wird Zeitlimit überschritten -> neue Requests werden abgebrochen
- Nach einem Timeout öffnet siche der Proxy wieder und prüft

-> Vorteil: Wenige Ressourcen werden verschwendet bei einem Ausfall

4.1.5. Technologien und Framworks

- HTTP Server direkt in Applikation integriert
  - Jetty
  - Dropwizard
  - Jakarta EE
    - Schlanke Runtimes, geringer Memory-Footprint, schnelle Startzeiten, kleine Deployments

Frameworks

- Springboot:
  - Vereinfacht Erstellung von Microservices stark
  - Schwäche: Basiert auf Laufzeit-Konfiguration
- Micronaut.io:
  - Spezialisert auf Microservice
  - Compile-Time Annotation (kleinere Runtime, schnellere Startzeiten)
  - Dynamische Entwicklung

    - Starke und flexible Integration von Infrastrukturdiensten
- Quarkus ❤️
    - Kubernetes Stack auf OpenJDK oder GraalVM Applikation
    - Kompiliert Applikation
    - Optimiert für Kubernetes

Nativ komilierte Java-Anwendungen

- Basis: GraalVM
    - Alternative Runtime VM
    - Mehrsprachig
    - Applikation kann native erzeugt werden
    - Startzeit wird extrem reduziert

4.1.6. Zusammenfassung

Vorteile

- Überschaubar, unabhängig und gut wartbar
- Können relativ schnell und flexibel angepasst werden
- Können von kleine Teams entwickelt werden
- Sind für/in/mit unterschiedlichen Plattformen, Sprachen und Technologien realisierbar

Nachteile

- Hohe Anforderungen an das Operating
- Höhere Gesamtkomplexität durch Asynchronität und Resilienz
- Neue Herausforderungen durch cross-cutting-concerns wie Sicherheit, Überwachung, Monitoring und Logging

4.2. Herausforderungen von Microservices

4.2.1. Deploymentanforderungen

- Möglichkeiten bei klassischen Services:
  - Eigene, dedizierte HW
  - Voller VM Stack mit eigenem Betriebssystem
  - Individuell gepflegte, spezifische Docker-Container
- Microservices:
  - Ressourceverbrauch zu hoch wenn pro Service nur ein Prozess laufen soll
  - Arbeitsaufwand für Erstellung, Wartung und Betrieb zu hoch
  - DevOps zwingend

4.2.2. Deployment von Java-basierten Services

- Klassische Applikationscontainer (Java EE, WAR oder EAR) sind für das Deployment von mehreren Services ausgelegt und bieten Infrastrukturdienste an
  - konsumieren viele Ressource
  - lange Startzeiten
- Java kann Microservices seit Java 8 (2017). Davor:
  - Java Runtime ca. 40 MB gross
  - Memory Management für grosse, langlaufende Applikation ausgelegt

4.2.3. Umgang mit Transaktionen

Microservices und Transaktion vertragen sich nicht

- Transaktionen verursachen eine enge Kopplung

Lösung: Transaktionen aufbrechen und mit SAGA Pattern arbeiten

- zu stark getrennte Services wieder zusammensetzen
- Transaktionen minimalisieren

Monolithische Architekturen: Transaktionen einfach

Servicebasierende Architektur: Transaktionen auf den Service beschränken oder die globalen Transaktionen vom Container nutzen

Microservices: keine gemeinsamen Container mehr…

Teile eines strikt synchronen Ablaufes werden herausgebrochen und neu asynchron, unabhängig und zu einem späteren Zeitpunkt ausgeführt

4.2.4. Logging, Metrics und Tracing

In klassischen Applikationen

1. Durchgängiges Logging von fachlichen und systemtechnischen Ereignissen
2. Durchgängige Verfolgbarkeit von Prozessen über mehrere Schritte in verteilten Services und Systemen
3. Überwachung der Performance und der Ressourcen zur Feststellung von Engpässen ind einzelnen Services oder bei Teilausfällen

Grund für die Einfachheit:
- Wenige Architekturmittel zur Entkopplung existieren
- Homogene Plattformen, Sprachen und mit einheitlichen Libraries und Frameworks
- Architekturbedingte Zentralisierung der Prozess und Daten
- Application-Server verfügen über entsprechende Infrastrukturdienste

Microservices

- nicht sequenziell, asynchrone Aufrufe
- Entkopplung über Queues
- Services können Ausfallen oder Stören

Herausforderung:
- Logging findet auf verschiedenen Systemen mit verschiedenen Sprachen statt
- Stark verteilte Datenhaltung und Netzwerkinfrastruktur
- durch lose Kopplung verliert man die Nochverfolgbarkeit von Businessprozessen

4.2.4.1. Logging

Problematik: Angriffe werden mangels Überwachung und Logging gar nicht oder erst zu spät bemerkt

Empfehlungen:
- Konzept festlegen:
  - Was wird geloggt?
  - In welchem Format wird geloggt?
- Eigene Library ist nur beschränkt sinnvoll
- Einhaltung von Konventionen verlangt sehr hohe Disziplin der Entwickler
- Aufpassen bei Logging-Fassaden

Beispiel: log4j
- Konfiguration über XML Datei
- Flexibles Konzept der Appender
- Anbindung an Clouddienste möglich

Möglichst gut lesbar aber trotzdem maschinell auswertbare Logformate verwenden

4.2.4.2. Metriken

Explizit in die Software eingebaute Messpunkte welche ein Rückschluss auf die Leistung eines Systems erlauben

Arten von Messpunkten:
- Zähler (counter): Anzahl Ereignisse
- Messwert (gauge): Absoluter Wert
- Meter (meter): Messwert pro Zeiteinheit
- Historgramme: Statistische Verteilung von Messwerten

Grösste Herausforderung:
- Auswahl eines geeigneten Messpunkt für eine bestimmte Grösse in einem System
- Betrifft sowohl Art als auch Ort des Messpunktes

Java Beispiel: Metrics:
- Zentrale Registry, Identifikation über Strings
- Verschiedene Metriken
- Vorbereitete Metriken für einige bekannte Frameworks
- Diverse Adapter zur Extrahierung der Messdaten

```java
// Zentrale Metrik-Registry aktivieren…
final MetricRegistry metrics = new MetricRegistry();
// …und eine konkrete Metrik (meter) registrieren.
metricPrimeProbe = metrics.meter("ch.hslu.swe.primefinder.prime-probe");
…
// Ein Ereignis (Metrik) markieren (typisch n-fach).
metricPrimeProbe.mark();
…
// Manuelle Auswertung / Abfrage der Metrik (z.B. über Logging!)
LOG.info("Generate[Total:{}, Rate:{}/s]",
            metricPrimeProbe.getCount(),
            metricPrimeProbe.getMeanRate());
```

Beispiele für Auswertung und Visualisierung:
- Prometheus
- InfluxData
- Grafana

Empfehlungen:
- Vorhandene Tools und Werkzeuge nutzen
- Häufig werden solche Services direkt in Frameworks integriert
- Vernünftigen Einstieg wählen:
  - Aussagekräftige Werte, sukzessive Erweiterung
  - So viel wie nötig, so wenig wie möglich
  - Achtung: Metriken kosten auch Laufzeit
- Auswertung and Dritt-Tools delegieren
- Metriken sind kein Ersatz für detailliertes Profiling

4.2.4.3. Tracing

- Mit Asynchronitität und Queues ist die Nachverfolgbarkeit von einem Prozess sehr schwierig
- Analyse der Logs aufgrund vom fehlenden Zeitkontext schwierig
- Lösungsansatz: Jede Request bekommt eine Correlation-ID
  - Unabhängig vom zeitlichen und örtlichen Kontext wird ein Request nachvollziehbar

Correlation IDs mit UUID:

- Ziel: Informationen in verteilten Systemen eindeutig zu kennzeichnen
- 16-Byte-Zahl die hexadezimal notiert und in 5 Gruppen unterteilt wird
- Beispiel: `java.util.UUID` -> `4e6abea6-d18a-49c1-a7b0-4f57702e6602`
- Schnelle Generierung möglich, typisch kein Problem

Empfehlungen:

- Correlation IDS nicht auf Ebene der fachlichen Schnittstelle einfügen
- Für Remote Schnittstellen in den Kontext verschieben
  - z.B. http Header
  - reine Infrastrukturlösung
- Technische Correlation IDs einsetzen wenn kein verwendbares, eindeutiges, fachliches Attribut vorhanden ist
- Einfache, simple, schnelle Formate verwenden

Tracing Server

- Erlauben es fachliche Prozesse über das ganze System zu Verfolgen und zu analysieren
  - „Langläufer“, Hotspots und „Flaschenhälse“ Identifizieren

Produkte:

- Zipkin
- Jaeger

Architekturentscheidung: Einzelne Dienste nutzen doer Konsolidierung auf eine Technik

4.2.4.4. Service Discovery

Klassische Architektur:

- IP Adressen und Ports sind fix
- Unterhalt verschiedener Umgebungen über ausgelagerte Konfiguration
- Applikationsserver verfügen über eigene Namensdienste

Microservices:

- Viele Dienste
- Docker: Port Mapping, Netzwerke
- Spezielle Dienste: z.B Consul

Service Discoveris vergleichbar mit einem lokalen DNS -> Logische Namen werden einem Service (IP und Port) zugewiesen

- auch eine Art Load Balancing möglich

5. API Design und REST

5.1. Einleitung - API

DEFINITION: Ein API ist eine Schnittstelle die explizit darauf ausgelegt ist, eine Softwareeinheit (Modul, Library, System) nutzbar zu machen.

- Soll eine möglichst **einfache Nutzung** ermöglichen.
- Soll den **Aufwand** für die Nutzende **minimieren**.
- Soll **Unabhängigkeit** von einer konkreten Implementation bieten.
- Hat ein implizites Mass an Öffentlichkeit.

Die API wird typisch von Nutzenden verwendet und von Anbietenden implementiert. Beispiele:
- JPA (Java Persistence API) als Schnittstelle zur Persistierung.
- SLF4J (Simple Log Facade for Java) als Schnittstelle für Logging.

Eine API besteht typisch aus mehreren Klassen / Interfaces.
- Definiert gemeinsame Funktionen (Interfaces) und Datentypen.

5.1.1. Wann wird eine Schnittstelle zum API?

- Eigentlich nur eine Frage des Scopes
- Eine (rein technisch betrachtet) identische Schnittstelle kann:
  - Im (Detail-)OO-Design in einer Implementation existieren.
  - Die Schnittstelle zu einer Komponente darstellen.
  - Die Schnittstelle zu einem Framework darstellen.
  - Die Schnittstelle zu einem Teilsystem darstellen.
  - Die Schnittstelle zu einem vollständigen System darstellen.

-> Abstraktion nach unten

Die Definition welchen Scope, und somit welche «Wichtigkeit» eine Schnittstelle hat, erfolgt auf **Architektur-Ebene**. (Somit nicht primär technisch getrieben.)

Manchmal werden Schnittstellen zu APIs (gemacht/genutzt) die nicht wirklich dafür konzipiert wurden.

5.1.2. Motivation für gute APIs

Gute APIs haben einen grossen Wert
- Fördern die Modularität und Entkoppelung
- Machen komplizierte Dinge viel einfach nutzbar.
- Erlauben Implementationen einfacher auszutauschen
- machen Produkt attraktiver für Benutzende
- binden (im positiven Sinne) die Nutzenden
  - Nutzende investieren z.B. selber in eine API (Schulung, Entwicklung)

Schlechte APIs verursachen Ärger und Kosten
- Komplizierte, missverständliche Nutzung, aufwändige Fehlersuche, Support.
- Verführen dazu, eine bereits existierende, gute Lösung nicht wiederzuverwenden, sondern es abermals selber zu machen.
- Verführen zu komplizierten Umwegen und unnötigen Fassaden oder Adaptern.

5.1.3. API - Herausforderungen

- öffentlich verfügbare APIs leben (fast) ewig
  - somit leben Fehler auch ewig
  - APIs sind nur mit grossem Aufwand / Konsequenzen änderbar
  - meist nur eine Chance es richtig zu machen

TL;DR: Es ist viel einfacher ein schlechtes API zu erkennen, als ein wirklich gutes API selber zu entwerfen!

Grosse Kunst: APIs so entwerfen, dass sie entwicklungsfähig ist (KISS- and YAGNI-Prinzipien anwenden)
- KISS: Keep it simple stupid
- YAGNI: You Aren't Gonna Need It

5.1.4. Hauptziele einer guten API
- Maximale Entkopplung (lose Kopplung) der Implementation.
- Maximales Information Hiding (keine Implementationsdetails veröffentlichen).
- Maximale Kohäsion, in sich schlüssig konzipiert.

-> Prinzipien wiederholen sich auf anderer Ebene Gefahr bei Fehlern ist viel Grösser!

5.1.5. API - Qualitätsanforderungen - Empfehlungen
- **Konsistenz**: Namensgebung und -regeln konsequent einhalten.
- **Namensgebung**: Einfache, eingängige, treffende Namen wählen.
- **Verhalten**: Klare Erwartungen erfüllen, ohne Nebeneffekte.
- **Erweiterbarkeit**: Offen für Weiterentwicklung (der API).
- **Dokumentation**: Einfache, hilfreiche, kompakte Dokumentation.
- **Perspektive** (Anbieter vs. Nutzer): Einfache Nutzung.
- **KISS-Prinzip**: Schnittstelle möglichst einfach halten.
- **Sicherheit**: Die Nutzung ist sicher zu gestalten.

5.1.6. API - Patterns
Werden verwendet, um in APIs die Namensgebung und das Verhalten möglichst vorhersehbar und durchgängig zu gestalten.

Beispiele:
- **Repetition**: Verwenden immer den gleichen Namen für die gleichen Dinge.
- **Periodisch**: Verwenden immer die gleiche Bezeichnung für gleiches Verhalten.
- **Symmetrie**: Biete wenn möglich symmetrische Methoden an, z.B. für das Öffnen und Schliessen, oder das Belegen und Freigeben.

5.1.7. API - Mehr als nur Nutzer und Implementation
- eine API hat nicht nur eine Art von Nutzende, auch die Anbietenden sind Nutzende der API
- Beispiel: JDBC als API zu Datenbanken
  - als Nutzende verwendet man nur die Schnittstelle
  - konkret: man nutzt JDBC für den Zugriff auf eine Datenbank
- häufig gibt es mehrere Implementationen für ein und dieselbe API
  - Beispiel 1: JDBC wird von verschiedenen DB-Herstellern implementiert, es gibt unterschiedliche Treiber für diverse Datenbanken
  - Beispiel 2: Die SLF4J Logging-Fassade bietet verschiedene (Adapter-)Implementationen für unterschiedliche Logging-Frameworks an.

5.1.8. SPI - Die „API" für den Anbieter einer Implementation
- Service Provider Interface (SPI): eine spezielle Teilmenge oder Ergänzung zu einer API
- SPI ist für die Anbieter von Implementationen einer API gedacht
  - erlaubt z.B. sich in einer Factory der API zu registrieren
  - oder verschiedene (Basis-) Konfigurationen anzubieten
- SPI muss nicht zwingend Class-based sein, sondern kann auch:
  - eine Datei-Schnittstelle sein Property-Datei, JSON, YAML etc.)
  - eine Menge von (Environment-)Variablen oder Properties sein.

–> Die bewusste Abgrenzung durch ein explizites SPI verbessert die Gesamt-API und schützt sie diese massgeblich vor falscher Nutzung.

5.2. Class - based API

DEFINITION: API die über Interfaces und Klassen realisiert werden (1..n).

- typisch in einem Package bzw. Modul abgeschlossen
- vollständige und saubere Dokumentation (JavaDoc)

Gefahr: Klassen, die als formale Parameter oder Return genutzt werden, sind als mitgelieferte Hilfsklasse (z.B. Factories) automatisch Bestandteil der API

Geschichtete Architektur: häufig ein API Layer (z.B. Service-Layer nach Fowler)

Wichtig: Datentypen zwischen den Schichten brechen -> pro Layer eigene Datentypen: VO (Value Objects), DTO (Data Transfer Objects) [[DTO - Data Transfer Object]] -> sonst entsteht eine Koppelung über / zwischen APIs

5.2.1. Empfehlungen

- möglichst einfach kurze Parameterliste (clean code)
- für Rückgabetypen wenn immer möglich auch Interfaces nutzen
- ab drei Parametern ein Builder-Pattern in Betracht ziehen (Effective Java, Item 2)
- API-Stil an die Bedürfnisse / Sprache / Situation anpassen
  - Usage First - API aus der Perspektive des Nutzers entwerfen
  - Test First -> eat your own dog food

5.3. Beispiel: Student API - CRUD Operationen

```java
public interface StudentService {
  Student getById(long id);
  List<Student> findByLastName(String lastName);
  List<Student> getAll();
  Student create(Student student);
  boolean update(long id, Student student);
  boolean delete(long id); }
```

CRUD= create, read, update, delete

Naming:
- get: wenn man wirklich etwas erhält
- find: es kann auch eine leere Liste geben

Optional: evtl. bei getById

5.3.1. Java API - `null`, `Optional`, empty-list oder Exceptions?

```java
Student getById(long id
```

- die Methode **muss** etwas finden
- `null` oder Exception ist absolut legitim
- **nicht** `Optional` verwenden

```java
List<Student> findByLastName(String lastName)
```

- keine Daten sind ein Normalfall
- leere Liste (empty-list) ist die korrekte Wahl
- **nicht** `null` oder Exception

5.3.2. Java API - Verwendung von `Optional`

- seit Java 8
- `Optionals` machen explizit sichtbar, dass die Existenz eines Wertes optional ist
- **nicht** für API- oder Remote-Schnittstellen konzipiert
  - Optionals können z.B. schlecht als JSON serialisiert werden
  - Optionals sind gegen aussen „Implementationsdetail“

5.4. Rest API (Representional State of Transfer)

DEFINITION: REST ist ein Architekturstil und definiert einen Satz von Prinzipien, welche eine REST-konforme Architektur einhalten soll.

Die fünf REST-Prinzipien sind:
1. **Statuslose** Kommunikation (Zustandslosigkeit).
2. Verwendung von (http(s)-)**Standardmethoden**.
3. **Ressourcen** mit eindeutiger Identifikation (URI).
4. Unterschiedliche **Repräsentationen** von Ressourcen.
5. **Verknüpfungen** / Hypermedia (HATEOAS).

Eine REST-konforme Architektur basiert zwangsläufig auf einer verteilten Client/Server Architektur, meist mit http(s)-Protokollen. RESTfull nach RMM(Richardson Maturity Model) Level 3 verlangt aber eigentlich, dass unterschiedliche Protokolle genutzt werden können.

5.4.1. REST - Zustandslosigkeit

- Kommunikation **muss** zustandlos sein
  - Ein (client-)spezifischer, auf dem Server temporär gehaltener Zustand über die Grenze eines Requests hinaus ist nicht erlaubt (vgl. «Session»).
- Ein Zustand muss entweder vom Client gehalten, oder (viel besser!) vom Server in einer veränderten **Ressource** abgebildet werden.
- Der Status der Applikation ergibt sich ausschliesslich aus der Ressourcenrepräsentation (URI) und dem Ressourcenstatus.
  - Es dürfen somit keine Sitzungsinformationen existieren.
  - Also keine serverseitige «Sessioninformationen».
- Vorteile
  - Setzen von Bookmarks ist problemlos möglich.
  - Bessere horizontale Skalierbarkeit der Infrastruktur.

5.4.2. REST - Standardmethoden (am Beispiel http(s))

- Eine REST-konforme Applikation verwendet die http-Methoden GET, PUT, PATCH, POST, DELETE etc. in verbindlicher Weise

HTTP Methode	Beschreibung
GET	Fordert die angegebene Ressource vom Server an
PUT	Falls die angegebene Ressource existiert, wird sie aktualisiert, sonst neu angelegt
DELETE	Löscht die angegebene Ressource
POST	Fügt (erstellt) eine neue Ressource hinzu

- GET, PUT, PATCH und DELETE sollen beliebig oft aufgerufen werden können, ohne Seiteneffekte
  - **idempotente** Methoden.
- POST darf **nicht** mehrfach (mit denselben Daten) aufgerufen werden, hat Seiteneffekte
  - **nicht idempotente** Methode.

5.4.3. REST - Identifizierung von Ressourcen

DEFINITION: Ressource = Informationseinheit, welche über ein URI eindeutig identifiziert bzw. angesprochen werden kann.

- Ressource kann beliebige Datenformate nutzen (XML, JSON, Bild, PDF etc)
  - häufig: XML und JSON
  - Ideal: Repräsentation über «Accept: MIME-Type» wählbar.
- REST-konformer URI enthält keine Namen von Operationen!
- Beispiele:
  - Alle Studenten: http://localhost:8090/api/v1/students
  - Student mit ID «2»: http://localhost:8090/api/v1/students/2
  - Suche nach Studenten mit dem Namen «Zweifel»🥔: http://localhost:8090/api/v1/students?lastname=Zweifel

5.4.4. REST - Schnittstellendesign / Namensgebung

URIs für REST-Schnittstellen folgen strikten Konventionen:

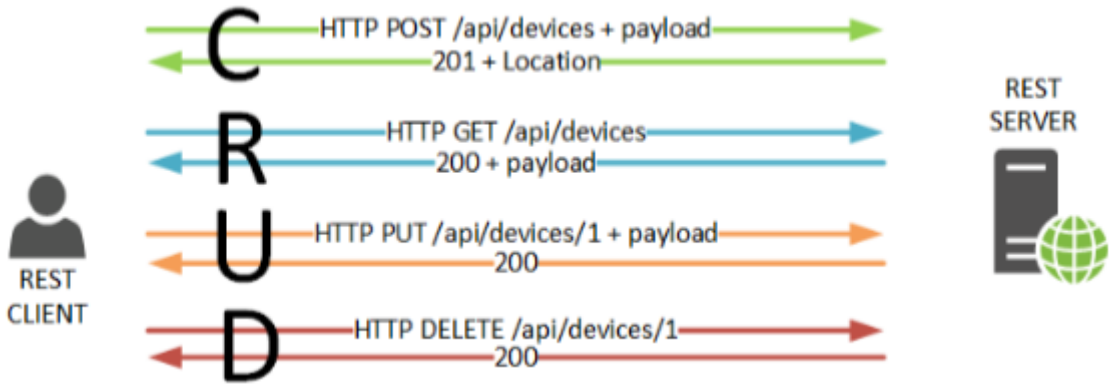
- Bezeichner werden typisch in **Mehrzahl** geschrieben
- IDs (Schlüssel) sind **immer** Bestandteil des Pfades (URI).
  - **Nie** als Attribute übermitteln

Konsequenz aus diesen Forderungen:

- Mit ein und derselben URI kann man unterschiedliche Aktionen ausführen
- Die Entscheidung der Operation geschieht nur über die verwendete Protokoll-Methoden (Beispiel http: GET, PUT etc.), und den Payload.

5.4.5. Rückgabewerte über Body und http-Statuscode

- Mehrere Rückgabemöglichkeiten
  - Content/Payload (Body) vom Request
  - http-Statuscode
- auf eine einheitliche, klare Zuordnung achten
- Konsequenz: Keine Fehlermeldungen/Code über den Content, sondern konsequent über den http-Statuscode



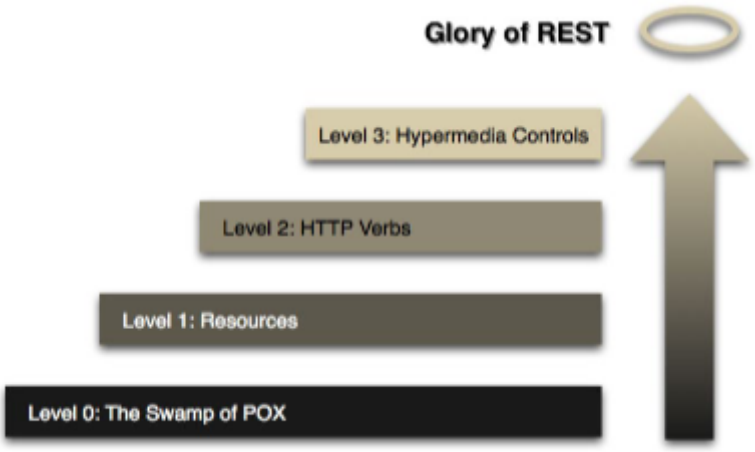
5.4.6. REST - Hypermedia-Prinzip

DEFINITION: Konzept von Verknüpfungen von unterschiedlichen Ressourcen über Links. Wenn der Client eine Antwort bekommt, kann er die enthaltenen Verknüpfungen verwenden bzw. ihnen dynamisch „folgen“.

```xml
<bestellung href='http://fbs.hslu.ch/bestellungen/73'>
  <datum>2017-12-18</datum>
  <betrag>1694.65</betrag>
  <produkt href='http://fbs.hslu.ch/produkte/53' />
  <kunde href='http://fbs.hslu.ch/kunden/421' />
</bestellung>
```

5.4.7. RESTfull - Richardson Maturity Model (RMM)

- Applikation ist erst RESTfull, wenn die wesentlichen REST-Prinzipien eingehalten wurden



Hinweis: die meisten Applikationen kommen nur bis Level 2.

5.4.8. RESTfull - The glory of REST

- keine Voraussetzung für feste Ressourcennamen oder Hierarchien
  - Server soll Struktur der URIs ändern können, ohne dass der Client angepasst werden muss
- Client muss nur die Einstiegs-URI kennen
  - alle weiteren (spezifischen) URIs werden dem Client vom Server übermittelt
  - HATEOAS: Hypermedia As The Engine Of Application State
- keine Abhängigkeit von einem spezifischen Kommunikationsprotokoll
  - Anwendung muss mit http-URIs, ftp-URIs und auch mit file-URIs funktionieren
- bereits definierten Protokolle dürfen nicht geändert werden (GET bleibt GET, PUT bleibt PUT etc).

-> gibt noch mehr

5.4.9. REST - Versionierung

- einfache Versionierung (paralleler Betrieb verschiedener Endpunkte möglich)
- Bsp.
  - http://fbs.hslu.ch/api/v1/orders - Version 1 der Schnittstelle.
  - http://fbs.hslu.ch/api/v2/orders - Version 2 der Schnittstelle.
- Wenn möglich werden alte Versionen durch eine Wrapper-Implementation (Translator) auf die neue Version ersetzt
  - So vermeidet man Coderedundanzen und erhöhten Wartungsaufwand, und hat trotzdem einen «zentralen» Zugriffspunkt.

6. Testing

6.1. Testarten

- **Unit- und Komponententest** (Entwicklertest)
  - schnelle, einfache Tests von einzelnen Klassen / Komponenten während Entwicklung
  - ohne Abhängigkeiten
  - automatisiert
  - überall lauffähig
  - beliebig wiederholbar
- **Integrations- und (Teil-)Systemtest**
  - Testen eines gesamten (Teil-)Systems innerhalb seiner (Teil-) Systemumgebung, auf Basis der Spezifikation, auf seine fachliche Korrektheit.
  - Aufgrund der Komplexität und des Aufwandes werden solche Tests noch viel zu häufig manuell durchgeführt, speziell z.B. bei GUI-Tests.
  - Automatisierung birgt aber sehr grosses Potential

6.2. Ziele für gute (System-) Tests

- Testausführung so weit wie möglich **automatisieren**
  - Frameworks verwenden
- **Testscripts** erstellen
  - Qualität von Testcode wie bei produktivem Code behandeln
- **frühe Erstellung** der Teststories auf Basis der Use Cases
  - **Test Frist** (für alle Ebene möglich)
- möglichst **kurze** und **infache Testfälle**
  - grössere Anzahl von freigranularen Testfällen
  - schnelle Lokalisation eines Fehlers
- Use Cases / Funktionen möglichst **einzeln** testen
  - Zu lange «Teststories» sind instabiler (auf Veränderungen), und somit unerwünschte Folgefehler viel häufiger.

Testfälle

- Testen der **normalen** Abläufe als Basis, Beispiele:
  - korrektes (erlaubtes) Login
  - Suchen (und finden) von vorhandenen Daten
  - Erfassen (und konkretes speichern) von Daten
- absichtlich provozierte **Fehler**, **Ausnahmen** und nicht erlaubte Vorgänge testen, Beispiele
  - falsches und nicht zulässiges Login
  - ungenügende Rechte
  - korrekter Abbruch einer Aktion (ohne Veränderung der Daten)

TL;DR: Testfälle gerade so komplex wie nötig, und so einfach wie möglich halten!

6.3. Tests in Schichtenarchitektur

Gefahr: man testet immer alle unteren Schichten (integrativ) mit
- Ausführungszeit steigt sukzessive an
- Selektivität sinkt stetig (es wird redundant immer alles getestet)
- Abhängigkeiten steigen –> Integrationstests statt Unit Tests

Ziel: Schichten möglichst **einzeln** und **entkoppelt** voneinander testen
- Entwicklung „von oben nach unten“ ist effektiver

Lösung: Einsatz von **Test Doubles**
- Untere Schichten werden für Tests dynamisch ersetzt.
- Vereinfachung durch wohldefinierte(s) Testdaten und Verhalten.

6.4. Testen von DB-Applikationen

Herausforderung: Datenmanipulation durch Testfälle (Reproduzierbarkeit muss gewährleistet sein)

Anforderungen
- Zustand der Datenbank muss vor und nach der Ausführung der Testfälle definiert werden/sein.
- Effiziente Verifikation des Datenbankinhaltes.
- Parallele Testausführung (Anforderung / Notwendigkeit?)

Grundlage
- Testdatenmanagement: Bereitstellung und aktive Wartung von Testdaten, spezifisch für die einzelnen Testfälle.

Lösungsansätze
- Initialisierung mit Testdaten, Testfälle, Verifikation
  - Verwendung von Frameworks (Bsp. DBUnit)
- Verwendung mehrere Schemas auf zentralem DBMS
  - Schemas und Daten synchron halten
  - Konflikte durch parallele Durchführung der Tests möglich
- Lokale Datenbank (pro Entwickler / Buildserver etc.)
  - je nach DBMS: Aufwand (Installation, Lizenzen etc.)
- In-Memory-Datenbank für jeden Test.
  - Schneller. Ist der Test aber noch genügend repräsentativ?
- Produktive Datenbank (Abzug) als Docker-Test-Container.
  - Identische Datenbank, immer sauber Initialisiert

6.5. Testen von REST-(Micro-) Services

6.5.1. Basis
- für REST-Services gibt es eine grosse Zahl von Testframeworks
- für simple Fälle reicht JUnit und Java-Bordmittel aus
  - oder sogar curl
- Jüngere Libraries für REST-Clients sind so abstrakt und kompakt, dass man damit «direkt» testen kann.
  - Beispiele: OkHttp, Unirest

6.5.2. Erweitert
- Die (meisten) Microservice-Frameworks (SpringBoot, Micronaut.io, Quarkus etc.) bieten eigene, direkt integrierte Ergänzungen für (Unit-)Testing an.
  - Vereinfacht und beschleunigt das Testen
- Bei Integrations-Tests in komplexeren Umgebungen -> Testcontainer

6.5.3. Test von REST-Services mit Clients
- **Swagger UI und Open API**
  - https://swagger.io/tools/swagger-ui/
  - https://www.openapis.org/
  - Beide dienen primär zur Dokumentation einer REST-API

- Webclient zum «Testen» (ausprobieren) der Schnittstelle, kann direkt in den Service integriert werden
- Kein automatisiertes Testing, sondern nur manuelle Versuche.

- **Postman**: https://www.postman.com/
  - Platform (mit Applikation) zur Erstellung, Planung, Verwaltung und «Testing» von REST-APIs.
  - Scripting für (Teil-)automatisierte Testsuiten möglich.
  - Interaktive Nutzung über Rich-Client.
- **Bruno** 🐶: https://www.usebruno.com/
  - Open Source und frei verfügbare Alternative zu Postman.

6.6. Testing - Bilanz

HINWEIS:
- «Es testen nur die Besten!» - Zitat von Dominique Portmann, Modul SWT
- Wenn man wirklich Testen will, findet man immer einen Weg!
- Testen Sie pragmatisch, selektiv und effizient

WICHTIG: TESTEN SIE IHRE APPLIKATION!

7. Modellieren

7.1. Requirements Engineering

Vorgehen beim Spezifizieren und Verwalten von Anforderungen an ein System, ein Produkt oder eine Software.

Hilft folgende Fragen zu beantworten:
- Was sind die Anforderungen – also die Fähigkeiten und Eigenschaften – des gewünschten Produkts?
- Wie lassen sich diese Anforderungen spezifizieren – also erheben, analysieren, dokumentieren und validieren?
- Wie sollten Anforderungen verwaltet – also freigegeben, evtl. verändert und rückverfolgt – werden?

7.1.1. Aufgaben und Ziele

Aufgaben
- Ermittlung von Anforderungen
  - inkl. Detaillierung und Verfeinerung
- Dokumentation
- Prüfung und Abstimmung von Anforderungen
- Verwaltung von Anforderungen
  - Requirements Management

Ziele
- Risikominimierung
- gemeinsames Verständins der Stakeholder

7.2. Stakeholder

- **Projektbeteiligte**
- Personen mit **Interesse am Verlauf oder Ergebnis**

Informationslieferanten für Ziele, Anforderungen und Randbedingungen an ein zu entwickelndes System oder Produkt

Beispiele für Stakeholder:
- Management
- Tester
- Gesetzgeber
- Sicherheitsbeauftragte
- Kunden
- Entwickler

7.3. Anforderungen

Nutzungsanforderungen:
- effiziente Erbringung eines Ergebnisses mit einem interaktiven System

Gesetzliche Anforderungen:
- Gesetzgebern und Behörden
- Richtlinien, Gesetzte, Verordnungen, Normen

Fachliche Anforderungen:
- Vollständigkeit und Korrektheit eines Arbeitsergebnisses

Organisatorische Anforderungen:
- Verhalten von Personen oder Organisationseinheiten bei der Erbringung von Arbeitsergebnissen

Marktanforderungen:
- Anforderungen die für die Kaufentscheidung entscheidend / relevant sind

7.3.1. funktionale / nicht-funktionale Anforderungen

funktional -> **was**

nicht-funktional -> **wie**
- Performance
- Zuverlässigkeit (Reliability) und Verfügbarkeit (Availability)
- Sicherheit (Security)
- Wartbarkeit (Maintainability)
- Portierbarkeit (Portability)

7.3.2. Qualitätskriterien
- Verständlichkeit, Klarheit
- Eindeutigkeit (keine Missverständnisse)
- Vollständigkeit
- Konsistenz
- Korrektheit
- Gültigkeit (aktuell)
- Verfolgbarkeit / Traceability
- Testbarkeit/Prüfbarkeit
- Machbarkeit/Umsetzbarkeit
- Bewertet (Prio, Aufwand, Risiko…)

7.3.3. Methoden / Techniken für Anforderungen
Methoden / Techniken zum Erheben, Festhalten oder Analysieren von Anforderungen:

- Kontext-Diagramm
- Domain-Modell
- Use Case Modell und Beschreibungen
- Geschäftsprozess-Modell
- Feature-Listen
- User Stories

7.4. User Story
textliche Anforderung an das System

Als „**Stakeholder**“ möchte ich „**Ziel / Wunsch**“ um „**Nutzen**“.

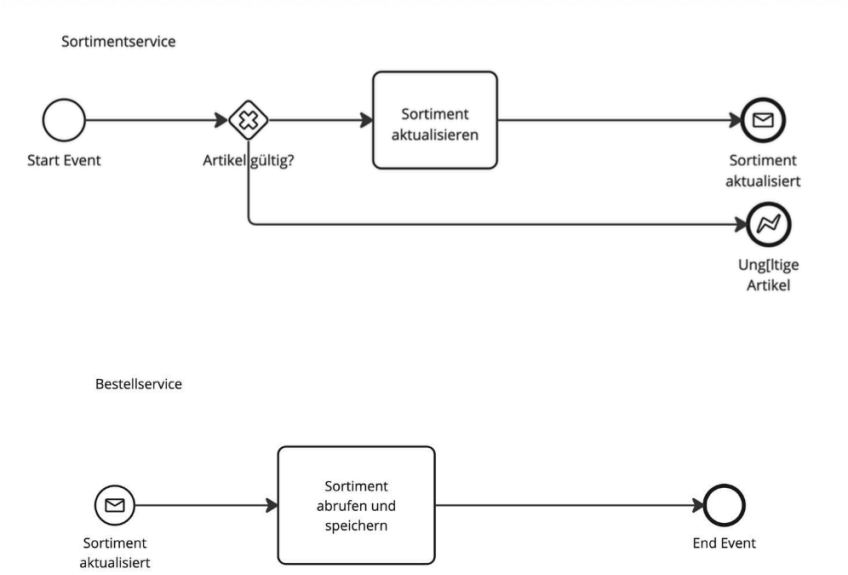
- aus **Anwendersicht**
- sollte **klaren Nutzen** erbringen
- evtl **weiter zerlegt**, damit in einem Sprint umgesetzt werden kann (Story Splicing)

7.4.1. Akzeptanzkriterien
beschreibt wann eine US umgesetzt ist

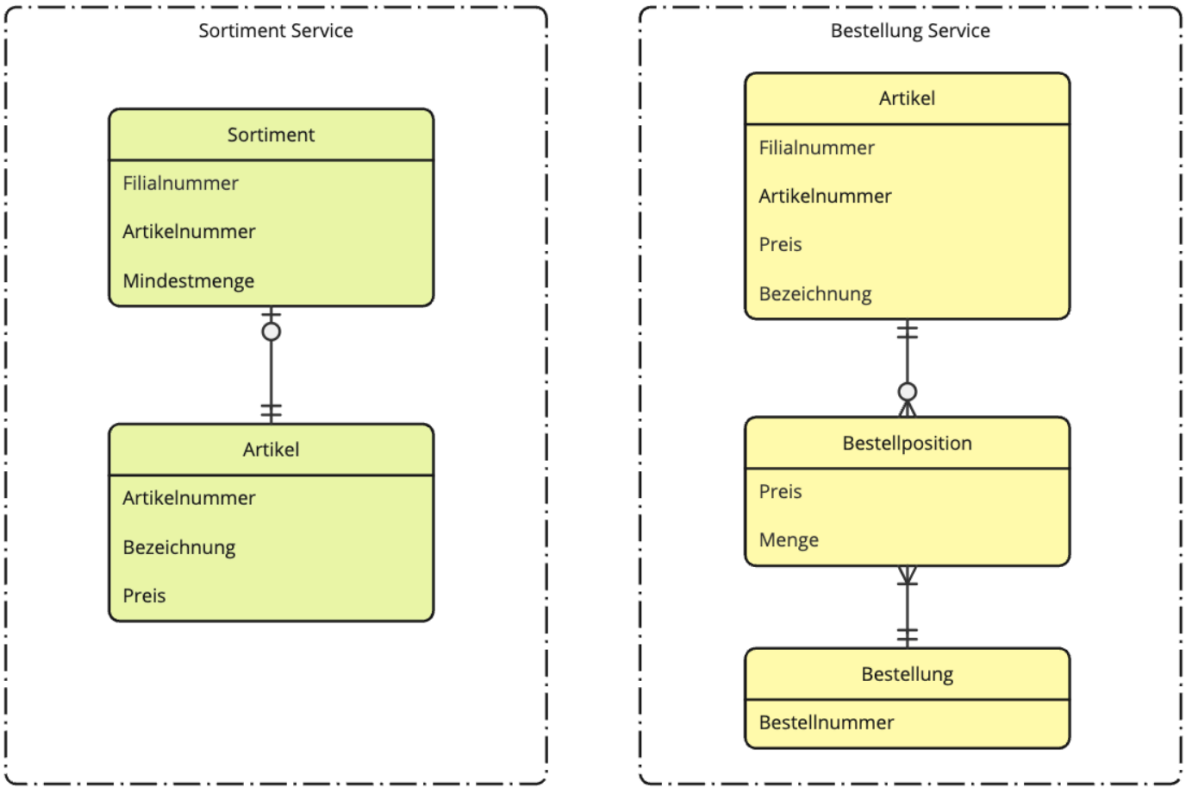
- jede **US soll mind. 1 Akzeptanzkriterium** habe
- **vor Implementierung** geschrieben
- unabhängig **testbar**
- klares **Pass / Fail**
- konzentriern sich auf das **was**, *nicht* wie
- funktionale & nicht-funktionale

7.5. OpenAPI
Requirements Engineering auf **REST-API** Ebene

7.6. Dokumentation - Prozess mit BPMN



7.7. Domain Model



Domain Model auf Sortiment-Seite ist nicht gleich wie das auf der Bestellungs-Seite

7.8. Weitere Informationen

Alles das **zwischen User-Story und Code** liegt, soll **dokumentiert** werden.

Wie sieht die API aus? -> Dokumentation

Wie sieht das Datenmodell aus? -> Dokumentation

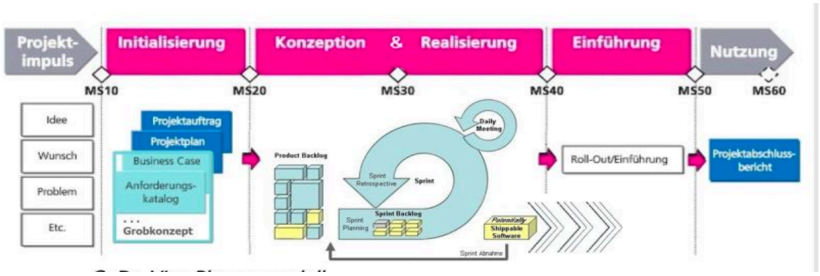
Was ist mit Artikel gemeint? -> Dokumentation

Welche Attribute und welche Validierungen gibt es? -> Dokumentation

Welche Events werden ausgelöst? -> Dokumentation

7.9. Vorgehen im Projekt

nach SODA:



7.9.1. Sprint Planning

Ein Sprint Planning ist das Planungstreffen zu Beginn eines Sprints, bei dem das Team festlegt:

- Was im Sprint erreicht werden soll (Sprint-Ziel)
- Wie die Arbeit umgesetzt wird (Aufgabenplanung)

7.9.2. Product-Backlog

- Liste aller Anforderungen, Ideen, Features
- Vom Product Owner gepflegt
- **Priorisiert nach Wert und Dringlichkeit**
- Grundlage für Sprint Planning
- Enthält User Stories und Akzeptanzkriterien

7.9.3. Sprint Review

- Installierte & funktionierende Software
- Produkt/Software entspricht den in den Stories definierten Anforderungen
- Software kann vorgeführt werden
- Dokumentation und Diagramme aktualisiert

7.10. User Story slicing

Story überschneidet mehrere architektonische Ebene

Store slicing: dünnere / kleinere Stories erstellen (immer noch **vertikal**)

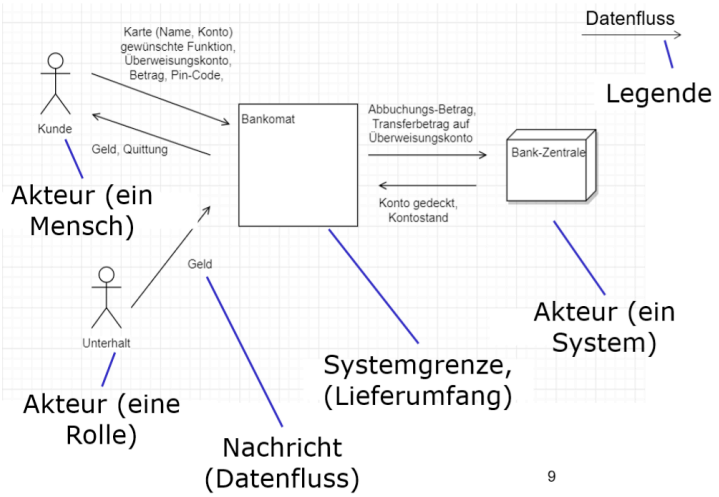
Wieso?

- schneller Mehrwert für Nutzer
- einfacher verständlich
- leichter testbar
- präzisere Aufwandsschätzung
- bessere Planbarkeit
- geringeres Risiko bei Änderungen
- schnelleres Feedback
- ermöglicht parallele Arbeit im Team
- frühzeitige Fehlererkennung
- inkrementelle, kontinuierliche Entwicklung

7.11. Modelle

7.11.1. Kontext Diagramm

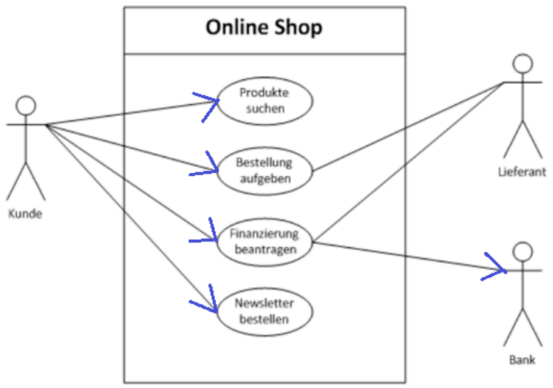
- Im Kontextdiagramm werden alle externen Akteure und alle Nachrichten, welche Sie mit dem System austauschen, dargestellt.
- Das Diagramm dient der Systemabgrenzung.



7.11.2. Anwendungsfall Diagramm / Use Case

beschreibt alle möglichen Szenarien zur Zielerreichung eines Akteurs mit dem System

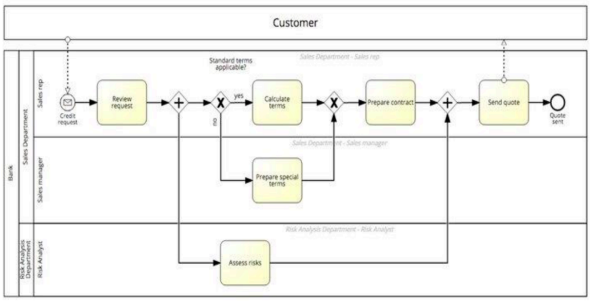
- Ziel: fachliches Ziel (Business Goal) erreichen
- Abstraktion: beschreibt was passiert, nicht wie technisch
- Use-Case-Diagramm: zeigt Akteure + Use Cases + Verbindungen
- Pfeile: zeigen, welcher Akteur welchen Use Case anstösst



7.11.3. Geschäftsprozess Modell / BPMN

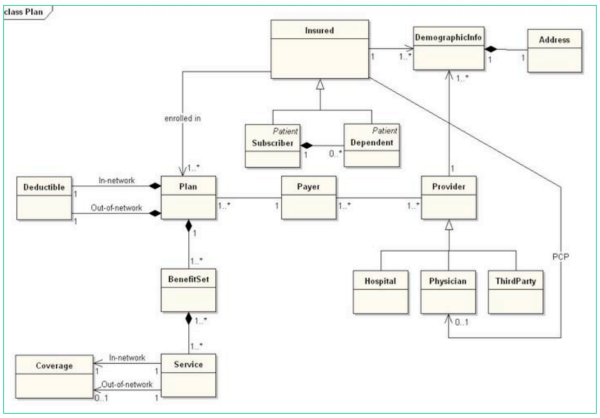
Abbildung von Geschäftsprozessen

- Zweck: systematische, strukturierte Darstellung von Abläufen
- Inhalt: zeitlich-sachlogische Abfolge von Funktionen
- Ziele:
  - Dokumentation
  - Analyse
  - Gestaltung von Prozessen
  - Automatisierung (Workflow-Management)
  - Kommunikation über Prozesse



7.11.4. Domain Modell

- Anwendungsdomäne: abgegrenztes Problemfeld oder Einsatzbereich einer Software
- Synonyme: Fachdomaine, Problemdomaine, Domain
- Domain-Model: konzeptionelles Modell der Domaine
- Inhalt: Daten + Beziehungen
- Darstellung: meist als Klassendiagramm



7.11.5. Feature Liste

- Auflistung aller Funktionen bzw. Merkmale eines Systems
- Beschreibt was das System kann (nicht wie)

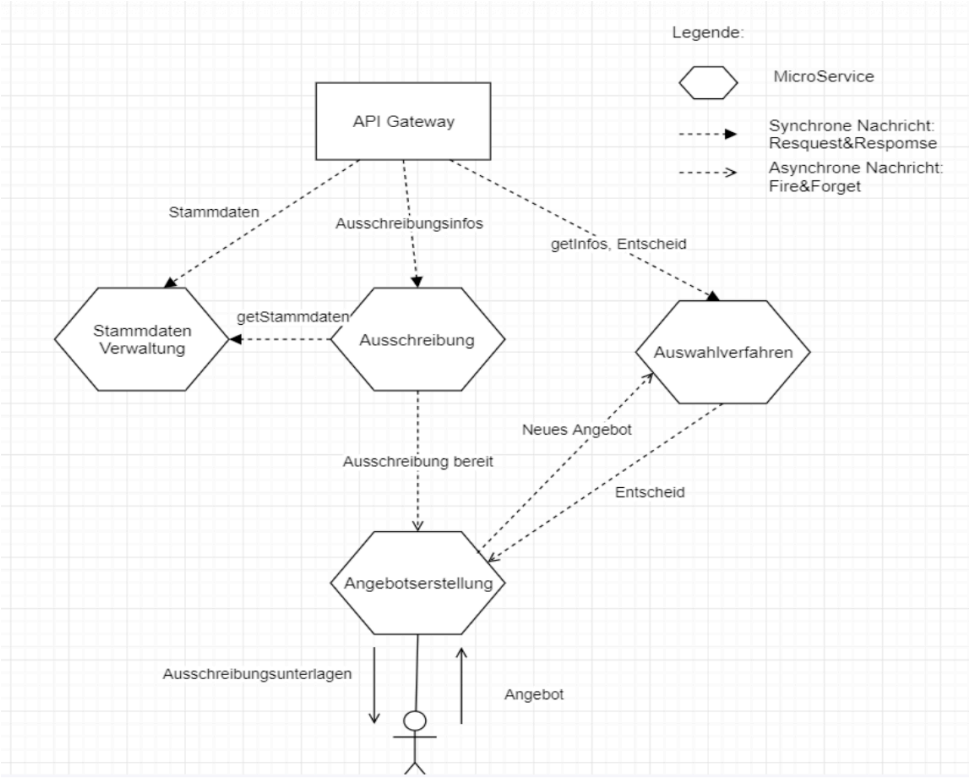
7.11.6. C4 Modell

- `C`ontext
- `C`ontainers
- `C`omponents
- `C`ode

7.11.7. Microservice Diagramm - SWDA spezifisch

- Welche Microservices gibt es?
- Wie sieht die Kommunikation dieser aus?

- Synchron
- Asynchron
- Welche Daten werden ausgetauscht?



7.11.8. Event Catalog

Sammlung aller Events in einem System