Kapitel 4 Integration

albion.eu www.tectrain.ch www.accso.de

< Kapitel 3 Modularisierung

Kapitel 5 Installation und Roll Out >

Kapitel 4 Integration

FLEX Lehrplan

4 Integration

Dauer: 90 Min Übungszeit: 30 Min

4.1 Begriffe und Konzepte

Frontend Integration, Legacy Systeme, Authentifizierung, Autorisierung, (lose) Kopplung, Skalierbarkeit, Messaging Patterns, Domain Events, dezentrale Datenhaltung.

4.2 Lernziele

4.2.1 Was sollen die Teilnehmer können?

- Die Teilnehmer sollen eine Integrationsstrategie wählen, die auf das jeweilige Problem am besten passt. Das kann beispielsweise eine Frontend-Integration, eine Integration über RPCMechanismen, mit Message-orientierter Middleware, mit REST oder über die Replikation von Daten sein.
- Die Teilnehmer sollen einen geeigneten Ansatz für das Umsetzen von Sicherheit (Autorisierung/Authentifizierung) in einem verteilten System konzeptionieren können.
- Anhand dieser Ansätze sollen die Teilnehmer eine Makroarchitektur entwerfen können, die zumindest Kommunikation und Sicherheit abdeckt.
- Für die Integration von Legacy-Systemen muss definiert werden, wie mit alten Datenmodellen umgegangen wird. Dazu kann der Ansatz des Strategic Designs mit wesentlichen Patterns wie beispielsweise Anti-Corruption Layer genutzt werden.
- Die Teilnehmer können abhängig von den Qualitätszielen und dem Wissen des Teams eine geeignete Integration vorschlagen.

4.2.2 Was sollen die Teilnehmer verstehen?

- Die Teilnehmer sollen die Vor- und Nachteile verschiedener Integrationsmechanismen kennen. Dazu zählen Frontend-Integration mit Mash Ups, Integration auf dem Middle Tier und Integration über Datenbanken oder Datenbank-Replikation.
- Die Teilnehmer sollen die Konsequenzen und Einschränkungen verstehen, die sich aus der Integration von Systemen über verschiedenen Technologien und Integrationsmuster z. B. in Bezug auf Sicherheit, Antwortzeit oder Latenz ergeben.
- Die Teilnehmer sollen ein grundlegendes Verständnis für die Umsetzung von Integrationen mit Hilfe von Strategic Design aus dem Domain Driven Design und wesentliche Pattern kennen.
- RPC bezeichnet Mechanismen, um Funktionalität in einem anderen Prozess über Rechnergrenzen hinweg synchron aufzurufen. Dadurch entsteht Kopplung in vielerlei Hinsicht (zeitlich, Datenformat, API). Diese Kopplung hat negative Auswirkungen auf Verfügbarkeit und Antwortzeiten des Systems. REST macht Vorgaben, die diese Kopplung reduzieren können (Hypermedia, standardisierte API). Die zeitliche Kopplung bleibt jedoch grundsätzlich bestehen.

- Bei der Integration mittels Messaging kommunizieren Systeme durch den asynchronen Austausch von Nachrichten. Die Systeme werden somit zeitlich entkoppelt. Technisch wird dies mittels Indirektion über eine Middleware erreicht. Nachrichten können optional persistiert,
  gefiltert, transformiert etc. werden. Es gibt verschiedene Messaging Patterns wie Request/Reply, Publish/Subscribe oder Broadcast.
- Eine Integration über Daten ermöglicht hohe Autonomie, die allerdings über die Notwendigkeit zur redundanten Datenhaltung und die damit notwendige Synchronisation erkauft wird. Es darf nicht angenommen werden, dass andere Systeme dieselben Schemata nutzen, weil das eine unabhängige Weiterentwicklung der Schemata verhindert. Daher muss für die Integration eine angemessene Transformation vorgesehen werden.
- Bei einer Event-Driven Architecture (EDA) wird RPC vermieden oder reduziert, indem Domain Events publiziert werden. Domain Events beschreiben Zustandsänderungen. Interessierte Systeme können diese Nachrichten verarbeiten (Publish/Subscribe). Dieses Vorgehen hat Auswirkungen darauf, wie der Zustand gespeichert wird. Während bei einer RPC-basierten Integration der Server die Daten speichern muss, liegt diese Verantwortung bei EDA beim Subscriber.
- Somit entstehen Replikate (dezentrale Datenhaltung). Die Subscriber fungieren dadurch als Cache für den Publisher. Es können weitere Subscriber hinzugefügt werden, ohne den Publisher zu beeinflussen (außer durch Polling). Monitoring der Event-Flüsse ist wichtig.
- Domain Events können via Messaging publiziert werden. Dabei pusht der Publisher die Nachrichten in ein Messaging System. Alternativ können die Nachrichten auch beim Publisher gepollt werden (bspw. Atom/RSS). Beim Einsatz eines Messaging Systems können Subscriber die Nachrichten per Push oder Pull erhalten. Dies hat Auswirkungen auf den Umgang mit Backpressure.

4.2.3 Was sollen die Teilnehmer kennen?

- Typische verteilte Sicherheitsmechanismen wie OAuth oder Kerberos
- Ansätze für die Front-End-Integration
- Techniken für die Integration von Services: REST, RPC, Message-orientierte Middleware
- Herausforderungen bei der Nutzung gemeinsamer Daten
- Datenbank-Replikationsmechanismen mit ETL-Tools oder anderen Ansätzen
- Messaging Patterns (Request/Reply, Publish/Subscribe etc.)
- Messaging Systeme (RabbitMQ, Kafka etc.), Protokolle (AMQP, MQTT, STOMP etc.) und APIs (JMS)

4.3 Referenzen

- Eric Evans: Domain-Driven Design: Tackling Complexity in the Heart of Software, AddisonWesley, 2003, ISBN 978-0-32112-521-7
- http://oauth.net/
- Gregor Hohpe, Bobby Woolf: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, Addison-Wesley, 2003, ISBN 978-0-32120-068

Inhalte

(A) Motivation und Treiber

1. Ebenen von Integration

Der Begriff der Integration bezieht sich auf die Notwendigkeit, die über Modularisierung aufgeteilten Komponenten wieder zusammenzuführen, v. a. auf die **Remote-Kommunikation** in verteilten Anwendungen oder verteilten Anwendungsteilen (z.B. Services). Aufrufe innerhalb eines Monolithen (also Aufrufe im selben Prozessraum) sind in diesem Kapitel i.d.R. nicht adressiert (da über Komponentenkonfiguration/-komposition bereits abgedeckt).

Integration in diesem Sinne bezieht sich auf zwei Ebenen:

1. Mikro-Ebene (innerhalb einer Anwendung / eines Systems): Zum anderen bezieht sich Integration auf die Anwendungs-interne Integration der Teilstrukturen der Anwendungen, also der (Micro-)Services auf verschiedenen Layern und Ebenen der Architektur. Die in Kapitel 3 vorgestellten Schritte hin zu einer modularen Anwendung erfordern nachfolgend Integrationstechniken, die die Einzelteile wieder zu einem Ganzen "zusammenstecken".
2. Makro-Ebene (zwischen Anwendungen / Systemen): Es sind separate Anwendungen zu integrieren, die bereits existieren und als Nachbar- oder Dritt-Systeme aus dem eigentlichen Kern-Scope herausfallen. Solche Grenzen sind idealerweise fachlich (andere Domäne), oft aber auch organisatorisch (andere Abteilung) bzw. leider oft auch (zu oft) technisch motiviert (z.B. Altsystem auf Mainframe). Das Management der Grenzen ist Thema für Enterprise-Architektur und EA-Management. Dies wird unten im Abschnitt zu Legacy-Systemen vertieft.

2. Erst Verteilung, dann Integration

Die Integration von Softwaresystemen wird durch eine Vielzahl von Faktoren bestimmt, wie Geschäftsanforderungen, Datenintegration, Skalierbarkeit, Zuverlässigkeit, Sicherheit und Benutzerfreundlichkeit. Die Trennung und Verteilung von Teilen des Systems auf verschiedene Teile wie Sub-Systeme oder Services ermöglicht die Bewältigung erhöhter Mengengerüste bzw. Datenvolumina, bietet Redundanz, Failover-Fähigkeiten, bessere Sicherheit, Wiederverwendbarkeit und Spezialisierung.

Die Trennung und Verteilung von Teilen des Systems auf verschiedene physisch verteilte Systeme, die remote kommunizieren müssen, ist aus mehreren **Gründen** notwendig:

1. **Geschäftsanforderungen:** Geschäftsanforderungen können die Integration verschiedener Softwaresysteme erforderlich machen, um die Effizienz zu steigern, Prozesse zu automatisieren oder neue Erkenntnisse aus Daten zu gewinnen.
2. **Scope:** Durch die Aufteilung eines Systems in verschiedene Systeme ist es möglich, jede Komponente zu spezialisieren, um bestimmte Funktionen oder Daten zu verarbeiten.
3. **Datenintegration:** Die Notwendigkeit, Daten zwischen verschiedenen Systemen auszutauschen, z. B. zwischen einer Kundendatenbank und einem Auftragsverwaltungssystem, kann die Notwendigkeit der Integration dieser Systeme begründen.
4. **Skalierbarkeit:** Durch die Aufteilung eines Systems auf verschiedene Systeme ist es möglich, jede Komponente unabhängig zu skalieren, was zur Bewältigung eines erhöhten Datenverkehrs oder Datenvolumens beitragen kann.
5. **Verlässlichkeit:** Durch die Aufteilung eines Systems auf verschiedene Systeme ist es möglich, Redundanz und Failover-Funktionen bereitzustellen, was dazu beitragen kann, Ausfallzeiten zu minimieren.
6. **Sicherheit:** Durch die Aufteilung eines Systems auf verschiedene Systeme ist es möglich, eine sicherere Umgebung zu schaffen, indem sensible Daten und Funktionen isoliert werden. Ein zentrales Sicherheitsmodell kann dabei genutzt werden.
7. **Verbesserte Benutzerfreundlichkeit:** Die Integration von Systemen kann dazu beitragen, die Benutzererfahrung zu verbessern, indem sie eine nahtlose und kohärente Erfahrung über verschiedene Systeme hinweg bietet (Beispiel: Portale, Microfrontends).
8. **Wiederverwendbarkeit:** Durch die Aufteilung eines Systems auf verschiedene Systeme ist es möglich, diese Komponenten in anderen Projekten oder Systemen wiederzuverwenden.

Bei der Verteilung und Integration von Systemen liegt daher der Fokus vor allem auf diesen **Qualitätseigenschaften** (siehe auch Kapitel 2 (B)): Skalierbarkeit, Verlässlichkeit, Sicherheit, Leistung, Wartbarkeit, Interoperabilität, Flexibilität, Testbarkeit, Benutzerfreundlichkeit. Durch die Berücksichtigung dieser Qualitätsmerkmale wird sichergestellt, dass das System anpassungsfähig bleibt und dauerhaft mit anderen Systemen und Technologien kommunizieren kann.

3. Legacy-Anwendungen / ihre Integration

Die Integration von **Legacy-Anwendungen** ist notwendig, da diese Altanwendungen oft wertvolle Daten und Funktionen enthalten, die von der Organisation noch genutzt werden, und mit neuen Systemen integriert werden müssen, um ein Gesamtzusammenspiel zu ermöglichen.

Eine Legacy-Anwendung ist ein älteres System, eine ältere Anwendung oder eine ältere Technologie, die noch in Gebrauch ist, aber nicht mehr aktiv entwickelt oder unterstützt wird. Legacy-Anwendungen sind oft schwer zu warten, zu aktualisieren oder in neuere Systeme zu integrieren und können Sicherheitslücken oder andere Probleme aufweisen, die ihren Einsatz riskant machen.

Einige gemeinsame Merkmale von Legacy-Anwendungen sind:

- Sie sind oft in älteren Programmiersprachen geschrieben und verwenden veraltete Technologien.
- Sie können schwer zu verstehen oder zu ändern sein, weil die ursprünglichen Entwickler nicht mehr verfügbar sind oder die Codebasis nicht gut dokumentiert ist.

- Sie lassen sich aufgrund von Inkompatibilitäten bei Datenformaten, Protokollen oder Schnittstellen nur schwer in neuere Systeme integrieren.
- Sie können Sicherheitslücken aufweisen, die nicht (mehr) behoben werden, weil der Hersteller die Technologie nicht mehr unterstützt.

Legacy-Anwendungen können ein erhebliches Risiko für ein Unternehmen darstellen, weil sie möglicherweise nicht in der Lage sind, moderne Arbeitslasten oder Sicherheitsbedrohungen zu bewältigen, und weil sie schwierig oder teuer in der Wartung und Aktualisierung sind. Einige Unternehmen entscheiden sich dafür, Altsysteme durch neuere Systeme zu ersetzen, während andere sie modernisieren oder umgestalten, um ihre Nutzungsdauer zu verlängern und sie sicherer und effizienter zu machen.

Die **Integration von Legacy-Anwendungen** mit anderen Anwendungen der Anwendungslandschaft kann eine schwierige Aufgabe sein, aber es gibt ein paar gängige Ansätze:

1. Middleware:
   a. Middleware ist eine Softwareschicht, die zwischen der Legacy-Anwendung und der modernen Anwendung angesiedelt ist und als Brücke zwischen den beiden fungiert.
   b. Middleware kann verwendet werden, um Daten zwischen den verschiedenen Formaten zu übersetzen, die von den alten und modernen Anwendungen verwendet werden, und um Protokoll- und Schnittstelleninkompatibilitäten zu behandeln.
2. APIs: Die Entwicklung von APIs für die Legacy-Anwendung kann modernen Anwendungen den einfachen Zugriff auf ihre Funktionen und Daten ermöglichen.
   a. Dies kann durch die Erstellung eines Webdienstes oder einer RESTful-API geschehen, der/die die Legacy-Anwendung umhüllt und ihre Funktionen über das Netzwerk zugänglich macht.
   b. REST: Unterscheide dabei verschiedene Reifegrade (Richardson Maturity Model, https://martinfowler.com/articles/richardsonMaturityModel.html)
3. Containerisierung und Virtualisierung:
   a. Legacy-Anwendungen können containerisiert oder virtualisiert werden, so dass sie in einer isolierten Umgebung - getrennt vom Host-Betriebssystem - laufen und in moderne Anwendungen integriert werden können, indem der Container oder die virtuelle Maschine für die neue Anwendung verfügbar gemacht wird.
   b. Beispiele: Lift&Shift, Containerisierung, Screen-Scraping.
4. Datenintegration:
   a. Ein anderer Ansatz besteht darin, nur die Daten aus dem Altsystem in das moderne System zu integrieren.
   b. Dazu werden Datenintegrationstools oder ETL-Prozesse (Extrahieren, Transformieren, Laden) verwendet, um Daten aus dem Altsystem zu extrahieren, sie in ein Format umzuwandeln, das vom modernen System verwendet werden kann, und sie in den Datenspeicher des modernen Systems zu laden.

Legacy-Anwendungen sind sehr individuell: Welches also der beste Integrationsansatz ist, hängt vom jeweiligen System, seinen Anforderungen und den verfügbaren technischen wie organisatorischen Ressourcen ab.

Wenn eine Integration auf Dauer schwierig oder zu kostspielig ist, so sind Ablösung und Migration die Wahl: Legacy-Anwendungen können in kleinere, unabhängige Dienste aufgeteilt werden. Diese kleineren Services können dann in moderne Anwendungen leichter integriert werden. Dies kann ein guter Ansatz sein, wenn die Legacy-Anwendung monolithisch und schwer zu ändern ist, entsprechend Refactoring und Modernisierung teuer ist.

4. Kopplung - warum ist das schlecht?

Mit **"Kopplung"** bezeichnet man den Grad, in dem eine Komponente eines Systems von anderen Komponenten abhängig ist (siehe Kapitel 3). Eine hohe Kopplung (auch: enge Kopplung) liegt dann vor, wenn ein Modul oder eine Komponente stark von anderen Modulen oder Komponenten abhängt und sich folglich Änderungen in einem Modul oder einer Komponente auf andere Module oder Komponenten direkt bzw. unmittelbar auswirken. Eine lose Kopplung hingegen ist also wünschenswert, da sie eine größere Flexibilität, Wartbarkeit und Wiederverwendbarkeit ermöglicht.

Was in dieser Form für Komponenten-Abhängigkeiten gilt, gilt um so mehr analog für verteilte Systeme - wenn auch aufgrund der Verteilung aber nicht über die statische (Code-)Abhängigkeit.

Eine fachlich enge Kopplung bei "loser" Verteilung (also Teilen, die lose über technischer Remote-Schnittstellen verbunden sind), ist eine besonders schlechte Kombination.

5. Migrationsvorgehen und -szenarien

Info
Martin Lehmann, Dr. Kristine Schaal: "Es gibt keine grüne Wiese. Wie migriere ich von ALT nach NEU?", IT-Tage 2019

Harry M. Sneed et al: "Software-Migration in der Praxis: Übertragung alter Softwaresysteme in eine moderne Umgebung", dpunkt, 2010

Carola Lilienthal, Henning Schwentner: "Domain-Driven Transformation: Monolithen und Microservices zukunftsfähig machen", dpunkt, 2023

Martin Fowler: "Strangler Fig Pattern", https://martinfowler.com/bliki/StranglerFigApplication.html

Sam Newman: "Monolith To Microservices. Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith", O'Reilly, 2019

Martin Fowler et al: "Patterns of Legacy Displacement", March 2024, https://martinfowler.com/articles/patterns-legacy-displacement/

Bei der Ablösung eines Legacy-Systems (Monolith) können verschiedene Migrationsvorgehen zum Einsatz kommen. Bekannte **Migrationsvorgehen** sind Butterfly, Chicken Little, Database First, Database Last und Strangler. (Diese Patterns sind unabhängig von der Deployment-Strategie im Go-Live. Für diese siehe Kapitel 5 (F), Abschnitt 8. "Rolling Updates). Die Vorgehen legen entweder Priorität auf "Kosten bzw. Dauer" oder sind "risiko-getrieben".

Kriterien für die Auswahl des richtigen **Migrationsvorgehens** sind u.a. (Auswahl):

- Kriterien der Architektur: Umfang und Komplexität des Systems: wie groß oder klein ist das System? je größer, desto schwieriger wird eine Ablösung als Big-Bang
- Kriterien des POs: Featuredruck - wie dringend werden neue Features benötigt, die sich nicht oder nur unzureichend im Altsystem implementieren lassen?
- Kriterien des Betriebs: Parallelbetrieb - ist ein solcher denkbar (um früh Erfahrungen mit neuer Technologie in Produktion zu machen?) oder nicht gewollt (weil Kosten/Kapazitäten dagegen sprechen?)

Abbildung: Entscheidungskriterien für ein Migrationsvorgehen (Links: Big-Bang, rechts: iterativ-inkrementell)
(aus: Martin Lehmann, Dr. Kristine Schaal: "Es gibt keine grüne Wiese. Wie migriere ich von ALT nach NEU?", IT-Tage 2019)

Unterscheide:

1. Turnover / Deployment: entweder in vielen kleinen Schritten (iterativ-inkrementell, wie Strangler) oder in einem großen Schritt ("Big-Bang").
2. Entwicklungsvorgehen: wasserfallig oder agil (also in kleinen Zyklen, wie Sprints). Das Entwicklungsvorgehen ist unabhängig vom Turnover / Deployment. Es ist durchaus denkbar und legitim, ein System in kleinen Sprints zu entwickeln, aber dennoch als Big-Bang live zu schalten. In diesem Fall bekommt man kein frühes Feedback (aus dem produktiven System) zu den in den Sprints entwickelten Features.

Vor allem das **Strangler (oder: Strangler Fig Pattern)** ist weithin bekannt, da man in ihm das Alt-System schrittweise, also iterativ-inkrementell ablöst: Neue Funktionalitäten oder Technologien werden schrittweise in eine bestehende Anwendung integriert, indem man sukzessive Teile der alten Anwendung durch neue ersetzt, bis die ursprüngliche Anwendung "erstickt" ist und vollständig durch die neue ersetzt wird. Gemeinhin wird angenommen, dass viele kleine Einzelschritte das Risiko minimieren, tatsächlich verteilen sie es (nur) auf der Zeitachse. Viele Einzelschritte benötigen Integration zwischen ALT und NEU, ebenfalls mit Potential für Fehler, sowie Aufwand für Tests der Integration. Außerdem müssen Nutzer ggf. mit zwei Systemen arbeiten (im schlimmsten Fall mit zwei UIs).

Im Idealfall wird eine fachliche Struktur gewählt, anhand von Daten, Nutzern, Nutzungsszenarien (vgl. Kapitel 3.(B), 10.), anhand deren sich einzelne fachliche Komponenten (im besten Fall über alle technischen Schichten hinweg) aus dem Altsystem herausoperieren lassen, um diese im neuen System zu implementieren. Siehe dazu auch die Konzepte von "Monolith to Microservices".

(B) Integration auf verschiedenen Tiers/Layern

Info
Schulung "Enterprise Application Integration Patterns (EIP / EAI)", Roger Rhoades und Martin Lehmann für die Landeshauptstadt München 2018 (nicht öffentlich)

Prof. Dr. Markus Voß u.a.: "Quasar Enterprise. Anwendungslandschaften serviceorientiert gestalten", dpunkt-Verlag

1. Varianten der Integration von Komponenten

Es gibt verschiedene Varianten der Integration für Teile und Komponenten von Software, darunter:

1. **Funktionale Integration:** Bei dieser Art der Integration geht es darum, sicherzustellen, dass die einzelnen Softwarekomponenten korrekt funktionieren, wenn sie miteinander kombiniert werden.
2. **Datenintegration:** Bei dieser Art der Integration geht es darum, Daten aus verschiedenen Quellen, wie Datenbanken oder APIs, zu einer einzigen kohärenten Ansicht zusammenzuführen.
3. **Schnittstellenintegration:** Bei dieser Art der Integration wird sichergestellt, dass die Schnittstellen zwischen verschiedenen Softwarekomponenten kompatibel sind und nahtlos zusammenarbeiten.
4. Integration in eine **SOA**: Bei dieser Art der Integration werden Webdienste verwendet, um verschiedene Softwarekomponenten miteinander zu verbinden und ihnen die Kommunikation und gemeinsame Nutzung von Daten zu ermöglichen.
5. **Systemintegration:** Bei dieser Art der Integration werden verschiedene Hardware- und Softwaresysteme zu einem kohärenten und funktionalen Ganzen kombiniert.
6. **Prozessintegration:** Bei dieser Art der Integration wird sichergestellt, dass Geschäftsprozesse zwischen verschiedenen Systemen und Softwarekomponenten integriert werden.

2. Integration auf dem Middle-Tier / Logik- / Anwendungsebene

Bei der **Integration auf dem Middle-Tier und der Anwendungsebene** werden in der Regel verschiedene Softwarekomponenten und System(-teile) miteinander verbunden, um Daten und Funktionen gemeinsam zu nutzen. Quasar Enterprise nennt dies "Logikintegration". Diese Art der Integration stellt die wichtigste Integrationsart dar und bietet die meiste Flexibilität, um Services, System(-teile), interne wie externe (Dritt-) Systeme, Legacy-Systeme und Anwendungen von Drittanbietern, ggf. auch Standardprodukten zu integrieren.

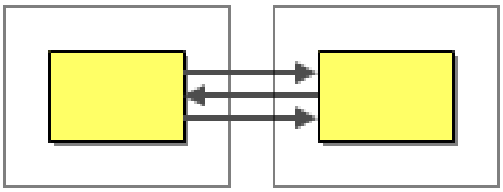


Abbildung: Business-to-Business Integration
(Quelle: https://www.enterpriseintegrationpatterns.com/patterns/messaging/Chapter1.html)

Dies kann mit verschiedenen Technologien und Protokollen wie Webdiensten, Nachrichtenwarteschlangen und API-Aufrufen erfolgen und umfasst , um Daten und Funktionen gemeinsam zu nutzen. Die spezifische Implementierung der Integration hängt von den verwendeten Technologien und Systemen sowie von den Anforderungen der Anwendung ab. Für diese Art der Integration wird v.a. benötigt (vgl. Quasar Enterprise, Kapitel 7.1):

- Kommunikation mit Adressierung, Transport, Protokollierung, Dienstgütezusicherung ("Quality of Service")
- Transformation, technisch wie fachlich
- Repositories wie Service-, Prozess-, Organisations-Repository
- Prozesssteuerung bei Orchestrierung mit Prozess-Engine, Transaktionssteuerung, Business Activity Monitoring

Als Infrastruktur muss eine Middleware als Brücke zwischen verschiedenen Anwendungen oder Systemen fungieren und ihnen die Kommunikation und gemeinsame Nutzung von Daten und Funktionen ermöglichen. Damit wird die Kommunikation zwischen den verteilten System(-teilen) ermöglicht, um diese zu verbinden und einen reibungslosen Kommunikations- und Datenfluss zwischen ihnen zu gewährleisten. Es gibt verschiedene Arten von Middleware, darunter:

- Kommunikations-Middleware, für den Austausch von Nachrichten zwischen Anwendungen oder System(-teilen),
- Transaktions-Middleware, die den Datenfluss koordiniert und sicherstellt, dass Transaktionen auf konsistente und zuverlässige Weise ausgeführt werden,
- serverseitige Container (Middleware wie JEE (Enterprise Java Beans), Corba (mit Object Request Brokern), die es Objekten in verschiedenen Systemen ermöglicht, über ein Remote-Protokoll (RMI, IIOP, ...) miteinander zu kommunizieren,
- ereignisgesteuerte Middleware, die es Anwendungen ermöglicht, miteinander zu kommunizieren, indem sie Ereignisse auslöst und verarbeitet.

3. Integration auf dem Frontend-Tier / Präsentationsebene

Info
Ansgar Brauner: "Eine konkurrenzfähige Architektur für den Lebensmittelhandel", Talk bei der JUG Darmstadt - zu Microservices und UI-Integration, 2019, https://www.jug-da.de/2019/01/Microservices-Architektur/

embarc Software Consulting GmbH: "Architekturspicker Microservices", Seite 4 mit Abschnitt zu Strukturierungsoptionen und Technologien für das UI, https://www.architektur-spicker.de/

Luca Mezzalira: "Microfrontends Anti-Patterns: Seven Years in the Trenches", https://www.infoq.com/presentations/microfrontend-antipattern/

Till Schulte-Coerne: "Optionen der Frontend-Integration", https://www.innoq.com/de/articles/2019/08/frontend-integration/

Die **Integration auf der Präsentations-/Frontend-Schicht** umfasst in der Regel die Erstellung oder Änderung der Benutzeroberfläche, um Daten aus einem bzw. mehreren Backends anzuzeigen und dort Benutzerinteraktionen zu verarbeiten. Das Frontend kann mit verschiedenen Technologien erfolgen, typischerweise web-basiert. Die Integration erfolgt client- oder serverseitig, typische Technologien sind SPA (mit Type /JavaScript und Frameworks wie React, Angular oder Vue.js); Client-Side-Includes; Server-Side-Includes. Ein clientseitiger Integrationsprozess setzt Anfragen an eine Backend-API für Abfrage und Anzeigen von Daten ab, übernimmt Daten z.B. von und für Formulare zum Senden von Daten an das oder die Backend(s).

Eine solche Integration setzt i.d.R. voraus, dass es nötig ist, ein oder mehrere Frontends für Nutzer aufzubereiten, wenn diese noch nicht durch die Services im Backend (siehe Diskussion zu Microservices und SCS in Kapitel 3) selbst umfassend abgebildet werden. Dies kann beispielsweise dann nötig sein, wenn im UI sehr viele verschiedene Daten aus den Backend-Services zusammenzuführen, darzustellen, ggf. auch zu aggregieren oder transformieren sind. Außerdem können in diesen integrierten Frontends querschnittliche Dienste wie Authentifizierung /Autorisierung oder Session-Management zentral geregelt werden.

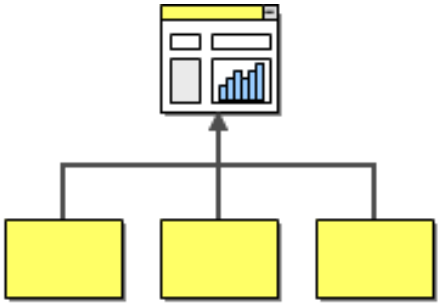


Abbildung: Information Portal
(Quelle: https://www.enterpriseintegrationpatterns.com/patterns/messaging/Chapter1.html)

Beispiele für solche Frontend-Integrationslösungen sind Portalanwendungen und Microfrontends:

- **Portalanwendungen** und Portlets sind kleine, in sich geschlossene Webanwendungen, die miteinander kombiniert werden können, um eine größere Webseite oder Anwendung zu erstellen. Sie werden in der Regel im Zusammenhang mit Java-basierten Unternehmensportalen und Web-Content-Management-Systemen verwendet. Ein Portlet ist eine Java-basierte Webkomponente, die gemäß der Java Portlet Specification (JSR-286) erstellt wurde und im Kontext eines Portals ausgeführt wird. Ein Portal fungiert als Container für Portlets, so dass diese auf einer einzigen Seite kombiniert werden können und über eine gemeinsame API miteinander interagieren. Jedes Portlet hat sein eigenes HTML-Markup, JavaScript und Style Sheets, nutzt aber die Infrastruktur des Portals für Authentifizierung, Personalisierung und andere Dienste. Portlets können mit verschiedenen Frameworks entwickelt werden und können hinzugefügt, entfernt oder ersetzt werden, ohne den Rest des Portals zu beeinträchtigen. Die Portlets können unabhängig voneinander entwickelt und separat verwaltet werden. Dies ermöglicht eine größere Flexibilität und Skalierbarkeit sowie eine modularere und wartungsfreundlichere Codebasis. Portlets werden häufig in Unternehmensumgebungen verwendet, um Intranet- und Extranet-Anwendungen zu erstellen, aber auch, um benutzerdefinierte Dashboards, Workflow-Management-Systeme und andere geschäftsspezifische Anwendungen zu entwickeln.
- **Microfrontends** sind ein architektonisches Muster für den Aufbau von Webanwendungen, bei dem ein monolithisches Frontend in kleinere, unabhängig voneinander einsetzbare Komponenten zerlegt wird. Diese Komponenten, oder Microfrontends, können getrennt voneinander entwickelt und eingesetzt werden und lassen sich über eine komponierbare Benutzeroberfläche in eine größere Anwendung integrieren. Jedes Microfrontend ist für ein bestimmtes Merkmal oder eine bestimmte Funktionalität der Gesamtanwendung verantwortlich und kann mit einem anderen Technologie-Stack oder Framework als die anderen entwickelt werden. Dies ermöglicht eine größere Flexibilität und Skalierbarkeit sowie eine modularere und besser wartbare Codebasis. Es gibt verschiedene Möglichkeiten, Microfrontends zu implementieren, darunter die Verwendung von Iframes, Webkomponenten oder Bibliotheken wie Single-spa. Dieser Architekturstil gibt Teams die Möglichkeit, unabhängig voneinander an verschiedenen Komponenten der Anwendung zu arbeiten, und um die Entwicklung und Bereitstellung verschiedener Teile der Anwendung zu entkoppeln.

Wie die UI fachlich und technisch in das Gesamtsystem der Services integriert ist, lässt sich wie folgt unterscheiden:

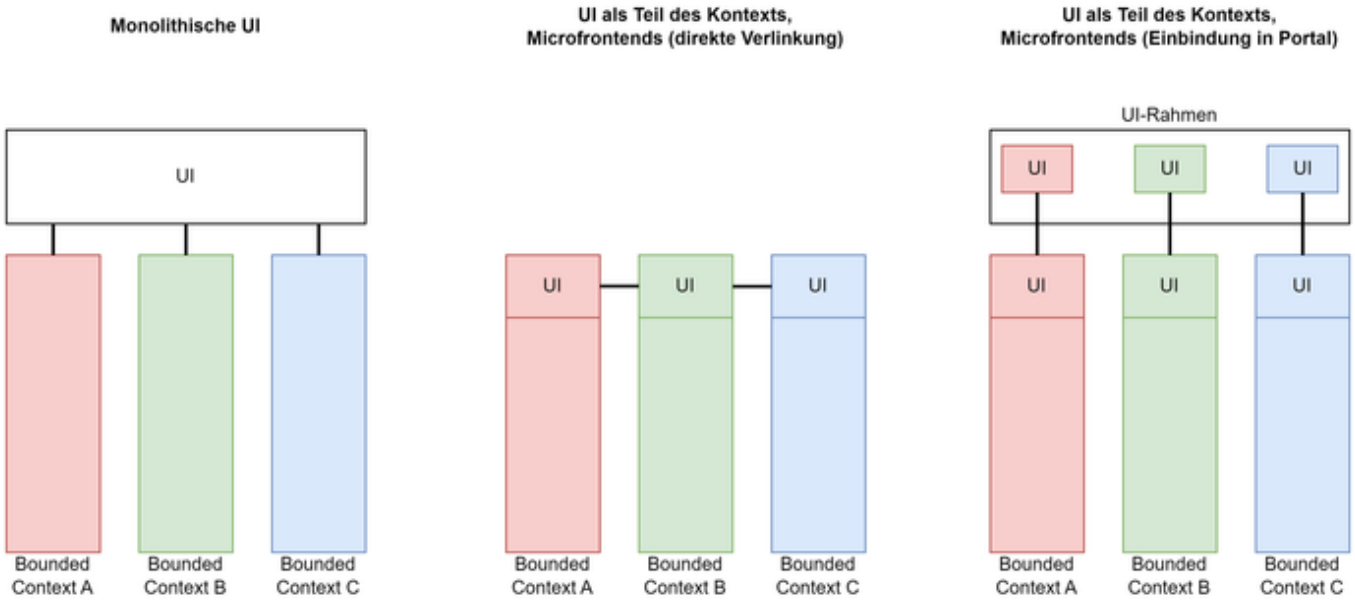


Abbildung: Monolithische UI - Microfrontends - mit Portal

4. Integration auf der Daten-Tier / Daten-Ebene

Bei der **Integration von Systemen auf der Datenebene** geht es in der Regel darum, den Datenaustausch / die Integration über gemeinsam genutzte Datenquellen zu implementieren.

Umsetzung ist möglich durch:

- Gemeinsam genutzte Datenbanken im Sinne einer **Shared Database**, z. B. relationalen Datenbanken wie MySQL, PostgreSQL oder NoSQL-Datenquellen wie MongoDB
- Integration über filebasierte Datenquellen wie CSV-Dateien, Excel-Tabellen oder andere flache Dateien und Lesen von Daten aus diesen Dateien.
- Verbindung zu zentralen Datenquellen, wie zu Amazon S3
- Verbindung zu Data-Warehousing-Systemen wie beispielsweise Amazon Redshift oder Google BigQuery
- Verbindung zu Data Lakes, wie Azure Data Lake Storage
- Verbindung zu anderen Datenquellen, wie Suchmaschinen, Graph-Datenbanken oder Key-Value-Stores, und Abfrage von Daten über APIs oder andere Schnittstellen.

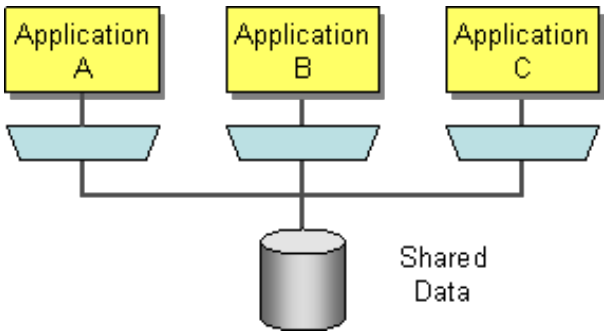


Abbildung: Shared Database
(Quelle: https://www.enterpriseintegrationpatterns.com/patterns/messaging/SharedDataBaseIntegration.html)

Die Integrationslösung kann die Datenumwandlung, Datenvalidierung und Datenmapping bereits umfassen oder zumindest optional anbieten, um sicherzustellen, dass die Daten das richtige Format und die richtige Struktur haben und von anderen Systemen genutzt werden können.

Alternativ werden Daten zwar nicht "shared" genutzt, aber über Replikation in verschiedene Kopien überführt. Dann kommt i.d.R. **Extract-Transform-Load (ETL)** zum Einsatz. Mit ETL bezeichnet man die Schritte, um Daten aus einer oder verschiedenen Quellen zu lesen / zu sammeln, sie so umzuwandeln, dass sie den Anforderungen des Zielsystems entsprechen, und sie dann zur Analyse und Berichterstattung in das Zielsystem zu laden.

1. Extract: Der erste Schritt von ETL besteht darin, Daten aus verschiedenen Quellen zu extrahieren. Dazu können Datenbanken, Flat Files oder APIs gehören. Die Daten werden in der Regel in ihrem Rohformat extrahiert und liegen möglicherweise nicht in einem Format vor, das vom Zielsystem problemlos verwendet werden kann.
2. Transform: Der nächste Schritt ist die Transformation der extrahierten Daten. Dies kann die Bereinigung und Normalisierung der Daten sowie die Konvertierung in ein Format umfassen, das in das Zielsystem geladen werden kann. Dieser Schritt kann auch die Anwendung von Geschäftslogik und Berechnungen auf die Daten umfassen.
3. Load: Der letzte Schritt besteht darin, die transformierten Daten in das Zielsystem zu laden. Dies kann das Laden von Daten in ein Data Warehouse, einen Data Lake oder ein Berichtssystem umfassen. Die Daten sind nun für die Analyse und Berichterstattung bereit.

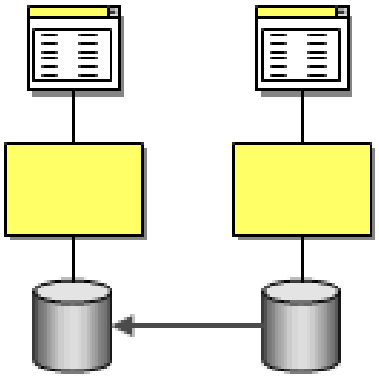


Abbildung: Data Replication
(Quelle: https://www.enterpriseintegrationpatterns.com/patterns/messaging/Chapter1.html)

ETL wird häufig für die Datenintegration und das Data Warehousing verwendet und dient in der Regel dazu, Daten zwischen verschiedenen Systemen und Formaten zu verschieben und umzuwandeln.

In skalierbaren, service-orientierten Architekturen ist eine Shared-Database-Lösung i.d.R. ein Anti-Pattern, da die einzelnen Services voneinander technisch wie fachlich unabhängig (weiter)entwickelt werden sollen.

Mathias Verraes bringt das in seinem Tweet ironisch auf den Punkt:

Abbildung: Shared Database
(Quelle: Tweet von Mathias Verraes
https://twitter.com/mathiasverraes/status/711168935798902785)

5. Organisatorische Patterns: Drehstuhl und Turnschuhnetzwerk

Es gibt eine Reihe von Anti-Patterns, mit denen man versucht, über organisatorische Mittel eine unzureichende oder fehlender IT-Integration zu kompensieren. Beispiele:

Das **"Swivel Chair"-Muster** ist ein organisatorisches Integrationsmuster, das häufig in Unternehmen verwendet wird, in denen verschiedene Systeme und Anwendungen integriert werden müssen, aber keine zentrale IT-Abteilung oder ein spezielles Integrationsteam vorhanden ist. Der Name des Musters leitet sich von der Vorstellung ab, dass Nutzer in ihren Stühlen sich von einer zur anderen Anwendung "drehen" und so die Integration manuell vornehmen, also Daten doppelt pflegen.

Das **"Turnschuhnetzwerk"** ist ein weiteres solches organisatorisches Integrationsmuster, das häufig verwendet wird, wenn größere Datenmengen nicht einfach automatisch zwischen Systemen transferiert werden können (weil eine technische Schnittstelle nicht vorhanden ist, mit Mengengerüsten oder Detailanforderungen nicht klarkommt etc.). Der Datentransfer erfolgt dann "per Turnschuh", also über manuelle Übertragung und organisiert einen solchen Datentransfer z.B., in dem die Daten per USB-Stick hin- und hergetragen werden.

Abbildung: xkcd #949, "File Transfer"
(Quelle: https://xkcd.com/949/)

6. Architekturstile zur Aufteilung auf Makro- oder Enterprise-Ebene

Info
Prof. Dr. Markus Voß u.a.: "Quasar Enterprise. Anwendungslandschaften serviceorientiert gestalten", dpunkt-Verlag

Es gibt verschiedene **Integrationsmuster und -architekturen,** die zur Integration von Systemen/Anwendungen bzw. ganzer Anwendungslandschaften verwendet werden können. Dies geht typischerweise über die Fragestellungen von System-Integration und Schnittstellen zwischen Anwendungen oder Anwendungsteilen hinaus und betrachtet eine grobgranularere Ebene.

Integration wird notwendig, wenn ein Monolith in Teile aufgeteilt wird, die für sich eigenständig laufen (Services, Microservices etc.) und dann remote kommunizieren müssen. Jeder Dienst ist für eine bestimmte Aufgabe oder Funktion zuständig und kann von anderen Diensten oder Systemen aufgerufen werden. Getrennte Dienste ermöglichen es idealerweise, dass jeder Dienst unabhängig entwickelt und bereitgestellt werden kann, und erleichtert die Skalierung und Weiterentwicklung der Anwendung im Laufe der Zeit, somit die lose Kopplung zwischen Diensten /Systemen und damit auch technische wie auch fachliche als auch organisatorische Skalierung.

Auf grobgranularer Ebene bieten sich diese Architekturstile an:

- Die **Microservices-Architektur** (bzw. SCS, siehe Kapitel 3) unterteilt eine Anwendung in eine Sammlung kleiner, lose gekoppelter Dienste, die über ein Netzwerk unter Verwendung von leichtgewichtigen Protokollen wie Rest/HTTP (synchron) oder Message-Queueing (asynchron) kommunizieren.
- **Serviceorientierte Architektur** (SOA) ist ein Architekturstil, bei dem eine Anwendung in eine Reihe von Diensten unterteilt wird, die über ein Netz mit Hilfe von Webdiensten oder anderen Protokollen aufgerufen werden können.
- **Pipes- und Filter-Architektur:** Ein Architekturstil, bei dem die Daten durch eine Reihe von Filtern geleitet werden, die die Daten transformieren und verarbeiten/weiterreichen, typischerweise zustandslos. Dieses Muster wird häufig in Datenintegrationsszenarien verwendet, in denen Daten umgewandelt und validiert werden müssen, bevor sie von anderen Systemen genutzt werden können.
- **Shared Database:** Ein Architekturstil, bei dem Daten zwischen Systemen und Anwendungen unter Verwendung einer zentralisierten Datenbank gemeinsam genutzt werden. Dieses Muster wird häufig in Szenarien verwendet, in denen Daten über verschiedene Systeme und Anwendungen hinweg gemeinsam genutzt werden müssen. Eine zentrale Datenbank wird als "die Quelle der Wahrheit" verwendet, und alle Systeme greifen über diese Datenbank auf die Daten zu. Entsprechend sind alle Services an und über diese Datenquelle eng aneinander gekoppelt, was Änderungen sehr schwierig macht.
- **Event-driven Architecture** (EDA): Siehe Abschnitt 4 H
- **API-Gateway:** Ein Architekturstil, bei dem ein einziger Einstiegspunkt für externe Verbraucher bereitgestellt wird, damit diese auf Backend-Dienste zugreifen können. Dieses Muster wird häufig verwendet, um zentral Sicherheits-, Ratenbegrenzungs- und Routing-Funktionen bereitzustellen.

7. Patterns für Enterprise Application Integration (EAI)

Info
Gregor Hohpe und Bobby Woolf: "Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions", Addison Wesley, 2004

Martin Fowler (Hrsg.): "Patterns für Enterprise Application-Architekturen", mitp, 2003

https://www.enterpriseintegrationpatterns.com/patterns/messaging/IntegrationStylesIntro.html

https://www.enterpriseintegrationpatterns.com/patterns/messaging/toc.html

https://camel.apache.org/components/3.20.x/eips/enterprise-integration-patterns.html

Enterprise Application Integration (EAI) Patterns sind eine Reihe bewährter Lösungsmuster für die Integration verschiedener Anwendungen.

Die Standard-Literaturreferenz für EAI-Muster ist "*Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*" von Gregor Hohpe und Bobby Woolf. Dieses Buch bietet einen umfassenden Überblick über EAI-Muster und enthält detaillierte Erklärungen und Beispiele aus der Praxis und hat auch eine Symbolsprache für die Patterns etabliert. Es gilt als das Referenzwerk auf diesem Gebiet.

Die Patterns nennen **Integrationsstile** (File Transfer, Shared Database, RPC und Messaging) sowie mehr als 60 **Messaging Patterns** mit

- Messaging Systems (darunter Pipes & Filters)
- Channels (darunter Point-to-Point, Publish-Subscribe)
- Message Konstruktion (u.a. von Command oder Event Messages),
- Message Routing (darunter Filter, Splitter, Aggregator, Broker, Dynamic Router)
- Message Transformation (darunter Wrapper, Enricher, Normalizer)
- Message Endpoints (darunter Gateway, Polling Consumer, Dispatcher)

Diese Patterns können kombiniert und zusammen verwendet werden, um ein breites Spektrum an Integrationsproblemen zu lösen. Diese Muster bieten eine Abstraktion der zugrundeliegenden Technologien und Protokolle, so dass sich die zugrundeliegenden Technologien leicht ändern lassen, ohne dass der Anwendungscode geändert werden muss.

Die wichtigsten Patterns sind:

Message Router: Dieses Pattern wird verwendet, um Nachrichten auf der Grundlage bestimmter Kriterien wie Nachrichteninhalt oder Absender an das richtige Ziel zu leiten.

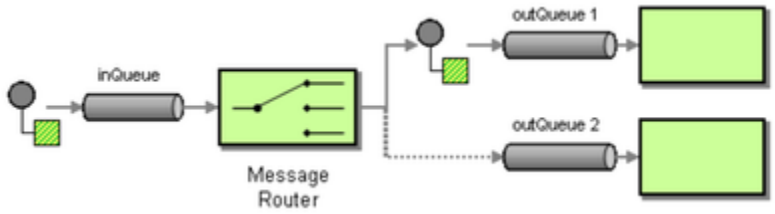


Abbildung: Message Router
(Quelle: https://www.enterpriseintegrationpatterns.com/patterns/messaging/MessageRouter.html)

Message Translator: Dieses Pattern ermöglicht die Konvertierung von Nachrichten von einem Format in ein anderes, wodurch die Kommunikation zwischen verschiedenen Systemen ermöglicht wird.

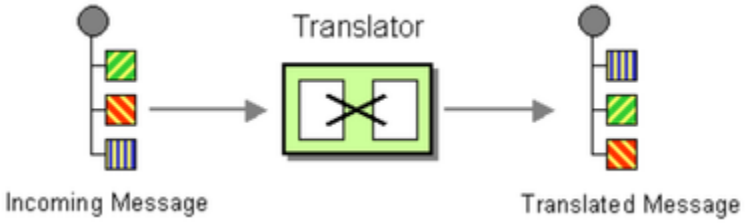


Abbildung: Message Translator
(Quelle: https://www.enterpriseintegrationpatterns.com/patterns/messaging/MessageTranslator.html)

Message Endpoint: Dieses Pattern wird zum Empfangen und Senden von Nachrichten verwendet.

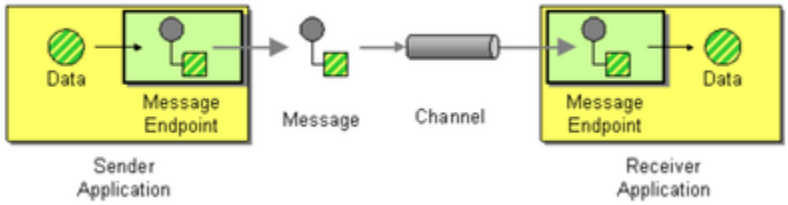


Abbildung: Message Endpoint
(Quelle: https://www.enterpriseintegrationpatterns.com/patterns/messaging/MessageEndpoint.html)

Message Channel: Dieses Pattern definiert den Kanal, über den Nachrichten gesendet und empfangen werden, z. B. über eine Nachrichtenwarteschlange oder eine direkte Verbindung.

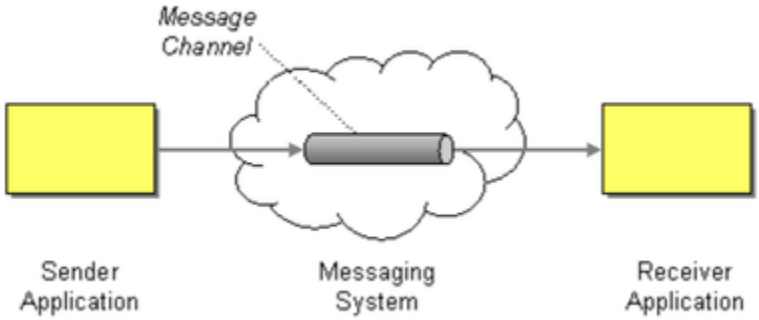


Abbildung: Message Channel
(Quelle: https://www.enterpriseintegrationpatterns.com/patterns/messaging/MessageChannel.html)

Message Selector: Dieses Pattern wird verwendet, um Nachrichten nach bestimmten Kriterien zu filtern, z. B. nach dem Absender oder dem Inhalt der Nachricht.

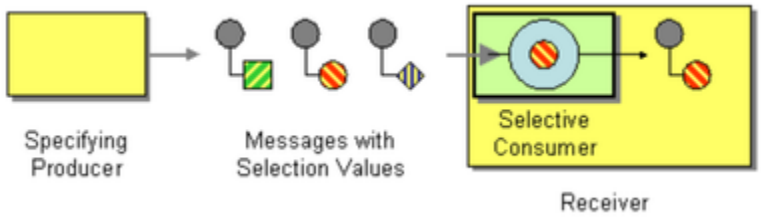


Abbildung: Message Selector
(Quelle: https://www.enterpriseintegrationpatterns.com/patterns/messaging/MessageSelector.html)

Message Sequence: Mit diesem Pattern können Nachrichten in einer bestimmten Reihenfolge gesendet werden, um sicherzustellen, dass sie in der richtigen Reihenfolge verarbeitet werden.

Message Dispatcher: Dieses Pattern wird verwendet, um Nachrichten an mehrere Empfänger zu verteilen.

Message Broker: Dieses Pattern fungiert als zentrale Drehscheibe für die Weiterleitung und Verwaltung von Nachrichten und bietet eine Möglichkeit, verschiedene Systeme und Anwendungen miteinander zu verbinden.

(C) Integration: Kern-Konzepte der Integration auf Anwendungsebene

Info
Für die Integration auf **Anwendungsebene** (auch: Logikebene) sind verschiedene **Muster und Eigenschaften der Kommunikation zwischen den Integrationspartnern** zu unterscheiden:

- Anzahl der beteiligten Kommunikationspartner (auf beiden Seiten)
- Muster der Interaktion (siehe unten in Abschnitt 1.)
  - zustandslos vs. zustandsbehaftet
  - synchron vs. asynchron
- Muster der Kommunikation
  - Kommunikationsmuster wie Request-Response, Fire-and-Forget (als Punkt-zu-Punkt oder Multicast oder Broadcast), Publish-Subscribe
  - Umsetzung der Kommunikation wie Methodenaufruf, Messaging, Ressourcen, Dateiübertragung (siehe unten in Abschnitt 2.)
- Push vs. Pull sowie Datenfluss vs. Kontrollfluss (siehe unten in Abschnitt 4.)
- Quality of Service

1. synchron - asynchron

Die **synchrone Integration** bezieht sich auf eine Integrationsform, bei dem zwei Systeme oder -teile miteinander kommunizieren und der anfragende Teil nach Absenden der Anfrage (des Requests) blockiert wird, bis eine Antwort (eine Response) empfangen wird.

- Beispiel "Online-Transaktionen": Wenn ein Kunde einen Kauf auf einer E-Commerce-Website tätigt, muss das System die Verfügbarkeit des Produkts prüfen, die Zahlung bestätigen und den Bestand in Echtzeit aktualisieren. Die synchrone Integration wird verwendet, um sicherzustellen, dass der Kunde sofort eine Bestätigung der Transaktion erhält.

Bei der Implementierung einer synchronen Schnittstelle sollten verschiedene Faktoren explizit berücksichtigt sein, die durch die Blockierung des Anfragers verursacht werden:

1. Die synchrone Kommunikation erfordert eine geringe Latenzzeit. Daher müssen die Systeme so konzipiert und eingesetzt werden, dass die Zeit, die für die Bearbeitung einer Anfrage und die Rückgabe einer Antwort benötigt wird, möglichst gering ist.
2. Da der Prozess blockiert wird, bis eine Antwort eintrifft, müssen die Systeme hochverfügbar und fehlertolerant sein, um sicherzustellen, dass die Kommunikation nicht unterbrochen wird. Das System muss in der Lage sein, Fehler, wie z. B. Zeitüberschreitungen und fehlgeschlagene Anfragen, zu behandeln, um sicherzustellen, dass die Kommunikation nicht unterbrochen wird.
3. Skalierbarkeit: Das System muss in der Lage sein, die erwartete Last zu bewältigen, sowohl in Bezug auf die Anzahl der Anfragen als auch auf den Umfang der ausgetauschten Daten. Es muss mehrere Anfragen gleichzeitig bearbeiten und die Ressourcen effizient verwalten können, um eine Überlastung des Systems zu vermeiden.

Die synchrone Request/Response-Denkweise ist gut verstanden und damit für Integration oft erste Wahl, da sie gut verstandene lokale Aufrufe in eine Remote-Kommunikation "anhebt". Sie trägt aber nicht weit: Die Latenzzeiten in verteilten Systemen liegen um Größenordnungen über denen lokaler Aufrufe innerhalb eines Monolithen. Solche Latenzen können sich akkumulieren wegen transitiver Abhängigkeiten und Aufrufe, mit entsprechend schlechter Gesamtperformance (... von Fehlerszenarien an dieser Stelle zu schweigen, mehr dazu siehe unten).

Asynchrone Kommunikation erfolgt dagegen zeitlich unabhängig und ermöglicht den Austausch von Informationen ohne direkte "Echtzeit"-Interaktion zwischen den Teilnehmern. Anders als synchrone Kommunikation erfordert dies also keine gleichzeitige Anwesenheit aller Beteiligten und kein unmittelbares Feedback. Asynchrone Kommunikation bietet Flexibilität und die Möglichkeit, Nachrichten zu senden und zu empfangen, ohne auf eine sofortige Antwort warten zu müssen, während synchrone Kommunikation die direkte Interaktion erfordert.

Vor- und Nachteile:

- Die synchrone Request/Response-Denkweise ist attraktiv, da gut verstanden: Oft wählt man für eine Integration eine synchrone Kommunikation, da sie die gut verstandene und "gewohnte" lokalen Aufrufe in eine Remote-Kommunikation "anhebt".
- Über synchrone Integration kann Anfrage-Antwort-Kommunikation einfach abgebildet werden, da der synchrone Call mit einer Antwort endet. Dagegen ist eine Antwort bei asynchroner Kommunikation aufwändiger, da ja zunächst keine Antwort auf die Anfrage zurückgeschickt wird. Möglich ist eine Antwort jedoch: Sendet ein Client eine asynchrone Anfrage an einen Server, so verarbeitet dieser die Anfrage und kann zu einem späteren Zeitpunkt eine Antwort an den Client zurücksenden. Dieser muss dann aber eine solche späte Antwort der Anfrage auch zuordnen können, z.B. über Korrelation mittels einer Request-Id.
- Asynchrone Kommunikation ist im Allgemeinen flexibler und performanter, da sie keine blockierende Kommunikation zwischen System(-teilen) erfordern. Dies ermöglicht einen flexibleren Datenfluss und die Bewältigung eines höheren Anfragevolumens.
- Asynchrone Kommunikation kann zwischengepuffert werden, z.B. über eine Message-Queue (oder Topic). Anfragen werden als Nachrichten zuerst in die Queue gestellt, bevor sie von dort entnommen und abgearbeitet werden. Ein solches Vorgehen in Messaging-Systemen mit JMS, bei Queueing-Systemen wie ActiveMQ oder bei Eventbus-Systemen wie Apache Kafka erlauben dabei auch Persistierung und transaktionale Nachrichten-Verarbeitung.

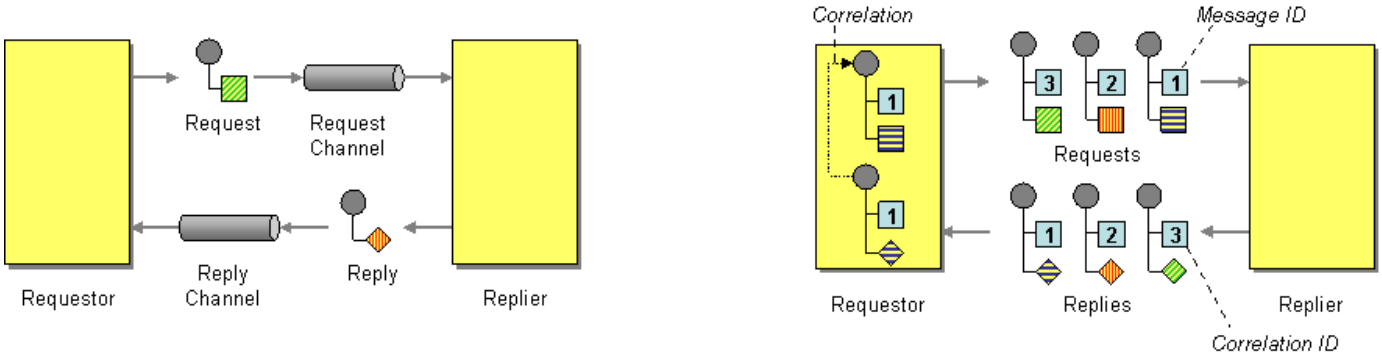
2. zustandslos - zustandsbehaftet

Zustandslos bedeutet, dass das angefragte System keine Informationen über frühere Anfragen oder Sitzungen speichert.

Zustandsbehaftet oder **-abhängig** dagegen bedeutet, dass das System Informationen über frühere Anfragen oder Sitzungen aufbewahrt. Zustandslose Services skalieren deutlich besser.

3. Muster der Kommunikation

Request-Response (oder: Request-Reply) ist ein Kommunikationsmuster, bei dem eine Anfrage durch eine explizite Antwort beantwortet wird. Dies kann im einfachsten Fall synchron umgesetzt sein, aber auch eine asynchrone Umsetzung ist möglich - dann muss aber die Response-Nachricht der Request-Nachricht zuordenbar sein, z.B. über ID oder über einen Correlation-Identifier).



Abbildungen: Request-Response; Correlation-ID
(Quelle: https://www.enterpriseintegrationpatterns.com/patterns/messaging/RequestReply.html, https://www.enterpriseintegrationpatterns.com/patterns/messaging/CorrelationIdentifier.html)

Fire-and-Forget (als Punkt-zu-Punkt oder Multicast oder Broadcast) ist dagegen ein Kommunikationsmuster, bei dem eine Anfrage nicht oder höchstens durch eine Eingangsbestätigung ("Acknowledgement") beantwortet wird.

Publish-Subscribe (Pub-Sub) ist ein Messaging-Pattern, das eine Entkopplung der Systeme ermöglicht, die Daten produzieren und konsumieren. Bei diesem Pattern sendet eine Komponente (der Producer oder Publisher) eine Nachricht, und eine andere Komponente (der Consumer oder Subscriber) empfängt sie. Der Publisher muss nicht wissen, wer die Subscriber sind oder wie viele es gibt, und die Subscriber müssen nicht wissen, wer der Publisher ist.

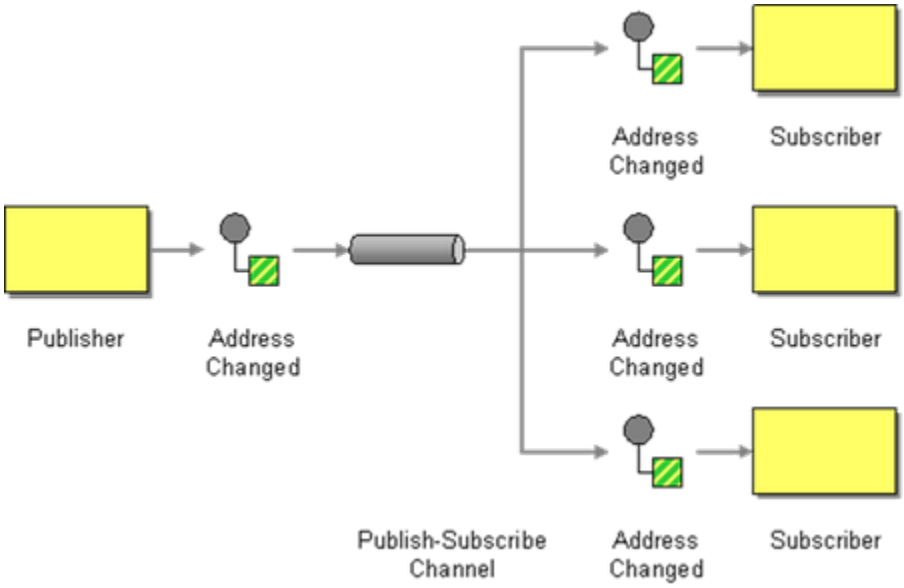


Abbildung: Publish Subscribe
(Quelle: https://www.enterpriseintegrationpatterns.com/patterns/messaging/PublishSubscribeChannel.html)

Das Hauptmerkmal von Pub-Sub ist, dass der Publisher eine Nachricht in ein Topic oder einen Channel sendet und eine beliebige Anzahl von Subscriber dieses Topic abhören kann, ohne dass der Publisher wissen oder sich darum kümmern muss, wer sie sind. Dies ermöglicht ein flexibleres und skalierbares System, da die Anzahl der Subscriber dynamisch wachsen und schrumpfen kann, ohne dass dies Auswirkungen auf den Publisher hat. Vgl. Kafka in Kapitel 7.

Es gibt verschiedene Arten von Pub-Sub-Messaging-Systemen, einige verwenden Messaging-Warteschlangen, andere Nachrichten-Broker. Eine Nachrichten-Warteschlange speichert Nachrichten, bis ein Subscriber bereit ist, sie zu empfangen, während ein Nachrichten-Broker Nachrichten

auf der Grundlage vordefinierter Regeln an die Subscriber weiterleitet. Pub-Sub wird üblicherweise in Szenarien verwendet, in denen Daten an mehrere Systeme gesendet werden müssen bzw. in denen Daten parallel verarbeitet werden müssen, wobei die lose Kopplung von Sender und Empfänger gewahrt werden soll.

4. Umsetzung der Kommunikation

Kommunikation kann über diese Varianten umgesetzt werden (siehe auch EAI-Integrationsstile, siehe bei 4 (F), 3. "EAI"):

- **Methodenaufrufe:** System(-teile) stellen Teile ihrer Funktionalität nach außen zur Verfügung (lokal bzw. remote per RPC). Tendenziell erfolgt dies über enge Kopplung und synchron.
- Beim **Messaging** werden Nachrichten (Messages) ausgetauscht. Tendenziell erfolgt dies über lose Kopplung und asynchron.
- **Ressourcen** sind als digitale Objekte über eine eindeutige ID (z.B. per URI) ansprechbar und stehen über Aufrufe zur Verfügung.
- **Dateiübertragung** nutzt filebasierte Kommunikation.

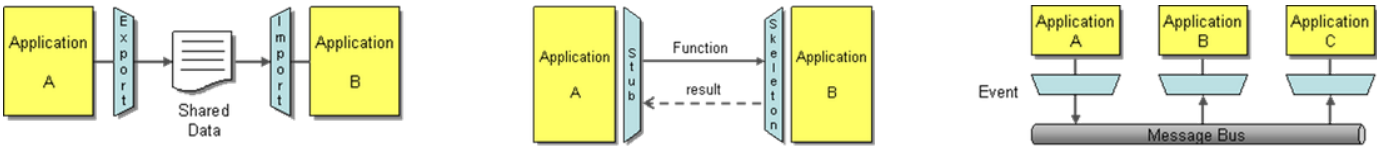


Abbildung: Dateiübertragung - Remote Procedure Invocation - Messaging,
(Quelle: https://www.enterpriseintegrationpatterns.com/patterns/messaging/IntegrationStylesIntro.html)

Für diese Varianten kommen unterschiedliche Middleware-Systeme und -Produkte zum Einsatz.

	Dateiübertragung	Remote Procedure Invocation	Messaging	Ressourcen
Kopplung	eher lose	eher eng	eher lose	eng oder lose
Synchronizität	asynchron	synchron	asynchron	eher synchron
Datenmengen für Anfrage/Ergebnis	beliebig	eher klein	eher klein	eher klein
Verständnis des Modells	intuitiv, einfach	intuitiv, einfach	schwieriger	schwierig
Technologieunabhängigkeit	gering	gering	hoch	hoch
Persistenz?	ja	nein	möglich	nein
Transaktionalität?	i.d.R. nein	nein	möglich	nein
Anzahl Kommunikationspartner	1:1 oder 1:n	nur 1:1	beliebig: entweder 1:1 oder n:m	n:1
Beispiel	FTP- oder SFTP-Server für den Dateiaustausch von typischerweise großen Datenmengen	Remote Method Invocation (RMI)	Java Messaging System (JMS) Apache Kafka	Representational State Transfer (REST), HTTP-Anfragen für POST (Erstellen), PUT (Aktualisieren), GET (Lesen) und DELETE (Löschen)

5. Push - Pull

Push und Pull bezeichnen die Konzepte der Datenübertragung in einem System.

- **Pull** bezieht sich auf eine Methode, bei der das empfangende System aktiv Daten vom sendenden System anfordert. Zum Beispiel eine Client-Anwendung, die Daten von einem Server abruft, oder eine Service, der Daten aus einer Datenbank abruft. Typisches Kommunikationspattern ist Request-Response, i.d.R. als synchroner Aufruf.
- **Push** hingegen bezieht sich auf eine Methode, bei der Daten von einem System an ein anderes gesendet werden, ohne dass das empfangende System diese explizit anfordert. Typisches Kommunikationspattern ist Publish-Subscribe, über asynchrone Nachrichten.

Im Allgemeinen ist Push effizienter für die Datenübertragung in Echtzeit und für Systeme, die mehrere Clients über dieselben Aktualisierungen informieren müssen, während Pull eher für Systeme geeignet ist, bei denen der Client für die Anforderung der von ihm benötigten spezifischen Daten verantwortlich sein möchte.

6. Datenfluss - Kontrollfluss

Datenfluss und Kontrollfluss sind verschiedene Konzepte, die sich auf den Fluss von Informationen und Anweisungen in einem System beziehen.

- Der **Datenfluss** bezieht sich auf die Bewegung von Daten durch ein System, von der Quelle bis zum Ziel. Er beschreibt, wie Daten umgewandelt, gespeichert und zwischen verschiedenen Teilen eines Systems bewegt werden.
- Der **Kontrollfluss** hingegen bezieht sich auf die Reihenfolge und Initiatoren, in der Anweisungen in einem System gestartet und ausgeführt werden. Er beschreibt die Abfolge der Schritte, die ein Programm oder System unternimmt, um eine Aufgabe zu erfüllen oder eine bestimmte Funktion auszuführen.

Bei der Integration von System(teil)en ändert sich der Datenfluss nicht, sobald die fachlichen Verantwortlichkeiten festgelegt sind. Dagegen ändert sich der Kontrollfluss je nach Integrationsart. Beispiel: Service B benötigt Daten von Service A. Der Datenfluss ist demnach A -> B. Der Kontrollfluss unterscheidet sich nach Art der Integration, z.B.:

- Ruft B den Service A auf und "pullt" sich aktiv die Daten, so ist der Kontrollfluss B -> A.
- Wird Publish-Subscribe benutzt, so "pusht" A aktiv die Daten und initiiert so die Kommunikation. B ist dann der (oder ggf. auch nur ein) Subscriber. Der Kontrollfluss ist hier A -> B.

7. Orchestrierung - Choreography

Info
https://solace.com/blog/microservices-choreography-vs-orchestration/

Tobias Flohre: "Wer Microservices richtig macht, braucht keine Workflow Engine und kein BPMN", https://blog.codecentric.de/wer-microservices-richtig-macht-braucht-keine-workflow-engine-und-kein-bpmn

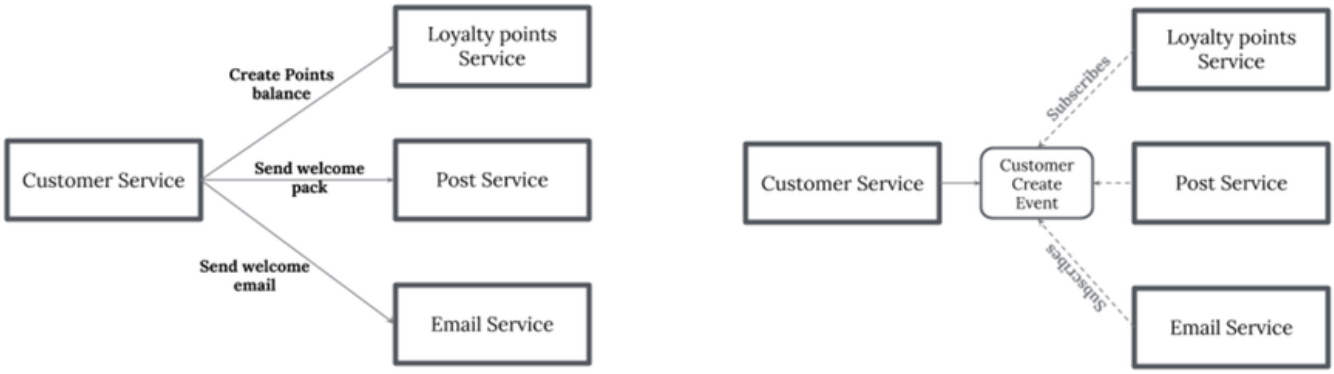


Abbildung: Orchestrierung vs. Choreographie
(Quelle: Sam Newman, "Building Microservices: Designing Fine-Grained Systems")

Orchestrierung und Choreographie sind zwei Ansätze zur Koordinierung der Interaktionen zwischen verschiedenen Softwarekomponenten, Diensten oder Systemen.

- Mit **Orchestrierung** bezeichnet man die zentralisierte Steuerung und Verwaltung verschiedener Komponenten oder Dienste. Sie beinhaltet eine zentrale "Orchestrator"-Komponente ähnlich zu einem Dirigenten in einem Musik-Orchester. Dieser Orchestrator ist für die Verwaltung der Interaktionen zwischen anderen Komponenten oder Diensten verantwortlich. Dieser Orchestrator verfügt in der Regel über ein detailliertes Wissen über die interne Funktionsweise der anderen Komponenten und ist für die Koordinierung ihrer Interaktionen und die Sicherstellung ihres korrekten Zusammenwirkens verantwortlich.
- Dagegen bezeichnet man mit **Choreografie** einen dezentralen Ansatz, bei dem jede Komponente oder jeder Dienst für die Koordinierung ihrer Interaktionen mit anderen Komponenten oder Diensten selbst verantwortlich ist. Die Komponenten oder Dienste kommunizieren direkt miteinander, ohne über einen zentralisierten Orchestrator gesteuert werden zu müssen.

Daher ist asynchrone Kommunikation in verteilten und hoch-skalierbaren Anwendungen erste Wahl. Asynchronität als Prinzip erfordert dann Choreographie und keine zentrale Orchestrierung, die letztlich einen "Flaschenhals" darstellt.

8. Quality of Service

Der "**Quality of Service**" stellt dar, welche **Dienstgütezusicherung (Delivery Guarantees)** die Infrastruktur zusichert**.**

Bei der Kommunikation und dem Nachrichtenaustausch zwischen den Kommunikationspartnern ist "die **Qualität"** der Zustellung wichtig:

- At-Most-Once: Nachrichten werden maximal einmal zugestellt, können also verloren gehen.
- At-Least-Once: Nachrichten werden mindestens einmal, ggf. mehrfach zugestellt. Ein Konsument muss mit Nachrichtendoppelung umgehen können, sprich idempotent sein (siehe auch Kapitel 8).
- Exactly-Once: Nachrichten werden garantiert einmal zugestellt.

Die Performance dieser drei Stufen nimmt von oben nach unten ab.

Darüber hinaus gibt es noch die Frage nach **Reihenfolge-Garantie ("Ordering Guarantees"):** Werden Nachrichten in der richtigen Reihenfolge zugestellt? Ist also die Reihenfolge des Nachrichten-Versands diesselbe wie die des Nachrichten-Empfangs?

(D) Nutzen und Kosten verteilter Systeme (Konsistenz, Latenz, Resilience, Service Discovery, Skalierbarkeit)

Info
Wenn Modularisierung des Systems (vgl. Kapitel 3) die Einzelteile einer Anwendung oder einer Anwendungslandschaft so zerlegt, dass Integration des verteilten Systems nötig ist, so bringt dies **Herausforderungen** mit sich:

1. **Konsistente** Sicht auf die Daten im verteilten System
2. **Latenz:** Die Integration verschiedener Systeme und Anwendungen führt zu Latenz, also Verzögerungen bei der Datenverarbeitung und Kommunikation. Dies kann besonders in Echtzeit- oder zeitkritischen Szenarien kritisch sein. Durch die Überprüfung der Latenz in Last/Performance/Integrationstests bzw. im Betrieb ist sicherzustellen, dass das System unter den erwarteten Belastungen und Nutzungsszenarien gut funktioniert.
3. **Skalierbarkeit:** Die Integration verschiedener Systeme und Anwendungen stellt im Hinblick auf die Skalierbarkeit eine Herausforderung dar, vor allem, wenn das Lastverhalten wenig vorhersagbar ist bzw. über einen Zeitverlauf schwankt bzw. so groß ist, dass es mit "normalen" Mitteln nicht einfach zu beherrschen ist. Dies kann bedeuten, dass ein hohes Datenvolumen, eine hohe Geschwindigkeit und eine große Vielfalt an Daten bewältigt werden müssen und/oder dass das System mit steigenden bzw. schwankenden Lasten und Nutzungsmustern zurechtkommen muss.
4. **Fehlerszenarien (und Resilience):** Durch die Integration verschiedener Systeme und Anwendungen können auch neue Fehlerszenarien entstehen, die in den einzelnen Komponenten nicht vorhanden waren, darunter Ausfälle, Mehrfachaufrufe, Timeouts etc.
5. **Sicherheit:** Authentifizierung und Autorisierung sind zwingend in einer solche Schnittstelle mit zu designen und zu implementieren. Je offener die Schnittstelle ist, desto größer werden die Herausforderungen (vgl. auch Zero Trust). Siehe Abschnitt 4 (J)
6. **Verfügbarkeit**: Siehe Abschnitt 6 (C)

Neben Sicherheit, Latenz und Fehlerszenarien gibt es noch viele andere, organisatorisch-operative Herausforderungen bei der Integration, darunter:

1. **Datenintegration:** Die Integration verschiedener Datenquellen, -formate und -strukturen kann eine Herausforderung sein. Dies kann den Umgang mit Dateninkonsistenzen, Datenduplizierung und Datentransformation beinhalten.
2. **Interoperabilität:** Die Integration verschiedener Systeme und Anwendungen, die unterschiedliche Protokolle und Standards verwenden, kann den Umgang mit Kompatibilitätsproblemen und die Überbrückung der Kluft zwischen verschiedenen Technologien notwendig machen.
3. **Integration von Geschäftsprozessen:** Die Integration unterschiedlicher Geschäftsprozesse und Arbeitsabläufe kann den Umgang mit Inkonsistenzen in der Geschäftslogik, die Handhabung unterschiedlicher Granularitätsebenen und die Koordinierung verschiedener Geschäftseinheiten beinhalten.
4. **Steuerung und Verwaltung:** Die Integration verschiedener Systeme und Anwendungen kann aus Sicht der Unternehmensführung und des Managements eine Herausforderung darstellen. Dies kann den Umgang mit mehreren Anbietern, Service Level Agreements und die Notwendigkeit einer zentralisierten Verwaltungs- und Überwachungslösung beinhalten.
5. **Änderungsmanagement:** Die Integration verschiedener Systeme und Anwendungen kann eine Herausforderung darstellen, wenn es um die Verwaltung von Änderungen geht. Dabei geht es um Versionskontrolle, Abwärtskompatibilität und die Gewährleistung, dass sich Änderungen an einem Teil des Systems nicht negativ auf die anderen Teile auswirken.

Diese Herausforderungen können spezialisierte Lösungen und Fachwissen erfordern, und ihre Lösung kann komplex und zeitaufwändig sein. Sie müssen proaktiv angegangen werden, um sicherstellen, dass die Integrationslösungen robust und zuverlässig sind und den Anforderungen und Qualitätseigenschaften gerecht werden.

1. Konsistenz

Konsistenz ist in daten-orientierten (verteilten) Systemen wichtig, da sie sicherstellt, dass **alle Knoten im System die gleiche Sicht auf die Daten** haben. Dies ist besonders wichtig in Systemen, in denen mehrere Benutzer gleichzeitig dieselben Daten aktualisieren, da es Konflikte verhindert und sicherstellt, dass alle Benutzer dieselben Daten sehen.

Konsistenz kann auf verschiedene Weise erreicht werden, z. B.

- durch die Verwendung von **Sperren** zur Verhinderung gleichzeitiger Aktualisierungen,
- durch die Verwendung eines **Consensus-Algorithmus** zur Lösung von Konflikten
- oder durch die Verwendung eines **Master-Slave-Replikationsmodells.**

1.1. Konsistenz: ACID vs. BASE

Info
Andrew Tanenbaum, Marten van Steen, Distributed Systems – Principles and Paradigms, Prentice Hall, 2nd Edition, 2006

Leslie Lamport, The Part-Time Parliament, ACM Transactions on Computer Systems 16, 2 (May 1998), 133-169

Eric Brewer, Towards Robust Distributed Systems, PODC Keynote, July-19-2000

Mikito Takada, Distributed Systems for Fun and Profit, http://book.mixu.net/distsys/

Uwe Friedrichsen: "Real World Consistency explained"

- https://www.slideshare.net/ufried/realworld-consistency-explained-80984942
- https://www.herbstcampus.de/2018/veranstaltung-7089-datenkonsistenz-in-der-realen-welt0ae2.html?source=&id=7089
- Blog-Serie https://www.ufried.com/blog/no_acid_1/ https://www.ufried.com/blog/no_acid_2/ https://www.ufried.com/blog/no_acid_3/ https://www.ufried.com/blog/no_acid_4/

ACID und BASE sind zwei unterschiedliche Ansätze für den Umgang mit Daten in einem Datenbanksystem. ACID beschreibt eine Reihe von Eigenschaften, die sicherstellen, dass Datenbanktransaktionen zuverlässig verarbeitet werden, wobei der Schwerpunkt auf Konsistenz und Isolierung liegt. Hingegen konzentriert sich BASE auf Verfügbarkeit und Partitionstoleranz und ist für verteilte Systeme konzipiert , bei denen hohe Schreibanforderungen zu erwarten sind und die Konsistenz nicht so entscheidend ist wie die Verfügbarkeit.

ACID (Atomicity, Consistency, Isolation, Durability) ist eine Reihe von Eigenschaften, die gewährleisten, dass Datenbanktransaktionen zuverlässig verarbeitet werden, v.a. für Situationen, wenn mehrere Transaktionen gleichzeitig stattfinden und das System sicherstellen muss, dass sie sich nicht gegenseitig beeinträchtigen.

- A = Atomicity/Atomarität stellt sicher, dass eine Transaktion als eine einzige, unteilbare Arbeitseinheit (atomar) behandelt wird und dass entweder alle ihre Operationen abgeschlossen werden oder keine von ihnen.
- C = Consistency/Konsistenz gewährleistet, dass eine Transaktion die Datenbank von einem gültigen Zustand in einen anderen bringt.
- I = Isolation/Isolierung stellt sicher, dass konkurrierende Transaktionen sich nicht gegenseitig beeinträchtigen.
- D = Durability/Dauerhaftigkeit stellt sicher, dass die Auswirkungen einer festgeschriebenen Transaktion alle nachfolgenden Ausfälle überdauern.

Mit Einsatz einer ACID-Datenbank allein ist es nicht getan: Der **Isolation Level** der Datenbank bestimmt und ist entscheidend, inwieweit eine Transaktion die Änderungen einer anderen Transaktion sehen kann (vgl. Uwe Friedrichsens Talk). Höhere Isolationsebenen bieten eine größere Datenkonsistenz, können aber zu einer geringeren "Concurrency" führen, während niedrigere Isolationsebenen eine höhere "Concurrency" bieten, zum Preis einer geringeren Datenkonsistenz. Die Wahl der Isolationsebene hängt von den spezifischen Anforderungen der Anwendung ab. Der Default-Level ist typischerweise READ COMMITTED. Die vier **Standardisolationsebenen** sind:

- READ UNCOMMITTED ermöglicht es Transaktionen, nicht bestätigte (uncommitted) Änderungen anderer Transaktionen zu sehen.
- READ COMMITTED ermöglicht es Transaktionen, nur die von anderen Transaktionen vorgenommenen bestätigten (committed) Änderungen zu sehen.
- REPEATABLE READ stellt sicher, dass eine Transaktion immer denselben Datensatz sehen kann, auch wenn eine andere Transaktion die Daten zwischendurch ändert.
- SERIALIZABLE sorgt dafür, dass Transaktionen seriell ausgeführt werden, wodurch Konflikte zwischen Transaktionen vermieden werden.

BASE (Basically Available, Soft-State, Eventually Consistent) ist ein Ansatz für die Datenpartitionierung, bei dem die Partitionierung auf der Art der Daten und der damit durchzuführenden Operationen basiert und nicht auf der Struktur der Datenbank. Es wurde für Situationen entwickelt, in denen ein verteiltes System ein hohes Maß an Schreibanfragen bewältigen muss und in denen höherere Anforderungen an Verfügbarkeit als an Konsistenz gestellt werden.

- BA = Basically available / grundsätzlich verfügbar bedeutet, dass das System die meiste Zeit die Verfügbarkeit der Daten garantiert.
- S = Soft-State bedeutet, dass sich der Zustand des Systems im Laufe der Zeit ändern kann, auch ohne Eingaben.
- E = Eventually Consistent bedeutet, dass das System irgendwann konsistent wird, aber der Zeitrahmen, in dem dies geschieht, kann nicht garantiert werden.

Es ist wichtig zu beachten, dass die Wahl eines bestimmten Konsistenzmodells von den Anforderungen der Anwendung und den akzeptablen Kompromissen abhängt.

- ACID ist in Situationen erforderlich, in denen die **Datenkonsistenz und -integrität** kritisch sind und mehrere **Transaktionen gleichzeitig** stattfinden. Diese Art von Systemen erfordert in der Regel ein hohes Maß an Datenkonsistenz und -isolierung, z. B. bei Finanztransaktionen, bei denen jede Transaktion genau und unabhängig von anderen Transaktionen verarbeitet werden muss.
- Andererseits ist **BASE** in Situationen ausreichend, in denen die **Datenkonsistenz fachlich nicht kritisch** ist und die Systemanforderungen ein gewisses Maß an **Dateninkonsistenz** tolerieren kann. Diese Art von Systemen erfordert in der Regel ein **hohes Maß an Verfügbarkeit und Partitionstoleranz**, z. B. bei Social-Media-Plattformen, wo das System eine große Anzahl von Schreibanfragen bewältigen muss und wo eine (kleine) Verzögerung bei der Datenkonsistenz akzeptabel ist.

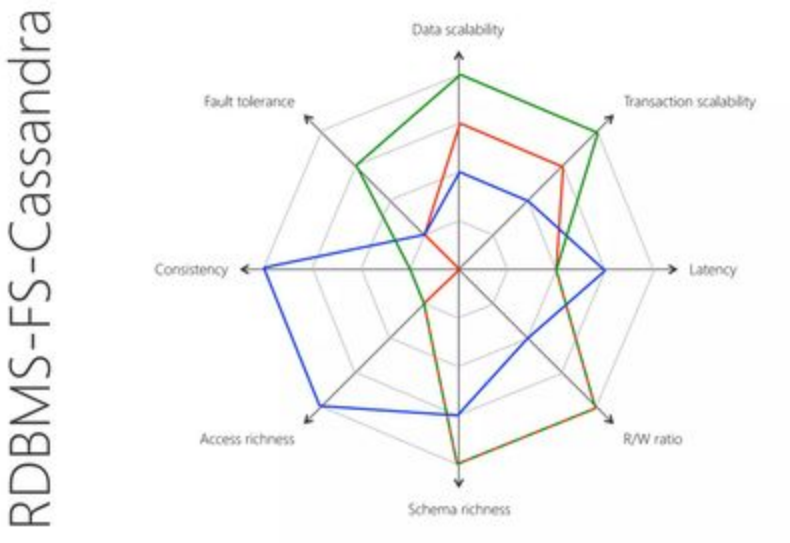


Abbildung: Eigenschaften von RDBMS (blau), File-Systemen (rot) und Cassandra (grün) als eine NoSQL-DB (Quelle: https://www.slideshare.net/ufried/realworld-consistency-explained-80984942)

1.2. Konsistenz: Eventual Consistency

Info
Susanne Braun: verschiedene Vorträge und Videos zu Eventual Consistency, darunter:

- https://www.youtube.com/watch?v=ujwdaEHjba4
- https://www.youtube.com/watch?v=VO7Fip5VuSc
- Folien u.a. https://www.slideshare.net/SusanneBraun2/eventual-consistency-jug-da

Eventual Consistency ist eine schwächere Form der Konsistenz: In einem solchen System haben alle Knoten **letztendlich** ("eventually") die gleiche Sicht auf die Daten - aber es gibt bis dahin eine (längere) Zeitspanne, in der einige Knoten veraltete oder inkonsistente Daten haben. Ein solches Konsistenzmodell ist ein Kompromiss für erhöhte Verfügbarkeit und Skalierbarkeit.

Eventual Consistency wird häufig in verteilten Systemen verwendet, die hohe Schreiblasten aufweisen und horizontal skaliert werden müssen, wie NoSQL-Datenbanken, oder in Systemen, die auch dann funktionieren müssen, wenn einige der Knoten nicht verfügbar sind, wie Peer-to-Peer-Netzwerke.

Es ist wichtig zu beachten, dass die Konsistenz nur ein Aspekt ist, der in verteilten Systemen berücksichtigt werden muss. Andere Qualitätsmerkmale wie Verfügbarkeit, Leistung und Skalierbarkeit sind wichtig und müssen mit der Konsistenz abgeglichen werden (vgl. Qualitätsmerkmale und Trade-Offs).

1.3. Konsistenz: NoSQL-Datenbanken

Info
Pramod J. Sadalage, Martin Fowler: "NoSQL Distilled - A Brief Guide to the Emerging World of Polyglot Persistence", Addison-Wesley

Martin Fowler zu NoSQL:

- Introduction to NoSQL, Vortrag auf der GOTO 2012, https://www.youtube.com/watch?v=qI_g07C_Q5I
- Info-Sammlung, https://martinfowler.com/data/index.html#nosql

NoSQL-Datenbanken können grob in die folgenden **Hauptkategorien** eingeteilt werden:

1. **Document Databases (dokumentenorientiert)** speichern Daten in halbstrukturierten Dokumentenformaten, wie JSON oder XML. Beispiele hierfür sind MongoDB, Couchbase und RethinkDB.
2. **Column Databases** (mit **Spaltenstruktur)** speichern Daten in Tabellen mit Zeilen und Spalten, ähnlich wie relationale Datenbanken, aber mit flexibleren Datenmodellen. Beispiele sind Apache Cassandra und Hbase.
3. **Key-Value-Datenbanken** speichern Daten als Schlüssel-Wert-Paare, wobei der Schlüssel ein eindeutiger Bezeichner für den Wert ist. Beispiele hierfür sind Redis und Riak.
4. **Graph-Datenbanken** speichern Daten als Knoten und Kanten in einem Graphen, wobei Beziehungen zwischen Datenelementen als Kanten dargestellt werden. Beispiele hierfür sind Neo4j und ArangoDB.
5. In **Objektdatenbanken** werden Daten als Objekte gespeichert, ähnlich wie in objektorientierten Programmiersprachen. Beispiele hierfür sind ZODB, db4o und Versant Object Database.

6. **Zeitserien-Datenbanken** speichern Daten als eine mit einem Zeitstempel versehene Folge von Werten. Sie sind für die Verarbeitung zeitabhängiger Daten optimiert und werden zur Speicherung von Daten wie Metriken, Sensordaten und Finanzdaten verwendet. Beispiele hierfür sind InfluxDB und TimescaleDB.

Einige der Datenbanken fallen in mehrere dieser Kategorien. Jede NoSQL-Datenbank/-kategorie hat ihre eigenen spezifischen Stärken, Schwächen und Anwendungsfälle, so dass die Wahl der richtigen Datenbank von den spezifischen Anforderungen der Anwendung und den Arbeitslasten abhängt, die die Datenbank bewältigen muss.

Einige der bekanntesten und verbreitesten **NoSQL-Produkte** sind:

1. MongoDB: Eine dokumentenorientierte Datenbank, die ein flexibles JSON-ähnliches Schema verwendet.
2. Couchbase: Eine dokumentenorientierte Datenbank, die ein flexibles JSON-ähnliches Schema verwendet und Volltextsuche unterstützt.
3. RethinkDB: Eine quelloffene, dokumentenorientierte Datenbank, die Daten-Push in Echtzeit und flexible Abfragen unterstützt.
4. Cassandra: Eine hoch skalierbare und verfügbare Datenbank, die große Datenmengen über viele Server hinweg verarbeiten kann.
5. Hbase: Eine spaltenbasierte NoSQL-Datenbank, die auf dem Hadoop Distributed File System aufbaut.
6. Redis: Ein In-Memory-Key-Value-Store, der als Datenbank, Cache und Message Broker verwendet werden kann.
7. Neo4j: Eine Graphdatenbank, die ein flexibles Eigenschaftsgraphenmodell zur Speicherung und Abfrage von Daten verwendet.
8. DynamoDB: Ein vollständig verwalteter NoSQL-Datenbankdienst, der von Amazon Web Services bereitgestellt wird.
9. CosmosDB: Ein global verteilter Multi-Model-Datenbankdienst, der von Microsoft Azure bereitgestellt wird.
10. Elasticsearch: Eine verteilte Suchmaschine - als dokumentenorientierte NoSQL database, speziell design für Volltextsuche und Real-time Analytics.

1.4. Konsistenz: Wie kann Konsistenz in einem verteilten System *erreicht* werden?

Info
Martin Kleppmann: "Designing Data-Intensive Applications", O'Reilly. https://www.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/

M. Tamer Özsu , Patrick Valduriez: "Principles of Distributed Database Systems", Springer 2020, https://link.springer.com/book/10.1007/978-3-030-26253-2

https://microservices.io/patterns/data/transactional-outbox.html

Das Erreichen von Konsistenz in einem verteilten System ist typischerweise sehr komplex und mit Trade-Offs in Bezug auf Leistung und Skalierbarkeit verbunden. Welcher Ansatz am besten geeignet ist, hängt von den spezifischen Anforderungen und Beschränkungen des Systems ab. Es gibt mehrere Techniken, die zur **Erreichung von Konsistenz in einem verteilten System** eingesetzt werden können:

1. **Zwei-Phasen-Commit** (2PC): Dieses Protokoll stellt sicher, dass sich alle Knoten in einem verteilten System auf einen einzigen Wert für ein bestimmtes Datenelement einigen. Es erfordert, dass alle Knoten miteinander kommunizieren, bevor sie eine Änderung festschreiben. 2PC hat diverse Nachteile, vor allem mit Skalierbarkeit und Durchsatz:

- Problem mit Skalierbarkeit: 2PC basiert auf Koordinierung zwischen den beteiligten Parteien. Insbesondere muss jede am 2PC teilnehmende Partei die Vorbereitungsphase und die Übergabe bestätigen. Sobald eine Partei bestätigt hat, dass sie zur Übergabe bereit ist, muss sie blockieren, bis der Transaktionskoordinator die Commit- oder Rollback-Nachricht sendet.
- Außerdem muss eine Partei - sobald sie bestätigt hat, dass sie bereit ist, die Transaktion zu committen - auch in der Lage sein, dies selbst dann zu tun, wenn sie zwischenzeitlich abgestürzt ist. Dies erfordert ein persistentes Checkpointing und dieses schränkt daher i.d.R. den Durchsatz ein.
- Performance und Durchsatz der Transaktionen sind durch das langsamste "Service-Glied in der Kette" limitiert.
- Ein 2PC wird i.d.R. durch einen zentralen Koordinator orchestriert. Dieser ist ein potentieller Single-Point-of-Failure.

2. Statt des 2PC wird gerne das "**Outbox Pattern**" implementiert

- Gleichzeitig mit der Persistenz in eine lokale (oft: relationale) Datenhaltung wird in *derselben* Datenbank eine Nachricht in eine Messaging-Datenbanktabelle geschrieben, damit fachliche Daten und Nachricht in *derselben* Datenbank-Transaktion geschrieben werden.
- Aus dieser Outbox-Queue liest dann das andere System die Daten aus. Die Outbox-Tabelle dient somit als leichtgewichtiger Ersatz für eine externe Infrastruktur (wie Kafka).
- Folglich ist *kein* 2PC über zwei Datenbanken nötig.

3. Andere Techniken, allesamt nicht-trivial und komplex:

- **Paxos** und **Raft** sind Consensus-Algorithmen, die es einer Gruppe von Knoten ermöglichen, sich auf einen Wert zu einigen. Sie werden häufig in verteilten Systemen eingesetzt, um die Konsistenz der replizierten Daten zu gewährleisten.
- **Quorum-basierte Replikation:** Bei diesem Ansatz werden Daten über mehrere Knoten repliziert, wobei die Konsistenz durch ein Quorum von Knoten sichergestellt wird. Er kann in Situationen verwendet werden, in denen die Mehrheit der Knoten einem Wert zustimmen muss, bevor er als konsistent angesehen werden kann.
- Ein **Versionsvektor** ist eine Datenstruktur, die die Versionen von Daten auf verschiedenen Knoten in einem verteilten System aufzeichnet. Er kann verwendet werden, um Inkonsistenzen in replizierten Daten zu erkennen und aufzulösen.
- **Konfliktfreie replizierte Datentypen** (Conflict-free Replicated Data Types, **CRDT)** sind eine Klasse von Datenstrukturen, die über verschiedene Knoten in einem verteilten System repliziert werden können und garantieren, dass letztendlich alle Replikate gleichwertig sind; sie benötigen keine zentrale Koordination.

- **Synchrone Replikation:** Dieser Ansatz stellt sicher, dass alle Knoten in einem verteilten System die gleichen Aktualisierungen zur gleichen Zeit erhalten. Es erfordert, dass alle Knoten miteinander kommunizieren, bevor sie eine Änderung vornehmen, wie bei 2PC. Nachteile und Probleme: Siehe Cap-Theorem und Netzwerk-Partitionierung unten!

1.4. Konsistenz: Wie kann Konsistenz in einem verteilten System *kompensiert* werden?

Info
https://microservices.io/patterns/data/saga.html

https://www.baeldung.com/cs/saga-pattern-microservices

Uwe Friedrichsen, "Beyond the SAGA pattern", 2016ff, https://speakerdeck.com/ufried/beyond-the-saga-pattern

Die Kompensation mangelnder Konsistenz in einem verteilten System ist mit Abstrichen bei der Datengenauigkeit und -verfügbarkeit verbunden. Welcher Ansatz am besten geeignet ist, hängt von den spezifischen Anforderungen und Beschränkungen des Systems ab. Es gibt verschiedene Techniken, die verwendet werden können, um **mangelnde Konsistenz in einem verteilten System zu kompensieren**:

1. **SAGA-Pattern:**
   - Das SAGA-Pattern ist ein Architekturansatz, um verteilte Transaktionen in Microservice-Systemen zu koordinieren. Es besteht aus einer Sequenz von lokal durchgeführten Transaktionen (als lokale (technische) Transaktionen abgebildet).
   - Diese Transaktionen können als *kompensierende* Aktionen rückgängig gemacht werden, falls eine der Transaktionen fehlschlägt: Ein Zurückrollen erfolgt also durch eine zweite Aktion als "fachliche Kompensation". (Beispiel: "Betrag auf Konto buchen" Kompensation mit: "Betrag von Konto abbuchen").
   - Im SAGA-Pattern müssen Transaktionen idempotent und wiederholbar sein.
2. **Read Repair:** Bei dieser Technik wird die Konsistenz der Daten beim Lesen geprüft und die gefundenen Inkonsistenzen werden repariert. Die Daten werden im Laufe der Zeit konsistent.
3. **Auflösung von Konflikten:** Bei diesem Ansatz geht es darum, Konflikte zu erkennen und aufzulösen, wenn sie auftreten. Dies kann mit Techniken wie Versionsvektoren, CRDTs oder einer benutzerdefinierten Konflikt-Lösungslogik geschehen.
4. **Read your own writes**: Bei dieser Technik werden die Daten von demselben Knoten gelesen, auf den sie geschrieben wurden, wodurch die Gefahr des Lesens veralteter Daten verringert wird.
5. **Hinted Handoff:** Bei dieser Technik wird eine temporäre Kopie der Daten auf einem anderen Knoten gespeichert, für den Fall, dass der Knoten, der die Daten empfangen sollte, ausfällt oder nicht verfügbar ist. Wenn der Knoten wieder in Betrieb ist, werden die Daten dann an ihn geliefert.
6. **Last Write wins**: Bei diesem Ansatz werden Konflikte dadurch gelöst, dass immer der letzte Schreibvorgang als korrekter Wert gewählt wird.
7. **Time-to-live (TTL)**: Bei diesem Ansatz wird ein Zeitlimit für die Gültigkeit der Daten festgelegt, nach dem sie als veraltet gelten und aktualisiert werden sollten.

1.6. Konsistenz: CAP-Theorem und Partitionstoleranz

Das **CAP-Theorem** besagt, dass es für ein verteiltes System unmöglich ist, alle drei der folgenden Garantien gleichzeitig zu bieten: **Konsistenz (Consistency), Verfügbarkeit (Availability) und Partitionstoleranz**.

- Konsistenz bedeutet, dass alle Knoten die gleichen Daten zur gleichen Zeit sehen (siehe oben)
- Verfügbarkeit garantiert, dass jede Anfrage eine Antwort erhält, ohne aber die Garantie, dass diese die aktuellste Version der Informationen enthält.
- Partitionstoleranz bedeutet, dass das System weiterhin funktioniert, auch wenn eine beliebige Anzahl von Nachrichten durch das Netz zwischen den Knoten gestoppt (oder in der Kommunikation stark verzögert) wird.
  Partitionstoleranz stellt sicher, dass das System auch bei Netzausfällen noch funktioniert, wenn es zu Ausfällen oder Unterbrechungen zwischen den verschiedenen Komponenten kommt und so Nachrichten verlorengehen. Sie bedeutet also, dass das System in der Lage ist, Netzwerkpartitionen zu tolerieren, bei denen einige Teile des Systems nicht mit anderen Teilen kommunizieren können. Die (getrennten) Einzelknoten arbeiten also weiter und reagieren auf Kundenanfragen, auch wenn die Netzwerkkommunikation zwischen ihnen unterbrochen ist. Dadurch bleibt das System auch bei Netzwerkausfällen oder -unterbrechungen verfügbar und reaktionsfähig.

Nach dem CAP-Theorem kann ein verteiltes System höchstens zwei dieser drei Garantien gleichzeitig bieten, siehe Abbildung:

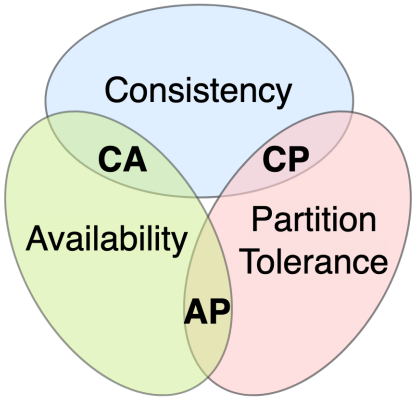


Abbildung: CAP Theorem
(Quelle: https://en.wikipedia.org/wiki/CAP_theorem#/media/File:CAP_Theorem_Venn_Diagram.png)

Verkürzt gesagt kann also nach dem CAP-Theorem ein verteiltes System höchstens zwei der drei Eigenschaften C, A und P gleichzeitig bieten.

Diese Aussage ist jedoch verkürzt und eigentlich falsch, denn man kann "P" nicht gezielt an- oder abwählen.

- Grund: Netzwerkausfälle passieren - das ist nicht zu verhindern, sondern die Regel, mit der man umgehen muss.
- Im Falle einer solchen Netzwerk-Partitionierung muss sich das System also zwischen "C" (Konsistenz) und "A" (Verfügbarkeit) entscheiden (also CP oder AP). Ein System, das wie eine relationale Datenbank die Konsistenz und Partitionstoleranz in den Vordergrund stellt, kann beispielsweise keine hohe Verfügbarkeit bei Netzwerkpartitionen garantieren. Andererseits kann ein System, das Verfügbarkeit und Partitionstoleranz in den Vordergrund stellt, wie eine NoSQL-Datenbank, keine starke Konsistenz garantieren.
- Überspitzt: Wenn man Consistency *und* Availability benötigt, darf man kein verteiltes System designen, in dem Partitionierung auftreten kann, muss also ein monolithisches System bauen.

2. Latenz

Info
Latenz: https://imgur.com/8LIwV4C

https://speakerdeck.com/gernotstarke/zahne-putzen-haare-kammen

Latenz kann enorme Auswirkungen haben, wie der Vergleich in der Abbildung veranschaulicht. Latenz muss daher immer sorgfältig überwacht werden.

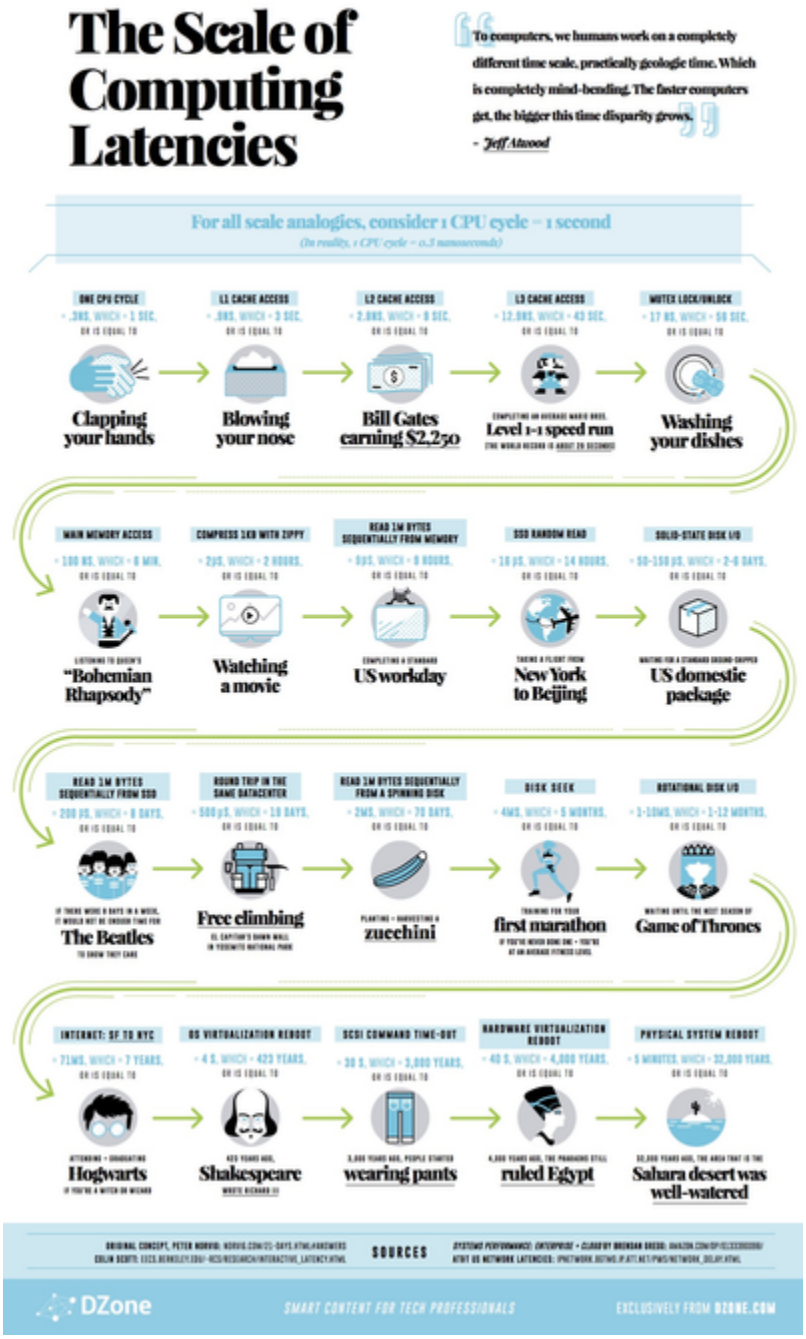


Abbildung: Latenz im Vergleich - 1 CPU-Cycle = 1 Sekunde
(Quelle: https://imgur.com/8LIwV4C)

Die Überwachung der **Latenzzeit** in einem verteilten System kann mit verschiedenen Techniken und Werkzeugen erfolgen, darunter:

1. Netzwerk-Überwachungstools: Mit diesen Tools kann der **Netzwerkverkehr** überwacht werden, einschließlich Paketverlust, Jitter und Umlaufzeit, die allesamt Indikatoren für Latenzprobleme sein können.
2. Tools zur Überwachung der **Anwendungsleistung:** Mit diesen Tools kann die Leistung der einzelnen Komponenten des verteilten Systems überwacht werden, einschließlich der Anwendungs-, Datenbank- und Netzwerkschichten. Sie können detaillierte Metriken über die Antwortzeit von Anfragen liefern, was ein Indikator für hohe Latenz sein kann.
3. Werkzeuge zur **Protokollanalyse:** Mit diesen Tools können Protokolldaten des verteilten Systems analysiert werden, die Informationen über das Timing von Anfragen und Antworten liefern, die zur Ermittlung von Latenzproblemen verwendet werden können.
4. **End-to-End-Tests:** Mit dieser Technik kann die reale Nutzung des verteilten Systems simuliert und die Zeit gemessen werden, die Anfragen für die Durchquerung des Systems benötigen.
5. **Verteiltes Tracing:** Die verteilte Rückverfolgung ist eine Methode zur Verfolgung des Flusses einer Anfrage durch die verschiedenen Komponenten eines verteilten Systems. Mit Hilfe eines Tracing-Tools, das die Zeitinformationen einer Anfrage erfassen kann, lässt sich die Ursache von Latenzproblemen ermitteln.
6. Eine **synthetische Überwachung** ermöglicht die Simulation von Benutzerinteraktionen und die Messung der Antwortzeit einer Anwendung. Sie kann dazu verwendet werden, Latenzprobleme zu erkennen und zu melden, die mit anderen Überwachungstechniken möglicherweise nicht sichtbar sind.

Es ist wichtig zu beachten, dass die Überwachung der Latenz in einem verteilten System einen umfassenden Ansatz erfordert, der die Überwachung auf verschiedenen Ebenen des Systems und die Verwendung und Kombination mehrerer Tools und Techniken umfasst. Welcher Ansatz am besten geeignet ist, hängt von den spezifischen Anforderungen und Beschränkungen des Systems ab. Außerdem ist es wichtig, eine Baseline für das normale Systemverhalten zu haben und Warnungen und Benachrichtigungen einzurichten, wenn die Latenzzeit diese überschreitet.

3. Fehlerbehandlung

Die **Fehlerbehandlung** bei synchroner und asynchroner Integration ist elementar unterschiedlich:

Synchrone Kommunikation blockiert den Anfrager. Kommt keine oder nicht ausreichend schnell eine Antwort, so wird i.d.R. per Timeout die Anfrage abgebrochen. Sind nun System(-teile) nicht verfügbar oder kommt es zu dauernden bzw. sich akkumulierenden Verzögerungen, so hat dies Verzögerungen oder Ausfälle im gesamten Prozess zur Folge. Siehe auch Resilience-Patterns in Kapitel 8.

Bei der asynchronen Integration werden die Anfragen und Antworten in eine Warteschlange gestellt, was zu robusteren und fehlertoleranteren Systemen führt. Auch ist damit eine parallele Verarbeitung von Anfragen möglich, was die Skalierbarkeit des Systems deutlich erhöht. Dabei lassen sich die Ressourcen durch die Pufferung besser verwalten, da das System eine Vielzahl von Anfragen verarbeiten kann, ohne überlastet zu werden.

4. Umgang mit Überlast

Info
Phuoc Tran-Gia: "Analytische Leistungsbewertung verteilter Systeme". Springer-Verlag

Unabhängig von der Synchronizität der Kommunikation kann eine Situation in der Kommunikation der Kommunikationspartner entstehen, in dem der Empfänger der Anfragen/Daten überlastet wird, also zu viel Last wegen zu vieler (oder wegen zu großer und komplexer) Anfragen erhält:

- In einem **synchronen** Kommunikationsszenario werden dann die Anfrager blockiert, weitere Anfragen führen nur zu weiteren Blockaden.
- In einem **asynchronen** Kommunikationsszenario mit Pufferung der Nachrichten (über eine Queue o.Ä.) läuft in einem solchen Überlastszenario die "Infrastruktur voll".

4.1. Backpressure

In letzterem Szenario hilft **Backpressure** als Mechanismus, der in Systemen eingesetzt wird, um weiteren Datenfluss zu kontrollieren und zu verhindern, dass angefragte System mit weiteren und noch mehr Anfragen / Daten weiter überlastet wird. Er ermöglicht, den Anfragern zu signalisieren, dass es nicht in der Lage ist, eingehende Daten mit der aktuellen Geschwindigkeit zu verarbeiten, und das Senden von Daten zu verlangsamen oder zu stoppen, bis das System wieder in der Lage ist, sie zu verarbeiten. Der Rückstau wird also an den Anfrager weitergegeben (ggf. auch transitiv über mehrere Ebenen).

Backpressure ist wichtig, wenn Daten performant verarbeitet werden müssen und die Gefahr besteht, dass das System mit zu vielen Daten überlastet wird. Es trägt dazu bei, dass das System stabil und reaktionsfähig bleibt und Datenverluste vermieden werden.

Backpressure kann auf verschiedene Weise implementiert werden:

- Durch die Verwendung eines Puffers zur Speicherung der eingehenden Daten, wobei jeweils nur eine bestimmte Menge an Daten gespeichert werden darf.
- Durch die Verwendung eines Flusskontrollprotokolls. TCP nutzt einen Fenstermechanismus , um den Datenfluss zwischen dem Sender und dem Empfänger zu regulieren.

Es ist nicht trivial, die Größe eines Puffers vorab festzulegen. Die Größe kann entweder experimentell bestimmt werden, oder sie wird analytisch mittels Warteschlagentheorie ermittelt, siehe Little's Law unten.

4.2. Little's Law

Das **Little's Law** ist nach John D.C. Little benannt, der es 1961 erstmals veröffentlichte. Es wird häufig in den Bereichen Operations Research und Warteschlangentheorie zur Analyse und Optimierung von Systemen mit einem stetigen Artikelfluss verwendet, z. B. in Produktionsanlagen, Call Centern und Transportnetzen.

Das Gesetz besagt, dass die durchschnittliche Anzahl von Gegenständen in einem System (z. B. Anzahl Requests in einem Server, Kunden in einem Geschäft, Fahrzeuge in einem Stau, ...) gleich der durchschnittlichen Ankunftsrate (pro Zeiteinheit) ist, mit der Gegenstände in das System gelangen, multipliziert mit der durchschnittlichen Zeit, die jeder Gegenstand im System verbringt:

Durchschnittliche Anzahl = "Durchschnittliche Ankunftsrate" x "Durchschnittliche Zeit im System"

Das Little'sche Gesetz geht davon aus, dass sich das System in einem stabilen Zustand befindet, was bedeutet, dass die Ankunftsrate und die Abfertigungsrate über die Zeit konstant sind. In Fällen, in denen sich das System nicht in einem stabilen Zustand befindet, ist das Gesetz möglicherweise nicht anwendbar.

Mit dem Little's Law kann man die durchschnittliche Anzahl der abarbeitbaren Anfragen in einem System berechnen und umgekehrt die Zielkapazität des Systems festlegen. Mit Hilfe des Little'schen Gesetzes lassen sich Rückstau-Probleme erkennen und abmildern. Unter Rückstau versteht man eine Situation, in der die Geschwindigkeit, mit der Gegenstände in einem System produziert werden, die Geschwindigkeit übersteigt, mit der sie von nachgelagerten Systemen verarbeitet werden können, was zu einer Anhäufung von Gegenständen im System führt.

Die Skalierbarkeit von Serveranwendungen (z.B. im Thread-per-Request-Modell) wird durch Little's Law bestimmt, das somit Latenz, Concurrency und Durchsatz miteinander in Beziehung setzt:

- Bei einer gegebenen Dauer der Anfragebearbeitung (d. h. Latenzzeit) muss die Anzahl der von einer Anwendung gleichzeitig bearbeiteten Anfragen (d. h. Concurrency) proportional zur Ankunftsrate (d. h. Durchsatz) wachsen.
- Beispiel: Angenommen, eine Anwendung mit einer durchschnittlichen Latenzzeit von 50 ms erreicht einen Durchsatz von 200 Anfragen pro Sekunde, indem sie 10 Anfragen gleichzeitig verarbeitet. Damit diese Anwendung auf einen Durchsatz von 2000 Anfragen pro Sekunde skaliert werden kann, muss sie 10x mehr, also 100 Anfragen gleichzeitig verarbeiten.

5. Resilience

Info
Michael T. Nygard: "Release It! Design and Deploy Production-Ready Software", Pragmatic Programmers, 2017

Uwe Friedrichsen: "Resilient Functional Service Design", https://www.slideshare.net/ufried/resilient-functional-service-design

Uwe Friedrichsen: "Patterns of Resilience", *https://www.slideshare.net/ufried/patterns-of-resilience*

Uwe Friedrichsen: "Resilience reloaded - more resilience patterns", https://www.slideshare.net/ufried/resilience-reloaded-more-resilience-patterns

Resilientes Software Design adressiert die Entwicklung von Softwaresystemen, die mit Ausfällen, Fehlern und unerwarteten Bedingungen umgehen und sich von ihnen erholen können. Diese Art von Design stellt sicher, dass die Software auch bei "widrigen" Umständen im Fehlerfall weiter funktioniert und dass sie sich (schnell) von auftretenden Störungen erholen kann. Ein solches Design setzt Techniken wie Redundanz, Failover, Load Balancing und "Selbstheilung" ein, um sicherzustellen, dass die Software auch dann noch funktioniert, wenn einzelne Komponenten ausfallen. Dies adressiert folgende Aspekte:

- **Stabilität** bezieht sich auf die Fähigkeit eines Systems, auch bei hoher Belastung oder anderen Arten von Stress stabil und funktionsfähig zu bleiben. Techniken wie Rate Limiting oder Load-Balancing tragen dazu bei, dass das System auch bei hoher Belastung reaktionsfähig bleibt.
- **Availability, Verfügbarkeit:** Dieses Pattern bezieht sich auf die Fähigkeit eines Systems, für die Benutzer verfügbar und zugänglich zu bleiben, auch bei Ausfällen oder anderen Störungen. Dies kann durch Techniken wie Redundanz, Replikation und Failover erreicht werden. Diese Techniken tragen dazu bei, dass das System verfügbar bleibt, auch wenn einzelne Komponenten ausfallen (siehe auch Kapitel 6 (C)).
- **Graceful Degradation** beschreibt die Fähigkeit eines Systems, weiter zu funktionieren, auch wenn einige seiner Komponenten nicht verfügbar sind. Dies kann durch Techniken wie Fallback und Circuit Breaking erreicht werden. Diese Techniken ermöglichen es dem System, einen alternativen Dienst oder ein reduziertes Dienstniveau bereitzustellen, wenn ein primärer Dienst nicht verfügbar ist.
- **Failover** erlaubt einem System, schnell und automatisch auf eine Backup- oder Sekundärkomponente umzuschalten, wenn eine Primärkomponente ausfällt. Dies erfordert Techniken wie Replikation und Redundanz erreicht werden.

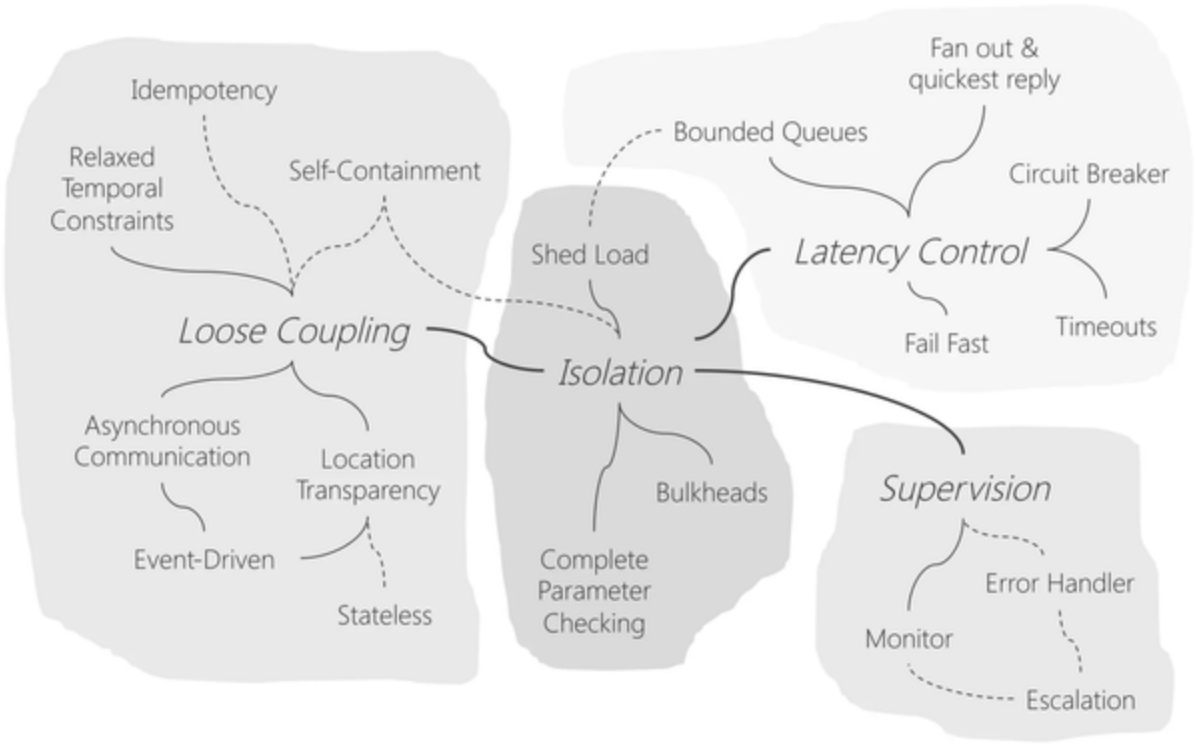


Abbildung: Resilience Patterns
(Uwe Friedrichsen: "Patterns of Resilience",
https://www.slideshare.net/ufried/patterns-of-resilience)

Dazu gibt es verschiedene **Standard-Patterns:**

1. Das **Circuit-Breaker-Pattern** verhindert, dass ein System wiederholt versucht, auf einen Dienst zuzugreifen, der mit Fehlern oder nicht antwortet. Es öffnet einen "Stromkreis", wenn es einen Fehler feststellt, verhindert so weitere Anfragen an den Dienst und ermöglicht dem System eine Wiederherstellung.
2. Das **Retry-Pattern** ermöglicht es dem System, fehlgeschlagene Operationen mehrfach automatisch zu wiederholen, um die Wahrscheinlichkeit eines erfolgreichen Abschlusses zu erhöhen. Typischerweise wird nach einer maximalen Anzahl von Wiederholungen beendet, außerdem wird der Zeitabstand zwischen zwei Retries immer größer (z.B. als Exponential Backoff).
3. Das **Bulkhead-Pattern** stellt sicher, dass Ausfälle in einem Teil des Systems sich nicht auf das gesamte System auswirken (ähnlich "Schotten" in einem Schiff). Es ermöglicht die Isolierung von Fehlern, so dass sich ein Fehler in einem Teil des Systems nicht auf den Rest des Systems auswirkt.
4. Das **Timeout-Pattern** setzt eine zeitliche Grenze für die Dauer eines Vorgangs. Dauert die Operation länger als das Zeitlimit, kann das System davon ausgehen, dass die Operation fehlgeschlagen ist und entsprechende Maßnahmen ergreifen.
5. Pattern für **Health Checks** ermöglichen dem System, den Zustand seiner Komponenten regelmäßig zu überprüfen, Fehler zu erkennen und entsprechende Maßnahmen zu ergreifen.

Diese Patterns bedienen den Fehlerfall. Proaktive Design-Techniken können damit ergänzt, um das System verfügbar und skalierbar zu gestalten:

1. Leader Election: Dieses Pattern ermöglicht es einem verteilten System, unter allen Knoten einen Leader zu wählen, der für die Koordinierung der Aktionen der Gruppe verantwortlich ist (z.B. Broker in Kafka).
2. Sharding-Pattern: Dieses Pattern ermöglicht eine horizontale Skalierung des Systems durch Aufteilung der Daten auf mehrere Knoten.
3. Replikation ermöglicht es dem System, mehrere Kopien von Daten über verschiedene Knoten hinweg zu verwalten, wodurch Redundanz und eine höhere Verfügbarkeit erreicht werden.

Die oben genannten Patterns (Retry, Bulkhead, Timeout etc.) sind für synchrone Integration/Kommunikation relevant.

Bei der **asynchronen Integration** und Kommunikation (z.B. Messaging, Eventbus) bietet die anders gelagerte Infrastruktur bereits grundlegend andere Eigenschaften, so dass folgende Konzepte und Patterns Anwendung finden:

1. **Message Queuing** verwendet eine Nachrichten-Queue, um Nachrichten zwischen Systemen zu puffern, so dass sie auch dann weiterverarbeitet werden können, wenn ein System vorübergehend nicht verfügbar ist. Dies stellt sicher, dass keine Nachrichten verloren gehen und die Systeme unabhängig voneinander weiterarbeiten können.

2. **Dead-Letter-Queues/Topic:** Bei diesem Pattern wird eine separate Queue für Nachrichten eingerichtet, die nicht erfolgreich verarbeitet werden können. Dies ermöglicht die Überwachung und Fehlersuche bei fehlgeschlagenen Nachrichten und deren eventuelle erneute Verarbeitung. Nachrichten aus dem Dead Letter müssen i.d.R. manuell korrigiert und bei Bedarf in die Kommunikation zurückgespielt werden.
3. **Ereignis-Sourcing:** Bei diesem Pattern werden alle Änderungen am Anwendungsstatus als eine Folge von Ereignissen gespeichert, anstatt den Status direkt zu aktualisieren. Dadurch kann sich die Anwendung von Fehlern erholen, indem sie die Ereignisse wieder abspielt und den Zustand rekonstruiert.
4. **Kompensations-Transaktionen:** Bei diesem Pattern wird eine Reihe von Aktionen ausgeführt, die im Falle eines Fehlers frühere Aktionen rückgängig machen oder "kompensieren". Dies kann dazu beitragen, dass das System auch bei Fehlern in einem konsistenten Zustand bleibt.
5. **Idempotenz** stellt sicher, dass mehrere identische Anfragen die gleiche Wirkung haben wie eine einzige Anfrage. Delivery-Garantie von effizienter Infrastruktur (z.B. Kafka) ist üblicherweise "At-least-once". I.d.R. garantiert die Infrastruktur keine Exactly-Once-Zulieferung garantiert, da dies erhebliche Performance-Einbussen zur Folge hätte.

Info
https://docs.spring.io/spring-kafka/api/org/springframework/kafka/listener/DefaultErrorHandler.html

https://github.com/spring-projects/spring-retry und https://www.baeldung.com/spring-retry

https://resilience4j.readme.io/docs und https://www.baeldung.com/spring-boot-resilience4j

Die Flexinale nutzt in der Variante "Distributed" den Spring-Boot-Errorhandler, siehe de.accso.flexinale.common.infrastructure.eventbus.KafkaConsumerErrorHandler

Resilience in Java- und Spring-Boot-Applikationen:

- Spring Boot bringt bereits einige **Standard-Techniken** mit, z.B. für Fehlerbehandlung mit **Retries, Backoff und Dead-Letter-Topic** für kafka-basierte Kommunikation.
- **Spring Retry** ist ein weiteres Projekt im Spring-Ökosystem.
- **Resilience4j** ist eine Java-Bibliothek zur Implementierung verschiedener Resilienzmuster, darunter Circuit Breaker, Rate Limiter, Retry, Bulkhead und Cache.

5.1. Resilience vs. Chaos Engineering

Info
Casey Rosenthal, Nora Jones: "Chaos Engineering", 2020, O'Reilly Media

https://github.com/Netflix/SimianArmy

https://github.com/netflix/chaosmonkey

Beim **Chaos-Engineering** werden absichtlich kontrollierte Ausfälle und Unsicherheiten in ein System eingebracht, um dessen Belastbarkeit zu testen und potenzielle Schwachstellen zu ermitteln. Durch die Simulation verschiedener Arten von Fehlern und die Beobachtung der Reaktion des Systems kann das Team über das Verhalten des Systems lernen und **Erkenntnisse darüber gewinnen, wie seine Zuverlässigkeit und Belastbarkeit** verbessert werden kann.

Konzepte des Chaos Engineering den Teams sollen demnach helfen, potenzielle Probleme in ihren Systemen proaktiv zu erkennen und anzugehen, die Widerstandsfähigkeit des Systems zu verbessern und die Reaktion auf Zwischenfälle und die Zuverlässigkeit insgesamt zu erhöhen.

Man erhofft sich damit folgende Verbesserungen:

1. **Identifizierung potenzieller Fehlerpunkte:** Durch die absichtliche Einführung von Fehlern können Teams potenzielle Fehlerpunkte im System erkennen, die ihnen sonst vielleicht nicht aufgefallen wären. Dies kann Teams dabei helfen, diese Probleme proaktiv anzugehen, bevor sie kritisch werden.
2. Verbesserung der **Widerstandsfähigkeit** des Systems: Durch das Testen der Reaktion des Systems auf verschiedene Arten von Fehlern können die Teams lernen, wie sie die Widerstandsfähigkeit des Systems verbessern und es robuster gegenüber realen Fehlern machen können.
3. **Verbesserung der Reaktion** auf Vorfälle: Durch die Simulation von Ausfällen können die Teams ihre Verfahren zur Reaktion auf Vorfälle üben und verbessern, was ihnen helfen kann, effektiver auf reale Ausfälle zu reagieren, wenn diese auftreten.
4. **Verbesserte Überwachung** und **Beobachtbarkeit:** Durch die Durchführung von Chaos-Experimenten können Teams Lücken in ihrer Überwachung und Beobachtbarkeit erkennen und ihre Fähigkeit verbessern, Ausfälle zu erkennen, zu diagnostizieren und zu beheben.
5. **Verbesserung** der **Kommunikation** und **Zusammenarbeit:** Durch die Einbeziehung verschiedener Teams und Interessengruppen in den Chaos-Engineering-Prozess können die Teams die Kommunikation und Zusammenarbeit verbessern, was ihnen helfen kann, effektiver auf Ausfälle zu reagieren und die allgemeine Zuverlässigkeit des Systems zu verbessern.
6. **Verbesserung der Gesamtkultur:** Durch die Einführung der Konzepte des Chaos Engineering können die Teams eine Kultur des Experimentierens, Lernens und der kontinuierlichen Verbesserung schaffen, die für den langfristigen Erfolg des Systems unerlässlich ist.

Die **Netflix Simian Army** ist eine Reihe von Open-Source-Tools, die von Netflix entwickelt wurden, um die Widerstandsfähigkeit ihrer Infrastruktur zu testen und zu verbessern.

Abbildung: Netflix Simian Army Logo

Die Tools simulieren verschiedene Ausfallszenarien, wie z. B. Netzwerkpartitionen und Instanzausfälle, um sicherzustellen, dass das System diese problemlos bewältigen kann. Diese Army umfasst eine Reihe von sogenannten **"Monkeys" für verschiedene Testzwecke**:

1. Chaos Monkey - beendet virtuelle Maschineninstanzen nach dem Zufallsprinzip, um zu testen, wie das System mit Ausfällen umgeht.
2. Latency Monkey - führt Netzwerklatenz ein, um zu testen, wie das System mit langsamen Antworten umgeht.
3. Conformity Monkey - identifiziert Instanzen, die sich nicht an die Best Practices halten, und schaltet sie ab.
4. Security Monkey - identifiziert Sicherheitsschwachstellen und Richtlinienverstöße in der Cloud-Umgebung.
5. Doctor Monkey - erkennt und repariert Instanzen, die ungesund sind oder nicht richtig funktionieren.

Man könnte die Simian-Armee im Zyklus der Anwendungsentwicklung einsetzen, indem man diese Tools zumindest in seine **Test- und Präproduktionsumgebungen gezielt integriert**. Auf diese Weise können die Entwickler sehen, wie sich ihre Anwendung bei Fehlern verhält, und etwaige Probleme vor der Bereitstellung in der Produktion erkennen und beheben.

6. Service Discovery

Service Discovery ist der Prozess der automatischen **Erkennung und Lokalisierung von Diensten in einem verteilten System**.

In einem verteilten System können Dienste auf verschiedenen Hosts laufen, gestartet und gestoppt werden, an andere Standorte wechseln oder ihre IP-Adresse verändern. Service Discovery ermöglicht es, diese Änderungen automatisch zu erkennen und seine Konfiguration entsprechend zu aktualisieren, so dass das System und seine -teile stets über den Standort und den Status aller Dienste informiert ist. Service Discovery ermöglicht es demnach, Dienste dynamisch zu erkennen, zu lokalisieren und mit ihnen zu interagieren, ohne dass eine manuelle Konfiguration oder fest kodierte IP-Adressen erforderlich sind.

Damit wird eine dynamische Skalierung und Entwicklung des Systems ermöglicht. Das stellt sicher, dass das System auch bei Ausfällen weiterarbeiten kann und dass es sich schnell von auftretenden Störungen erholen kann.

Je nach den spezifischen Anforderungen des Systems gibt es verschiedene Möglichkeiten, die Diensteerkennung zu implementieren. Einige beliebte Ansätze sind:

1. Auf dem **Domain Name System (DNS)** basierende Dienstsuche: Bei diesem Ansatz wird DNS verwendet, um Dienstnamen auf IP-Adressen abzubilden. Clients können DNS abfragen, um die IP-Adresse eines Dienstes zu ermitteln.
2. **Service Registry** implementiert eine zentrale Registry verwendet, um den Standort und den Status von Diensten zu speichern und zu verwalten. Clients können die Registry abfragen, um den Standort eines Dienstes zu ermitteln.
3. Ein **Service Mesh** ist eine konfigurierbare Infrastrukturschicht für Microservices-Anwendungen, die die Kommunikation flexibel, zuverlässig und schnell macht. Es bietet Funktionen wie Service Discovery, Lastausgleich und Sicherheit.
4. **Peer-to-Peer-Service-Erkennung:** Bei diesem Ansatz wird ein verteiltes Protokoll wie das Multicast Domain Name System (mDNS) oder das Simple Service Discovery Protocol (SSDP) verwendet, damit Dienste sich gegenseitig finden können, ohne dass eine zentrale Registrierung erforderlich ist.
5. **Cloud-basierte Dienstsuche** verwendet Cloud-basierte Service Discovery Tools wie Amazon ECS Service Discovery, Kubernetes Service Discovery und Consul Service Discovery. Sie ermöglicht die Erkennung von Diensten anhand ihres Namens und Namensraums und ist so konzipiert, dass sie nahtlos mit Container-Orchestrierungsplattformen zusammenarbeitet.

Unter **Konfigurationsmanagement** versteht man den Prozess der Verwaltung und Pflege der Konfiguration eines Systems oder Dienstes (siehe Kapitel 5 (E)). Im Zusammenhang mit der Diensterkennung wird das Konfigurationsmanagement verwendet, um sicherzustellen, dass Service Discovery über die aktuelle Konfiguration der Dienste im verteilten System informiert ist. Damit in einem verteilten System das System stets über den Standort und den Status aller Dienste informiert ist, enthalten Service Discovery Tools häufig eine Konfigurationsmanagement-Komponente, mit der Administratoren die Konfiguration von Diensten, einschließlich ihres Standorts und Status, definieren und verwalten können. Diese Komponente wird verwendet für:

1. **Registrierung neuer Dienste:** Wenn ein neuer Dienst bereitgestellt wird, muss er im Service Discovery System registriert werden, damit andere Dienste ihn finden können.
2. **Vorhandene Dienste aktualisieren:** Wenn ein Dienst verschoben wird oder sich seine IP-Adresse ändert, muss das Service Discovery System mit den neuen Informationen aktualisiert werden.
3. **Entfernen von Diensten:** Wenn ein Dienst außer Betrieb genommen wird, muss er aus dem System zur Ermittlung von Diensten entfernt werden, damit andere Dienste nicht versuchen, ihn zu finden.
4. optional **Dienst-Überwachung:** Das Service Discovery System sollte in der Lage sein, den Zustand der Dienste zu überwachen und den Betrieb zu benachrichtigen, wenn ein Dienst nicht erreichbar ist oder ausfällt.

Durch die Bereitstellung eines zentralen Kontrollpunkts für die Konfiguration von Diensten in einem verteilten System ermöglicht die Diensterkennung mit Konfigurationsmanagement die dynamische Skalierung und Weiterentwicklung des Systems, ohne dass eine manuelle Konfiguration oder fest kodierte IP-Adressen erforderlich sind. So wird sichergestellt, dass das System auch bei Ausfällen weiter funktioniert und sich von auftretenden Störungen schnell erholen kann.

7. Skalierbarkeit

Die Integration verschiedener Systeme und Anwendungen stellt im Hinblick auf die Skalierbarkeit eine Herausforderung dar, vor allem, wenn das Lastverhalten wenig vorhersagbar ist bzw. über einen Zeitverlauf schwankt bzw. so groß ist, dass es mit "normalen" Mitteln nicht einfach zu beherrschen ist. Dies kann bedeuten, dass ein hohes Datenvolumen, eine hohe Geschwindigkeit und eine große Vielfalt an Daten bewältigt werden müssen und/oder dass das System mit steigenden bzw. schwankenden Lasten und Nutzungsmustern zurechtkommen muss.

Die **Skalierbarkeit** ist bei verteilten Systemen wichtig, da sie es dem System ermöglicht, steigende Arbeitslasten und Benutzerzahlen zu bewältigen, ohne dass es zu einem erheblichen Leistungsabfall kommt. Dies ist besonders wichtig für Systeme, die voraussichtlich wachsen werden oder eine große Anzahl gleichzeitiger Benutzer unterstützen müssen. Skalierbarkeit bezieht sich auf die Fähigkeit eines Systems, zu wachsen und mehr Arbeitslast und Benutzer zu bewältigen, wenn die Nachfrage steigt (bzw. auch dynamisch zu skalieren, also Ressourcen bei sinkender Last auch wieder abzubauen).

Skalierbarkeit ist nicht gleichbedeutend mit **Performance** und **Durchsatz**:

- Performance bezieht sich i.d.R. auf die Fähigkeit eines Systems, bestimmte Anforderungen an die Antwortzeit zu erfüllen.
- Der Durchsatz bezieht sich auf die Anzahl der Anfragen, die ein System pro Zeiteinheit bearbeiten kann.

Ein System kann zum Beispiel eine gute Performance und einen guten Durchsatz haben, wenn es eine kleine Anzahl von Benutzern verarbeitet, aber seine Leistung kann sich verschlechtern, wenn die Anzahl der Benutzer steigt. Ein solches System ist dann nicht skalierbar.

Unterscheide weiterhin:

- Beim **Scale-up,** auch als **vertikale Skalierung** bezeichnet, werden einem einzelnen Knoten im System mehr Ressourcen (z. B. CPU, Speicher, Storage) hinzugefügt. Dadurch kann das System mehr Arbeitslast und Benutzer bewältigen, ist aber durch die physischen Ressourcen des Knotens begrenzt. Man kann z. B. einem einzelnen Server mehr Speicher hinzufügen, um mehr Anfragen zu bearbeiten. Die Scale-up-Strategie ist vorteilhaft, wenn das System die Grenze der maximale Anzahl von Knoten erreicht hat, die hinzugefügt werden können.
- Beim **Scale-out,** auch als **horizontale Skalierung** bezeichnet, werden dem System weitere Knoten hinzugefügt. Dadurch kann das System mehr Arbeitslast und Benutzer bewältigen, indem die Last auf mehrere Knoten verteilt wird. Die Scale-Out-Strategie ist prinzipiell zu bevorzugen, weil sie dem System erlaubt, mehr Arbeitslast und Benutzer zu bewältigen, indem die Last auf mehrere Knoten verteilt wird. Dadurch kann das System eine höhere Nachfrage bewältigen und die Fehlertoleranz und Verfügbarkeit verbessern. Durch Scale-out kann die Systemlast auf mehrere Ressourcen verteilt werden, wodurch das Risiko von Ressourcenkonflikten verringert und die Leistung verbessert wird.

Die beiden Konzepte können kombiniert werden.

8. Trade-offs

Die Qualitätsmerkmale eines verteilten Systems können im Widerspruch zueinander stehen, und die Architektur des Systems muss Prioritäten setzen, welche Merkmale am wichtigsten sind.

- **Konsistenz und Verfügbarkeit** stehen oft im **Widerspruch** zueinander. Ein System, das die Konsistenz in den Vordergrund stellt, kann möglicherweise keine hohe Verfügbarkeit bei Netzwerkunterbrechungen oder -ausfällen garantieren. Ein System, das die Verfügbarkeit in den Vordergrund stellt, kann möglicherweise keine hohe Konsistenz garantieren.
- Ein System, bei dem die **Sicherheit** im Vordergrund steht, ist möglicherweise starrer und weniger anpassungsfähig an sich ändernde Anforderungen, während ein System, bei dem die **Flexibilität** im Vordergrund steht, eine geringere Sicherheit aufweisen kann.

Eine Architektur muss die **Qualitätsattribute** auf der Grundlage der Anforderungen der Anwendung und der **akzeptablen Kompromisse priorisieren**. Bei einer E-Commerce-Website kann beispielsweise die Konsistenz und Verfügbarkeit im Vordergrund stehen, um sicherzustellen, dass die Benutzer immer auf die Website zugreifen und die neuesten Preise und Bestände einsehen können, während bei einer Social-Media-Website die Skalierbarkeit und Leistung wichtiger sein kann, um eine große Anzahl gleichzeitiger Benutzer und hohe Schreiblasten zu bewältigen.

Es ist wichtig zu beachten, dass sich die Qualitätsattribute nicht gegenseitig ausschließen. Eine gute Architektur sollte danach streben, alle diese Attribute so gut wie möglich zu berücksichtigen und auszugleichen. Je nach den spezifischen Anforderungen der Anwendung kann es jedoch erforderlich sein, bestimmten Attributen Vorrang vor anderen einzuräumen.

(E) Domain-driven Design - Integrationsmuster in DDDs Strategischem Design

1. DDDs Strategisches Design

Info
https://www.heise.de/news/software-architektur-tv-Strategisches-DDD-Grundlegende-Patterns-im-Blick-6174848.html

Das **Strategische Design** in DDD stellt einen Ansatz dar, um eine Domäne top-down in kleinere und handhaberere Subdomänen aufzuteilen, um somit die Komplexität zu reduzieren und die fachliche Struktur zu ermitteln. Der Fokus des Strategischen Designs liegt daher auf dem Big Picture einer Anwendungslandschaft. Beim Strategic Design geht es also um das große Ganze, d.h. um die Gesamtheit eines Systems oder einer Systemlandschaft. Faktisch basiert Strategic Design erst mal auf einem Konstrukt namens Bounded Context und den Domänen und Subdomänen. Die Domäne ist der "große Rahmen", die Subdomänen sind die ersten Untergliederungen - um dort Bounded Contexts zu schneiden. Ein Bounded Context ist zunächst eine fachliche Klammer um eine bestimmte Fachlichkeit und definiert so Grenze und Gültigkeit eines bestimmten Modells. ("EIn Kunde in Context 'Rechnungsstellung' ist nicht derselbe Kunde in Context 'Vertrieb' oder in 'Versand').

Zu den wichtigsten Konzepten von DDD gehören (siehe auch Kapitel 3):

1. **Bounded Context**: Ein Bounded Context ist eine Möglichkeit, Domänenobjekte zu gruppieren, die unterschiedliche Bedeutungen, Zwecke oder Anwendungsfälle haben, und sie von anderen Domänenobjekten zu trennen, die andere Bedeutungen haben. Dies ermöglicht unterschiedliche Modelle für ein und denselben Begriff, je nach Verwendungskontext.
2. **Context-Mapping**: Beim Context Mapping werden die Beziehungen und Abhängigkeiten zwischen verschiedenen Bounded Contexts in der Problemdomäne identifiziert. Es hilft dabei, das Softwaredesign an der Geschäftsdomäne auszurichten und stellt sicher, dass die verschiedenen Bounded Contexts auf konsistente und wartbare Weise integriert werden.

2. Integrationmuster

DDD definiert verschiedene **Integrationsmuster** in seinem Strategischen Design:

1. Customer/Supplier: Es handelt sich um eine Beziehung zwischen Kunde (Downstream) und Server (Upstream), bei der die Teams in kontinuierlicher Integration arbeiten.
2. Conformist: Es handelt sich um ein Szenario, an dem vor- und nachgelagerte Teams beteiligt sind. In diesem Modell hat das vorgelagerte Team jedoch keine Motivation, die Anforderungen des nachgelagerten Teams zu erfüllen.
3. Partnership: In diesem Szenario sind die Teams voneinander abhängig und müssen eine kooperative Beziehung eingehen, um die Entwicklungsanforderungen beider Systeme zu erfüllen.
4. Shared Kernel: Ein von zwei oder mehr Teams gemeinsam genutzter Kontext, der die Duplizierung von Code reduziert - aber die Koppelung erhöht bzw. manifestiert. Änderungen müssen zwischen allen Teams abgestimmt werden.
5. Anti-Corruption-Layer: siehe unten

3. Upstream vs. Downstream

Im Domain-Driven Design (DDD) werden die Begriffe "Upstream" und "Downstream" verwendet, um die Beziehung zwischen den verschiedenen Teilen des Systems in Bezug auf ihre Geschäftsdomäne und den Fluss von Daten und Logik zu beschreiben (genauer: dem "Modelfluss" zwischen Bounded Contexts).

Diese beiden Begriffe lassen sich nicht den Begriffen "Datenfluss" oder "Kontrollfluss" zuordnen.

- **"Upstream"** bezieht sich auf die Teile des Systems, die näher an der Kerndomäne liegen und für die Implementierung der Kerngeschäftslogik und das Treffen von Entscheidungen auf der Grundlage des Domänenwissens verantwortlich sind. Diese Teile des Systems sind in der Regel komplexer und verfügen über ein besseres Verständnis der Geschäftsdomäne.
- **"Downstream"** bezieht sich auf die Teile des Systems, die weiter von der Kerndomäne entfernt sind und für die Schnittstellen zu externen Systemen oder Technologien wie Datenbanken, externen APIs oder Benutzerschnittstellen zuständig sind. Diese Teile des Systems sind in der Regel einfacher und haben ein begrenzteres Verständnis der Geschäftsdomäne.

Der Model Flow wird zwischen Up- und Downstream dargestellt. Upstream gibt dabei das Modell vor - und Downstream darf - je nach Beziehungspattern (Open-Host, Conformist, Customer-Supplier, Partnership etc.) dabei nicht/wenig oder viel mitbestimmen.

Änderungen am Modell des Upstream-Bounded-Context haben Auswirkung auf den Downstream-Bounded-Context, nicht aber umgekehrt. Durch eine klare Trennung zwischen den vor- und nachgelagerten Teilen des Systems trägt DDD dazu bei, dass die Logik der Kerndomäne vor Änderungen in externen Systemen oder Technologien geschützt wird und das System als Ganzes flexibler und wartbarer ist.

4. Anti-Corruption Layer

Ein **Anti-Corruption Layer (ACL)** ist ein Integrationsmuster, um ein System von Änderungen in externen Systemen, Diensten oder Technologien abzuschirmen. Bei der Integration verschiedener Systeme oder Dienste kann ein ACL helfen, die Schnittstelle eines Systems von der Implementierung eines anderen zu entkoppeln. Der ACL ist in diesem Sinne eine Isolationsschicht, die das eigene Modell vor ungewünschten engen Kopplungen an externe Modelle sowie der Vermischung von Fachlichkeit schützt.

Dies ermöglicht mehr Flexibilität bei Änderungen an einem System, ohne das andere zu beeinträchtigen, und kann auch dazu beitragen, die Beschädigung von Daten oder Funktionen zu verhindern, die auftreten können, wenn verschiedene Systeme inkompatible Schnittstellen haben.

Darüber hinaus kann eine ACL auch Funktionen wie Datenvalidierung, Datenzuordnung und Fehlerbehandlung enthalten, um sicherzustellen, dass die Daten bei der Weitergabe zwischen Systemen ordnungsgemäß übersetzt und verarbeitet werden.

Dieses Verfahren ist insbesondere für die Kommunikation mit Legacy-Anwendungen oder der Schnittstelle eines Drittanbieters gut geeignet.

(F) Interface Design, Schnittstellenvertrag

Info
Renato Vinga-Martins: "Consumer-Driven Contracts mit Pact: Keine Angst vor der Integration", IT-Spektrum 5/2021, 27. August 2021

1. Inhalte eines Schnittstellen-Vertrags

Ein **Schnittstellenvertrag** ist ein Dokument, das die Regeln und Erwartungen für die Kommunikation zwischen zwei Systemen festlegt und die verwendete API definiert.

Dies ist insbesondere dann relevant, wenn es a) bilaterale Beziehungen der Schnittstellenpartner gibt und setzt b) eine Customer-Supplier-Beziehung (siehe DDD, strategisches Design) der Partner "auf Augenhöhe" voraus, um den Vetrag "auszuhandeln" und zu erfüllen bzw. bei Nicht-Erfüllung entsprechend reagieren zu können.

Der **Inhalt** eines Schnittstellenvertrags dokumentiert technisch-architektonische Kommunikation - idealerweise auch mit Beispielen - eine Reihe von Anforderungen:

A) Strukturelle Sicht des Schnittstellenvertrags (ggf. generierbar)

1. **Format** von Request- und Response (siehe unten bei 2): Dazu gehören die Struktur und das Format der Daten, die zwischen den Systemen ausgetauscht werden, wie z. B. die Datentypen, Feldnamen und Validierungsregeln, Angabe der Datenstruktur (z. B. JSON, XML usw.) und etwaiger Beschränkungen für die Daten (z. B. maximale Länge, erforderliche Felder usw.), dazu auch Encoding. Siehe auch unten.
2. **Syntax:** Die Syntax der Remote-Schnittstelle sollte klar, einheitlich und leicht verständlich sein. Dazu gehört auch die Definition der Struktur der von der Remote-Schnittstelle gesendeten und empfangenen Nachrichten sowie die Struktur der verwendeten Befehle oder Abfragen.
3. **Semantik:** Die Semantik der Remote-Schnittstelle sollte klar definiert sein, einschließlich aller Regeln oder Einschränkungen für die gesendeten und empfangenen Nachrichten.
   a. Dazu gehört auch die Angabe der Bedeutung von Codes, Fehlermeldungen und anderen Daten, die in den Nachrichten enthalten sein können. Dies sollte auch festlegen, welche Vorbedingungen für einen erfolgreichen Schnittstellenaufruf nötig sind.
   b. Beispiele: erzwungene Reihenfolge, fortlaufende Zähler, Kontextinformationen bzw. Zustand - besser vermeiden, da zustandslose Dienste einfacher handhabbar sind und besser skalieren.
4. **Komprimierung:** Können (und werden) Nachrichten auf dem Infrastrukturlayer verlustfrei komprimiert übertragen werden?
5. **Versionierung:** Versionsnummer der API, die Versionsstrategie und die Art und Weise, wie das System mit Änderungen an der API im Laufe der Zeit umgehen wird. Idealerweise ist sichergestellt, dass neue Versionen der Schnittstelle mit vorhandenen Clients kompatibel sind und dass umgekehrt die Clients mit verschiedenen Versionen der Schnittstelle umgehen können. Eine solche Kompatibilität sollte zumindest für einige ältere Versionen über einen gewissen Zeitraum sichergestellt sein.
6. **Fehlerbehandlung:** Format und Struktur von Fehlernachrichten mit allen Fehler-Codes und -Meldungen, die zur Anzeige von Fehlern verwendet werden.

B) Verhaltenssicht des Schnittstellenvertrags

1. **Endpunkte,** den das System zum Senden und Empfangen von Anfragen verwendet. (Beispiele: URLs, Dateiordner, Kafka-Topics etc.)
2. **Authentifizierung** und **Autorisierung:** Dazu gehören die Methode und die Anforderungen für die Authentifizierung und Autorisierung der Systeme für den Zugriff auf die Schnittstelle.
3. **Mengengerüste** und **Ratenbegrenzung** (Rate Limits): Mengengerüste über erwartete Anzahl Requests und die Größe der Nutzlast (in Summe, im Mittel, zu bestimmten Tageszeiten, im Peak etc.). Dazu die Begrenzung der Anzahl von Anfragen, die innerhalb eines bestimmten Zeitraums an die Schnittstelle gestellt werden können (in Summe, zu bestimmten Tageszeiten, im Peak, zu welchen Peak-Zeiten etc.), und die Art und Weise, wie das System Anfragen behandelt, die diese Grenze überschreiten.
4. **Validierungen:** Es sollten Validierungsregeln beschrieben sein, über die der Schnittstellenanbieter valide Nachrichten von invaliden Nachrichten unterscheiden kann.
5. Umgang mit **Zuliefergarantien:** Was passiert bei fehlenden, doppelt/mehrfachen Nachrichten? Resultieren solche Nachrichten in Fehler? Wo wird beispielsweise Idempotenz eingesetzt, um Mehrfachaufrufe erkennen zu können?
6. **Fehlerbehandlung:**
   a. Umgang mit **Fehlern** (z.B. Nachrichten in ein Dead-Letter-Topic, Retry-Verhalten)
   b. Umgang mit technisch bedingten **Fehlersituationen** (z.B. falls Aufrufe in einen Timeout laufen).
7. **Tests:** Dazu gehört, wie, wo, wieviel/oft das System vor der Inbetriebnahme getestet wird (im besten Fall automatisiert).
8. **Support** und **SLAs:** Dazu gehören die Kontaktinformationen für das Support-Team und Einzelheiten zu den SLAs für Reaktionszeiten und Problemlösung.

Im besten Fall ist ein solcher Vertrag in einer **Test-Suite** verdrahtet (vgl. Consumer Driven Contract Testing, CDC).

2. Nachrichtenformat

Bei der Wahl eines **Nachrichtenformats** sind mehrere Faktoren zu berücksichtigen.

1. **Größe:** Binäre Formate sind in der Regel kompakter als Textformate, was zur Verringerung der Netzwerkbandbreite und des Speicherbedarfs beitragen kann. Allerdings sind Textformate in der Regel besser lesbar, was die Fehlersuche und -behebung erleichtern kann.
2. **Kompatibilität:** Textformate wie JSON und XML werden von einer Vielzahl von Programmiersprachen und Plattformen unterstützt, was sie zu einer guten Wahl für die Interoperabilität machen kann.
3. **Leistung:** Binäre Formate lassen sich effizienter analysieren und generieren, was für Hochleistungssysteme von Vorteil sein kann. Textformate sind jedoch in der Regel besser lesbar und nachvollziehbar, was die Fehlersuche und -behebung erleichtern kann.
4. **Datentypen:** JSON und XML unterstützen komplexere Datentypen wie Arrays, verschachtelte Strukturen und Metadaten und sind in dieser Hinsicht vielseitiger, während Binärformate effizienter sind, aber möglicherweise nicht alle Datentypen unterstützen. Sind selbstdefinierte Teilstrukturen möglich (und so in Teilen wiederwendbar, z.B. durch includes)?
5. **Validierung:** Gibt es ein Schema, gegen das Nachrichten validiert werden können (Beispiel: XML-XSD)?
6. **Encoding**

Beispiele für gebräuchliche Nachrichtenformate sind:

- **XML** (eXtensible Markup Language): Eine Auszeichnungssprache, die zum Speichern und Übertragen von Daten verwendet wird. Sie wird häufig in Webdiensten und anderen Internetprotokollen verwendet. XML kann gegen ein XSD-Schema validiert werden.
- **JSON** (JavaScript Object Notation): Ein leichtgewichtiges Datenaustauschformat, das für Menschen leicht zu lesen und zu schreiben und für Maschinen leicht zu analysieren und zu erzeugen ist.
- **Protocol Buffers** von Google ist ein Binärformat, das zur Serialisierung strukturierter Daten verwendet wird. Es ist so konzipiert, dass es klein und effizient ist und mit einer breiten Palette von Programmiersprachen verwendet werden kann.
- Apache **Avro** bietet ein kompaktes Binärformat für Daten und einen umfangreichen Satz von Datenstrukturen.
- **CBOR** (Concise Binary Object Representation): Ein binäres Format, das JSON ähnelt und darauf ausgelegt ist, klein und effizient zu sein.
- **BSON** (Binary JSON): ist eine binär kodierte Serialisierung von JSON-ähnlichen Dokumenten und wird z.B. in MongoDB verwendet.

Je weniger über die Schnittstellenpartner bekannt ist bzw. vorgegeben/gesetzt ist (externe Drittsysteme, ggf. gar öffentliche Schnittstelle), desto mehr ist es notwendig, Nachrichten explizit zu **validieren** und gegen eine **Schema-Definition** zu vergleichen (z.B. durch Wahl von XML und XSD). Dagegen sind interne Schnittstellen, bei denen alle Schnittstellenpartner bekannt und die Kommunikation gut verstanden sind, weniger strikt definierbar (z.B. durch Wahl von JSON).

(G) Standards bzgl. Java und Spring-Boot-Anwendungen

1. Standards, Protokolle, Produkte für Integrationsstrategien

In Bezug auf Java- und Spring Boot-Anwendungen gehören zu den am häufigsten verwendeten **Standards** für die Integration:

1. Java Message Service **(JMS):** Dies ist ein Java-basierter Standard für Messaging, der es Java-Anwendungen ermöglicht, Nachrichten asynchron zu senden und zu empfangen.
2. Java Remote Method Invocation **(RMI):** Dies ist ein Java-basierter Standard für Remote Procedure Calls. RMI ermöglicht es Anwendungen, mit anderen Java-Anwendungen zu kommunizieren, die auf anderen Rechnern laufen.
3. Simple Object Access Protocol **(SOAP):** Hierbei handelt es sich um ein Web-Protokoll zum Austausch strukturierter Informationen bei der Implementierung von Webdiensten.
4. Representational State Transfer **(REST):** Hierbei handelt es sich um eine Reihe von Architekturprinzipien für die Erstellung von Webservices. RESTful Web Services basieren auf der REST-Architektur und können von Spring Boot-Anwendungen einfach erstellt und konsumiert werden.

Protokolle:

1. Das Advanced Message Queuing Protocol **(AMQP)** ist ein offener Standard für Message Queuing, der einen gemeinsamen Rahmen für Messaging-Systeme bietet. Er definiert ein drahtgebundenes Protokoll für die Nachrichtenübertragung und ist für die Unterstützung einer Vielzahl von Messaging-Systemen und -Plattformen konzipiert.
2. **MQTT** (Message Queue Telemetry Transport) ist ein leichtgewichtiges Messaging-Protokoll, das für den Einsatz in Netzwerken mit geringer Bandbreite und hoher Latenz konzipiert ist. Es wird häufig in IoT-Anwendungen (Internet der Dinge) verwendet und ist so konzipiert, dass es leichtgewichtig und einfach zu implementieren ist.
3. **STOMP** (Simple Text Orientated Messaging Protocol) ist ein einfaches textbasiertes Nachrichtenprotokoll, das für die Kommunikation mit Nachrichtenbrokern verwendet werden kann. Es ist einfach zu implementieren und wird von einer Vielzahl von Messaging-Systemen unterstützt, darunter RabbitMQ und ActiveMQ.

Es gibt viele **Middleware-Produkte und -Frameworks**, die mit Java- und Spring Boot-Anwendungen verwendet werden können:

- Apache **Kafka:** Eine Open-Source-Plattform für verteiltes Streaming, die häufig für das Streaming und die Verarbeitung von Echtzeitdaten verwendet wird. Siehe auch unten und Kapitel 7.
- Apache **ActiveMQ:** Ein Open-Source Message Broker, der eine Vielzahl von Messaging-Protokollen unterstützt, darunter JMS, STOMP und MQTT.
- **RabbitMQ:** Ein beliebter Open-Source-Message-Broker, der AMQP und andere Protokolle unterstützt und für seine Zuverlässigkeit und Skalierbarkeit bekannt ist.
- **Spring Integration:** Ein Framework, das Teil des Spring-Ökosystems ist und eine Reihe von Komponenten und Mustern für die Integration verschiedener Systeme und Anwendungen bietet.
- **Spring Cloud Stream:** Ein Framework, das auf Spring Integration aufbaut und ein einfaches und konsistentes Modell für den Aufbau ereignisgesteuerter Microservices bietet.
- **Apache Camel:** siehe unten

2. Eventbus / Messaging Plattform / Event-Streaming Plattform

Info
Vergleich Kafka - Pulsar - RabbitMQ bei Confluent, https://www.confluent.io/kafka-vs-pulsar/, Benchmarks unter https://www.confluent.io/blog/kafka-fastest-messaging-system/

Markus Günther, Boris Fresow: "Einführung in Apache Pulsar - Teil 1, Technisch-konzeptionelle Annäherung", Artikelserie im Java Magazin, Teil 1: https://entwickler.de/software-architektur/technisch-konzeptionelle-annaherung

Ein **Eventbus** ist ein Nachrichtensystem, das die Kommunikation zwischen verschiedenen Komponenten eines Softwaresystems ermöglicht. Es bietet den Systemteilen die Möglichkeit, Nachrichten zu senden und zu empfangen, und ermöglicht so eine entkoppelte, asynchrone Architektur.

Apache Kafka (siehe auch Kapitel 7) ist ein Open-Source-Eventbus, der verteilte, fehlertolerante und durchsatzstarke Nachrichtenübermittlung ermöglicht. Er wird häufig für den Aufbau von Echtzeit-Daten-Streaming-Anwendungen, die Verarbeitung von Streaming-Daten und die Verbindung verschiedener Komponenten einer Microservices-Architektur verwendet. In Kafka veröffentlichen Produzenten ("Producer") Nachrichten in Kafka-Topics, die dann von einem oder mehreren Konsumenten ("Consumer") konsumiert werden. Topics werden in Partitionen unterteilt, und jede Partition wird einem bestimmten Konsumenten zugewiesen. So können mehrere Verbraucher parallel verschiedene Nachrichten desselben Topics verarbeiten. Kafka bietet Funktionen wie Skalierbarkeit, Fehlertoleranz und hohen Durchsatz, die es für den Aufbau großer verteilter Systeme prädestinieren.

Apache Pulsar ist eine mögliche Alternative, die ein ähnliche Architektur verfolgt.

3. Enterprise Service Bus (ESB)

Ein **Enterprise Service Bus (ESB)** ist eine Middleware, die als **zentraler Dienst** die Kommunikation und Integration verschiedener Systeme und Anwendungen in einer Unternehmensumgebung erleichtert. Er fungiert als zentraler Knotenpunkt für die Weiterleitung und Umwandlung von Nachrichten zwischen Systemen und bietet eine Reihe gemeinsamer Dienste wie Sicherheit, Datenverwaltung und Skalierbarkeit.

Ein ESB verwendet meist ein Publish-Subscribe-Modell, bei dem verschiedene Systeme und Anwendungen Nachrichten an den Bus veröffentlichen können und andere Systeme und Anwendungen diese Nachrichten abonnieren können. Der ESB leitet dann die Nachrichten auf der Grundlage von Regeln und Filtern an die entsprechenden Abonnenten weiter.

Ein ESB kann für die Integration einer Vielzahl von Systemen verwendet werden, darunter Legacy-Systeme, Datenbanken und Anwendungen von Drittanbietern. Er kann auch dazu verwendet werden, die Funktionen dieser Systeme als Webdienste bereitzustellen, wodurch sie für andere Systeme und Anwendungen leichter zugänglich werden. Einige Beispiele für kommerzielle und **Open-Source-ESBs** sind:

- Apache ServiceMix
- Mule ESB
- JBoss ESB
- IBM WebSphere Message Broker
- Oracle Service Bus

ESBs können mit Java- und Spring Boot-Anwendungen integriert werden, indem die entsprechenden Konnektoren oder Bibliotheken der ESB-Anbieter oder JMS (Java Message Service) oder andere Messaging-Protokolle verwendet werden.

4. Application-Server, v.a. für Java Enterprise Edition (JEE)

Java Enterprise Edition (JEE) ist eine Reihe von Spezifikationen und technischen Standards für die Entwicklung und den Einsatz von **Unternehmensanwendungen mit Java.** Sie definiert APIs und Technologien für die Entwicklung verteilter, mehrschichtiger Anwendungen und umfasst Spezifikationen für Webdienste, Messaging, Sicherheit und Datenzugriff. JEE-Anwendungen werden in der Regel auf Appservern bereitgestellt und mit Frameworks wie Servlets, JavaServer Faces (JSF), Enterprise JavaBeans (EJB) und REST- oder SOAP-Services erstellt.

Ein **Java/JEE-Applicationserver** ist eine Softwareplattform, die eine Laufzeitumgebung und eine Reihe von Diensten für die Ausführung von Java´/JEE-basierten Anwendungen bereitstellt. Zu diesen Diensten gehören in der Regel Webserver-Funktionen, Transaktionsmanagement, Sicherheit und Datenzugriff. Einige beliebte Java-Anwendungsserver sind:

- Apache Tomcat bzw. TomEE: Open-Source-Appserver, der häufig für Webanwendungen verwendet wird und für seine schlanke und schnelle Leistung bekannt ist.
- JBoss/WildFly: Ein Open-Source-Appserver, der auf den Spezifikationen der Java Enterprise Edition (JEE) basiert und eine breite Palette von Diensten für die Erstellung und Bereitstellung von Unternehmensanwendungen bietet.
- GlassFish: Ein Open-Source-Appserver, der von Oracle entwickelt und gepflegt wird und mit den JEE-Spezifikationen konform ist. (Kommerzielle Variante: Payara).
- IBM WebSphere: Ein kommerzieller Appserver, der eine breite Palette von Diensten für die Erstellung und den Einsatz von Unternehmensanwendungen bietet.

5. Spring Boot

Spring Boot ist ein Framework, mit dem sich eigenständige, produktionsreife Spring-basierte Anwendungen erstellen lassen. Es bietet einen eigenen Ansatz für die Konfiguration und konfiguriert Spring und seine Bibliotheken automatisch auf der Grundlage der im Projekt enthaltenen Abhängigkeiten.

Eine der wichtigsten Funktionen von Spring Boot ist der eingebettete **Webserver** (standardmäßig z. B. Tomcat oder Jetty). Das bedeutet, dass kein separater Appserver erforderlich ist, um eine Spring Boot-Anwendung auszuführen. Der eingebettete Webserver ist leichtgewichtig und für den Produktionseinsatz optimiert, was ihn zu einer guten Wahl für den Einsatz von Spring Boot-Anwendungen in der Produktion macht.

Darüber hinaus bietet Spring Boot eine Reihe weiterer Funktionen, die dazu beitragen können, den "Bedarf" an einem separaten Application Server zu reduzieren, wie z. B.:

- Automatische Konfiguration von Spring und seinen Bibliotheken basierend auf den im Projekt enthaltenen Abhängigkeiten,
- Integrierte Unterstützung für allgemeine Entwicklungsaufgaben wie Logging, Metriken und Health Checks,
- Unterstützung für die Paketierung als JAR- oder WAR-Datei, was die Bereitstellung in einer Vielzahl von Umgebungen erleichtert,
- Unterstützung für die Paketierung als Docker-Image,
- Unterstützung für die Ausführung als eigenständige Anwendung oder als WAR-Datei,
- Integration mit verschiedenen Technologien für den Datenzugriff, Sicherheit, Messaging, Caching und mehr, ohne dass ein zusätzlicher Appserver erforderlich ist.

6. Integrationsframework Apache Camel

Abbildung: Apache Camel Logo

Info
Claus Ibsen, Jonathan Anstey: “Camel in Action”, Februar 2018, 2nd Edition, ISBN 9781617292934, https://www.manning.com/books/camel-in-action-second-edition

Apache Camel ist ein leistungsfähiges Open-Source-Integrationsframework, das bekannte Enterprise Integration Patterns anbietet. Camel unterstützt die meisten der "Enterprise Integration Patterns" aus dem hervorragenden, gleichnamingen Buch von Gregor Hohpe und Bobby Woolf (siehe Kapitel 4 F). Camel bietet Unterstützung für >80 Protokolle und Datentypen. Mit Camel kann man Enterprise Integration Patterns umsetzen, um Routing- und Mediationsregeln entweder in einer Java-basierten Domain Specific Language (oder Fluent API), über Spring-basierte Xml-Konfigurationsdateien oder über die Scala DSL zu implementieren. Camel ist leichtgewichtig, hat also nur einen kleinen Footprint.

Apache Camel ist kein ESB, es ist ein Framework und eine Bibliothek für die einfache Umsetzung der EAI-Patterns. Eine Anwendung nutzt Camel und kann damit dann in einen ESB oder Appserver deployt werden.

7. Service Meshes und Sidecar-Pattern

Info
Eberhard Wolff, Hanna Prinz: "Das Service Mesh. Die Lösung aller Microservice-Probleme?". Java-Magazin, 08/2019, online: https://www.innoq.com/de/articles/2019/10/service-mesh/

Hanna Prinz: "Service Mesh – ein kritischer Blick – Hanna Prinz", INNOQ Technology Lunch, 2020, Video online: https://www.youtube.com/watch?v=cTfp7It1eEo

Ein **Service Mesh** ist eine konfigurierbare Infrastrukturschicht für Microservices-basierte Anwendungen, die deren Kommunikation flexibel, zuverlässig und schnell macht. Es bietet Funktionen wie Diensterkennung, Lastausgleich und Sicherheit.

Ein Service Mesh besteht in der Regel aus einer **Data Plane (Datenebene)** und einer **Control Plane (Steuerungsebene)**: Die Datenebene ist für die eigentliche Kommunikation zwischen den Diensten zuständig, während die Steuerungsebene für die Verwaltung der Konfiguration des Service Meshes verantwortlich ist.

Services Meshes nutzen das **Sidecar-Pattern:** Ein Sidecar ist ein **separater Prozess, der neben einem primären Dienst** läuft und zusätzliche Funktionen für diesen Dienst bereitstellt. Ein Sidecar wird als **separater Prozess** ausgeführt, so dass er unabhängig vom Hauptdienst aktualisiert und konfiguriert werden kann. Dies ermöglicht mehr Flexibilität bei der Hinzufügung neuer oder der Änderung bestehender Funktionen, ohne den Hauptdienst zu beeinträchtigen. Ein Sidecar bietet verschiedene querschnittliche Funktionen an wie Service Discovery, Load Balancing, Proxying, Monitoring, Resilience, Security.

- Typischerweise unterstützen heutige Service-Mesh-Implementierungen nur **synchrone, HTTP-basierte** Kommunikation.
- Aber das Sidecar-Pattern kann prinzipiell auch in **asynchronen Kommunikation** verwendet werden: In einem asynchronen Kommunikationsszenario kann der primäre Dienst beispielsweise eine Anforderung verarbeiten und dann eine Nachricht zur weiteren Verarbeitung an eine Nachrichtenwarteschlange senden. Der Sidecar kann dafür verantwortlich sein, die Nachrichtenwarteschlange abzuhören und die Nachricht zu verarbeiten, z. B. indem er sie an einen anderen Dienst weiterleitet oder in einer Datenbank speichert. Alternativ verwendet ein Sidecar in der asynchronen Kommunikation als Dienst, der einen Teil der Kommunikation an einen anderen Dienst oder eine Datenbank auslagern muss, aber den primären Prozess nicht blockieren möchte, während er auf die Antwort wartet. In diesem Fall fungiert der Sidecar als Proxy, der die Kommunikation mit dem anderen Dienst oder der Datenbank abwickelt und die Antwort an den primären Dienst zurücksendet. Der primäre Dienst kann mit der Bearbeitung der nächsten Anfrage fortfahren, während der Sidecar auf eine Antwort wartet.

Zu den wichtigsten Funktionen eines Service Mesh gehören:

1. **Load Balancing** ermöglicht es dem Service Mesh, Anfragen auf mehrere Instanzen eines Dienstes zu verteilen, so dass das System auch bei Ausfall einer Dienstinstanz weiterarbeiten kann.
2. **Traffic Management** ermöglicht es dem Service Mesh, den Verkehrsfluss zwischen den Diensten zu steuern, z. B. durch Ratenbegrenzung, Unterbrechung der Verbindung und Anfrage-Timeout.
3. **Proxying** kann verwendet werden, um Anfragen an einen Dienst weiterzuleiten und bietet Funktionen wie Authentifizierung, Verschlüsselung und Ratenbegrenzung.
4. **Sicherheit:** Das Service Mesh bietet Funktionen für Authentifizierung und Autorisierung, um die Sicherheit der Kommunikation zwischen den Diensten zu gewährleisten.
5. **Monitoring, Observability:** Service Mesh bietet betriebliche Funktionen wie Monitoring, Metriken, Protokollierung, Logging und Tracing, um die Überwachung und Fehlersuche bei den Diensten und ihren Interaktionen zu ermöglichen.
6. Service Mesh implementiert i.d.R. **Resilience-Patterns** wie Retries, Timeouts und Circuit Breaker, um sicherzustellen, dass die Dienste widerstandsfähig gegen Ausfälle sind (siehe Kapitel 4(E)).
7. **Service Discovery:** siehe ebenfalls Abschnitt 4 (E)

Bekannte Services Meshes sind:

1. **Linkerd** ist ein Service-Mesh für Kubernetes und andere Frameworks, das Funktionen wie Service-Erkennung, Lastausgleich und Sicherheit bietet.
2. **Istio** ist ein Open-Source-Service-Mesh, das Funktionen wie Service-Erkennung, Lastausgleich und Sicherheit bietet.

Bekannte und verbreitete Service Discovery Tools sind :

1. **Consul:** Ein Open-Source-Tool zur Service-Erkennung, das zur Registrierung und Erkennung von Services in einer verteilten Umgebung verwendet werden kann.
2. **Eureka:** Ein von Netflix entwickeltes Open-Source-Tool zur Service-Erkennung, das häufig in Microservice-basierten Architekturen eingesetzt wird.
3. **ZooKeeper:** Ein verteilter Koordinationsdienst, der für die Erkennung von Diensten und das Konfigurationsmanagement verwendet werden kann (z. B. innerhalb von Kafka)
4. **etcd:** Ein verteilter Key-Value-Speicher, der für die Erkennung von Diensten und das Konfigurationsmanagement verwendet werden kann.
5. **Kubernetes:** Eine Open-Source-Plattform zur Orchestrierung von Containern, die integrierte Funktionen zur Erkennung von Diensten enthält.
6. **Amazon ECS:** Ein von Amazon Web Services bereitgestellter Container-Orchestrierungsdienst, der über integrierte Funktionen zur Service-Erkennung verfügt.
7. **HashiCorp Nomad:** Ein flexibler Multi-Cloud-Workload-Orchestrator, der Dienste in verschiedenen Umgebungen registrieren und erkennen kann.
8. **NGINX Plus:** Eine für Unternehmen geeignete Version von NGINX, die Funktionen zur Erkennung von Diensten und zum Lastausgleich enthält.

(H) Event-driven Architecture (EDA)

1. Event-driven Architecture (EDA)

Info
Marc Richards, Neal Ford: "Handbuch moderner Softwarearchitektur", 1. Auflage, O'Reilly Verlag - Kapitel 14

Ben Stopford: "Designing Event-Driven Systems", O'Reilly, eBook als PDF bei https://www.confluent.io/resources/ebook/designing-event-driven-systems/

Eine **Event-driven Architecture (EDA, auch: event-basierte Architektur)** ist ein Softwarearchitekturmuster, das sich auf die Erzeugung, Erkennung und Nutzung von Events (Ereignissen) konzentriert. Ein solches Ereignis ist eine Zustandsänderung oder eine Nachricht, die von einem System oder einer Komponente erzeugt und von einem anderen System oder einer anderen Komponente konsumiert wird (typischerweise fachliche Events, beginnend mit Domain Events oder Entity-Änderungen).

In einer EDA kommunizieren Systeme und Komponenten, indem Producer solche Ereignisse erzeugen und Consumer diese konsumieren. Wenn eine Komponente ein solches Ereignis erzeugt, wird dieses an alle interessierten Abnehmer weitergeleitet, die damit das Ereignis konsumieren und verarbeiten. Dieses Vorgehen ermöglicht eine lose Kopplung zwischen den beteiligten Komponenten, da diese nicht übereinander Bescheid wissen müssen, um miteinander zu kommunizieren.

Die event-basierte Architektur zeichnet sich durch mehrere Merkmale aus:

- **Entkopplung:** Systeme und Komponenten sind lose gekoppelt, da sie nicht voneinander wissen müssen, um miteinander kommunizieren zu können. Dies ermöglicht eine größere Flexibilität und einfachere Änderungen.
- Hoher **Durchsatz:** Event-basierte Architekturen können eine große Anzahl von Ereignissen mit hohem Durchsatz verarbeiten und ermöglichen so eine performante, skalierbare Echtzeitverarbeitung von Daten.
- Typisch ist die **Persistierung** von Events auf der Infrastrukturebene sowie die **asynchrone Kommunikation** mit Blick auf a) erhöhte Skalierbarkeit sowie b) Verfügbarkeit, da Systeme und Komponenten unabhängig voneinander weiterarbeiten können, auch wenn beteiligten Komponenten (einige Zeit) nicht verfügbar sind.

Eine EDA kann in verschiedenen Szenarien eingesetzt werden, z. B. in der Echtzeit-Datenverarbeitung, in verteilten Systemen und in der Microservices-Architektur. Sie wird häufig in Anwendungen eingesetzt, die eine hohe Skalierbarkeit und geringe Latenzzeiten erfordern, wie z. B. IoT, Spiele und Aktienhandel.

Ereignisgesteuerte Architekturen können mit verschiedenen Technologien wie Nachrichtenwarteschlangen, Ereignisbussen implementiert werden und nutzen i.d.R. das Publish-Subscribe-Pattern. Frameworks wie Apache Kafka, RabbitMQ und Spring Cloud Stream können für die Implementierung verwendet werden.

2. Design von Events

Info
Beispiel für Events siehe Case Study "Flexinale", dort in Distributed: Im Projekt "flexinale-distributed-common" sind die technischen Grundlagen (Interfaces, Eventbus-Implementierungen etc.), in den drei fachlichen Services jeweils unter *apicontract siehe die konkreten Events und TOs.

Der **Entwurf von Ereignissen** in einer event-basierten Architektur (EDA) umfasst mehrere Schritte:

1. **Definition** der Ereignisse, die in der EDA verwendet werden sollen.
   a. Dazu gehören der Ereignisname, die Daten, die das Ereignis enthält,
   b. ... und das Format der Meta- sowie der Nutzdaten.
2. Die Ereignisse sollten so konzipiert sein,
   a. dass sie in sich **geschlossen** sind und alle Informationen enthalten, die von den konsumierenden Systemen und Komponenten benötigt werden. (Alternativ sind "Delta"-Ereignisse denkbar, aber i.d.R. problematisch, da dies Reihenfolgegarantien voraussetzt, die die Infrastruktur i.d.R. nicht leisten kann.)
   b. Ereignisse sollten nur die Informationen enthalten, die für eine einzelne, spezifische Aktion erforderlich sind, und **keine Logik** oder **Ver arbeitung** enthalten.
3. Events sind in der Regel **fachliche Ereignisse,**
   a. also Ergebnisse dessen, was sich bereits in der Vergangenheit ereignet hat (Beispiel: `TicketKaufWurdeDurchgeführtEvent`).
   b. Abweichend kann man mit einer EDA auch über Commands kommunizieren (Beispiel: `GutschriftEinlösenEvent`), was i.d.R. eine 1:1-Kommunikation der Beteiligten bedeutet (also keine Multi/Broadcast). Solche Commands sind keine Ereignisse im (semantischen) eigentlichen Sinne, sondern "nur" Messages. Mögliche Abhilfe und semantischer Trick: Umbenamung in `GutschriftEinlösenBeau ftragtEvent`
4. Definition des **Ereignisschema:** Das Schema sollte den Ereignisnamen, die Daten, die das Ereignis enthält, und das Format der Daten enthalten.

5. Ereignisse sollten **versioniert** sein (siehe unten).
6. Ereignisse sollten **Metadaten** (Zeitstempel, Version etc.) von Nutzdaten in Form von Transportobjekte (TOs) trennen.
7. Ereignisse sollten einfach **(de)serialisierbar** sein (z.B. keine bidirektionalen Beziehungen zwischen TOs abbilden).
8. Alle Ereignisse sollten der gleichen **Struktur** folgen, damit sie für die Consumer leichter zu verstehen und zu verarbeiten sind.
9. Ein Ereignis muss eindeutig **identifizierbar** sein
    a. **Ereignis-ID:** Jedes Ereignis sollte eine eindeutige Kennung (ID) haben, mit der es auf seinem Weg durch das System verfolgt werden kann.
    b. **Correlation-ID:** Jedes Ereignis sollte eine Correlation-ID für fachlich logische Folge-Event aufweisen, damit so Event-Kausal-Ketten durch das System über diese ID nachverfolgbar sind. In der Regel sind solche Event-Ketten einfache Ketten ohne Verzweigungen oder Zusammenführungen.
    c. optional **Historie:** Ein Ereignis kann die Historie des/r auslösenden Ereignisse mit sich führen, mit mehr Informationen als nur über die Correlation-Id.

Dazu muss **Ereignistransport** (Queue, Kafka, synchron oder asynchroner Event-Bus) und **Format** der Ereignisse (Serialisierung, Text oder Binär, XML oder JSON etc.) definiert werden. Der Transportmechanismus sollte auf der Grundlage der System-Anforderungen der EDA wie Skalierbarkeit, Zuverlässigkeit und Sicherheit, ausgewählt werden.

3. Versionierung von Event-Typen

In Bezug auf Änderungen und Versionskontrolle ist es am besten, ein **Versionsfeld** (fortlaufende Nummer) im Ereignis zu haben, das es den Consumern ermöglicht zu wissen, wie der Typ des Ereignisses zu behandeln ist.

- Sobald eine Änderung an der Ereignisstruktur vorgenommen wird, sollte die Version erhöht werden.
- Bei abwärtskompatiblen Änderungen (Beispiel: neues Feld kommt hinzu), sind zunächst alle Consumer prinzipiell in der Lage, auch die neue Version zu verarbeiten (sofern eigener Consumer-Code bzw. benutzte Bibliotheken wie z.B. Data-Binding-Frameworks solche neuen Felder einfach ignorieren).
- Dieses Vorgehen erlaubt eine schleichende Migration der Consumer, die somit weitgehend unabhängig auf die neue Event-Version umgestellt werden können.
- Man kann z.B. bei Nutzung von Apache Kafka so vorgehen: Event-Typen und Versionen werden immer in getrennte Topics geschrieben (siehe unten bei 9.). Wenn der Producer einen neuen Event-Typ E mit Versionsnummer n+1 produziert, schreibt er das in ein dediziertes Topic.. Spezieller Migrationscode (letztlich auch nur Consumer) migriert jede Event-Nachrichten von E[n+1] in das Topic E[n], auf dem alle bestehenden Consumer arbeiten. Dieser Migrationscode läuft so lange, bis keine alten Consumer mehr produktiv sind.

4. Ordnung von Events

Info
https://en.wikipedia.org/wiki/Lamport_timestamp

Zusätzlich sollten Events über einen **Zeitstempel** verfügen, der *prinzipiell* eine zeitliche Sortierung von Events in Event-Ketten ermöglicht.

Uhren in einem verteilten System sind nicht synchronisiert, bilden also keine globale Ordnung ab. Sie können also differerieren, so dass die Zeitstempel von Events, die von verschiedenen Services "nacheinander" erzeugt werden, nicht zwingend auch in ihren Zeitstempeln diese Differenz der Zeitspanne auch abbilden. Mit Hilfsmitteln wie der "Lamport-Uhr" kann man den Ereignissen in einem Verteilten System aufgrund ihres Zeitstempels eine partielle kausale Ordnung zuzuweisen.

5. Herausforderungen im Datenfluss-Monitoring in EDA

In einer event-basierten Architektur fließen die Daten als Ereignisse zwischen verschiedenen Komponenten und Diensten.

Einige der Herausforderungen bei der Überwachung dieser Datenflüsse sind:

a) fachliche **Komplexität** und **Nachvollziehbarkeit:** Event-basierte Architekturen sind komplex, da viele verschiedene Komponenten und Diensten Ereignisse gleichzeitig senden und konsumieren. Dies erschwert das Verständnis und die Verfolgung von Datenflüssen. Hier hilft die zentrale Sammlung aller Events in einem Audit-Log, in dem man mit Event-IDs und **Correlations-IDs** eine Analyse der Event-Ketten implementieren (oder gar visualisieren) kann.

b) Dazu technisch:

1. **Asynchronität:** Ereignisse werden i.d.R. asynchron verarbeitet werden, was bedeutet, dass die Reihenfolge, in der sie ausgesendet und konsumiert werden, nicht vorhersehbar ist. Dies macht es schwierig, festzustellen, ob ein Ereignis korrekt verarbeitet wurde.
2. **Delivery-Garantien:** Je nach verwendeter Infrastruktur muss ein Consumer mit den angebotenen verschiedenen Delivery-Garantien zurechtkommen.
    a. Eine Exactly-Once-Garantie ist unperformant (z.B. bei Kafka nur mit Kafka Streams zu lösen).
    b. Als Trade-Off wählt man i.d.R. At-Least-Once-Semantik, womit Events mehrfach zugestellt werden können. Dazu müssen Consumer idempotent arbeiten.
    c. Bei unkritischer Fachlichkeit reicht ggf. auch At-Most-Once.
3. **Transaktionalität:** Producer und Consumer sollten transaktional arbeiten (ist der Fall für Kafka-Eventbus).
4. **Skalierung:** Event-basierte Architekturen können eine große Anzahl von Ereignissen verarbeiten, was das Sammeln und Analysieren aller relevanten Daten schon alleine wg. der Datenmengen und großen Anzahl der Messages zu einer Herausforderung macht.

5. **Latenz:** In einem verteilten System kann eine Überwachung der Datenströmen zu erhöhten Latenzen führen, die sich negativ auf die Gesamtleistung des Systems auswirken können.
6. **Datenqualität:** Da Ereignisse von vielen Quellen erstellt werden, kann es schwierig sein, zentral die Qualität und Konsistenz der Daten zu gewährleisten.

6. Apache Kafka als Eventbus (in der Case Study "Flexinale - Distributed")

Info
Wir nutzen Apache Kafka als verteilten und asynchronen Eventbus (nur in Distributed). Docker-Images für Apache Kafka, Zookeeper und Kafdrop stehen bereit.

Siehe "Chapter Integration 4 - Event-driven Architecture" (Link)

https://kafka.apache.org/

Neha Narkhede, Gwen Shapira, Todd Palino: "Kafka: The Definitive Guide", 2017, O'Reilly Media, Inc., https://www.oreilly.com/library/view/kafka-the-definitive/9781491936153/

Apache Kafka ist eine verteilte Streaming-Plattform, die häufig als Infrastrukturebene in einer Event-Driven Architektur verwendet wird. Sie ist für die Verarbeitung von Datenströmen mit hohen Mengengerüsten, hohem Durchsatz und niedriger Latenz konzipiert und eignet sich daher gut für den Einsatz in Ereignis-basierten Systemen.

Abbildung: Apache Kafka Logo

Kafka ist ein leistungsfähiges Tool für den Aufbau Ereignis-basierter Architekturen, da es eine verteilte, fehlertolerante und durchsatzstarke Nachrichten-Streaming-Plattform bietet. Die Entwicklung einer Kafka-basierten ereignisgesteuerten Architektur umfasst die Erstellung von Producern und Consumern, die mit Topics interagieren, sowie die Verwaltung des Kafka-Clusters, der Topics und der Consumer, die Überwachung des Systems und die Erstellung einer Backup-Strategie.

In Bezug auf die Entwicklung umfasst eine Kafka-basierte ereignisgesteuerte Architektur normalerweise die folgenden **Komponenten**:

- Broker: Kafka-Prozess für die Verwaltung der Infrastruktur.
- Producer: Anwendungen oder Dienste, die Nachrichten produzieren und diese an einen Kafka-Cluster senden.
- Topic: Logische Kanäle, an die Nachrichten gesendet und von denen sie konsumiert werden. Topics sind in Partitionen unterteilt, die über mehrere Broker in einem Kafka-Cluster repliziert werden.
- Consumer: Anwendungen oder Dienste, die Nachrichten aus Topics konsumieren und eine Aktion durchführen, z. B. eine Datenbank aktualisieren oder ein neues Ereignis auslösen.
- Consumer-Groups: Eine Gruppe von Consumern, die zusammenarbeiten, um Nachrichten aus einem Topic zu konsumieren. Jedem Consumer in einer Gruppe wird ein Satz von Partitionen zugewiesen, aus denen er Nachrichten konsumiert, was eine parallele Verarbeitung von Nachrichten ermöglicht, also gute Skalierbarkeit bei passender Aufteilung der Daten.

Die Infrastruktur-**Verwaltung** umfasst folgenden Themen:

- Cluster-Verwaltung: Einrichten und Verwalten eines Kafka-Clusters, einschließlich Hinzufügen oder Entfernen von Brokern, Konfigurieren der Replikation und Überwachen des Zustands des Clusters.
- Verwaltung von Topics: Erstellen, Löschen und Konfigurieren von Topics, einschließlich der Festlegung von Aufbewahrungsrichtlinien und Partitionierungsstrategien.
- Verwaltung von Consumern: Verwalten und Überwachen von Consumer-Groups, einschließlich Anpassen der Anzahl der Consumer, Überwachen des Consumer-Offset/Versatzes ("Lag") und Verwalten von Consumer-Verbindungen zum Cluster.
- Überwachung: Überwachung wichtiger Metriken wie Nachrichtenrate, Verzögerung der Consumer und Zustand des Clusters sowie Verwendung von Tools wie Kafka Manager, Kafdrop oder Prometheus zur Visualisierung dieser Metriken und zur Behebung von Problemen.
- Sicherung und Wiederherstellung: Erarbeitung einer Strategie für die Sicherung und Wiederherstellung von Daten im Falle eines Ausfalls.

Kafka bietet folgende **Zustell-Garantien**:

- At most once: Eine Nachricht wird höchstens einmal an den Kafka-Broker gesendet. Es besteht jedoch keine Garantie dafür, dass die Nachricht sicher an die Konsumenten zugestellt wird.
- At least once: Eine Nachricht wird mindestens einmal an den Kafka-Broker gesendet, was bedeutet, dass Nachrichten nicht verloren gehen. Es kann jedoch dazu führen, dass Nachrichten doppelt verarbeitet werden, wenn es zu einer Duplikation kommt. Konsumenten müssen dann mit diesen Duplikationen umgehen können.
- Exactly once: Eine Nachricht wird genau einmal zugestellt. Diese Zustellgarantie ist am sichersten, aber auch am komplexesten und kann in bestimmten Konfigurationen eine höhere Latenz aufweisen.

"At least once" ist wohl die am meisten genutzt Art - in der Regel möchte man, dass Nachrichten im System auch ankommen. Mit "Exactly once" sollte man vorsichtig sein, wegen der hohen Komplexität und der möglichen Latenz. Die Flexinale verwendet "At least once".

Kafka speichert Events und ermöglicht es so, Daten über einen bestimmten Zeitraum hinweg zu speichern. Damit kann Kafka bis zu einem gewissen Grad als Persistenz eingesetzt werden.

7. Überblick über Apache Kafka Admin Tools

Es gibt eine Reihe von Admin-Tools, typischerweise als webbasierte Tools zur Verwaltung und Überwachung von Apache Kafka. Ein solches Tool kann verwendet werden, um den Zustand eines Kafka-Clusters zu überwachen, Consumer- und Producer-Metriken anzuzeigen und administrative Aufgaben wie die Neuzuweisung von Partitionen und die Konfiguration von Aufbewahrungsrichtlinien durchzuführen.

Einige beliebte **Admin-Tools** (Monitoring) zur Verwaltung und Überwachung von Apache Kafka sind:

- Kafka Manager, siehe unten
- Kafdrop, siehe unten (wird in der Case-Study Flexinale/Distributed benutzt)
- Kafka Monitor: Hierbei handelt es sich um ein Open-Source-Tool, das eine webbasierte Benutzeroberfläche für die Überwachung von Kafka-Clustern bietet. Es umfasst Funktionen wie die Überwachung von Topics und Partitionen, die Verfolgung von Consumer-Offsets und Warnmeldungen.
- Kafka Tool: Hierbei handelt es sich um eine UI für die Verwaltung und Überwachung von Kafka-Clustern. Es umfasst Funktionen wie Topic Browsing, Message Browsing und Consumer Offset Tracking.
- Kafka-topics-ui: Dies ist eine webbasierte Benutzeroberfläche für die Verwaltung von Kafka-Topics. Sie ermöglicht Benutzern das Erstellen, Löschen und Auflisten von Topics sowie das Anzeigen von Topic-Konfiguration und Partitionsinformationen.
- Burrow: Hierbei handelt es sich um ein Überwachungstool, das den Fortschritt von Kafka-Consumeren und deren Verzögerung verfolgt. Es bietet eine webbasierte Benutzeroberfläche und kann in Alerting-Systeme integriert werden.
- kafka-exporter: Dies ist ein Prometheus-Exporter, der Metriken von Kafka-Clustern abrufen und für die Überwachung und Alarmierung exportieren kann.
- kafka-lag-exporter: Dies ist ein Prometheus-Exporter, der Consumer-Offset- und Lag-Metriken aus Kafka-Clustern auslesen und für die Überwachung und Alarmierung exportieren kann.
- Kafka-graphite-reporter: Dies ist ein Tool, das Metriken von Kafka-Clustern sammeln und zur Speicherung und Visualisierung an Graphite senden kann.
- KaDeck von Xeotek, https://www.kadeck.com/

Kafka Manager bietet eine Reihe von Funktionen, die für die Überwachung einer Kafka-basierten ereignisgesteuerten Architektur nützlich sein können, darunter:

- Cluster-Übersicht: Bietet eine Übersicht über den Zustand eines Kafka-Clusters, einschließlich der Anzahl von Brokern, Topics, Partitionen und Consumer-Groups.
- Details zu Topics und Partitionen: Ermöglicht die Anzeige detaillierter Informationen zu bestimmten Topics und Partitionen, einschließlich der Anzahl der produzierten und konsumierten Nachrichten und Bytes sowie des Rückstands der Consumer-Groups.
- Metriken für Consumern und Producern: Bietet detaillierte Metriken für Consumer und Producer, einschließlich der Anzahl der produzierten und konsumierten Nachrichten und Bytes sowie der Rate der produzierten und konsumierten Nachrichten.
- JMX-Metriken: Ermöglicht die Anzeige von JMX-Metriken für Broker, Topics und Consumer-Groups.
- Partitionen neu zuweisen: Ermöglicht die Neuzuordnung von Partitionen zu verschiedenen Brokern.

Kafdrop ist ein weiteres solches webbasiertes Tool zur Inspektion und Überwachung von Apache Kafka-Clustern und setzt um:

- Durchsuchen von Topics: Ermöglicht die Anzeige detaillierter Informationen zu bestimmten Topics, einschließlich der Anzahl der Partitionen, Replikate und Nachrichten.
- Durchsuchen von Nachrichten: Ermöglicht es, den Inhalt bestimmter Nachrichten innerhalb eines Themas anzuzeigen.
- Verfolgung des Consumer-Offsets: Ermöglicht die Anzeige des Offsets von Consumer-Groups, einschließlich der Verzögerung und Rate der verbrauchten Nachrichten.
- Konfig-Verwaltung: Ermöglicht das Anzeigen und Bearbeiten von Konfigurationen auf Topic-Ebene, wie z. B. Aufbewahrungsrichtlinien und Bereinigungsrichtlinien.
- JMX-Metriken: Ermöglicht es, JMX-Metriken für Broker und Topics anzuzeigen.
- Zookeeper-Integration: Ermöglicht die Anzeige von Zookeeper-bezogenen Informationen, wie z. B. den Status von Brokern und den Zustand des Clusters.

8. Limitierung der Nachrichtengröße

Kafka ist für hohen Durchsatz vieler, aber kleiner Nachrichten konzipiert. Das Limit solcher Nachrichten beträgt wenige Megabytes. Will man dennoch große Datensätze / BLOBs über Kafka verschicken (z.B. Bild-, Video-, Audiodateien), so gibt es einige wenige Lösungsansätze, darunter "**Reference-Based Messaging**".

Bei Reference-Based Messaging werden über Kafka die eigentlichen Events/Nachrichten zugestellt. Diese enthalten aber nicht die großen BLOBs, sondern verweisen stattdessen auf eine andere Ablage (Ordner in Netzwerk-Filesystem, in S3 etc.). Wenn die Kafka-Nachricht von einem Client verarbeitet wird, muss dieser also auch noch die BLOB-Nutzdaten von der anderen Ablage auslesen.

9. Das Apache Kafka Consumer-Lag ist wichtig für den Betrieb!

Das **Consumer-Lag** in einer ereignisgesteuerten Architektur auf Kafka-Basis ist eine, vielleicht *die* wichtigste Metrik, die überwacht werden muss, da sie Aufschluss darüber gibt, wie gut die Consumern mit der Nachrichtenproduktion Schritt halten und wie groß der Rückstand im System ist. Die Überwachung des Consumer Lag kann helfen, Leistungsprobleme zu erkennen und Korrekturmaßnahmen zu ergreifen, um sicherzustellen, dass die Nachrichtenproduktion und -Verarbeitung reibungslos und effizient läuft.

Das Kafka-Consumer-Lag bezieht sich auf die Differenz zwischen dem Offset einer Consumer-Group und dem Offset der letzten Nachricht in einer Topic-Partition. Er misst, wie weit eine Consumer-Group bei der Verarbeitung von Nachrichten in einer Topic-Partition zurückliegt, und ist demzufolge wichtig zu überwachen, da dieses Lag auf Probleme mit dem Durchsatz der ereignisgesteuerten Architektur hinweist.

- Ein hoher Lag der Consumer deutet i.d.R. darauf hin, dass die Consumer nicht in der Lage sind, mit der Geschwindigkeit der Nachrichtenproduktion Schritt zu halten. Dies kann durch eine Reihe von Faktoren verursacht werden, z. B. durch eine langsame Verarbeitung von Nachrichten, unzureichende Ressourcen oder Netzwerkverzögerungen. In diesem Fall hilft die Überwachung des Consumer Lag, das Problem zu identifizieren und Korrekturmaßnahmen zu ergreifen, wie z. B. die Skalierung Ihrer Consumer-Infrastruktur oder die Optimierung Ihres Nachrichtenverarbeitungscodes.
- Ein geringer Lag bei den Consumern deutet hingegen i.d.R. darauf hin, dass die Consumer die Nachrichten "zu schnell" verarbeiten. Das ist nicht per se schlecht, aber es ist überdenkenswert, ob nicht eine geringer Consumer-Zahl für die Datenmenge ausreicht.

Für einen idealen Ressourcenverbrauch erfolgt eine dynamische Skalierung von Consumern auf Basis des Lags: Bei wachsendem Lag werden mehr Consumer gestartet, bei sinkendem Lag werden Consumer gestoppt.

10. Events und Kafka - Zuordnung von Event-Typen auf Kafka-Topics

Jeder Ereignis-Typ sollte separiert, z.B. in einem **separaten Kafka-Topic** veröffentlicht werden, damit es von den Consumern leicht "gefunden" oder gefiltert werden kann (ohne in die einzelnen Messages "reinschauen" zu müssen). Bei der Zuordnung von Ereignissen zu Kafka-Topics ist also ideal, wenn **jeder Event-Typ eindeutig einem Topic** zugeordnet wird, also in einem Topic nicht mehrere Event-Typen vermischt sind.

Der **Name** des Topics sollte aussagekräftig sein und den Typ der Ereignisse widerspiegeln, die in ihm veröffentlicht werden.

- Dies erleichtert es einem Consumer, die Topics zu identifizieren, die es interessiert, und die Ereignisse nach ihrem Thema zu filtern
- Es hilft v.a. auch bei fachlichen Analysen.
- Es hilft auch dabei, zielgerichtet Lese/Schreib-Rechte auf die Topics zu managen.

Eine **Versionierung** kann auch über Topicnamen gelöst werden (und erlaubt einfaches Migrationstooling zwischen TOPICNAME-<Version-N> und TOPICNAME-<Version-N+1>), siehe dazu oben bei Abschnitt 3.

(I) Tests und Integrationstests

1. Testarten, Testpyramide(n), Testautomatisierung

Info
https://istqb-glossary.page/de/testpyramide/

Testarten, bzw. Testphasen werden häufig in einer **Testpyramide** dargestellt. **Automatisierte Tests** können einfach ausgeführt werden und in der CI/CD-Pipeline mitlaufen. Daher strebt man einen hohen Automatisierungsgrad der Tests an. Testautomatisierung ist jedoch auch aufwendig. Daher ist immer auch eine Abwägung zwischen Kosten und Nutzen motwendig.

Einige gängige **Testarten** sind:

1. **Unit-Tests** testen einzelne Code-Einheiten, wie einzelne Klassen oder Funktionen. Unit-Tests werden in der Regel zu Beginn der Pipeline ausgeführt und dienen dazu, Fehler im Code frühzeitig zu erkennen.
2. **Komponententests** testen größere Einheiten, also ganze Komponten. Sie werden mit oder nach den Unit-Tests ausgeführt.
3. **Integrationstests** testen eine übergreifende und integrative Sicht verschiedener Code-Einheiten, z.B. verschiedener Komponenten oder Dienste. Integrationstests werden seltener ausgeführt, da sie wesentlich mehr Aufwand an Vorbereitung und Durchführung bedeuten.
4. **Systemtests** testen ein gesamtes System (i.d.R. mit "echter Hardware", "echten Schnittstellen"). Die Abgrenzung zu Integrationstests ist dabei nicht immer scharf.

Testarten aus Sicht der Anwender:

1. **Funktionstests** testen die Funktionalität der Anwendung aus der Sicht des Benutzers. Funktionstests werden in der Regel nach den Integrations- oder Systemtests durchgeführt und dienen dazu, Fehler abzufangen, die bei der Verwendung der Anwendung in einem realen Szenario auftreten können.
2. **Abnahmetests** dienen der Abnahme des Systems. Sind sind vom Charakter ähnlich wie Funktionstests, haben aber auch einen formalen Charakter, eben der "Abnahme" eines Systems.

Tests, die Qualitätsanforderungen abtesten, sind beispielsweise

1. **Last-/Performance-Tests:** siehe unten. Ihre Integration in eine CI/CD-Pipeline ist möglich, aber häufig werden sie auch separat ausgeführt, unter anderem, weil sie hohe Anforderungen an ihre Umgebung haben und weil sie in der Regel lange laufen.
2. **Security-Tests:** Hier geht es um automatisierte Tests, die die Sicherheit der Anwendung prüfen, z. B. ob SQL-Injection oder Cross-Site-Scripting möglich ist.

Die Abbildung zeigt die gängige Darstellung der Testpyramide:

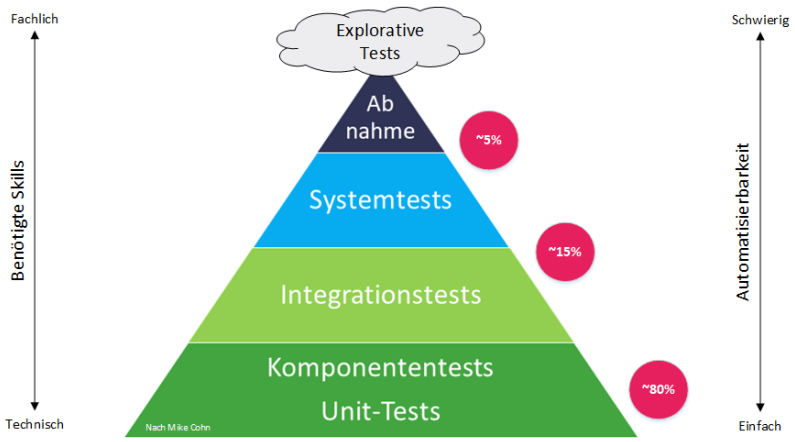


Abbildung: Testpyramide
(Quelle: Mirco de Rohni: "Automatisiertes Testen",
https://www.mircoderoni.ch/2019/08/31/automatisiertes-testen/)

Je weiter unten in obiger Pyramide eine Testphase liegt, desto schneller laufen die Tests und desto weniger Anforderungen an ihre Umgebung haben die Tests. Unittests haben in der Regel keine Anforderungen an ihre Umgebungen. Für Integrationstests müssen Teile einer Anwendung bereits gemeinsam laufen, sie benötigen eventuell auch schon Daten / eine Datenbank. Spätestens die Funktionstests benötigen einen oder mehrere vollständige Services und häufig Daten und damit eine Datenbank.

Damit sind die Tests, die weiter oben in der Pyramide liegen, auch schwergewichtiger in dem Sinne, dass sie aufwendiger zu schreiben sind, eine aufwändigere Umgebung brauchen und länger laufen (inkl. Testvorbereitung - z.B. Bereitstellung passender Testdaten). Daher kann eine

hohe Testabdeckung weiter unten in der Pyramide hilfreich sein. Das ist in der in obiger Testpyramide auch so angedeutet. Das ist jedoch ein zweischneidiges Schwert. Ein Aufwandstreiber sind Mocks, die ggf. zu schreiben sind: Wenn dafür (zu) viele Mocks zu erstellen sind, insbesondere wenn der Datenzugriff / die Datenbank weggemockt werden müsste, ist der Aufwand dafür (und das dadurch möglicherweise abweichende Verhalten der Anwendung) dagegen abzuwägen.

Entscheidend für die Effizienz ist ein möglichst **hoher Automatisierungs- und Reproduzierbarkeitsgrad** (der entsprechend voraussetzt, dass das System bzw. dessen Teile von außen gesteuert/angesprochen werden können und dass entsprechende Testszenarien und -daten, ggf. eine Datenbank, vorhanden und benutzbar sind). Dies ist insbesondere dann einfach(er) möglich, wenn Virtualisierung / Containerisierung das Testobjekt für solche Tests verfügbar machen (gut möglich für Backend-Systeme, weniger geeignet für Frontend-/Mobile Anwendungen).

Bei der Entscheidung, welche und wieviele Tests man wo in der Testpyramide planen sollte, ist eine weitere Frage interessant: Wie **aussagekräft ig** und damit (fachlich) wertvoll sind welche Tests, wie nachstehende Abbildung (und der verlinkte Artikel) aussagen:

- Man investiert gezielt in "inhaltliche Aussagekraft", nimmt also den Aufwand für integrative und umfassende Tests in Kauf, weil diese mehr Aussagekraft haben.
- Man investiert bewusst weniger in Unit-Tests, denn diese sind a) teilweise (fachlich) nichtssagend und b) aufwändig zu pflegen und bei Refactorings zu ändern.

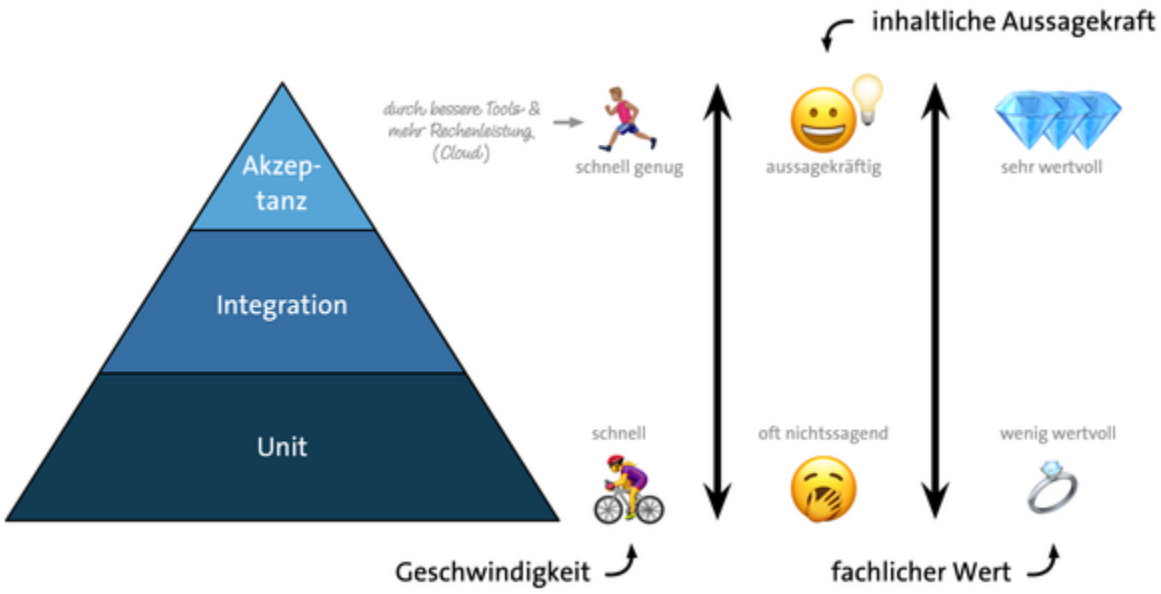


Abbildung: Testpyramide mit weiteren Kriterien
(Quelle: "Alles steht Kopf – Die umgedrehte Testpyramide",
neuland bfi, https://www.neuland-bfi.de/blog/alles-steht-kopf-die-umgedrehte-testpyramide)

2. Weitere Testarten

Info
Jan Baganz: "Evaluierung von Property-Based-Testing als Testmethodik in der praktischen Softwareentwicklung", Masterarbeit 2017, PDF online unter https://accso.de/app/uploads/2019/12/Masterarbeit_Jan-Baganz_201704_Property-Based-Testing_PBT.pdf

Dr. Lars Hupel: "Eigenschaftsbasiertes Testen mit „fast-check“. 1000 auf einen Streich", https://www.innoq.com/de/articles/2023/02/testing-fast-check/

https://en.wikipedia.org/wiki/Mutation_testing

Weitere Testarten und -verfahren wie Mutation Testing, Property Based Testing etc. können gezielt die Testabdeckung erweitern (hier nicht weiter ausgeführt).

3. Architekturtests

Info
Neal Ford, Rebecca Parsons, Patrick Kua: "Building Evolutionary Architecture", https://evolutionaryarchitecture.com

Dr. Kristine Schaal: "Wie fit ist deine Architektur? Automatisierte Architekturtests & statische Codeanalyse mit ArchUnit" https://speakerdeck.com/kristines/20240201-oop-schaal-fitnessfuncions-automatisierte-architekturtests-und-statische-codeanalyse-mit-archunit

https://de.wikipedia.org/wiki/Problem_des_Handlungsreisenden

Weiterhin sollten **Architekturtests** (z.B. mit ArchUnit) benutzt werden, die automatisiert sicherstellen, dass Architekturvorgaben sichergestellt sind. Hier wird - i.d.R. über statische Code-Analyse - eine automatisierte Bewertung des Systems und dessen Codestrukturen vorgenommen (z. B. Check auf erlaubte Abhängigkeiten).

Architekturtests kann man auch als sog. **Fitness Function** nutzen: Das Konzept der Fitness Function kommt aus dem Gebiet der Genetischen Algorithmen zur Lösung von Optimierungproblemen (z.B. „Travelling Salesman“): Eine Fitness Funcion misst hier eine Lösung, so dass Lösungen vergleichbar werden. Damit wird letztlich eine Metrik ausgedrückt (bei „Travelling Salesman“ ist dies die Länge der Route durch alle Städte).

Architekturtests messen, ob die Architekturregeln erfüllt werden. Mit Fitness Functions misst man, wie gut eine Architektur die Architekturvorgaben erfüllt sind. Beispiele sind Performanzmessungen, Security, Energieeffizienz. Es können einzelne Qualitätsanforderungen gemessen werden ("Atomic Fitness Function") oder mehrere in Kombination ("Holistic Fitness Function"). Architekturtests sind somit Fitness Functions für die (statische) Struktur des Codes. Sie können als echte Tests eine boole'schen Aussage liefern ("ok" - keine Regelverletzung; "not ok" - mindestens eine Regelverletzung), oder auch eine Zahl, z.B. die Zahl der Architekturverletzungen angeben.

Fitness Functions spielen für "Evolutionary Architecture" eine Rolle: Mit jedem Schritt in der Evolution und Weiterentwicklung misst man anhand der Fitness Function(s), wie gut bzw. um wieviel besser (oder schlechter) die neue Architektur die Qualitätsanforderungen erfüllt.

4. Last- und Performance-Tests

Last- und Performance-Tests sind Test-Techniken, mit denen sichergestellt werden kann, dass eine Anwendung die erwarteten Lasten bewältigen kann und unter realen Bedingungen gut funktioniert. Durch die Simulation verschiedener Belastungsszenarien und die Messung der Leistung und des Verhaltens der Anwendung können Last- und Leistungstests dazu beitragen, Probleme zu erkennen und zu beheben, die andernfalls möglicherweise unbemerkt bleiben würden.

1. Diese Tests können helfen, Engpässe in der Anwendung zu erkennen, wie z. B. langsame Datenbankabfragen, Speicherlecks oder Netzwerklatenz. Durch das **Erkennen und Beheben dieser Engpässe** kann die Anwendung leistungsfähiger gemacht werden.
2. Mit diesen Tests kann sichergestellt werden, dass die Anwendung **skaliert** werden kann, um steigende Lasten zu bewältigen, z. B. bei Lastspitzen oder wenn neue Funktionen hinzugefügt werden. Auf diese Weise kann sichergestellt werden, dass die Anwendung auch bei hoher Belastung den Anforderungen der Benutzer gerecht wird.
3. Verbesserung der **Benutzerfreundlichkeit:** Last- und Leistungstests können dazu beitragen, Probleme zu erkennen, die sich negativ auf die Benutzerfreundlichkeit auswirken könnten, wie z. B. langsame Ladezeiten oder schlechte Antwortzeiten. Durch das Erkennen und Beheben dieser Probleme kann die Anwendung reaktionsschneller und benutzerfreundlicher gestaltet werden.

Idealerweise kann - entsprechende Umgebungen, Infrastruktur und Testdaten vorausgesetzt - ein solches Testverfahren automatisiert ablaufen, ggf. sogar als Teil des Nightly Builds laufen: Werden dabei starke Abweichungen zwischen einzelnen Testläufen (zum Nightly davor) identifiziert, ist die Fehlerdiagnose gut eingrenzbar und die Fehlerursache auf die Commits eines kurzen Zeitraums eingeschränkt.

5. Tests und Automatisierung in Pipelines

Der Umfang der Tests und der Grad der **Automatisierung in CI/CD-Pipelines** hängen stark von den spezifischen Anforderungen der Anwendung ab.

Im Allgemeinen ist es eine gute Idee, so viele Tests und Automatisierungen wie möglich einzubeziehen, um sicherzustellen, dass die Anwendung gründlich getestet wird. Eine Vielzahl von automatisierten Tests in der Pipeline mit Unit-Tests, Integrationstests, Funktionstests, Leistungstests und Sicherheitstests hilft, eine breite Palette von Problemen frühzeitig zu erkennen.

Es ist jedoch auch wichtig, die Kompromisse zwischen dem Umfang der Tests und der Automatisierung und den für die Implementierung und Wartung verfügbaren Ressourcen zu berücksichtigen. Es muss ein **Gleichgewicht** gefunden werden **zwischen der Maximierung des Test- und Automatisierungsumfangs und der Minimierung des Zeit- und Arbeitsaufwands** für die Implementierung und Wartung.

Es empfiehlt sich, mit einem kleineren Test- und Automatisierungsset zu beginnen und die Abdeckung und Komplexität nach Bedarf schrittweise zu erhöhen, wobei die verfügbaren Ressourcen, die Anforderungen der Anwendung und das gewünschte Risiko- und Qualitätsniveau berücksichtigt werden.

6. Testbarkeit

Bereits beim Entwerfen der Architektur werden die Weichen gestellt, wie gut und einfach es hinterher ist, die Anwendung zu testen. Man sollte sich also frühzeitig Gedanken machen, wie man die Architektur so gestalten kann, dass sie gut und einfach **testbar** ist (siehe Kapitel 2 (A), 5. und 2 (B) "Testability").

Hier helfen ebenfalls Prinzipien der flexiblen Architektur:

- Kleine, unabhängige Komponenten lassen sich leichter testen.
- Ein funktionaler Stil erleichtert das Testen, d.h. Methoden, Klassen oder Schnittstellen sollten nur von ihren Eingabeparametern abhängen, ihre Ergebnisse zurückgeben und weder von Seiteneffekten abhängen noch sie auslösen.

7. Gekoppelte Systeme sind schwierig und aufwändig zu testen

Das Testen (eng) gekoppelter Systeme ist i.d.R. deutlich schwieriger als das Testen entkoppelter Systeme, da sich Änderungen in einem Teil des Systems auf andere Teile des Systems auswirken. Um gekoppelte Systeme zu testen, ist es oft effizienter, das gesamte System als Ganzes zu testen, anstatt einzelne Teile isoliert zu testen. Solche **Integrationstests** sind eine Technik, bei der verschiedene Teile des Systems gemeinsam, also als Ganzes, getestet werden. Auf diese Weise kann man testen, wie die verschiedenen Teile des Systems miteinander interagieren, und sicherstellen, dass das System als Ganzes korrekt funktioniert.

- Das Testen gekoppelter Systeme stellt aufgrund der Abhängigkeiten zwischen den Komponenten eine Herausforderung dar (je mehr Abhängigkeiten, je schwieriger). Dies bedeutet, dass das System möglicherweise nur dann korrekt funktioniert, wenn bestimmte Komponenten vorhanden sind oder bestimmte Bedingungen erfüllt sind. Dies kann es schwierig machen, das System in einer kontrollierten bzw. isolierten Umgebung zu testen und sicherzustellen, dass es in einer Produktionsumgebung korrekt funktioniert.
- Das Testen gekoppelter Systeme kann alleine aufgrund der Komplexität der Interaktionen zwischen den verschiedenen Komponenten eine Herausforderung darstellen. Dies kann es erschweren, das Verhalten des Systems als Ganzes zu verstehen und die Ursache für eventuell auftretende Probleme zu ermitteln.

8. Integrativ oder integriert? Integrationstests!

Integrationstests sind eine Art von Softwaretests, bei denen mehrere Komponenten oder Module eines Systems kombiniert und als Gruppe getestet werden, um sicherzustellen, dass sie wie erwartet zusammenarbeiten. Das Ziel von Integrationstests ist es, Probleme zu identifizieren und zu beheben, die bei der Integration verschiedener Komponenten oder Module auftreten können, und zu überprüfen, ob das System als Ganzes korrekt funktioniert.

Es gibt mehrere gängige **Testmuster,** die bei Integrationstests verwendet werden, darunter:

- Big-Bang-Tests: Alle Komponenten werden in einem Durchgang integriert und gemeinsam getestet. Dieser Ansatz wird in der Regel verwendet, wenn die Komponenten eng miteinander gekoppelt sind und es schwierig ist, sie separat zu testen.
- Top-Down-Tests: Das System wird beginnend mit den Komponenten der höchsten Ebene getestet und dann zu den Komponenten der unteren Ebene heruntergearbeitet. Dieser Ansatz wird in der Regel verwendet, wenn das System in einer mehrschichtigen Architektur aufgebaut ist.
- Bottom-Up-Tests: Das System wird beginnend mit den Komponenten der untersten Ebene getestet und arbeitet sich dann zu den Komponenten der höheren Ebene vor. Dieser Ansatz wird in der Regel verwendet, wenn das System mit einer mehrschichtigen Architektur aufgebaut ist.
- Sandwich-Tests (Hybrid-Tests): Eine Kombination aus Top-Down- und Bottom-Up-Ansatz, auch bekannt als hybrides Testen.
- Inkrementelles Testen: Das System wird in kleinen Schritten getestet, wobei neue Komponenten hinzugefügt und getestet werden, wenn sie entwickelt werden. Dieser Ansatz wird in der Regel verwendet, wenn das System in einem iterativen oder inkrementellen Entwicklungsprozess entwickelt wird.

Man spricht oft von "Integrationstests", ohne konkret bis ins Letzte zu beschreiben, was damit gemeint ist. Zu unterscheiden: **"Integrative Tests"** ("integrative") sollten von **"integrierten Tests"** ("integrated") unterschieden werden, auch wenn das wie eine etwas künstliche Abgrenzung wirkt. Beispiel:

In einem verteilten Event-driven System, in dem z.B. die Bestandteile über einen Eventbus kommunizieren, will man in solchen Tests das Gesamtzusammenspiel der Komponenten testen, um echte Ende-zu-Ende-Tests zwischen den Services und dem Eventfluss durchzuführen.

Eine besondere Herausforderung für Tests sind verteilte Architekturen. Daher sollte man hier die Tests aufteilen in

- "**integriert**": Die verteilten Services werden in einen Deployment-Monolithen zusammengeführt (z.B. eine Spring-Boot-App) und **fachlich** getestet (Beispiel: Kommen die Events richtig an, werden die richtigen Folgeevents erzeugt, ...?)
- "**integrative**" Tests: Hier wird "echt" verteilt getestet, v.a. mit Blick auf **Technik**, Infrastruktur und Verteilung (Beispiel: Kommen die Events richtig über Kafka von Producern an alle Consumer?)

Ein integrativer Test würde nun einen asynchronen Eventbus wie Kafka als echte Infrastruktur nutzen, um den Test durchzuführen. Die Systemteile kommunizieren über die Zielinfrastruktur. Ein integrierter Test startet dagegen alle Services (integriert) innerhalb einer monolithischen Anwendung und nutzt für die Kommunikation einen In-Memory-Bus, um die Einzelteile integriert zu testen. Damit ist zumindest der fachliche und lose gekoppelte Kommunikationsfluss via Events in den Tests sichergestellt.

Die Verteilung und asynchrone Kommunikation (über den Bus) sowie die damit verbundenen Latenzen oder Schwierigkeiten mit Mehrfachzustellung etc. sind so nicht testbar (vgl. auch "Law of Leaky Abstraction").

Die beiden Testarten sind also nicht deckungsgleich, sondern ergänzen einander.

9. Wie kann man gekoppelte Systeme testen? Mocking oder Container?

Info
https://site.mockito.org/

https://www.testcontainers.org/

Eine Möglichkeit, gekoppelte Systeme ein Stück weit isoliert zu testen, ist die Verwendung von **Mocking** und **Containern.**

Mocking ist eine Technik, bei der Dummy-Objekte oder -Funktionen verwendet werden, um das Verhalten von echten Objekten oder Funktionen zu simulieren. Dies ermöglicht es Entwicklern, den zu testenden Code vom Rest des Systems zu isolieren und den Code isoliert zu testen. Mocking-Frameworks wie mockito können zur Erstellung von Mock-Objekten verwendet werden, die beim Testen anstelle echter Objekte eingesetzt werden können. Diese Mock-Objekte können das Verhalten echter Objekte simulieren und so konfiguriert werden, dass sie bestimmte Werte zurückgeben oder bestimmte Ausnahmen auslösen, so dass die Entwickler verschiedene Szenarien und Randfälle testen können. Mocking ist nicht ideal, da es durchaus aufwändig sein kann, einen Mock für ein komplexes Verhalten nachzustellen. Außerdem steckt im Mock-Verhalten immer das bereits erwartete Verhalten des Gegenübers im Test, das aber nicht die Realität abbilden muss.

Eine Möglichkeit, Integrationstests durchzuführen, ohne alle externen Systeme einrichten zu müssen, ist die Verwendung von **Mock-Objekten.** Ei n Mock-Objekt ist eine simulierte (Minimal-)Version eines realen Systems oder einer Komponente, die während der Tests anstelle des tatsächlichen Systems oder der Komponente verwendet werden kann. Das Scheinobjekt ahmt das Verhalten des realen Systems oder der realen Komponente nach, führt aber keine "reale Arbeit" aus. Vorteile der Verwendung von Mock-Objekten:

- Isolierung: Mock-Objekte können verwendet werden, um das zu testende System von externen Abhängigkeiten zu isolieren, so dass das System in einer kontrollierten Umgebung getestet werden kann.
- Geschwindigkeit: Die Verwendung von Mock-Objekten kann das Testen erheblich beschleunigen, da das System nicht auf die Reaktion externer Systeme warten muss.
- Wiederholbarkeit: Mock-Objekte können so konfiguriert werden, dass sie konsistente Ergebnisse liefern, was die Wiederholbarkeit der Tests erhöht und die Wahrscheinlichkeit von Fehlalarmen verringert.
- Kosten: Wenn die realen externen Systeme teuer oder schwer zugänglich sind, lassen sich durch Mocking die Kosten für Integrationstests senken.

Es gibt mehrere Bibliotheken, die zur Erstellung von Mock-Objekten in verschiedenen Programmiersprachen verwendet werden können, z. B. jMockit, EasyMock, Mockito, usw. Es ist zu beachten, dass Mock-Objekte mit Bedacht eingesetzt werden sollten, da sie die Integrationstests weniger realistisch machen und möglicherweise nicht alle Probleme abfangen, die bei der Integration des Systems mit den echten externen Systemen auftreten können:

Integrationstests muss oft Infrastruktur wie Datenbanken oder Web/Application Server oder Messaging Systeme nutzen, um das Zusammenspiel im Test realistisch zu gestalten. Hier hilft **Containerisierung:** Testcontainer sind leichtgewichtige, wegwerfbare Instanzen eines Systems oder einer Komponente, die für Tests verwendet werden können. Diese Container können in verschiedenen Umgebungen ausgeführt werden, z. B. auf lokalen Entwicklungsrechnern, Testservern und Cloud-basierten Umgebungen. Wird eine Datenbank wie PostgreSQL oder ein Eventbus wie Kafka als Docker-Container gestartet, kann damit ein Integrationstests schnell und effizient durchgeführt werden. Werden die Inhalte des Containers nicht auf Volumes persistiert, so findet nach dem Test sogar ein Reset auf den Initialzustand statt und erlaubt Reproduzierbarkeit und Test-Isolation. **"Testcontainers** for Java" ist eine Java-Bibliothek, die JUnit-Tests unterstützt und leichtgewichtige, wegwerfbare Instanzen von gängigen Datenbanken, Selenium-Webbrowsern oder allem anderen, das in einem Docker-Container laufen kann, bereitstellt.

Weitere Alternativen:

- Test-Doubles: Test-Doubles sind Objekte, die das Verhalten von realen Objekten nachahmen, aber nicht die realen Objekte selbst sind. Zu den Test Doubles gehören Stubs, d.h. Objekte, die auf Methodenaufrufe vorgegebene Antworten zurückgeben, und Fakes, d.h. Objekte, die das Verhalten von realen Objekten nachahmen, aber nicht den vollen Funktionsumfang haben.
- Test-Harness: Ein Test-Harness ist eine Sammlung von Software und Testdaten, die zum Testen einer Softwarekomponente oder eines Systems verwendet wird. Mit Test-Harness lassen sich die Wechselwirkungen zwischen verschiedenen Komponenten oder Systemen simulieren, so dass sie isoliert getestet werden können.
- Test-Sandbox: Eine Test-Sandbox ist eine isolierte Umgebung, die zum Testen einer Softwarekomponente oder eines Systems verwendet werden kann. Dabei kann es sich um eine separate physische oder virtuelle Umgebung handeln, die von der Produktionsumgebung isoliert ist und zum Testen des Systems unter verschiedenen Bedingungen verwendet werden kann.

10. Testen mit Java und Spring / Spring Boot

Das wichtigste Test-Framework im Java-Umfeld ist **JUnit.** Es ist in Entwicklungsumgebungen und Build-Tools wie Maven und Gradle integriert. Damit kann man nicht nur Unit-Tests ausführen, sondern es eignet sich auch sehr gut als "Testtreiber" für alle möglichen anderen Tests, auch Integrationstests.

Mit **ArchUnit** kann man Architekturtests ausführen. ArchUnit nutzt ebenfalls JUnit zur Testausführung, damit können Architekturtests mit ArchUnit ganz einfach in die CI/CD-Pipeline integriert werden.

Spring bietet für den Test von Spring- oder Spring-Boot-Anwendung gute Unterstützung an, unter anderem

- Für die Tests wird der (über Annotationen konfigurierbare) Spring-Context hochgefahren. Damit wird in eine Art Smoketest überprüft, ob alle Spring-Beans eindeutig aufgelöst werden können.
- Man kann Spring-Profile nutzen, um die Umgebung, in der die Tests laufen, passend zu konfigurieren
- Man kann steuern, wann und wie oft der Context neu aufgebaut wird, also beispielsweise auch die Datenbank zurückgesetzt wird.
- Ein Datenbank-Schema kann für Tests "on the fly" erzeugt werden.

(J) Security

1. Security: Authentifizierung und Autorisierung

Bei der **Authentifizierung** wird die Identität eines Benutzers überprüft. Dies geschieht in der Regel durch eine Kombination aus Benutzernamen und Passwort, kann aber auch andere Methoden wie biometrische Daten oder Sicherheits-Token umfassen.

Die **Autorisierung** hingegen ist der Prozess der Gewährung oder Verweigerung des Zugangs zu bestimmten Ressourcen oder Aktionen auf der Grundlage der authentifizierten Identität eines Benutzers. Nach der Authentifizierung eines Benutzers wird anhand seiner Identität bestimmt, welche Aktionen er durchführen darf und auf welche Ressourcen er zugreifen kann. Dies kann Dinge wie das Lesen und Schreiben von Daten, den Zugriff auf bestimmte Seiten einer Website oder die Ausführung bestimmter Befehle auf einem Server umfassen.

- **RBAC** steht für Role-Based Access Control **(rollenbasierte Zugriffskontrolle).** Dabei handelt es sich um eine Methode zur Regelung des Zugriffs auf Computer- oder Netzwerkressourcen auf der Grundlage der Rollen der einzelnen Benutzer innerhalb einer Organisation. Bei diesem Ansatz werden den Benutzern Rollen zugewiesen, und jede Rolle ist mit einer Reihe von Berechtigungen oder Zugriffsrechten auf bestimmte Ressourcen verbunden. Benutzer können nur Aktionen ausführen, die durch die mit ihrer Rolle verbundenen Berechtigungen erlaubt sind. Dies erleichtert die Verwaltung des Ressourcenzugriffs, da es nicht notwendig ist, die Berechtigungen für jeden einzelnen Benutzer festzulegen und zu pflegen.
- **CBAC** steht für Context-Based Access Control **(kontextbezogene Zugriffskontrolle).** Dabei handelt es sich um eine Methode zur Regelung des Zugriffs auf Netzressourcen auf der Grundlage des Kontexts der Anfrage, wie z. B. der Quell- und Ziel-IP-Adressen und Portnummern sowie des verwendeten Protokolls. Bei diesem Ansatz werden Zugriffskontrollregeln so konfiguriert, dass bestimmte Arten von Netzwerkverkehr je nach Kontext zugelassen oder abgelehnt werden. CBAC verwendet Stateful Inspection, um den Status von Netzwerkverbindungen zu verfolgen, und kann so detailliertere Entscheidungen darüber treffen, welcher Datenverkehr zugelassen oder abgelehnt werden soll. Dies ermöglicht flexiblere und detailliertere Zugangskontrollrichtlinien als herkömmliche zustandslose Firewalls. CBAC wird in der Regel zusammen mit anderen Sicherheitsmaßnahmen, wie z. B. Intrusion Detection and Prevention Systemen, eingesetzt, um eine umfassende Sicherheitslösung zu schaffen.
- **ACL** steht für **Access Control List.** Es handelt sich um eine Methode zur Regelung des Zugriffs auf Ressourcen wie Dateien, Verzeichnisse oder Netzwerkressourcen auf der Grundlage der Identität des Benutzers oder des Systems, das den Zugriff beantragt. Bei diesem Ansatz wird eine Liste von Berechtigungen mit einer Ressource verknüpft, und jede Berechtigung gibt an, wer auf die Ressource zugreifen darf und welche Aktionen er durchführen darf. Zugriffskontrolllisten können auf Dateisystemebene, auf Netzwerkebene oder auf beiden Ebenen implementiert werden. ACLs können verwendet werden, um den Zugriff auf eine Ressource auf eine bestimmte Gruppe von Benutzern oder Systemen zu beschränken oder um den Zugriff auf eine Ressource für jeden Benutzer oder jedes System zuzulassen, der/das sich am System authentifizieren kann. Sie werden häufig verwendet, um den Zugriff bestimmter Benutzer oder Gruppen auf bestimmte Netzwerkprotokolle und Ports zu beschränken, um das Netzwerk vor unbefugtem Zugriff zu schützen.

2. OAuth und OAuth2

OAuth (Open Authorization) ist ein offener Standard für tokenbasierte Authentifizierung und Autorisierung. Er wird verwendet, um einen sicheren Zugriff auf Ressourcen zu ermöglichen. **OAuth2** ist die zweite Version des OAuth-Protokolls und ist sicherer und flexibler als die erste Version. Es ist genauso wie OAuth ein offener Standard und tokenbasiert.

Das Protokoll ermöglicht es, Anwendungen von Drittanbietern Zugriff auf ihre Ressourcen zu gewähren, ohne das Benutzer ihre Anmeldedaten weitergeben zu müssen. Anstatt der Drittanwendung seinen Benutzernamen und sein Passwort mitzuteilen, wird der Benutzer an den Ressourcenanbieter (z. B. Google oder Facebook) weitergeleitet, um die Zugriffsanfrage zu genehmigen. Der Ressourcenanbieter stellt dann ein Zugriffstoken aus, das die Drittanbieteranwendung für den Zugriff auf die Ressourcen des Nutzers verwenden kann.

OAuth2 definiert mehrere **"Grant-Typen",** die zum Erhalt eines Zugriffstokens verwendet werden können. Die gängigsten Grant-Typen sind:

- Autorisierungscode: Dieser Grant-Typ wird für Web- und mobile Anwendungen verwendet, bei denen der Benutzer den Zugriff gewährt, indem er auf die Website des Ressourcenanbieters weitergeleitet wird und die Anfrage genehmigt.
- Implizit: Dieser Grant-Typ ähnelt dem Grant-Typ Autorisierungscode, ist aber für die Verwendung mit clientseitigen JavaScript-Anwendungen und mobilen Apps gedacht. Er gibt das Zugriffstoken direkt an die Anwendung zurück, ohne dass die Anwendung es gegen einen Autorisierungscode austauschen muss.
- Ressourceneigentümer Passwort Credentials: Dieser Grant-Typ wird verwendet, wenn der Benutzer seine Anmeldedaten direkt an die Anwendung übermittelt. Dieser Grant-Typ ist weniger sicher als die anderen Grant-Typen und sollte nur im Bedarfsfall verwendet werden.
- Client-Anmeldeinformationen: Dieser Grant-Typ wird verwendet, wenn die Anwendung auf ihre eigenen Ressourcen zugreifen muss und keine Autorisierung des Benutzers benötigt.

OAuth2 bietet den Clients auch die Möglichkeit, das Zugriffstoken zu aktualisieren, wenn es abläuft, so dass der Benutzer die Anwendung nicht jedes Mal neu autorisieren muss, wenn das Token abläuft.

3. Kerberos

Kerberos ist ein Netzwerk-Authentifizierungsprotokoll, das für die sichere Authentifizierung von Client/Server-Anwendungen verwendet wird. Kerberos ist in Unternehmensumgebungen weit verbreitet, um eine sichere Authentifizierung für Netzwerkdienste wie E-Mail, Dateifreigabe und Fernzugriff zu ermöglichen.

Kerberos wurde entwickelt, um eine starke Authentifizierung für Client/Server-Anwendungen zu ermöglichen, indem es eine Verschlüsselung mit geheimen Schlüsseln verwendet.

Das Protokoll verwendet Tickets, die verschlüsselt sind und nur vom Empfänger entschlüsselt werden können, um Benutzer und Server zu authentifizieren. Die Tickets werden von einem zentralen Authentifizierungsserver, dem so genannten Key Distribution Center (KDC), ausgestellt und zur Authentifizierung von Benutzern und Servern verwendet, ohne dass diese ihre Anmeldedaten über das Netz übertragen müssen.

4. Typische Authentizierungs- und Autorisierungsprotokolle und -standards, v.a. hinsichtlich Java und Spring-Boot-Anwendungen

Java- und Spring Boot-Anwendungen verwenden in der Regel eine Vielzahl von **Authentifizierungs- und Autorisierungsprotokollen und -standards,** darunter:

- **Spring Security** ist ein leistungsfähiges und hochgradig anpassbares Authentifizierungs- und Autorisierungs-Framework, das zur Absicherung von Java- und Spring Boot-Anwendungen verwendet werden kann. Es bietet Unterstützung für eine breite Palette von Authentifizierungsmethoden, einschließlich Basic Authentication, OAuth2 und JWT, und lässt sich leicht in Spring Boot-Anwendungen integrieren.
- **Basic Authentication:** Hierbei handelt es sich um eine einfache Authentifizierungsmethode, bei der eine base64-kodierte Zeichenkette verwendet wird, die den Benutzernamen und das Passwort enthält und im Request-Header gesendet wird. Sie gilt als nicht sehr sicher, da die kodierten Anmeldedaten leicht entschlüsselt werden können, und sollte nur über HTTPS verwendet werden.
- **OAuth2:** Wie bereits erwähnt, ist OAuth2 ein offener Standard für Token-basierte Authentifizierung und Autorisierung, der es Nutzern ermöglicht, Anwendungen von Drittanbietern Zugriff auf ihre Ressourcen zu gewähren, ohne ihre Anmeldedaten weitergeben zu müssen. Spring Boot bietet Unterstützung für OAuth2 durch das "Spring Security OAuth2"-Projekt, das es Entwicklern ermöglicht, ihren Anwendungen einfach OAuth2-Authentifizierung und -Autorisierung hinzuzufügen.
- **JSON-Web-Token (JWT):** JWT ist ein JSON-basierter, offener Standard zur Erstellung von Zugriffstoken und wird häufig als Alternative zu OAuth2 verwendet. JWT-Tokens sind in sich geschlossen und enthalten alle Informationen, die zur Authentifizierung des Benutzers und zur Autorisierung des Zugriffs auf Ressourcen erforderlich sind. Spring Boot bietet Unterstützung für JWT durch das "Spring Security JWT"-Projekt.
- **LDAP** und **Kerberos** sind offene Standardprotokolle für die zentralisierte Authentifizierung. LDAP (Lightweight Directory Access Protocol) ist ein Verzeichnisdienstprotokoll, das für den Zugriff auf und die Verwaltung von Verzeichnisinformationsdiensten wie Active Directory verwendet wird, und Kerberos ist ein Netzwerkauthentifizierungsprotokoll, das für die sichere Authentifizierung von Client/Server-Anwendungen verwendet wird (siehe oben).

5. Ganzheitliche Betrachtung von Security

Ein Sicherheitskonzept für eine Anwendung erfordert eine ganzheitliche Herangehensweise, die neben Authentifizierung und Autorisierung weitere Aspekte beachtet:

Daten schützen:

- **Datenschutz**: Die Anwendung muss die geltenden rechtlichen oder unternehmerischen Anforderungen und Datenschutzbestimmungen erfüllen, insbesondere wenn personenbezogene Daten verarbeitet werden.
- **Verschlüsselung**: Die Vertraulichkeit und Integrität von Daten während der Übertragung und Speicherung wird durch Verschlüsselung gewährleistet. Dies umfasst die Verwendung von SSL/TLS für die sichere Kommunikation und die Verschlüsselung von sensiblen Daten in Datenbanken / Dateispeichern oder sogar ganzer Datenbanken und Datenspeicher.

Anwendung absichern:

- **Sicherheitspatches und Updates**: Alle Komponenten einer Anwendung sollen auf dem aktuellen Stand gehalten werden, mindestens Sicherheitspatches und Updates regelmäßig einspielen. Dies gilt sowohl für das Betriebssystem als auch für Anwendungssoftware und Frameworks.
- **Überwachung und Protokollierung**: Eine Überwachung und Protokollierung ist notwendig, um verdächtige Aktivitäten zu erkennen und aufzuzeichnen. Dies ermöglicht es, Sicherheitsvorfälle frühzeitig zu erkennen und darauf zu reagieren.
- **Schutz vor Denial-of-Service-Angriffen**: Mechanismen, um solche Angriffe zu erkennen und abzuwehren, sind notwendig. Das sind z.B. Firewalls, Intrusion Detection Systemen (IDS) und Load Balancern. Diese werden in der Infrastruktur implementiert (nicht in der Anwendung selber).
- **Redundanz und Failover**: Bei Ausfällen von Komponenten oder Netzwerken sollen keine Daten verloren gehen und die Anwendung weiter erreichbar sein. Welche Anforderungen hier gelten, muss explizit in den Qualitätsanforderungen festgelegt werden.
- **Security by Design**: Bereits in der Design- und Entwicklungsphase Sicherheitsaspekte berücksichtigen. Dies hilft, Sicherheitslücken von Anfang an zu vermeiden.

Planung:

- Ein **Notfallplan** beschreibt, wie auf Sicherheitsvorfälle reagiert wird. Dies sollte die Kommunikation, die Wiederherstellung von Daten und Systemen sowie die Zusammenarbeit mit Sicherheitsbehörden einschließen.
- **Verwaltung von Schlüsseln und Zertifikaten:** Schlüssel und Zertifikate müssen verwaltet und regelmäßig aktualisiert werden.

6. Security in verteilten Systemen

In einem verteilten System sind andere / zusätzliche Aspekte für Security zu berücksichtigen als in einem (Deployment-)Monolithen:

- **Zugriffskontrolle zwischen den (verteilten) Komponenten des Systems**: Komponenten im verteilten System müssen sich authentifizieren, um auf andere zugreifen zu können, und ihr Zugriffsberechtigungen müssen definiert und implementiert werden. Übliche Sicherheitsmechanismen sind API-Keys, OAuth oder Token-Authentifizierung.
- **Absicherung der Verteilungsinfrastruktur**: Häufig wird für die Verteilung von Daten zusätzliche Infrastruktur eingesetzt, wie bspw. Kafka als Eventbus. Diese Infrastruktur muss ebenfalls abgesichert sein, bspw. kann/muss ein Zugriffsschutz auf Kafka-Topics implementiert werden.
- **Verteilte Identitätsverwaltung:** Die Verwaltung von Benutzeridentitäten und Zugriffskontrollen in verteilten Systemen kann komplex sein, da verschiedene Komponenten möglicherweise unterschiedliche Authentifizierungssysteme verwenden. Gleichzeitig erwarten Benutzer, dass sie sich nicht an jeder Komponente neu oder anders authentifizieren müssen.
- **Kommunikation über unsichere Netzwerke:** In verteilten Systemen erfolgt die Kommunikation zwischen verschiedenen Komponenten häufig über öffentliche Netzwerke wie das Internet. Dies macht die Übertragung von Daten anfälliger für Abhören, Man-in-the-Middle-Angriffe und andere Netzwerkangriffe.
- **Verwaltung von Schlüsseln und Zertifikaten:** In verteilten Systemen spielen diese eine noch größere Rolle für die Sicherheit, sie werden an sehr vielen Stellen eingesetzt. Die Verwaltung gestaltet sich daher aufwändiger.
- Datenschutz: In verteilten Systemen muss besonders bei der Kommunikation auf Datensicherheit geachtet werden. Die Übertragung und Speicherung von Daten über verschiedene geografische Standorte und rechtliche Gerichtsbarkeiten kann zusätzliche rechtliche Anforderungen mit sich bringen.

Security in verteilten Systemen ist aufwändiger und schwieriger als in (Deployment-) Monolithen. Das ist als "Kostenpunkt" bei der Entscheidung für ein verteiltes System unbedingt zu beachten und "einzupreisen".