# ***SYNTHESE ARCHITEKTURMUSTER***

# ***Schichtenarchitektur***

**Definition**

Das Softwaresystem wird in Schichten strukturiert.

Einzelne Aspekte (z.B. Klassen oder Komponenten) eines Softwaresystems werden jeweils einer Schicht zugeordnet.

• Aspekte einer höheren Schicht dürfen ausschließlich Aspekte der gleichen oder einer niedrigeren Schicht verwenden.

O imagine care conține text, captură de ecran, Font, număr

Descriere generată automat

**3-Schichten-Architektur**

**O imagine care conține text, captură de ecran, Font

Descriere generată automat**

**Querschnittliche SchichtO imagine care conține text, captură de ecran, Font, proiectare

Descriere generată automat**

**Strikte vs. Nicht-strikte Schichtenarchitektur**

**O imagine care conține text, captură de ecran, Font, număr

Descriere generată automat**

**Nicht-strikt:** Trade-off zwischen Performance und loser Kopplung notwendig.

Bessere Performance,aber engere Kopplung

**Möglichkeiten der Realisierung**

# Paketstruktur

Eine Möglichkeit, die Schichtenarchitektur zu implementieren, ist durch eine logische Paketstruktur innerhalb eines Projekts:

Paketstruktur: Die Klassen und Module eines Projekts werden in verschiedene Pakete organisiert, die den verschiedenen Schichten entsprechen. Zum Beispiel könnte ein Projekt in Pakete wie presentation, service, domain, und data unterteilt werden, wobei jede dieser Schichten spezifische Aufgaben hat.

### Komponenten

Eine andere Möglichkeit ist die Organisation des Systems in wiederverwendbare, eigenständige Komponenten:

* **Komponenten**: Jede Schicht wird als eine Sammlung von Komponenten realisiert, die klar definierte Schnittstellen haben. Dies ermöglicht eine lose Kopplung und erleichtert das Testen und die Wiederverwendbarkeit.
* **Beispiel**:
  + **UI-Komponenten**: Knöpfe, Formulare, Widgets.
  + **Service-Komponenten**: Authentifizierungsservice, Zahlungsservice.
  + **Datenbank-Komponenten**: Repositories, Datenbankadapter.

### Namenskonventionen

Durch konsistente Namenskonventionen können Schichten klar voneinander abgegrenzt und leicht identifizierbar gemacht werden:

* **Namenskonventionen**: Namen von Klassen, Methoden, und Variablen können so gestaltet werden, dass ihre Zugehörigkeit zu einer bestimmten Schicht offensichtlich wird.
* **Beispiel**:
  + Klassen im Presentation Layer: UserController, ProductView.
  + Klassen im Service Layer: UserService, OrderService.
  + Klassen im Data Layer: UserRepository, ProductDAO.

### Frameworks und Bibliotheken

**Frameworks und Bibliotheken**: Viele moderne Frameworks unterstützen das Konzept der Schichtenarchitektur direkt und bieten Werkzeuge und Strukturen, um diese zu implementieren.

### In Kombination mit anderen Architekturmustern

Schichtenarchitektur kann mit anderen Architekturmustern kombiniert werden, um die Flexibilität und Wartbarkeit eines Systems zu erhöhen:

* **In Kombination mit anderen Architekturmustern**: Patterns wie Microservices, Clean Architecture, oder Hexagonal Architecture können zusammen mit der Schichtenarchitektur verwendet werden.
* **Beispiele**:
  + **Microservices**: Jede Schicht könnte innerhalb eines Microservices isoliert sein, was die Skalierbarkeit und Wartbarkeit erhöht.
  + **Hexagonal Architecture**: Trennt das System in verschiedene "Ports" und "Adapters", um die Abhängigkeiten zwischen den Schichten weiter zu reduzieren.

# ***Deployment-Varianten***

# *Monolithische Bereitstellung und Verteilte Bereitstellung*

|  |  |
| --- | --- |
| O imagine care conține text, captură de ecran, Font, număr  Descriere generată automat | • Alle Schichten werden gemeinsam auf einer Serverinstanz  oder einem Container ausgeführt (On-Premises oder in der Cloud)  • Eignet sich gut für kleinere Anwendungen mit einfacher  Architektur und geringen Anforderungen an Skalierbarkeit  • Debugging und Management der Anwendung oft einfacher  als bei verteilter Bereitstellung |
|  | • Alle oder einzelne Schichten werden auf separaten  Serverinstanzen oder Containern bereitgestellt (OnPremises oder in der Cloud)  • Bietet eine bessere Skalierbarkeit und Flexibilität, da jede Schicht unabhängig voneinander skaliert und verwaltet werden kann.  • Debugging und Verwaltung des Gesamtsystems können schwieriger auf aufwändiger werden. |

# ***Integration zweier Anwendungen auf Ebene der Präsentationsschicht***

O imagine care conține text, captură de ecran, proiectare

Descriere generată automat

• Technische Umsetzung zum Beispiel mittels IFrames (Inline Frames)

• Beispiel aus einer realen Anwendung: LinkedIn

• Bietet die Möglichkeit, Inhalte wie Videos, Präsentationen oder Artikel direkt in LinkedIn-Beiträge einzubetten • Nutzer können auf diese Beiträge direkt auf der Plattform zugreifen, ohne die Plattform zu verlassen

### Example from a Real Application: LinkedIn

**Features:**

* Provides the ability to embed content such as videos, presentations, or articles directly into LinkedIn posts.
* Users can access these posts directly on the platform without leaving LinkedIn.

# ***Integration zweier Anwendungen auf Ebene der Anwendungsschicht***

O imagine care conține text, captură de ecran, Telefon mobil, proiectare

Descriere generată automat

Die Integration zweier Anwendungen auf der Anwendungsschicht kann durch verschiedene Technologien und Ansätze erreicht werden. Hier sind drei gängige Methoden, die oft verwendet werden:

• Web-Services und APIs

• Middleware und Message-Broker

• SDKs (Software Development Kits)

### Web-Services und APIs

**Beschreibung:** Web-Services und APIs ermöglichen die Kommunikation und den Datenaustausch zwischen zwei Anwendungen über das Internet.

Web-Services sind standardisierte Schnittstellen, die auf Protokollen wie HTTP, SOAP oder REST basieren, während APIs allgemeiner sind und auch andere Protokolle nutzen können.

### Middleware und Message-Broker

**Beschreibung:** Middleware und Message-Broker fungieren als Vermittler zwischen zwei Anwendungen und erleichtern die Kommunikation und den Datenaustausch.

Middleware bietet allgemeine Dienste und Funktionen, die spezifische Details der Anwendungen abstrahieren.

Message-Broker wie RabbitMQ oder Apache Kafka ermöglichen die asynchrone Kommunikation durch Nachrichtenwarteschlangen.

**Beispiel:**

* **Middleware**: Ein Unternehmen verwendet eine Middleware-Plattform wie IBM WebSphere oder Oracle Fusion Middleware, um Geschäftsprozesse über verschiedene Anwendungen hinweg zu integrieren.
* **Message-Broker**: Ein Zahlungsabwicklungssystem sendet Zahlungsaufträge an eine Message Queue (z.B. RabbitMQ). Ein separates Abrechnungssystem konsumiert diese Nachrichten, um die Zahlungen zu verarbeiten.

### SDKs (Software Development Kits)

**Beschreibung:** SDKs sind Entwicklungswerkzeuge, die Bibliotheken, Dokumentation und Beispiele enthalten und Entwicklern helfen, Funktionalitäten einer anderen Anwendung oder Plattform zu integrieren. Ein SDK ermöglicht es Entwicklern, spezifische Funktionen einer Anwendung in ihre eigene Anwendung einzubauen.

**Beispiel:**

* **Google Maps SDK**: Eine mobile Anwendung verwendet das Google Maps SDK, um Kartendienste und Geolokalisierungsfunktionen zu integrieren.
* **Payment Gateway SDK**: Eine E-Commerce-Website integriert ein Payment Gateway SDK, um Kreditkartenzahlungen zu verarbeiten.

# ***Integration zweier Anwendungen auf Ebene der Datenzugriffsschicht***

O imagine care conține text, captură de ecran, număr, Font

Descriere generată automat

• Direkte Datenbankverbindung, Datenbank-Replikation, Datenbankschnittstellen und APIs, Datenbankpools und Datenbankservices

# **Einige Kritikpunkte und Fazit**

• Gefahr von technischen Schulden innerhalb von Schichten

• Overhead durch Schichten

Jede zusätzliche Schicht in der Architektur führt zu einem gewissen Overhead. Dies bedeutet zusätzlichen Aufwand für die Datenübertragung zwischen den Schichten, Verwaltung von Schnittstellen und möglicherweise auch Leistungsverlust.

This means additional effort for data transfer between layers, management of interfaces, and potentially performance loss.

• Starre Struktur

Eine zu strikte Einhaltung der Schichtenarchitektur kann zu einer starren Struktur führen, die es schwierig macht, auf Änderungen in den Anforderungen oder Technologien flexibel zu reagieren.

Trotz dieser Kritikpunkte wird die Schichtenarchitektur nach wie vor häufig in der Softwareentwicklung eingesetzt, da sie eine **klare Trennung von Verantwortlichkeiten** und eine **modulare Struktur bietet**, die die **Wartung, Skalierung und Weiterentwicklung** von Anwendungen **erleichtert.**

# ***Pipes & Filters***

• Basiert auf dem Konzept der Datenverarbeitungspipeline

• Die Architektur besteht aus

**Filter:** Verarbeitung von Daten

• Jeder Filter verarbeitet also eingehende Daten und gibt sie über eine Pipe an den nächsten Filter weiter.

• Verarbeiten kann bedeuten

Herausfiltern/Löschen von Daten

Hinzufügen von Daten

Verändern von Daten

**Pipes:** Ermöglichen den Datenfluss zwischen Filter

O imagine care conține Font, alb, tipografie, proiectare

Descriere generată automat

# ***Eigenschaften der Architektur***

• Jeder Filter erfüllt eine spezifische Aufgabe, dadurch Trennung von Verantwortlichkeiten

• Jeder Filter kann unabhängig von anderen Filtern entwickelt werden

• Ermöglicht Wiederverwendbarkeit von Filtern

• Einfache Konfiguration/Rekonfiguration des Gesamtsystems durch

Hinzufügen/Entfernen von Filtern

Änderung der Filter-Reihenfolge

# ***Gemeinsamer Speicher***

Gemeinsamer Speicher kann notwendig werden, falls

Zustandsbehaftete Verarbeitung erforderlich ist

Kommunikation über den Datenstrom hinaus notwendig ist (z.B. Statusinformationen)

Parallelisierung oder Skalierung erforderlich ist

O imagine care conține siglă, clipart, proiectare

Descriere generată automat

# **Beispiele für Anwendungsgebiete**

O imagine care conține text, captură de ecran, Font, document

Descriere generată automat

# ***Pipes und Filters in der Linux-Shell***

O imagine care conține text, Font, chitanță, captură de ecran

Descriere generată automat

# ***Blackboard -Architektur***

**Definition:**

Die Blackboard-Architektur ist hilfreich für Probleme, zu denen keine deterministische Lösungsstrategie bekannt ist.

O imagine care conține text, Font, algebră

Descriere generată automat

The Blackboard Architecture is particularly useful for solving problems for which no deterministic solution strategy is known. Here's a detailed explanation of why this architecture is effective for such problems:

1. **Complexity Beyond Deterministic Methods:**
   * **Explanation:** Some problems are inherently complex and cannot be effectively tackled using straightforward, deterministic methods. These problems may involve numerous variables, interactions, and potential states that make it difficult to predict outcomes using conventional algorithms.
2. **Unstructured Problems:**
   * **Explanation:** Certain problems lack a clear structure, rules, or patterns that can guide the solution process. This unstructured nature means that there is no predefined path or method to follow to arrive at a solution.
   * **Example:** Developing creative solutions or strategies in a business context, where the absence of fixed rules requires a more flexible and iterative approach.
3. **Uncertainty:**
   * **Explanation:** Problems involving uncertainty are those where even known methods cannot guarantee a predictable outcome. The unpredictability may stem from incomplete information, variable conditions, or inherent randomness in the problem domain.
   * **Example:** Predicting stock market trends, where numerous unpredictable factors influence market movements.

O imagine care conține text, Font, chitanță, alb

Descriere generată automat

**Flexible and Dynamic Approach:**

* **Explanation:** The architecture allows the process to adapt based on new and current information. As new data is added to the central blackboard, the approach can be adjusted dynamically to incorporate these updates.

# **Komponenten einer Blackboard-Architektur**

O imagine care conține text, captură de ecran, Font, linie

Descriere generată automat

# **Blackboard**

O imagine care conține text, Font, captură de ecran, alb

Descriere generată automat

 **Central Data Storage:**

* **Function:** The blackboard acts as a central repository where all data, information, and hypotheses related to the problem are stored.

 **Storage for Raw Data, Information, and Hypotheses:**

* **Function:** The blackboard stores all available raw data (e.g., sensor readings, input data), processed information (e.g., analyzed results, intermediate findings), and hypotheses (e.g., potential solutions or theories).

 **Accessible to All Participating Components:**

* **Function:** The blackboard is accessible to all knowledge sources or components involved in the problem-solving process. Each component can read from and write to the blackboard.

 **Central Interface for Collaboration:**

* **Function:** The blackboard serves as the main interface through which all components interact and collaborate.

 **Interfaces for Reading and Writing:**

* **Function:** The blackboard provides standardized interfaces or APIs (Application Programming Interfaces) that allow components to read data from and write data to the blackboard.

# **Wissensquellen**

* Eigenständige Komponenten oder Expertensysteme
* Stellen spezialisiertes Wissen oder Algorithmen für die Lösung eines Problems bereit
* Jede Wissensquelle ist für einen bestimmten Aspekt des Problems zuständig
* Erzeugt, analysiert oder verfeinert Hypothesen
* Interagieren über das Blackboard miteinander
* Signalisieren, unter welchen Bedingungen sie zu einer (Teil-) Lösung beitragen können

# **Controller**

* Überwacht und steuert den Ablauf des Lösungsprozesses
* Entscheidet, welche Wissensquellen aktiviert werden sollen, basierend auf
  + Aktuellen Bedingungen des Problems
  + Informationen auf dem Blackboard
  + Bewährten Lösungsstrategien

O imagine care conține text, captură de ecran, Font, document

Descriere generată automat

O imagine care conține text, diagramă, linie, captură de ecran

Descriere generată automat

# **Typische Anwendungsfälle**

O imagine care conține text, Font, captură de ecran, algebră

Descriere generată automat

# ***Broker -Architektur***

**Definition**  
Wird in verteilten Systemen verwendet, um die Kommunikation und Interaktion zwischen Komponenten zu erleichtern

• Zentrale Komponente: Broker (auch Mittelsmann/Vermittler) Vermittelt Nachrichten zwischen Komponenten

O imagine care conține text, diagramă, captură de ecran, linie

Descriere generată automat

# **Zentrale Konzepte**

Broker empfängt Nachrichten und leitet sie an Empfänger weiter

• Je nach Implementierung kann der Broker Nachrichten

* Transformieren
* Filtern
* Priorisieren

• Das Architekturmuster fördert die **Entkopplung von Komponenten**

# **Vorteile bei der Skalierbarkeit**

Zum Beispiel wird das Hinzufügen von Komponenten vereinfacht

In einer Broker-Architektur muss nur eine Schnittstelle programmiert werden.

Sonst - Im schlechtesten Fall müssten n Schnittstellen implementiert werden! (n = Anzahl der Komponenten des ursprünglichen Systems)

# **Herausforderungen bei der Implementierung des Brokers**

• Skalierbarkeit

• Nachrichtenverarbeitung und Durchsatz

• Sicherheit und Datenschutz

• Kommunikationsprotokolle und Interoperabilität

• Komplexität und Wartbarkeit

# **Frameworks und Technologien**

• Apache Kafka

• RabbitMQ

• Apache ActiveMQ

• Mosquitto (MQTT)

• NATS

# **Message Broker am Beispiel von RabbitMQ**

**Definition**

* Open-Source-Messaging-Broker für die Implementierung von zuverlässigen und skalierbaren Nachrichtenübertragungssystemen
* Wurde erstmals 2007 von der Firma Rabbit Technologies Ltd. entwickelt und später von Vmware übernommen.
* Wird von der RabbitMQ-Community aktiv weiterentwickelt
* Ist Bestandteil vieler Architekturen wie Microservices, Event-Driven Architekturen und Cloud-Anwendungen
* Basiert Advanced Message Queuing Protocol (AMQP) basiert
  + Industriestandard für die Messaging-Kommunikation
* Unterstützt auch Protokolle wie MQTT, STOMP und HTTP

# **Elemente von RabbitMQ**

O imagine care conține text, captură de ecran, Font

Descriere generată automat

# **Exchange Types**

O imagine care conține text, captură de ecran, Font

Descriere generată automat

# **Exchange Type – Fanout Exchange**

**Definition:**

Nachrichten werden an alle Warteschlangen weitergeleitet, die an den Exchange gebunden sind.

Ermöglicht eine einfache Broadcast- oder Multicast-Verteilung von Nachrichten an alle Warteschlangen, die sich an den Exchange angeschlossen haben.

O imagine care conține diagramă, captură de ecran

Descriere generată automat

# **Exchange Type – Direct Exchange**

**Definition:**

Leitet Nachrichten basierend auf einem festgelegten Routing-Schlüssel an eine oder mehrere Warteschlangen weiter.

Der Routing-Schlüssel einer Nachricht wird mit einem spezifischen RoutingSchlüssel einer Warteschlange verglichen, und die Nachricht wird an die Warteschlange weitergeleitet, wenn die Schlüssel übereinstimmen.

O imagine care conține captură de ecran, diagramă, text, linie

Descriere generată automat

# **Exchange Type – Topic Exchange**

**Definition**

Leitet Nachrichten basierend auf einem Muster oder Thema an eine oder mehrere Warteschlangen weiter

• Der Routing-Schlüssel einer Nachricht wird als Thema interpretiert, das aus

mehreren Wörtern oder Segmenten besteht, die durch Punkte getrennt sind

(z.B. "weather.usa.newyork").

• Warteschlangen können mit einem Routing-Muster konfiguriert werden, das dem Thema entspricht, und die Nachricht wird an alle Warteschlangen weitergeleitet, die mit dem entsprechenden Muster übereinstimmen.

O imagine care conține diagramă, captură de ecran, linie, cerc

Descriere generată automat

O imagine care conține text, captură de ecran, proiectare

Descriere generată automat

O imagine care conține text, captură de ecran, diagramă, proiectare

Descriere generată automat

# ***Serviceorientiere Architekturen***

O imagine care conține text, captură de ecran, Font, diagramă

Descriere generată automat

# **Eigenschaften**

* Services sind im Idealfall zustandsfrei(Idempotent)
  + Egal wie oft man sie mit den gleichen Eingabedaten aufruft, sie führen immer zu den gleichen Ergebnissen.
* Serviceschnittstellen besitzen Vertragscharakter und binden Servicekonsumenten an Serviceanbieter (der Serviceimplementierung).
  + Design-by-Contract
* Serviceanbieter sind austauschbar, sofern die Schnittstellenzusagen eingehalten werden.
* Services sind ortsunabhängig und können jederzeit und von jedem Ort aus aktiviert werden (Ortstransparenz).

# ***Model -View -Controller***

O imagine care conține text, diagramă, captură de ecran, Font

Descriere generată automat

# **Model:**

• Enthält die darzustellenden Daten

• Ist von Steuerung und Präsentation unabhängig

• Änderungen an relevanten Daten im Modell werden durch das Entwurfsmuster Observer bekannt gemacht

# **View:**

• Stellt Daten aus dem Modell dar und nimmt Benutzereingaben entgegen

• Meist wird die View über Änderungen im Modell benachrichtigt und holt sich die aktuellen Daten aus dem Model

# **Controller:**

• Verwaltet die Views und verarbeitet Benutzereingaben

• Zu jeder Präsentation existiert ein Modell

• Entscheidet, welche Daten im Modell geändert werden müssen

• Steuert die Abfolge der anzuzeigenden Fenster

# **Vorteile**

* Trennung von Verantwortlichkeiten
* Wiederverwendbarkeit
* Parallele Entwicklung
* Bessere Testbarkeit

# **Nachteile**

 **Komplexität:**

* **Beschreibung:** Die Struktur von MVC kann komplex werden, insbesondere in großen Anwendungen oder wenn nicht klar definiert ist, welche Komponente für welche Aufgaben zuständig ist.
* **Herausforderung:** Die Komplexität kann zu einer erschwerten Wartung und Weiterentwicklung der Anwendung führen

 **Verständnis und Implementierung:**

* **Herausforderung:** Die Implementierung von MVC erfordert von den Entwicklern eine klare Vorstellung davon, wie Daten zwischen den Komponenten ausgetauscht werden und wie die Trennung von Anwendungslogik und Präsentationsschicht erreicht wird.

 **Zusätzlicher Code**

 **Keine Modularisierung nach fachlichen Gesichtspunkten:**

* **Beschreibung:** MVC bietet keine natürliche Modularisierung nach fachlichen Gesichtspunkten. Stattdessen wird die Anwendung nach den technischen Komponenten (Model, View, Controller) strukturiert.

# ***Model -View -ViewModel*** ***O imagine care conține text, captură de ecran, Font, proiectare Descriere generată automat***

• Datenbindung ist Schlüsselkonzept

* Einweg-Bindung (One-Way Binding): Daten fließen nur vom ViewModel zur View
* Zweiweg-Bindung (Two-Way Binding): Daten fließen sowohl vom ViewModel zur View als auch umgekehrt
* Einweg-zu-Quelle Bindung (One-Way to Source Binding): Daten fließen nur von der View zum ViewModel

• Vorteile

* Klare Trennung von Verantwortlichkeiten
* Trennung UI-Design und klassischem Source Code
* Verbesserte Testbarkeit
* Anwendung kann unabhängig von der View getestet werden

# ***Event Sourcing***

**Definition**

Ein Architekturmuster zur **Implementierung der Speicherung von Zustandsänderungen in einem System**

Mehr dazu:

• Nicht der aktuelle Zustand eines Objektes wird gespeichert, sondern alle Änderungen, die an diesem Objekt auftreten (Ereignisse)

• Der aktuelle Zustand wird durch die sequenzielle Wiedergabe der Ereignisse rekonstruiert

# **Grundprinzipien**

* **Ereignisse Als Datenquelle**
* **Event-Store**
* **Zustandsrekonstruktion**
* **Unveränderlichkeit der Daten**
* **Historische Daten und Zeitreisen**

# **Umsetzungsmöglichkeiten eines Event Stores**

**• Relationale Datenbanken**

**• NoSQL-Datenbanken**

**• Spezielle Event-Store Frameworks, wie z.B.:**

**EventStoreDB**

**Riak**

**• Cloud-basierte Speicherlösungen**

**• Eigenentwickelte Lösungen**

# **Nachteile & Herausforderungen**

O imagine care conține text, captură de ecran, Font, carte de vizită

Descriere generată automat

O imagine care conține text, captură de ecran, Font

Descriere generată automat

# **Einsatzfelder**

**Event Sourcing** is a design pattern used in software development where changes to the state of an application are captured as a series of events. These events are stored as a log or journal, providing a chronological record of all actions that have occurred in the system. Here's an explanation of its application areas:

1. **Finance and Banking:**
   * In financial and banking systems, it's crucial to maintain a complete and auditable history of transactions, account changes, and other financial activities. Event sourcing ensures transparency and traceability of these actions.
2. **E-Commerce:**
   * E-commerce platforms often require tracking of orders, inventory changes, customer interactions, and payment transactions. Event sourcing allows for a detailed history of these events, facilitating analysis, auditing, and resolving disputes.
3. **Healthcare:**
   * Healthcare systems benefit from event sourcing to track patient records, medical treatments, diagnoses, and changes in health status. This ensures an accurate and comprehensive history of patient care, which is essential for compliance and decision-making.
4. **Social Networks and Community Platforms:**

* Social networks and community platforms record user interactions, content creation, comments, likes, and other social activities.

# **Beispiel**

**O imagine care conține text, captură de ecran, proiectare, smartphone

Descriere generată automat**

# ***CQRS***

Command Query Responsibility Segregation

|  |  |
| --- | --- |
|  |  |

# **Grundidee**

Verantwortlichkeiten für Lese- und Schreiboperationen in einer Anwendung zu trennen

# **Vorteile**

• Optimierte Daten-Schemata für jede Seite

• Einfachere Abfragen

• Unabhängige Skalierung

# **Herausforderung**

Synchronisation zwischen Schreib und Lesedatenbank

# **Kombination mit Event Sourcing**

CQRS und Event Sourcing werden häufig in Kombination verwendet

O imagine care conține text, Font, captură de ecran, algebră

Descriere generată automat

the views can be rebuilt or updated by reprocessing or "replaying" all the events from the event store.

# **Gründe für den Einsatz von CQRS laut Microsoft**

O imagine care conține text, Font, captură de ecran, alb

Descriere generată automat

# **Das Architekturmuster wird (von Microsoft) explizit nicht empfohlen, wenn**

• Die Domäne und die Geschäftsregeln einfach sind

• Wenn einfache CRUD-Operationen eine gute Lösung darstellen

# ***Hexagonale Architektur***

O imagine care conține text, diagramă, captură de ecran, Plan

Descriere generată automat

Die Trennung der Kernlogik der Anwendung von externen Systemen und Technologien (wie Datenbanken, Benutzeroberflächen und anderen externen Diensten)

Durch klare Abgrenzungen können Tests für die Kernlogik der Anwendung ohne die Komplexität externer Systeme geschrieben werden

Die Kernlogik der Anwendung ist nicht von externen Technologien abhängig, was es einfacher macht, diese Technologien zu wechseln oder zu aktualisieren, ohne die Kernlogik zu beeinflussen.

Modernisierung der Infrastruktur soll ohne Anpassungen an der Geschäftslogik möglich sein

### Schlüsselkomponenten der hexagonalen Architektur

1. **Kern-Domänenlogik (Innerhalb des Hexagons):**
   * Enthält die Kern-Geschäftslogik und Domänenregeln.
   * Sollte frei von Abhängigkeiten zu externen Systemen oder Frameworks sein.
2. **Ports (Schnittstellen):**
   * Definieren die Einstiegspunkte (Input-Ports) und Ausstiegspunkte (Output-Ports) für die Kern-Domänenlogik.
   * Input-Ports behandeln Anfragen an die Kernlogik, wie Service-Schnittstellen oder Anwendungsdienste.
   * Output-Ports definieren, wie die Kernlogik mit externen Systemen interagiert, wie z. B. Repositories oder externe Dienste.
3. **Adapter (Außerhalb des Hexagons):**
   * Implementieren die von den Ports definierten Schnittstellen, um mit externen Systemen zu interagieren.
   * Beispiele sind Datenbank-Adapter, Benutzeroberflächen-Adapter und externe Dienst-Adapter.
   * Adapter übersetzen zwischen den Protokollen des externen Systems und den Schnittstellen der Kernlogik.

O imagine care conține text, diagramă, captură de ecran, Plan

Descriere generată automat

O imagine care conține text, diagramă, captură de ecran, Plan

Descriere generată automat

# ***Microservices***

**Definition**

Architekturmuster bei dem komplexe Anwendungssoftware aus unabhängigen Prozessen komponiert wird, die untereinander mit sprachunabhängigen Programmierschnittstellen kommunizieren.

Die Dienste sind weitgehend entkoppelt und erledigen eine kleine Aufgabe. So ermöglichen sie einen modularen Aufbau von Anwendungssoftware.

These services are built around business capabilities and independently deployable

may be written in different programming languages and use different data storage technologies.

 **Unabhängige Prozesse:**

* Jeder Microservice ist ein eigenständiger Prozess, der unabhängig von anderen Microservices ausgeführt wird. Dadurch können sie unabhängig entwickelt, bereitgestellt und skaliert werden.

 **Sprachunabhängige Kommunikation:**

* Microservices kommunizieren über APIs, die keine bestimmte Programmiersprache voraussetzen. Dies ermöglicht die Verwendung verschiedener Technologien und Programmiersprachen für unterschiedliche Dienste, was eine größere Flexibilität bietet.

 **Weitgehend Entkoppelt:**

* Microservices sind lose gekoppelt, was bedeutet, dass Änderungen an einem Dienst nur minimale Auswirkungen auf andere Dienste haben. Dies erleichtert die Wartung und Weiterentwicklung der Anwendung.

 **Modularität:**

* Durch die Aufteilung der Anwendung in kleinere, spezialisierte Dienste kann die Software leichter gewartet, aktualisiert und erweitert werden. Neue Funktionen können hinzugefügt werden, ohne das gesamte System zu beeinflussen.

 **Kleine, spezialisierte Aufgaben:**

* Jeder Microservice konzentriert sich auf eine kleine, klar definierte Aufgabe. Diese Spezialisierung verbessert die Codequalität und erleichtert die Fehlersuche und Optimierung.

O imagine care conține text, captură de ecran, Font

Descriere generată automat

O imagine care conține text, captură de ecran, Font, diagramă

Descriere generată automat

O imagine care conține text, captură de ecran, Font, algebră

Descriere generată automat

# **Kritik an Microservice-Architekturen**

O imagine care conține text, captură de ecran, Font

Descriere generată automat

# **Monolith -> Modulith -> Microservices**

O imagine care conține schiță, diagramă, Plan, artă

Descriere generată automat

# **Modulith vs. Microservice**

O imagine care conține text, captură de ecran, Font, document

Descriere generată automat

# ***Cloud -Servicemodelle***

# **On-Premises (keine Cloud)**

O imagine care conține text, captură de ecran, Font, număr

Descriere generată automat

# **Infrastructure as a Service (IaaS)**

O imagine care conține text, captură de ecran, Font, proiectare

Descriere generată automat

# **Platform as a Service (PaaS)**

O imagine care conține text, captură de ecran, Font, număr

Descriere generată automat

### Platform as a Service (PaaS)

**Definition:** Platform as a Service (PaaS) is a cloud computing model that provides a platform allowing customers to develop, run, and manage applications without dealing with the complexity of building and maintaining the underlying infrastructure. PaaS delivers a framework for developers to create and deploy applications quickly and efficiently.

### Key Characteristics of PaaS

1. **Development Framework:**
   * Provides tools and libraries to support application development. This can include runtime environments, development tools, and pre-built components.
2. **Managed Infrastructure:**
   * The underlying infrastructure (servers, storage, networking) is managed by the PaaS provider, allowing developers to focus on writing code.
3. **Middleware:**
   * Offers middleware services that support application development and deployment, such as databases, messaging systems, and identity management.
4. **Scalability:**
   * Automatically scales the resources allocated to applications based on demand, ensuring performance and availability.
5. **Integrated Development Environment (IDE):**
   * Often includes an IDE or integrates with popular IDEs, providing a seamless development experience.
6. **Automated Deployment:**
   * Supports continuous integration and continuous deployment (CI/CD) pipelines, allowing for automated testing, deployment, and updates of applications.

**Middleware: Definition and Importance**

**Definition:** Middleware is software that sits between the operating system and the applications on each side of a distributed computing system. It provides common services and capabilities to applications outside of what's offered by the operating system. Middleware enables communication and data management for distributed applications.

**Key Characteristics of Middleware**

1. **Intermediary Role:**
   * Acts as an intermediary layer that facilitates communication and data exchange between different applications and services.
2. **Abstraction:**
   * Provides a layer of abstraction that simplifies the complexity of the underlying system and application interactions.
3. **Integration:**
   * Enables integration of diverse and distributed systems, allowing them to work together seamlessly.

# **Software as a Service (SaaS)**

O imagine care conține text, captură de ecran, Font, număr

Descriere generată automat

**Software as a Service (SaaS)**

**Definition:** Software as a Service (SaaS) is a cloud computing service model that delivers software applications over the internet. Users can access these applications through a web browser without needing to install or maintain the software on their local devices. The SaaS provider manages all the infrastructure, middleware, application software, and security.

**Key Characteristics of SaaS**

1. **Web-based Access:**
   * Applications are accessible via a web browser, which means they can be used from any device with internet access.
2. **Managed by Provider:**
   * The SaaS provider handles all aspects of the application, including hosting, maintenance, updates, and security.
3. **Subscription-based Pricing:**
   * Users typically pay a subscription fee to access the software, which can be based on usage or a flat rate.
4. **Scalability:**
   * SaaS solutions can easily scale to accommodate more users or increased data storage needs without significant changes to the infrastructure.
5. **Automatic Updates:**
   * Updates and new features are automatically deployed by the provider, ensuring users always have access to the latest version of the software.

O imagine care conține text, captură de ecran, Font, număr

Descriere generată automat

# *Hybride Cloud-Computing-Ansätze kombinieren die Nutzung von privaten (on-premises) und öffentlichen Cloud-Ressourcen.*

# ***Serverless Computing***

O imagine care conține text, captură de ecran, Font

Descriere generată automat

O imagine care conține text, captură de ecran, Font, algebră

Descriere generată automat

Serverless computing, also known as Function as a Service (FaaS), is a cloud computing model where cloud providers manage the infrastructure required to run and scale applications, allowing developers to focus solely on writing and deploying code without worrying about server management. Here's an explanation:

1. **No Server Management:**
   * In serverless computing, developers do not need to provision or manage servers. The cloud provider dynamically manages the allocation of resources based on the application's needs. Developers simply upload their code, and the provider handles the rest, including scaling, maintenance, and availability.
2. **Event-Driven Architecture:**
   * Serverless applications are typically event-driven, meaning they respond to triggers or events such as HTTP requests, database changes, file uploads, or scheduled tasks. Developers define functions that are executed in response to these events, allowing for efficient and scalable processing.
3. **Pay-Per-Use Pricing:**
   * Serverless platforms often employ a pay-per-use pricing model, where users are charged based on the number of invocations or the compute resources consumed by their functions. This model allows for cost-effective scalability, as users only pay for the resources they actually use, with no charges for idle time.
4. **Automatic Scaling:**
   * Serverless platforms automatically scale the underlying infrastructure to accommodate changes in workload demand. Functions are scaled out horizontally in response to increased traffic and scaled in when demand decreases, ensuring optimal performance and cost efficiency.
5. **Stateless Functions:**
   * Serverless functions are typically stateless, meaning they do not maintain any persistent state between invocations. Any required state must be managed externally, such as in a database or storage service. This statelessness facilitates scalability and fault tolerance.

 **Kaltstart (Cold Start):**

* **Explanation:** Cold start refers to the delay experienced when a serverless function is invoked for the first time or after a period of inactivity. During a cold start, the cloud provider needs to allocate resources and initialize the execution environment, which can result in increased latency.
* **Impact:** Cold starts can introduce latency into applications, particularly for functions that are invoked infrequently or have unpredictable usage patterns. Users may experience delays in response times during cold start periods.

 **Latenz (Latency):**

* **Explanation:** Latency refers to the delay between when a request is made to a serverless function and when the response is received. While serverless platforms are designed for scalability, the distributed nature of cloud environments and the overhead of function invocation can introduce latency.
* **Impact:** High latency can degrade user experience, especially for latency-sensitive applications such as real-time communication or interactive web applications. Minimizing latency is crucial for ensuring responsive and efficient application performance.

 **State-Management (State Management):**

* **Explanation:** Serverless functions are typically stateless, meaning they do not maintain persistent state between invocations. Managing state across multiple function invocations or coordinating stateful workflows can be challenging in a serverless environment.
* **Impact:** Lack of built-in state management capabilities can complicate the development of certain applications, particularly those that require maintaining session state, managing transactions, or coordinating distributed transactions across multiple functions.

 **Vendor-Lock-in (Vendor Lock-in):**

* **Explanation:** Vendor lock-in occurs when an organization becomes heavily dependent on a specific cloud provider's serverless platform, making it difficult to switch to an alternative provider or migrate to an on-premises environment. Serverless offerings often include proprietary features or integrations that are not easily portable to other platforms.
* **Impact:** Vendor lock-in can limit flexibility and hinder interoperability between different cloud services. Organizations may face challenges in negotiating pricing, managing service-level agreements (SLAs), or adapting to changes in the provider's offerings.

# ***Deploymentmuster***

**Definition**

Bewährtes Vorgehen, um Software von der Entwicklungsumgebung in die Produktionsumgebung zu übertragen und dort zu installieren oder zu aktivieren.

Es können verschiedene Aspekte im Bereitstellungsprozess abgedeckt werden

1. **Infrastruktur-Provisionierung:**
   * die Bereitstellung und Konfiguration der benötigten Infrastrukturkomponenten wie Server, Datenbanken, Netzwerke und Speicher
2. **Bereitstellungsstrategien:**
   * Deployment strategies define how the software is deployed and made available to end-users.
3. **Skalierbarkeit und Verfügbarkeit**
4. **Versionierung und Rollback:**
   * Die Versionierung beinhaltet das Verfolgen verschiedener Versionen der Software während des Bereitstellungszyklus. (Versioning involves tracking different versions of the software throughout the deployment lifecycle.)
   * Rollback-Mechanismen ermöglichen es, zu einer früheren Version der Anwendung zurückzukehren, falls während des Bereitstellungsprozesses Probleme oder Fehler auftreten.
5. **Überwachung und Logging:**
   * Deployment patterns include mechanisms for monitoring the deployed application's performance, health, and security in real-time.
   * Logging erfasst und analysiert System- und Anwendungsereignisse, um die Fehlersuche, Überprüfung und Einhaltung von Vorschriften zu erleichtern.

**Single-Server Deployment**

O imagine care conține text, captură de ecran, proiectare

Descriere generată automat

**Multi -Server Deployment**

**O imagine care conține text, captură de ecran, proiectare

Descriere generată automat**