

# Architektur verteilter Anwendungen

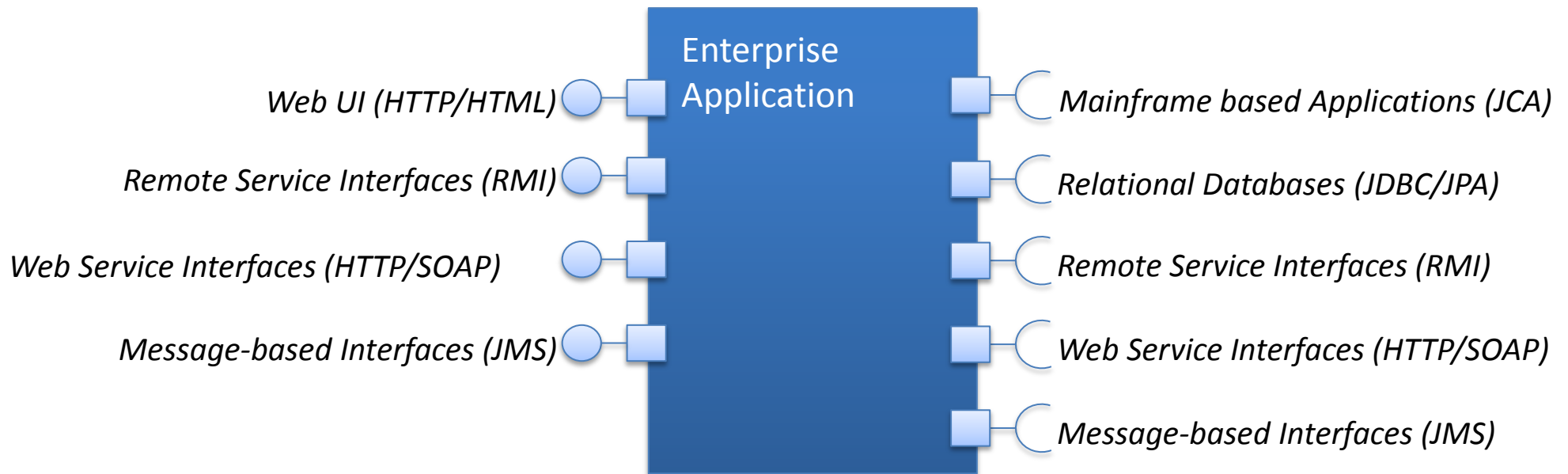
FWP Aktuelle Technologien zur  
Entwicklung verteilter Java-  
Anwendungen

Wer braucht schon einen Architekten?

# **ARCHITEKTUR VERTEILTER ANWENDUNGEN**

# Niemand ist eine Insel

- Applikationen benötigen anderer Systeme
- Applikationen unterstützen andere Systeme



*Wie wollen Sie diese Komplexität ohne Architektur bewältigen?*

# Schichtenmodell als Basis



# Drei prinzipielle Schichten

## Präsentation (*Presentation\**)

- Interaktion zwischen Anwendung und Benutzer
- Anzeige und Bearbeitung von Informationen

## Geschäftslogik (*Business, Domain\**)

- Kern der Anwendungen bestehend aus Diensten in einer Serviceschicht (*Service Layer*) und Domänenmodell (*Domain Model*)

## Integration (*Integration, Data Source\**)

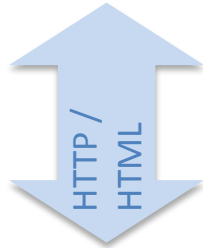
- Integration externer Ressourcen
- Transformation externes Domänenmodell / internes Domänenmodell

# Merkmale der Schichten

- Wohldefinierte Schnittstellen zwischen den Schichten
- Abhängigkeiten zwischen Schichten nur in eine Richtung
- Getrennte Verantwortlichkeiten
- Lose Kopplung / Hohe Kohäsion
- Verteilung der Schichten auf verschiedene Lokationen möglich

# Präsentationsschicht

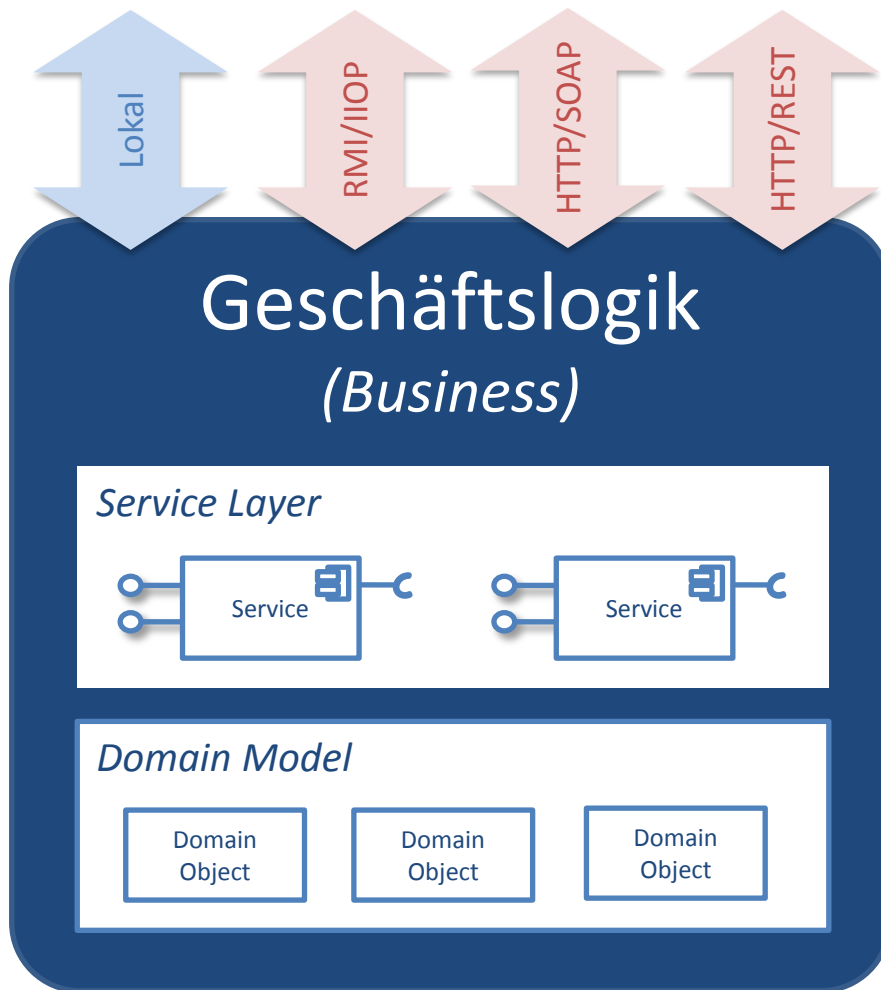
Benutzer



Präsentation  
*(Presentation)*

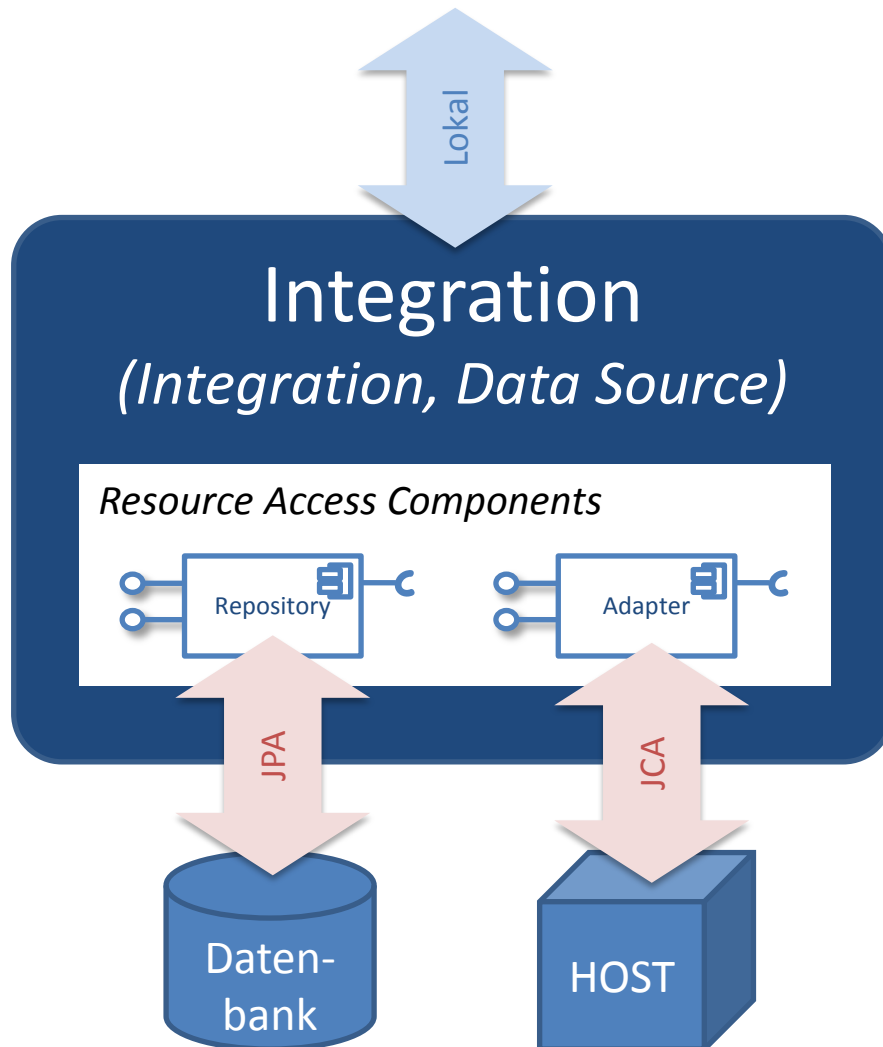
- Benutzeroberfläche für menschliche Benutzer
- Beinhaltet nur Präsentationslogik
- Höchste Änderungshäufigkeit innerhalb der Anwendung
- Rich Client oder Thin Client

# Geschäftslogikschicht



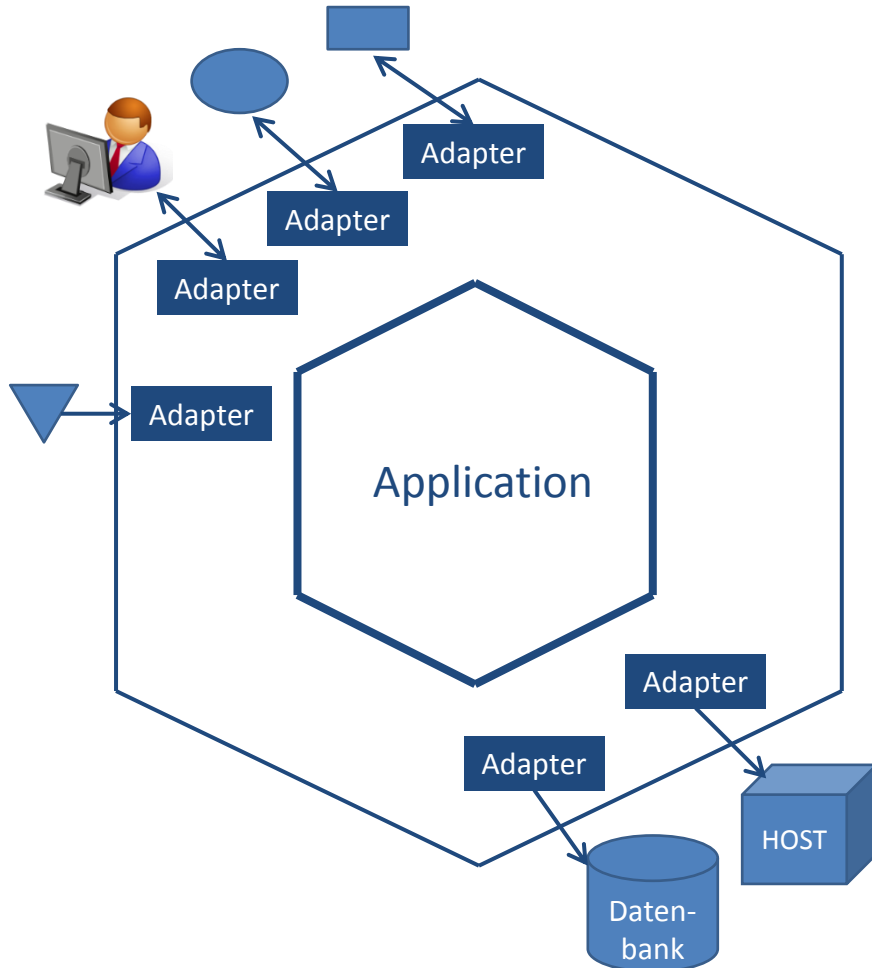
- Kern der Anwendung
- Bereitstellung von Services
  - ⊙ **Lokal**: Präsentationsschicht der eigenen Anwendung
  - ⊙ **Entfernt**: Andere Anwendungen
- Services operieren auf einheitlichem Domänenmodell
- Nutzt Integrationsschicht für Integration von externen Ressourcen

# Integrationsschicht



- Integriert externe Ressourcen
- Bildet externes Domänenmodell auf internes Domänenmodell ab
- Kapselt Information über konkrete Integration
  - Welche Datenbank?
  - Welches Kommunikationsprotokoll?
  - Welches externe System?

# Ausblick: Hexagonale Architektur

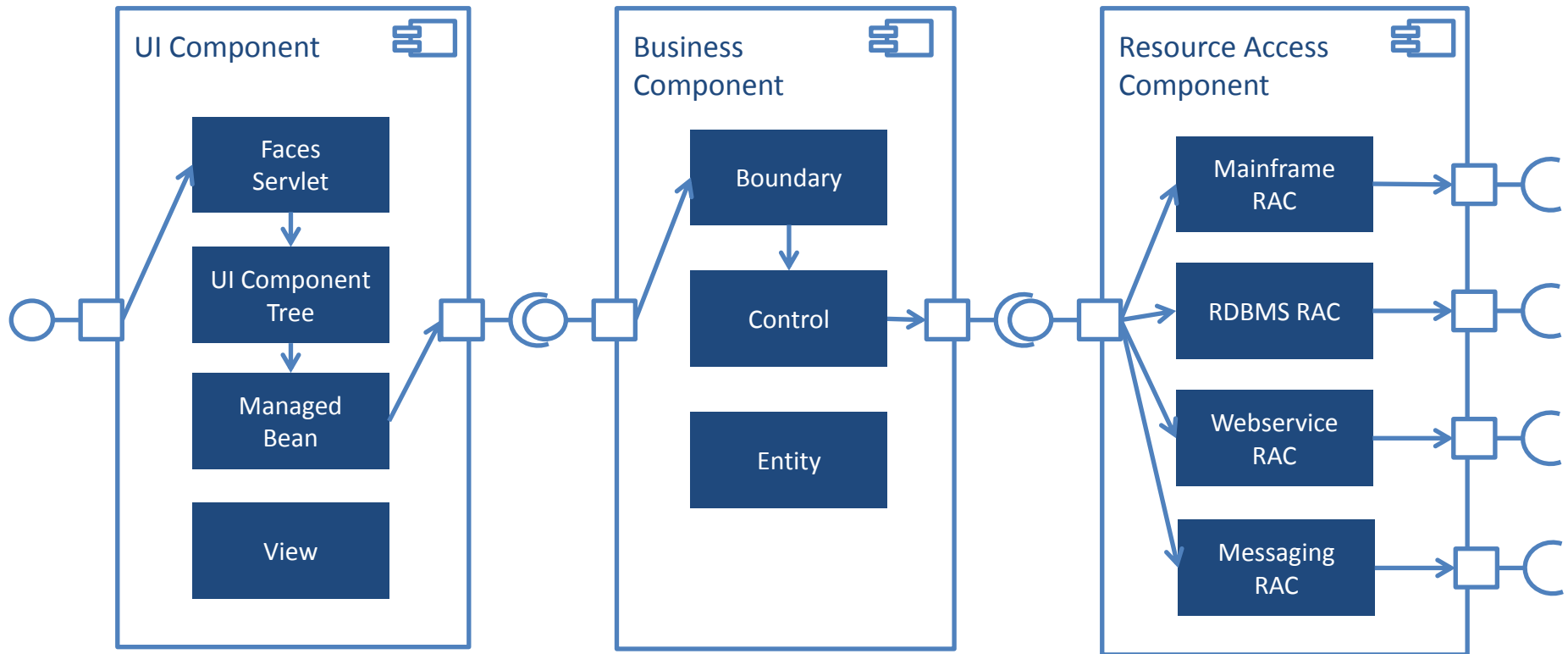


- Auch: Ports & Adapter
- Außenwelt kommuniziert mit Applikation über Ports
- Applikation kommuniziert mit Außenwelt über Ports
- Technologiespezifische Adapter hinter Ports übersetzen Kommunikation
- Applikation kennt keine Details der Außenwelt
- Außenwelt ist simulierbar

# Aufgebaut aus Komponenten



# Einheitliches Komponentenmodell



# Motivation und Umsetzung

- Ermöglicht Entkopplung, Kapselung und klare Trennung der Verantwortlichkeiten
- Erleichtert die Erstellung service-orientierter Applikationen
- Leicht umzusetzen
  - ⊙ 1 Implementierungsklasse pro Komponente
  - ⊙ 1 Interface pro Komponente (empfehlenswert)
  - ⊙ Gruppierung in Module/Packages gemäß Konvention

# Designmodelle

- Entscheidung für Designmodell zu Projektbeginn
- Wesentliche Modelle:
  - ⊙ Serviceorientiert (Service Oriented Architecture SOA)
  - ⊙ Domänengetrieben (Domain Driven Design DDD)
- Modelle können kombiniert werden, aber ein Modell sollte Hauptmodell sein

# Gegenüberstellung SOA - DDD

## Service Oriented Architecture

- Services enthalten alle Logik
- Domänenobjekte sind leere Datenträger
- Optimierung für Verteilung bestimmt internes Design
- Fördert Anti-Pattern **Anemic Domain Model** \*
- Widerspricht objekt-orientiertem Design

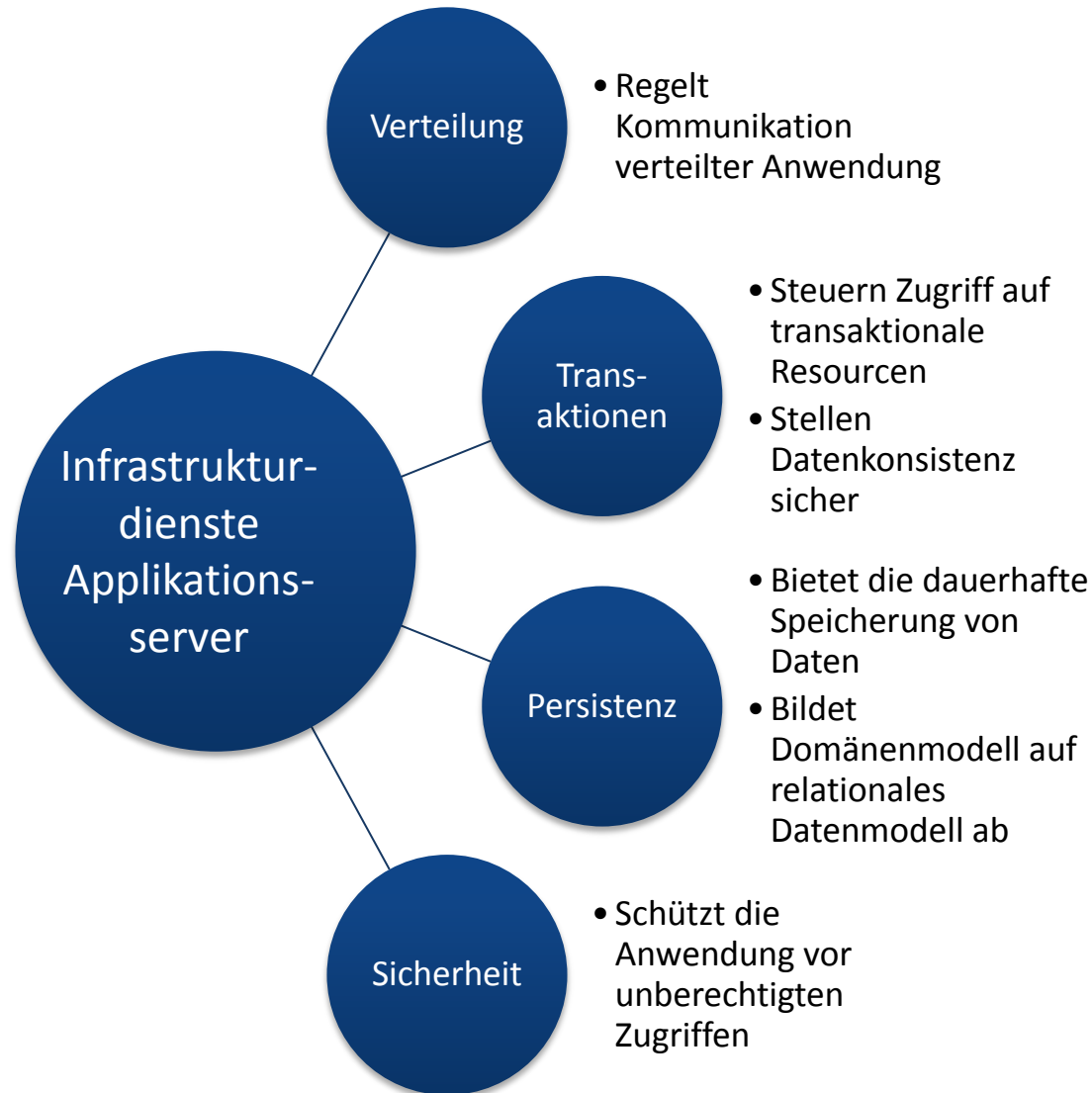
## Domain Driven Design

- Domänenobjekte enthalten Daten und Logik
- Services enthalten nur Logik zur Orchestrierung von Domänenobjekten
- Verteilter Zugriff über Adapter
- Entspricht objekt-orientiertem Design

\* Martin Fowler „AnemicDomainModel“ <http://www.martinfowler.com/bliki/AnemicDomainModel.html>

# **ALLGEMEINE INFRASTRUKTUR- DIENSTE**

# Infrastrukturdienste



# Verteilung

The image features a complex network of interconnected nodes and edges. The nodes are represented as dark, glossy spheres, and the edges are thin, glowing white lines. The network is dense and appears to be a random or semi-random graph. The overall aesthetic is high-tech and digital, with a dark, almost black background that makes the white lines and dark nodes stand out.

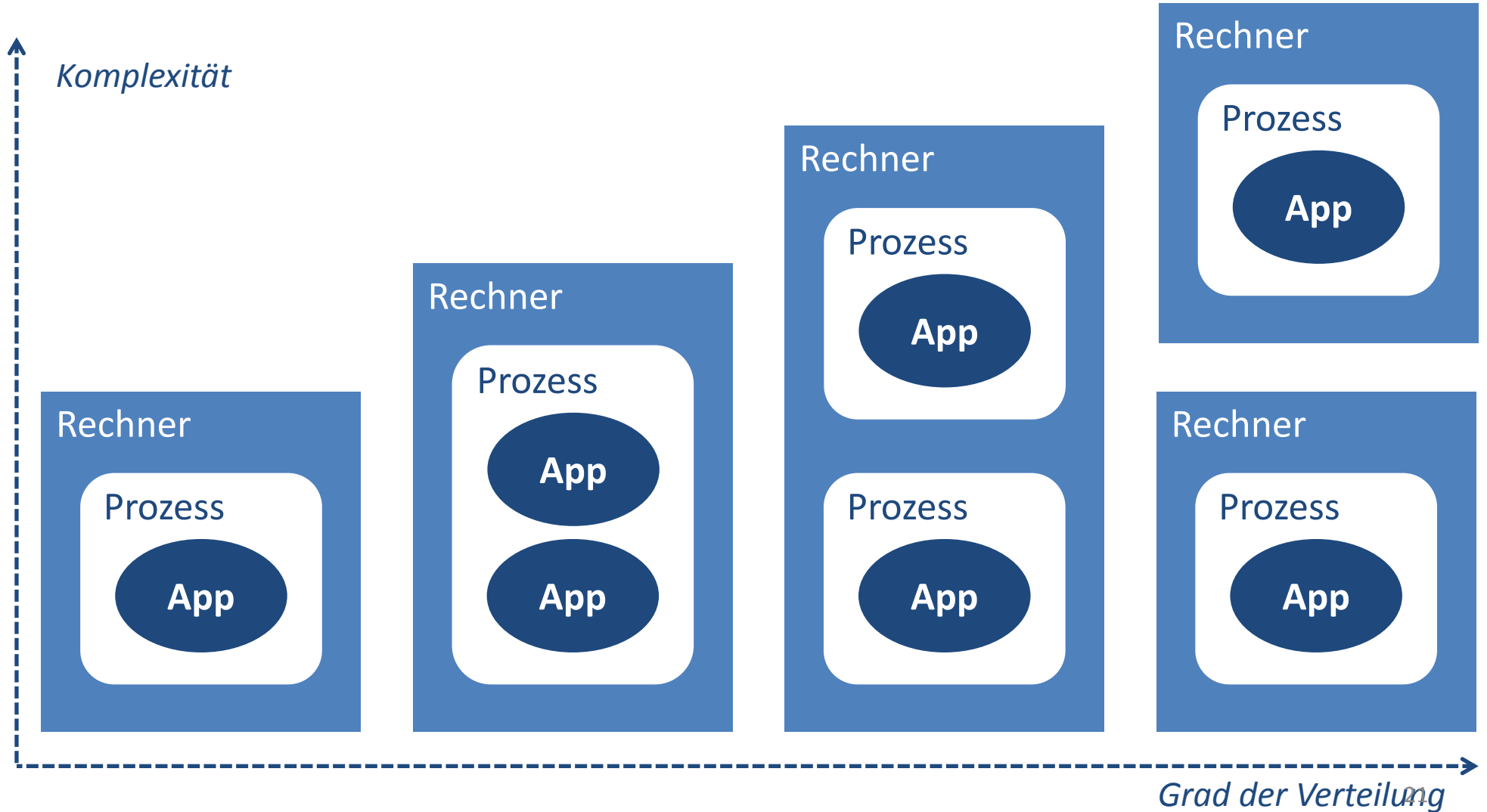
# Erstes Grundgesetz

***“Don’t distribute your objects!”***

# Irrtümer der verteilten Verarbeitung

- Das Netzwerk ist zuverlässig
- Latenz[zeit] ist 0
- Bandbreite ist unendlich
- Das Netzwerk ist sicher
- [Netzwerk-]Topologie ändert sich nicht
- Es gibt nur einen Administrator
- Transportkosten sind 0
- Das Netzwerk ist homogen

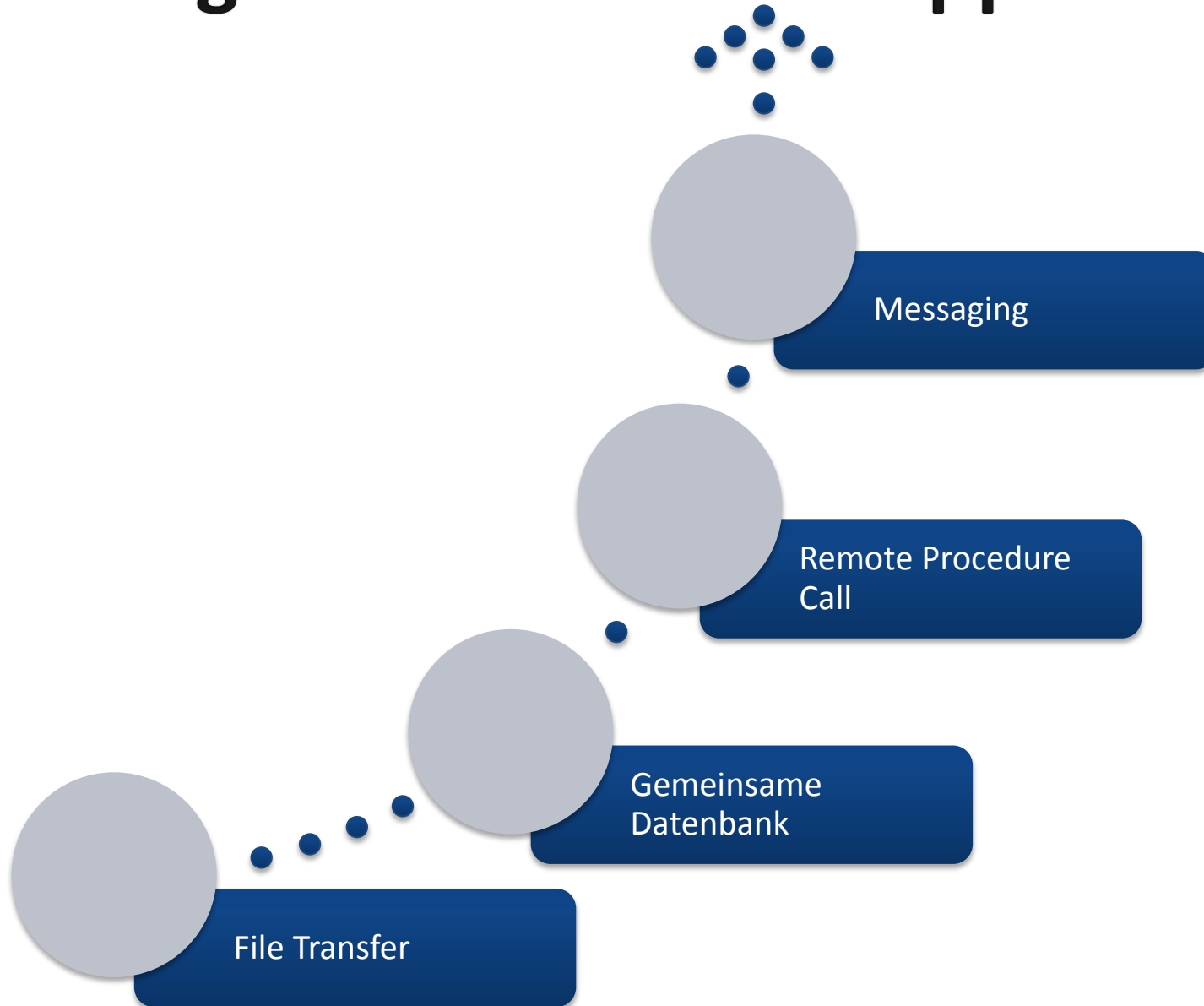
# Verteilung erhöht Komplexität



# Gründe für Verteilung

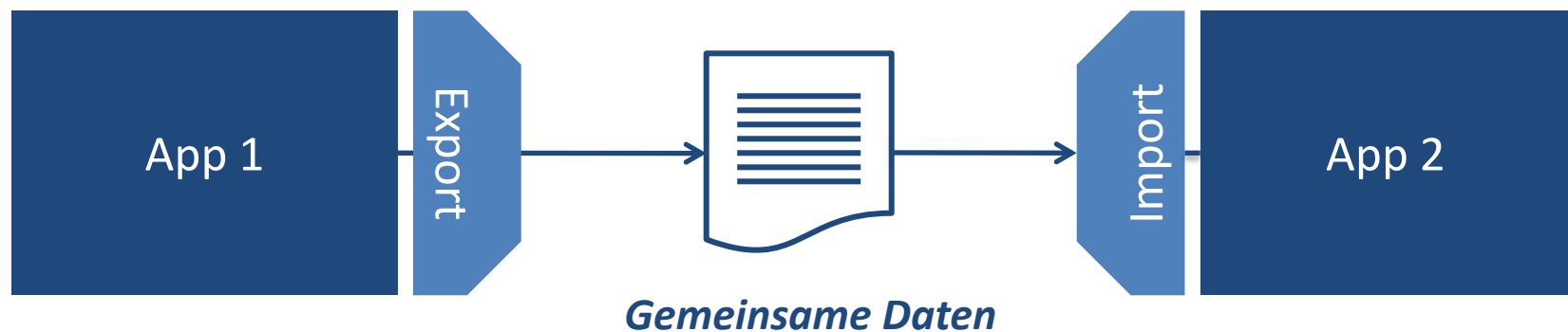
- Skalierbarkeit
- Ausfallsicherheit
- Technologiegrenzen
- Unterschiedliche Standorte
- Bereitstellung von Diensten für externe Consumer
- Komposition eigener Dienste aus Diensten externer Provider

# Integration verteilter Applikationen



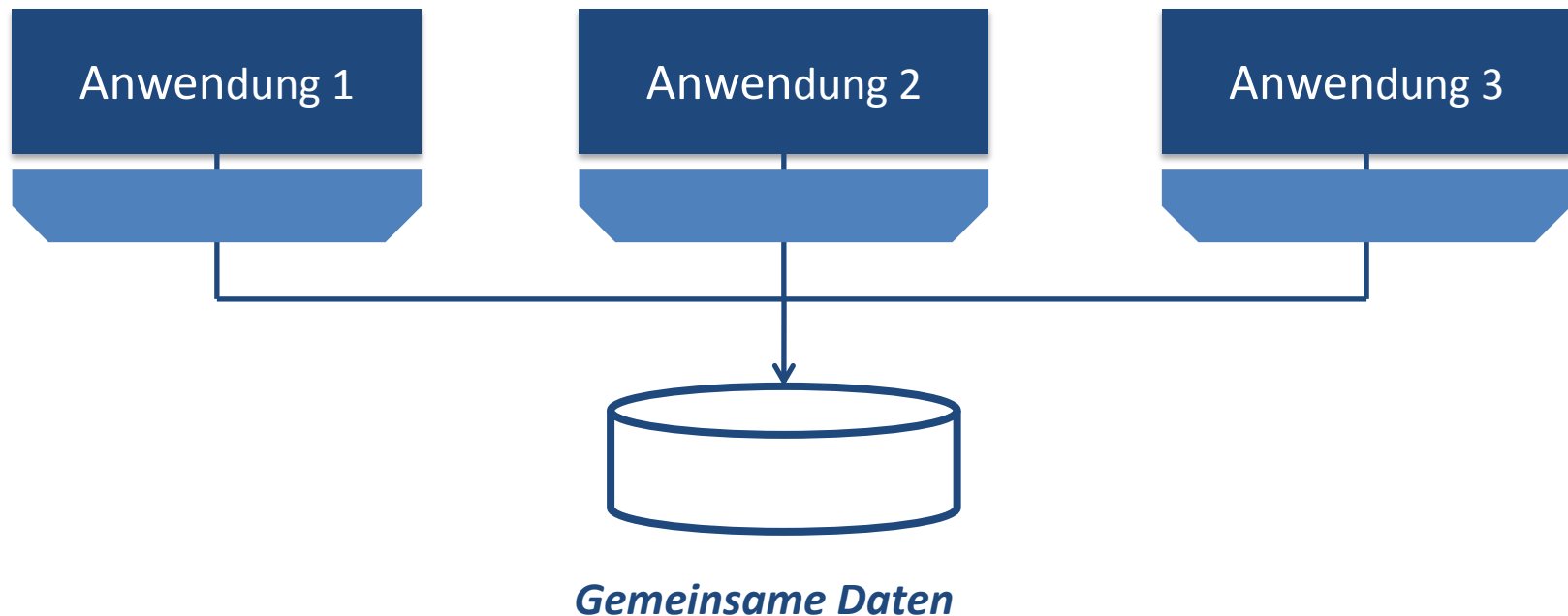
# File Transfer

- Anwendungen tauschen Informationen über Dateien aus
- Anforderungen bestimmen Häufigkeit des Austausches



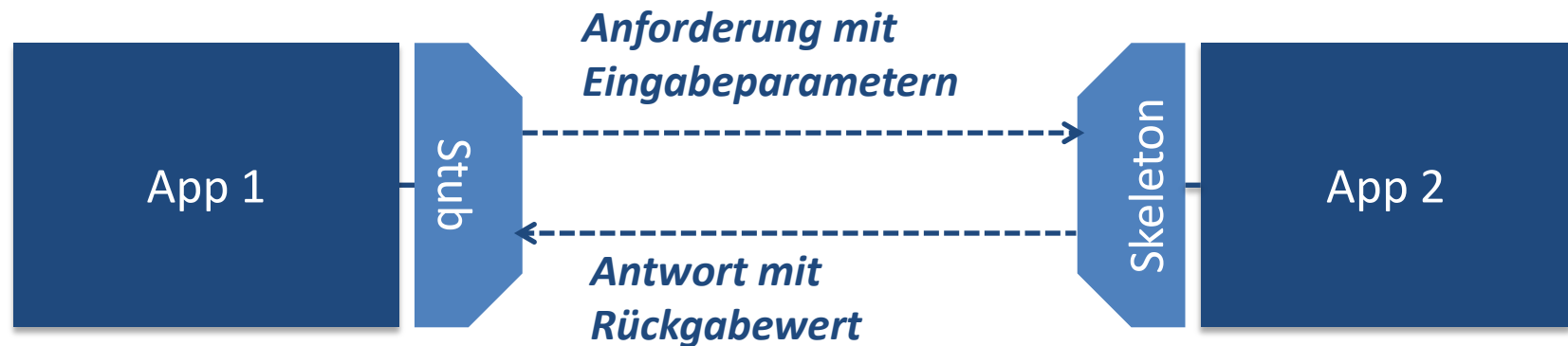
# Gemeinsame Datenbank

- Anwendungen tauschen Informationen über gemeinsame Tabellen aus
- Datenbank-Schema bestimmt Datenformat



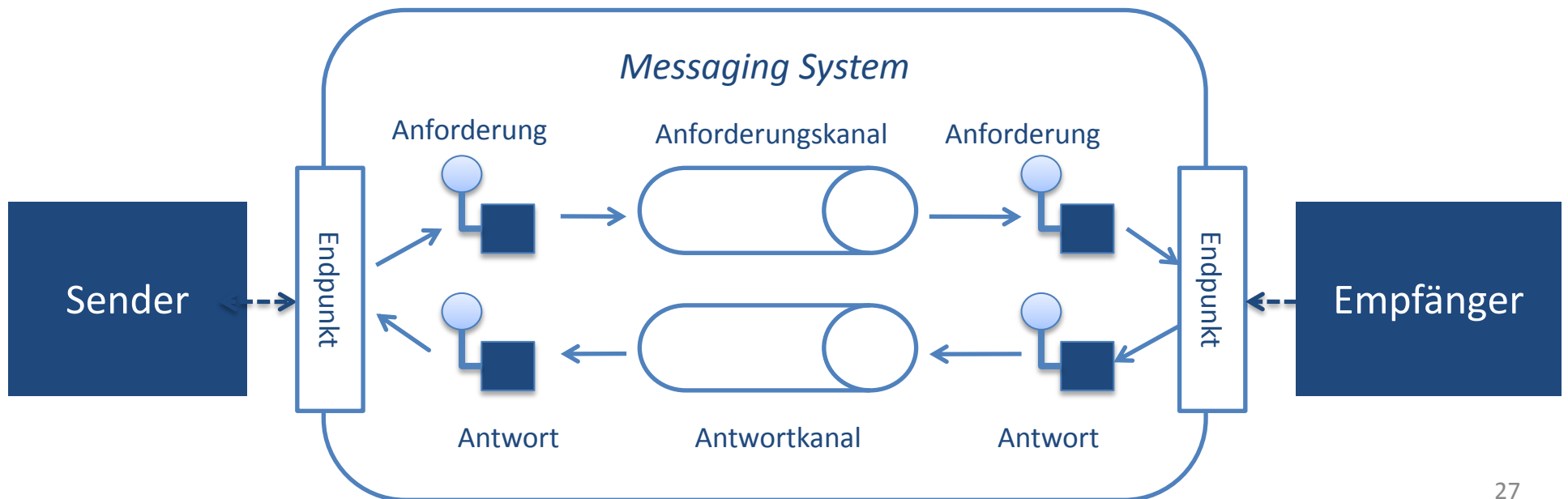
# Remote Procedure Call

- Anwendungen kapseln ihre Daten und stellen Interfaces für den Zugriff zur Verfügung
- Austausch der Informationen erfolgt über synchronen entfernten Methodenaufruf



# Messaging

- Häufiger, unmittelbarer, zuverlässiger und asynchroner Austausch von Nachrichten
- Flexibelste Integration, erfordert aber Umdenken



# Transaktionen



# ACID

## Atomicity

- Eine Transaktion wird entweder ganz oder gar nicht ausgeführt

## Consistency

- Eine Transaktion überführt eine transaktionale Ressource von einem bestehenden konsistenten Zustand in einen neuen konsistenten Zustand

## Isolation

- Das Ergebnis einer Transaktion darf für andere Transaktionen solange nicht sichtbar sein, bis diese Transaktion erfolgreich abgeschlossen worden ist.

## Durability

- Das Ergebnis einer erfolgreich abgeschlossenen Transaktion muss dauerhaft erhalten bleiben und Abstürze jeder Art überstehen.

# Isolationsgrad bestimmt Durchsatz

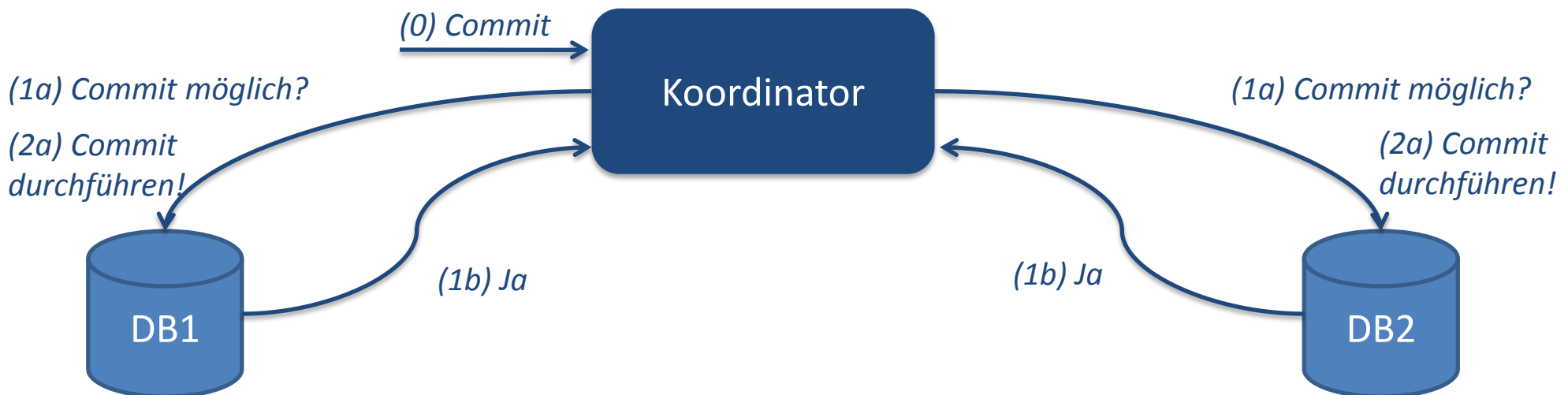


# Ablauf einer Transaktion

- **Begin**
  - ⊙ Transaktion beginnt
- **Commit**
  - ⊙ Transaktion endet erfolgreich
  - ⊙ Beteiligte transaktionale Ressourcen werden verändert
- **Rollback**
  - ⊙ Transaktion bricht nach Fehler ab
  - ⊙ Beteiligte transaktionale Ressourcen bleiben unverändert

# 2 Phase Commit (2PC)

- Transaktion mit mehreren beteiligten transaktionalen Ressourcen in zwei Phasen
  - ⦿ Phase 1: Commit vorbereiten
  - ⦿ Phase 2: Commit ausführen



# Propagation von Transaktionen

- Bei Aufrufen entfernter Anwendungen wird der Transaktionskontext mit übertragen
  - ⊙ Aufgerufene Methode kann die die Transaktion des Aufrufers teilen (muss aber nicht)
- Koordination der Transaktionen übernimmt der aufrufende Container (Applikationsserver)
- Umsetzung teilweise aufwändig bzw. unmöglich

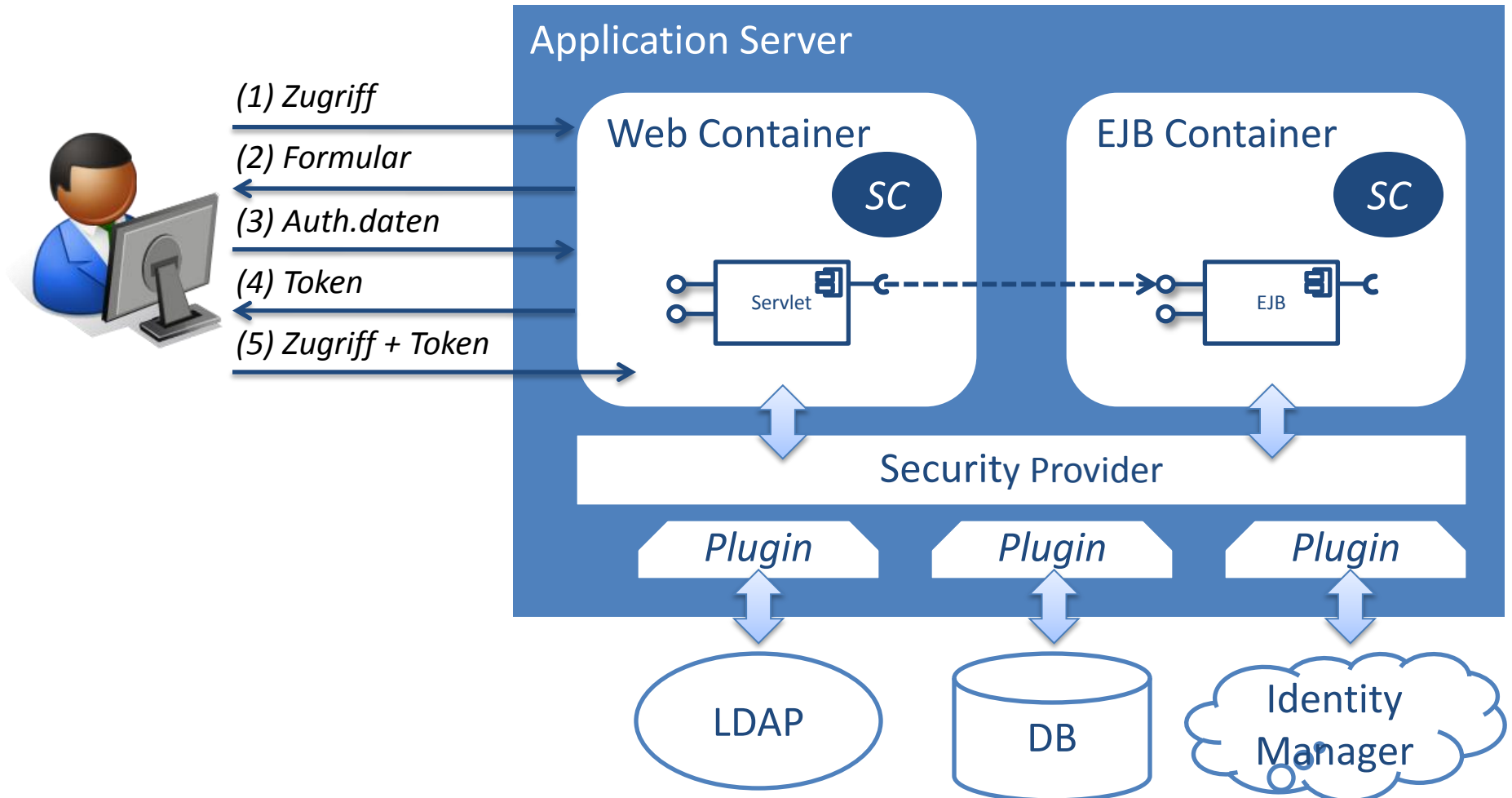
**Security**



# Authentisierung / Autorisierung

- Authentisierung == Wer bist du?
  - ⊙ Identifizierung eines Objekts und die Überprüfung der Identität
  - ⊙ Identität entspricht Benutzername
  - ⊙ Identifizierung erfolgt über Credentials
- Autorisierung == Was darfst du?
  - ⊙ Überprüfung von Berechtigungen eines angemeldeten Benutzers bezogen auf geschützte Ressourcen
  - ⊙ Berechtigungen entsprechen Rollen / Gruppen

# Ablauf einer Anmeldung



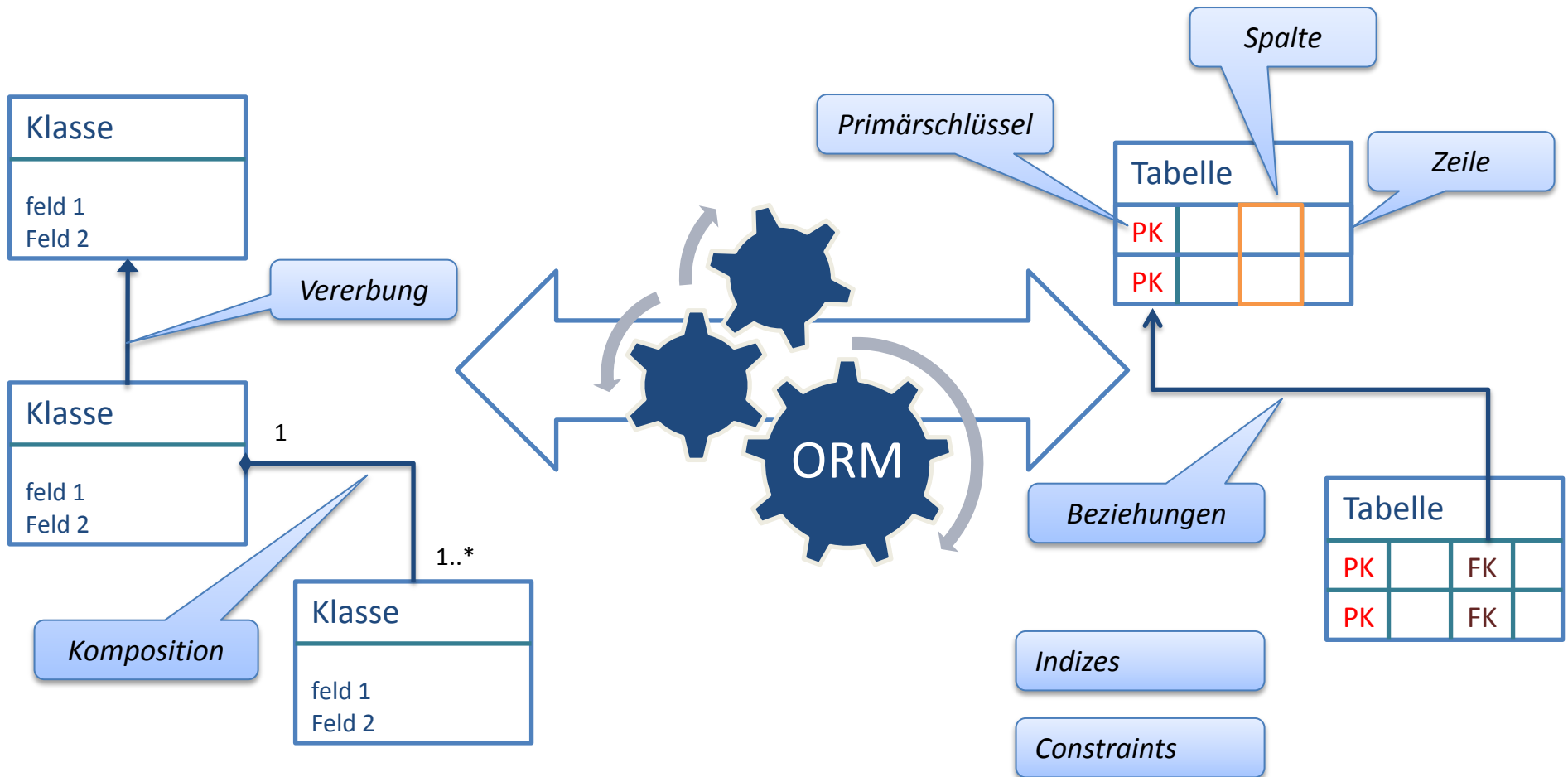
# Propagation von Securitykontexten

- Bei Aufrufen entfernter Anwendungen wird der Securitykontext mit übertragen
  - ⊙ Aufgerufene Methode läuft im gleichen Securitykontext wie der Aufrufer
  - ⊙ Benutzer muss sich nicht erneut anmelden (Single Signon/SSO)
- Beteiligte Container (Application Server) müssen einander trauen
- Umsetzung teilweise aufwändig

# Persistenz

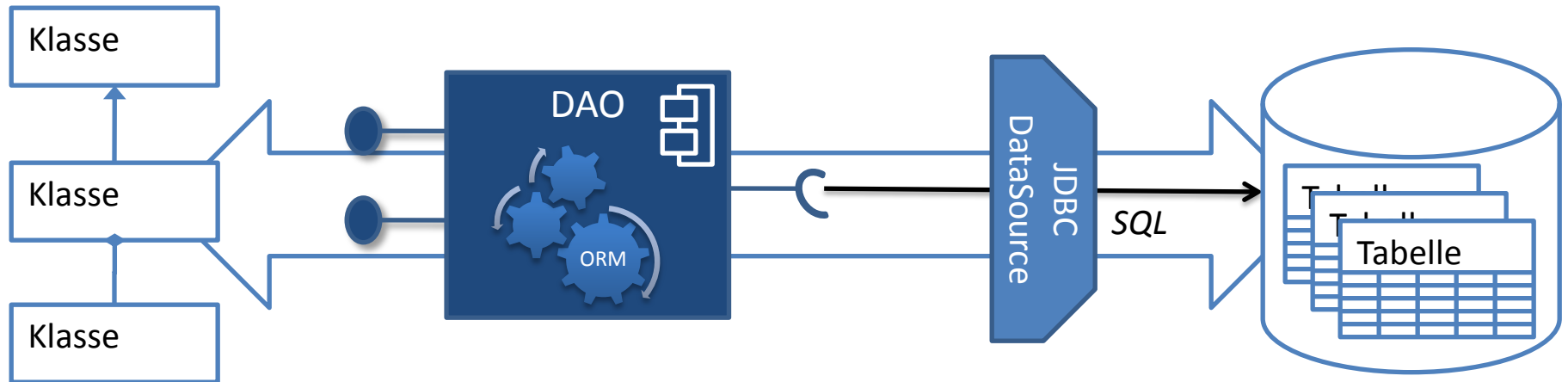


# Object Relational Mapping (ORM)



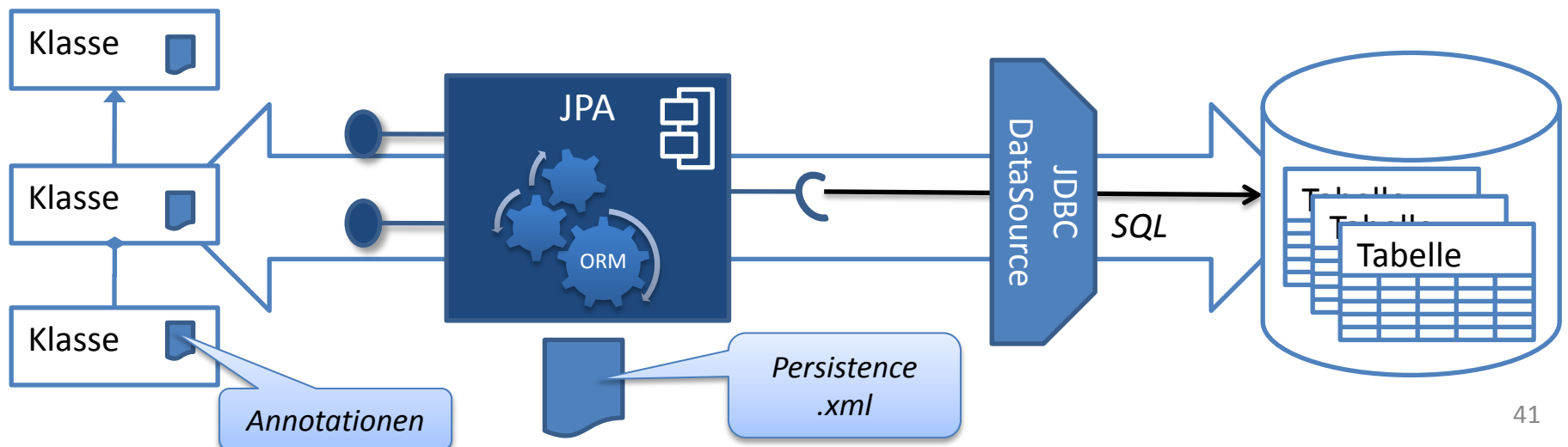
# Nativ mit JDBC

- Mapping muss programmiert werden
- Zugriff auf die Datenbank mit SQL muss programmiert werden



# Mit Persistenzframework (JPA)

- Persistente Klassen werden mit Mapping-Annotationen versehen und mit DataSource verknüpft
- Persistenzframework führt Mapping durch und generiert SQL zur Laufzeit



# Fragen?



# ANHANG

# Quellen

- Martin Fowler: ***Patterns of Enterprise Application Architecture***  
Addison Wesley 2003; ISBN 0-321-12742-0
- Adam Bien: ***Real World Java EE Patterns: Rethinking Best Practices***  
press.adam-bien.com September 2012; ISBN 978-0-300-14931-6
- Gregor Hohpe, Bobby Wolfe  
***Enterprise Integration Patterns***  
Addison Wesley, 2004, ISBN: 0-321-20068-3
- Eric Evans: ***Domain Driven Design:  
Tackling Complexity in the Heart of Software***  
Addison Wesley 2004; ISBN 0-321-12521-5



# Kontakt



## **Michael Theis**

Lehrbeauftragter Hochschule München

email        michael.theis@hm.edu

mobile      + 49 170 5403805

web         <http://www.tschutschu.de/Lehrauftrag.html>