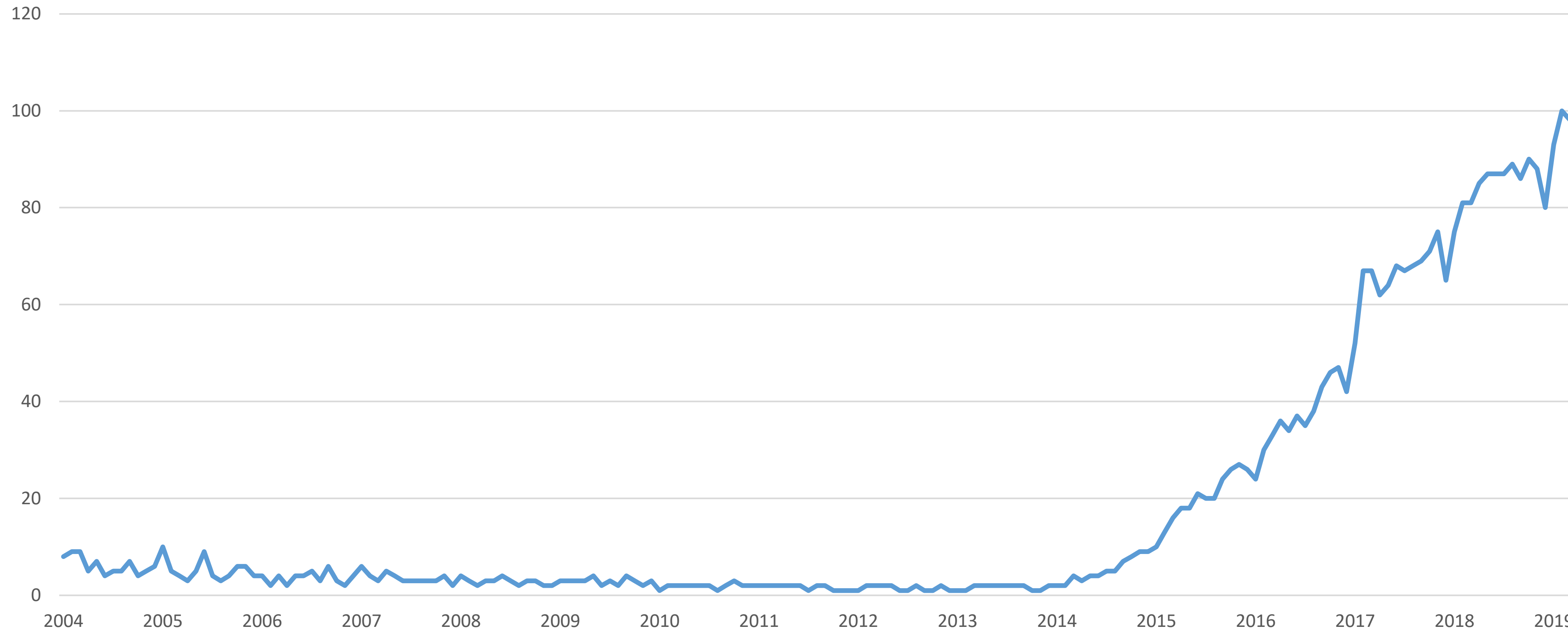
Charakteristiken von Microservices

Warum habe ich das Thema ausgewählt?

Warum solltet ihr über das Thema bescheid wissen?

Trend des Begriffs Microservices



Quelle: Google Trends

Agenda

1. Grundlagen: Monolith und Microservices
2. Charakteristiken von Microservices
3. Vom Monolithen zu Microservices
4. Herausforderungen
5. Zusammenfassung und Fazit

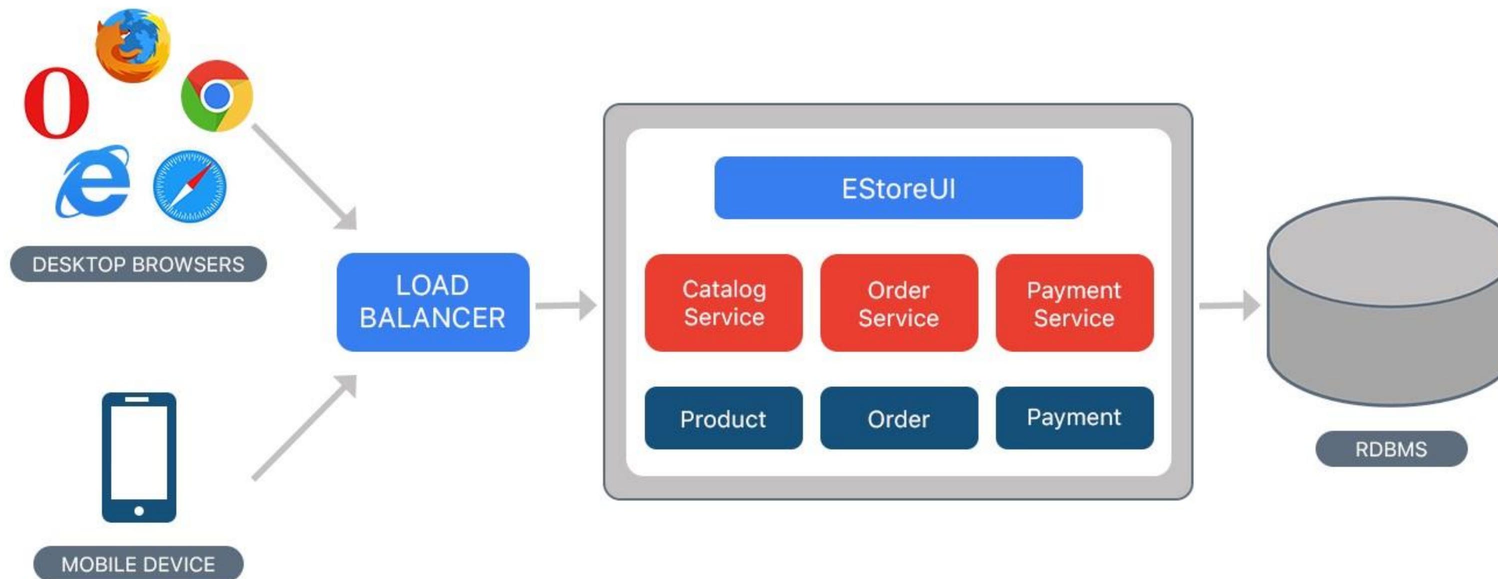
1. Grundlagen: Monolith und Microservices

Was ist ein Monolith?

- Gesamter Quellcode in einer Codebasis
- Beim Bauen entsteht ein Artefakt
  - Skalierung durch kopieren des Artefakts und erneutes bereitstellen

Was ist ein Monolith?

- Gesamter Quellcode in einer Codebasis
- Beim Bauen entsteht ein Artefakt
  - Skalierung durch kopieren des Artefakts und erneutes bereitstellen



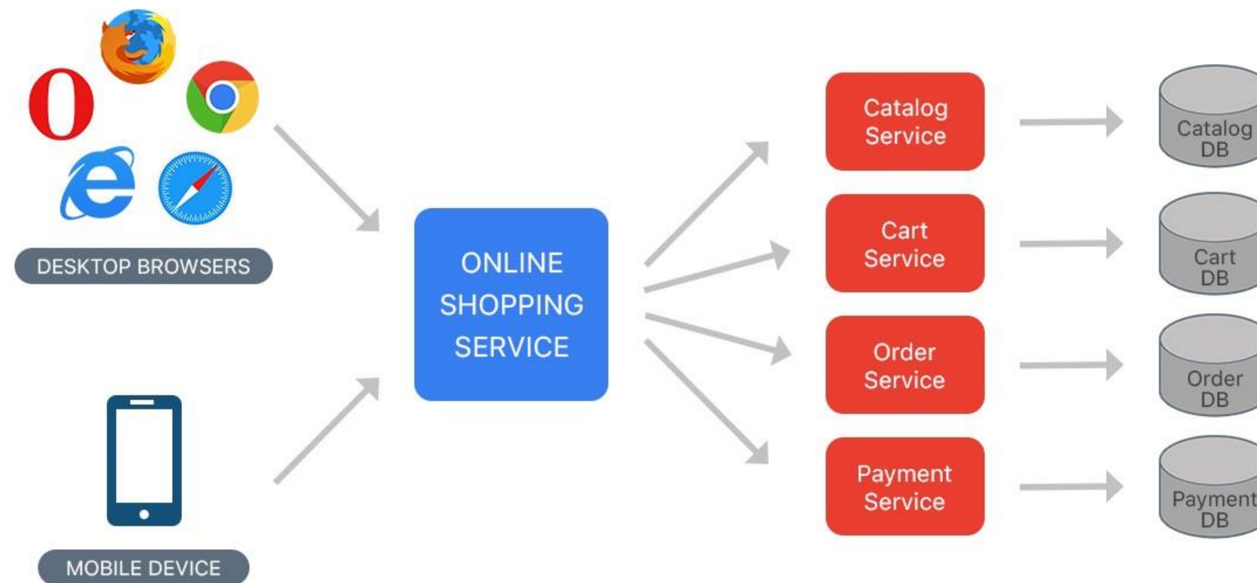
Quelle: Haq, S. Introduction to Monolithic and Microservice Architecture

Was sind Microservices?

- **Kleine** und **autonome** Services die voneinander **unabhängig bereitstellbar** sind
- Kommunikation zwischen Microservices über Netzwerke

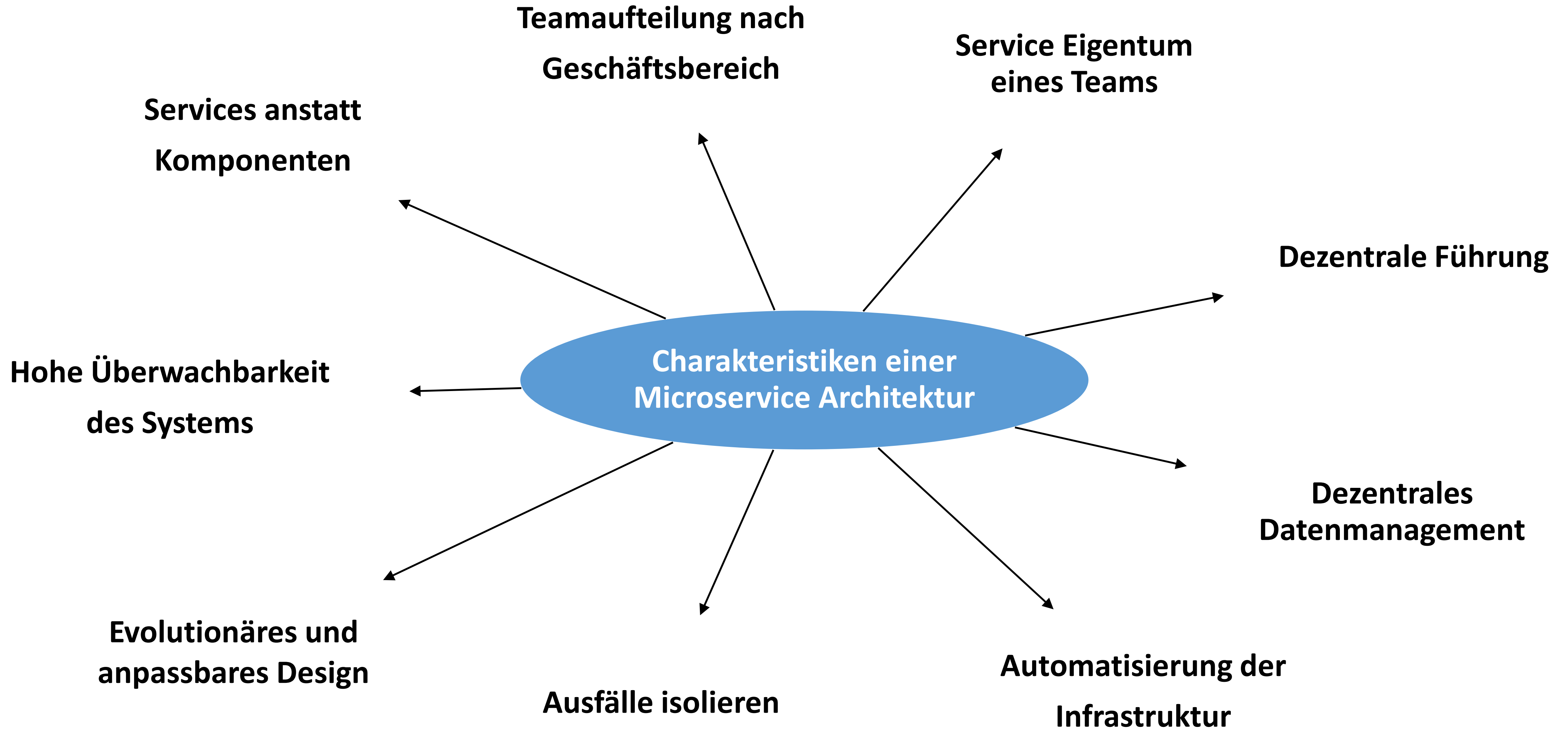
Was sind Microservices?

- **Kleine** und **autonome** Services die voneinander **unabhängig bereitstellbar** sind
- Kommunikation zwischen Microservices über Netzwerke



Quelle: Haq, S. Introduction to Monolithic and Microservice Architecture

2. Charakteristiken von Microservices


Teamaufteilung nach Geschäftsbereich
Service Eigentum eines Teams
Services anstatt Komponenten
Dezentrale Führung
Charakteristiken einer Microservice Architektur
Hohe Überwachbarkeit des Systems
Dezentrales Datenmanagement
Evolutionäres und anpassbares Design
Ausfälle isolieren
Automatisierung der Infrastruktur

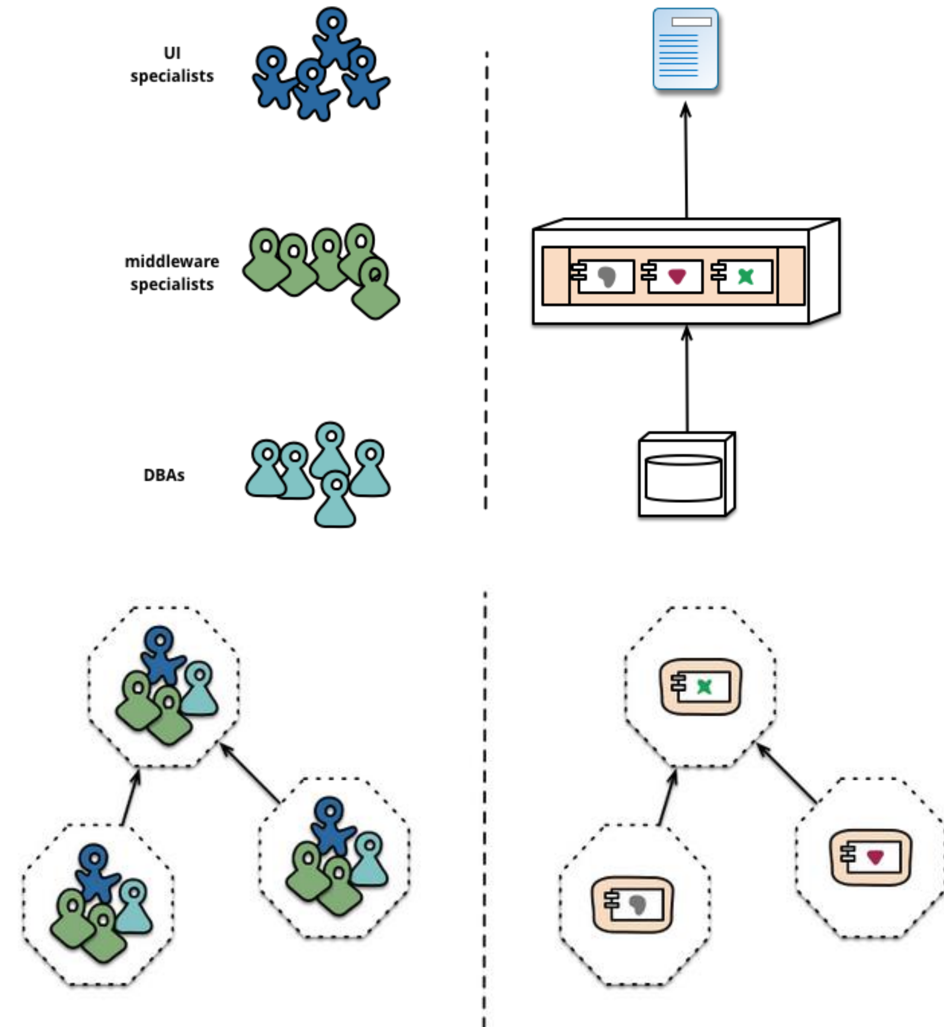
Services anstatt Komponenten

- Komponenten und Service
  - Beide lose gekoppelt und Kommunikation mit anderen Komponenten/Services über Schnittstellen
- Komponenten innerhalb eines Prozess → Kommunikation über Methodenaufrufe
- Services bilden eigene Applikation → Kommunikation über entfernte Schnittstellen
- Vorteile
  - Unabhängige Bereitstellung von Services möglich
  - Kopplung bleibt geringer und explizitere Schnittstellen

Teamaufteilung nach Geschäftsbereich

- Klassisch:
  - Teamaufteilung nach Technologie
  - Frontend-, Backend-, Datenbankteam

- Microservices:
  - Jedes Team besteht aus Experten auf den verschiedenen Teilgebieten
    → Ein Service pro Team



Quelle: https://martinfowler.com/articles/microservices.html

Service Eigentum eines Teams

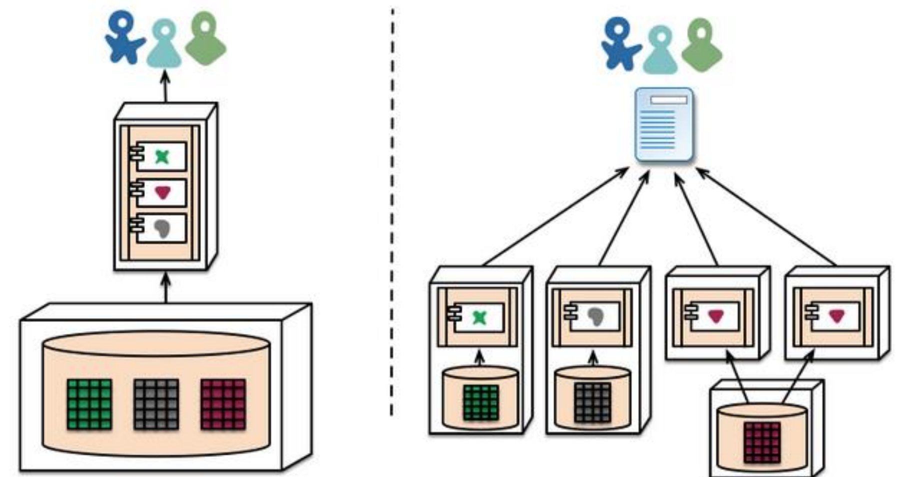
- Klassisch: Projektteam, QA Team, Produktion Team
  - Problem: Teams müssen sich neu mit Service vertraut machen
    → Einarbeitungszeit
- Microservices: Team besitzt einen Service
  - Zuständig für Planung, Bauen, Testen, Deployment und Monitoring
    → „You build it you run it“
- Vorteile
  - Keine Einarbeitungszeit
  - Service kann fortlaufend verbessert werden

Dezentrale Führung

- Klassisch: Zentrale Führung
  - Problem: Eine Person kann kein Experte in jedem Teil des Systems sein
    → Trifft vielleicht suboptimale Entscheidungen
- Microservices: Dezentrale Führung
  - Team trifft die Entscheidungen über einen Service
- Vorteil
  - Optimierte Technologiewahl
  - Neue Erkenntnisse oder Innovationen durch ausprobieren neuer Technologien
  - Bessere/schnellere Entscheidungsfindung

Dezentrales Datenmanagement

- Klassisch: Eine relationale Datenbank
  - Problem: Schlecht horizontal skalierbar und System durch Datenbank gekoppelt
- Microservices: Dezentrales Datenmanagement
  - Jeder Service hat eigene Datenbank und nur dieser hat direkten Zugriff
  - Andere Services: Datenbankoperationen über Schnittstellen
- Vorteil
  - Einfache horizontale Skalierbarkeit der DB
  - Unabhängig von der Datenbanktechnologie
  - Versionierung der Schnittstellen einfacher möglich



Quelle: https://martinfowler.com/articles/microservices.html

Automatisierung der Infrastruktur

- „Must have“ für Microservices
- Push eines Entwicklers stößt die Deploymentpipeline an
- Vorteil der Automatisierung:
  - Einfacher: Entwickler muss nur seine Änderungen pushen
  - Effizienter als manuelle Ausführung
  - Riskoärmer: Mensch macht Fehler

Ausfälle isolieren

- Bei Ausfall droht Gefahr einer Kettenreaktion im System
- Microservices verwenden mehr Maschinen und Netzwerke
  → Größeres Ausfallpotential
- Maßnahmen
  - Kurze Timeouts von Anfragen (3-4s)
  - Circuit Breakers (Sicherungen)
    → Fehlerhafte Anfragen zählen und nach gewisser Anzahl greift die Sicherung
    → Service neustarten und nach gewisser Zeit testweise Anfragen schicken
  - Bulkheads (Trennwand)
    → Isoliert fehlerhaften Service vom Rest des Systems

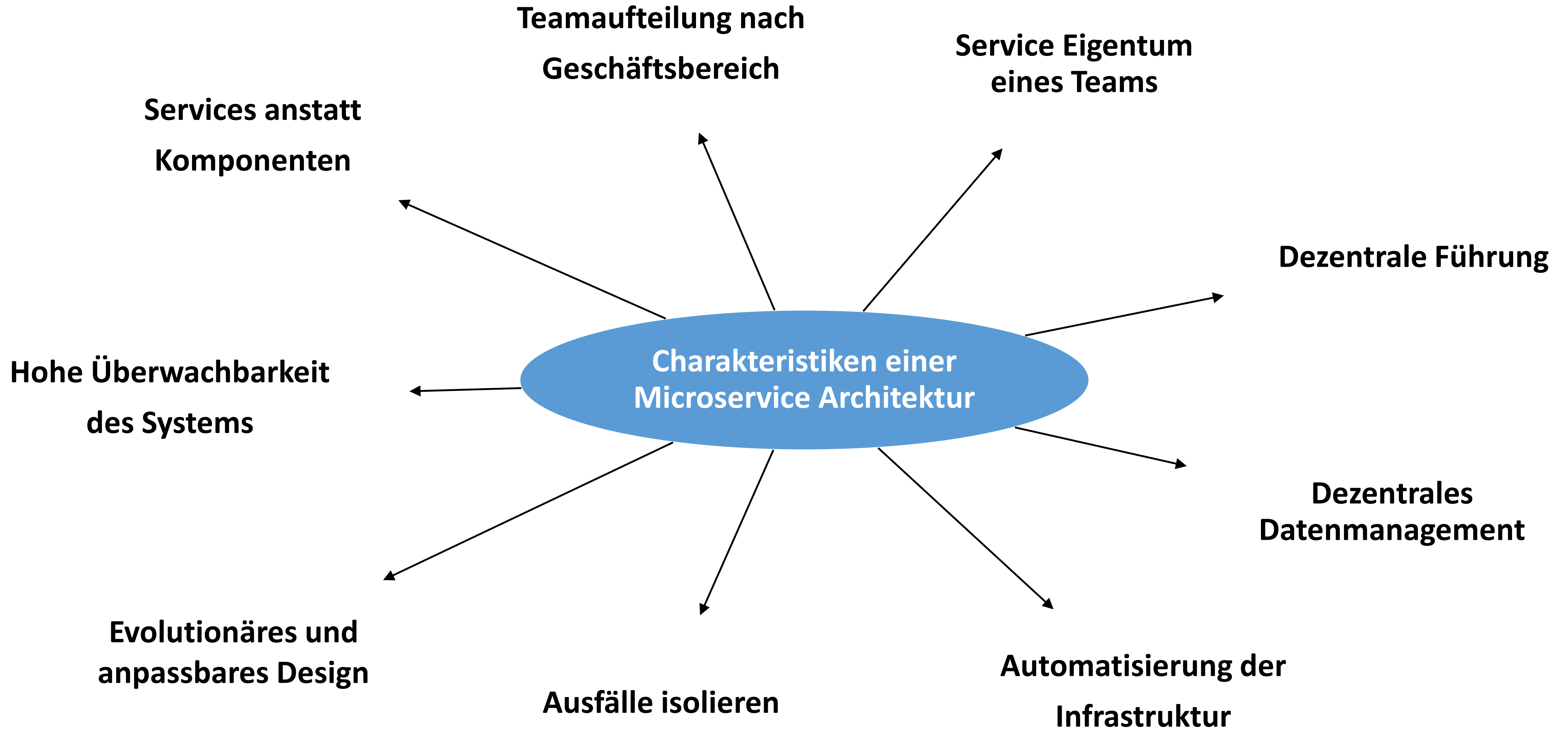
Hohe Überwachbarkeit des Systems

- Überwachung soll Ausfälle im System frühzeitig erkennen
- Problem
  - Logfiles auf verschiedenen Servern verteilt
  - Nachvollziehbarkeit des Nachrichtenfluss zwischen Microservices
- Lösung für verteilte Logfiles
  - Zusammenführen der Daten (mit z.B. Logstash, FluentD)
  - Analyse der Daten (mit z.B. Kibana)
- Lösung um Nachrichtenfluss nachzuvollziehen
  - Aktionen (z.B. Button Klick) eine ID zuweisen, diese im Header mitschicken und in Logs speichern

    → Graphische Darstellung des Nachrichtenfluss möglich

Evolutionäres und anpassbares Design

- Monolith
  - Langfristige Planung und Nutzung des Systems
  - Problem: Codebase wächst an → Änderungen brauchen immer länger
- Microservices
  - Services unabhängig bereitstellbar → Änderungen am Service können schnell bereitgestellt werden
  - Services sind austauschbar
  - Temporäre Services möglich
  - Neue Ideen können durch neuen Service ausprobiert werden


Charakteristiken einer Microservice Architektur
Teamaufteilung nach Geschäftsbereich
Service Eigentum eines Teams
Services anstatt Komponenten
Dezentrale Führung
Hohe Überwachbarkeit des Systems
Dezentrales Datenmanagement
Evolutionäres und anpassbares Design
Ausfälle isolieren
Automatisierung der Infrastruktur

3. Vom Monolithen zu Microservices

Warum Monolith aufteilen?

- Möchte von Charakteristiken profitieren
  - Unabhängige Bereitstellung
  - Evolutionäres Design
- Codebase wächst bei Monolith mit der Zeit → Komplexität steigt

Voraussetzungen

- Automatisierung der Infrastruktur
  - Deploymentpipline zur autonomen Bereitstellung
- Zentrale Überwachung des Systems möglich
- Neue Kontextanforderungen direkt als Microservices implementieren
- Service Eigentum eines Teams

  → Teams dafür umstrukturieren

Vorbereitung des Monolithen

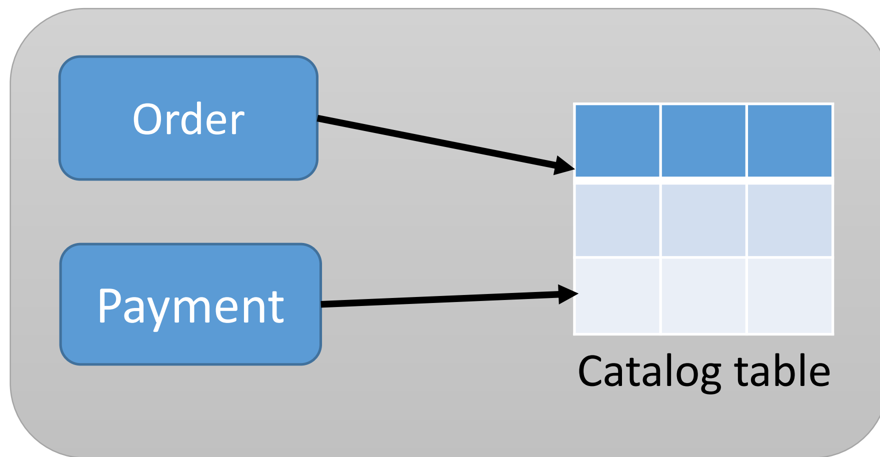
1. Monolith analysieren
    - Zusammengehörigen Kontext identifizieren
2. Ordner für jeden Kontext erstellen
    - Zusammengehörigen Sourcecode in die Ordner verschieben
3. Abhängigkeiten ausfindig machen
    - Methodenaufrufe/Klasseninstanziierung zwischen Ordnern ausfindig machen
    - Abhängigkeiten grafisch darstellen (z.B. Structure 101)

Aufteilung des Monolithen

- Wo fängt man mit der Aufteilung an?
  a) Unabhängige Microservices aus System ziehen
  b) Geschäftslogik aufbrechen die starke Abhängigkeiten besitzt
  c) Oft benutzten und häufig geänderter Kontext in Microservice auslagern
- Wie werden Abhängigkeiten aufgebrochen?
  - Methodenaufrufe und Datenbankoperationen durch entsprechende Schnittstellen ersetzen
  - Mehrfach benutzte Datenbanktabellen aufteilen und als eigene Services implementieren

Aufteilung des Monolithen

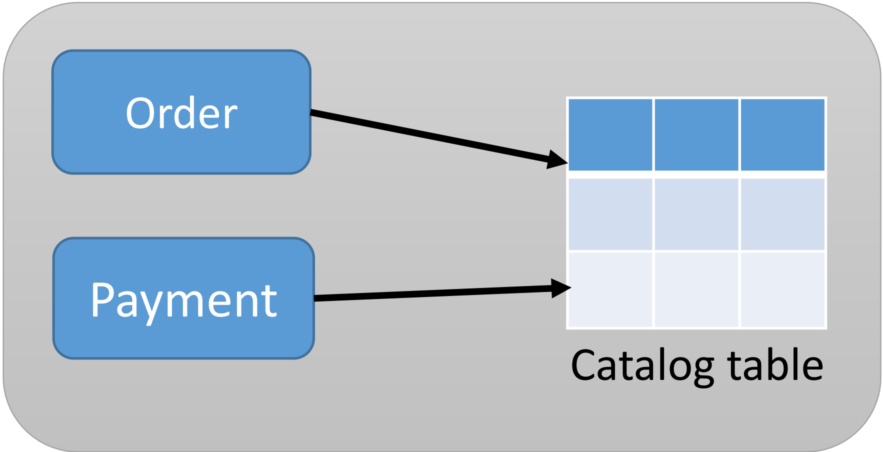
Monolith

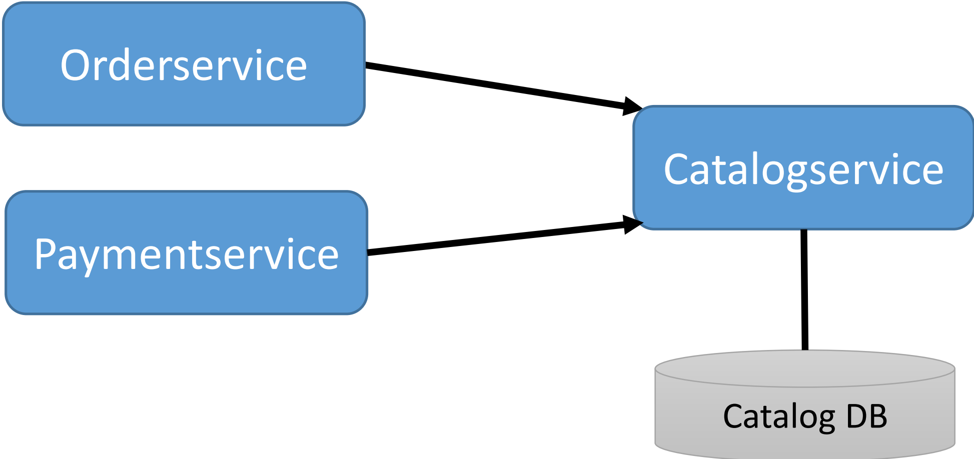


Aufteilung des Monolithen

Monolith

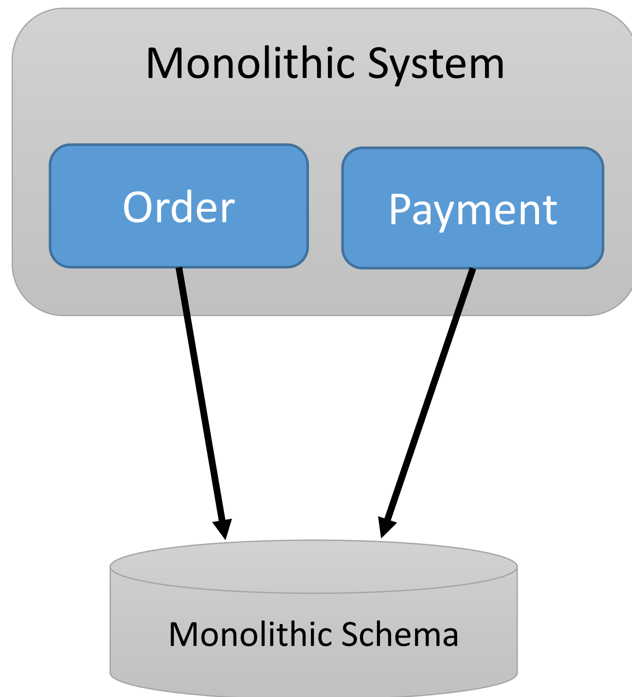
Microservices





Aufteilung des Monolithen

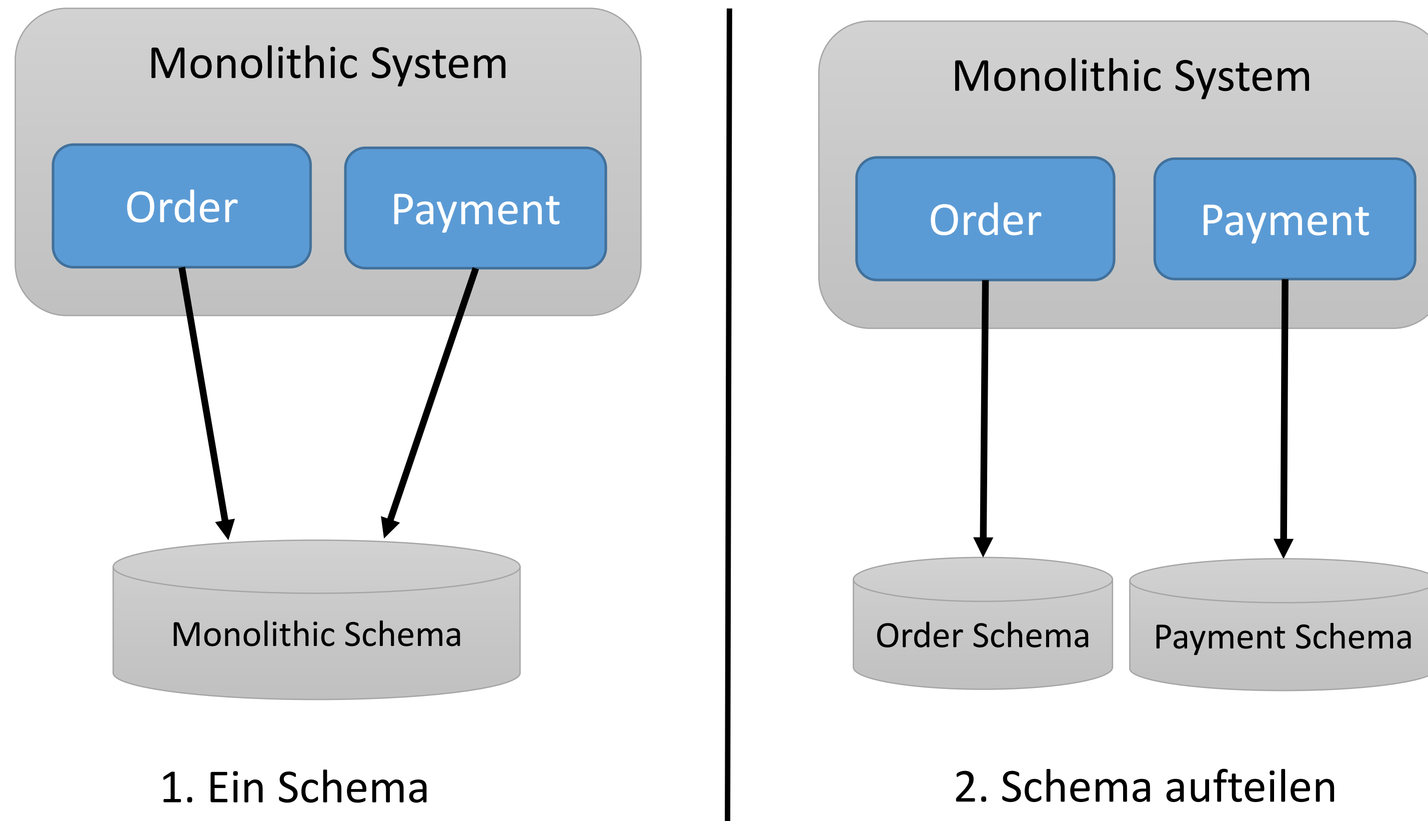
- Vorgehen bei der Aufteilung in Microservices



1. Ein Schema

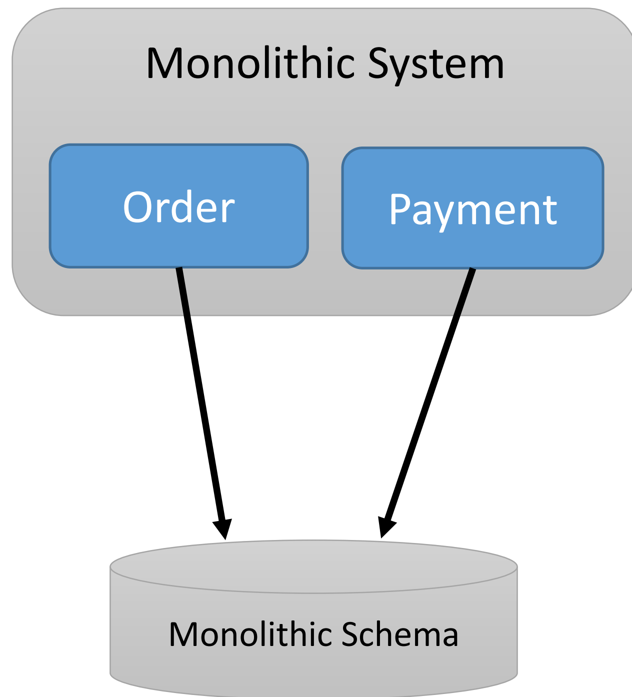
Aufteilung des Monolithen

- Vorgehen bei der Aufteilung in Microservices

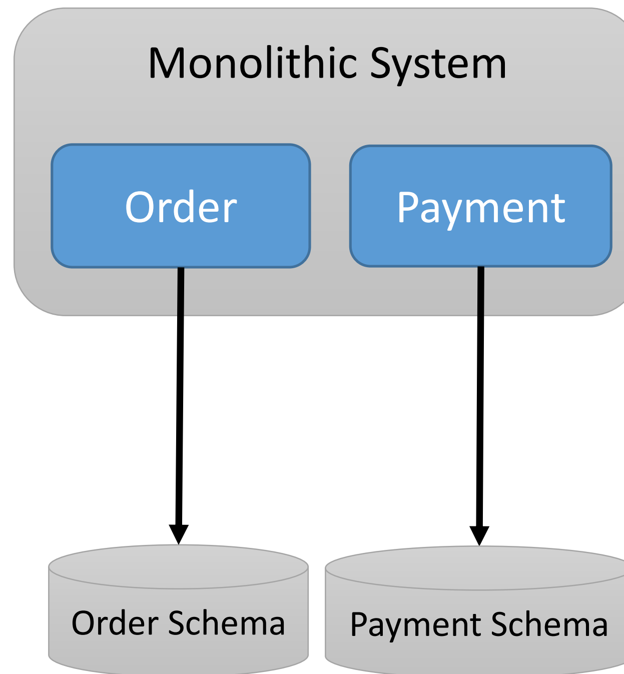


Aufteilung des Monolithen

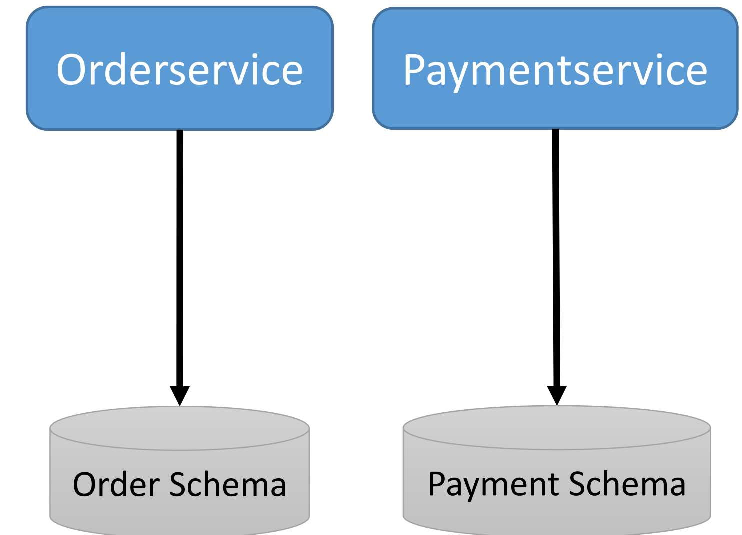
- Vorgehen bei der Aufteilung in Microservices



1. Ein Schema



2. Schema aufteilen



3. Applikation in Services aufteilen

4. Herausforderungen

Herausforderungen von Microservices

- Hohe Komplexität in der Entwicklungsphase
  - Monolith: Eine Applikation die auf eigenem Rechner ausgeführt werden kann
  - Microservices: Viele Applikationen die in Testumgebung laufen

    → Abhängig von Testumgebung

    → Debugging lokal viel einfacher als in Testumgebung
- Risiko schlechter Performance
  - Microservices sind viele eigene Applikationen → Overhead
  - Netzwerkkommunikation automatisch langsamer als Methodenaufrufe
- Höhere Instandhaltungskosten
  - Mehr Rechenleistung + Speicher → höhere Kosten

Herausforderungen von Microservices

- Komplexere Infrastruktur
  - Durch die hohe Anzahl an laufenden Applikationen
    → Fehlersuche im System schwer
- Hohe Anforderung an Entwickler
  - Ist für Planung, Entwicklung, Bereitstellung und Verwaltung verantwortlich
    → Weniger Spezialisierung auf Fachgebiete
- Unklare Kontextabgrenzung in der Echten Welt
  - Microservices manchmal schwierig voneinander abzugrenzen
  - Aufteilen des Datenbankschemas oft nicht möglich

5. Zusammenfassung und Fazit

Zusammenfassung

- Monolith
  - Eine Codebase, ein Artefakt und eine relationale Datenbank
- Microservices
  - Kleine unabhängige Services die autonom bereitgestellt werden können und über standardisierte Schnittstellen kommunizieren
- Charakteristiken
  - Automatisierung der Infrastruktur
  - Umstrukturierung der Teams
  - Team für einen Service verantwortlich
  - Dezentrale Führung
  - Eine Datenbank pro Service
  - Ausfälle isolieren → Kettenreaktionen vermeiden
  - Zentrale Überwachung des Systems

Zusammenfassung

- Aufteilung von Monolith
  - Voraussetzungen: Automatisierung und Zentrale Überwachung
  - Abhängigkeiten aufbrechen (Methodenaufrufe, Datenbank)
  - Erst Datenbankschema, dann Microservices herausbrechen
- Herausforderungen
  - Softwareentwicklungsprozess komplexer
  - Schlechte Performance durch Overhead der Applikationen und Kommunikation über Netzwerke möglich
  - Mehr Ressourcen notwendig → Höhere Instandhaltungskosten
  - Komplexere Infrastruktur
  - Größerer Aufgabenbereich für Softwareentwickler
  - Kontextabgrenzung in realer Welt nicht immer klar

Fazit

- Microservices sind vor allem für große Online Applikationen geeignet
- Softwareentwickler sollten im Mittelpunkt der Entscheidung

Fragen?