

DDD und Microservices bei Etengo im Einsatz

21.06.2018

Bastian Feigl

Sebastian Tuttas

Lars Alvincz

bastian.feigl@andrena.de

sebastian.tuttas@etengo.de

lars.alvincz@andrena.de

Agenda

1. Die Mission: Neuentwicklung mit Microservices
2. Fachlich: Microservices und DDD
3. Technisch: Entwicklung einer Microservice-Anwendung
4. Fazit

Die Mission: Neuentwicklung mit Microservices



- 160 Mitarbeiter
- 143 Millionen Euro Umsatz (Plan 2018)
- 975 Freelancer im Projekt
- Mehr als 6.000 erfolgreich abgewickelte Projekteinsätze bei Kunden
- 17 der 30 DAX-Konzerne sind bereits Bestandskunden von Etengo
- Rund 300 aktive Kunden vertrauen langfristig auf Leistungen von Etengo
- Platz 5 im deutschen Markt
- Pionier für Open-Book im Personaldienstleistungsmarkt
- Innovationsvorsprung durch schlanke, Software-unterstützte Prozesslandschaft



etengo
PERSONAL SERVICES



etengo
SMART FREELANCING

**Wir vermitteln freiberufliche
Experten aus IT und dem IT-nahen
Umfeld in Projekte.**



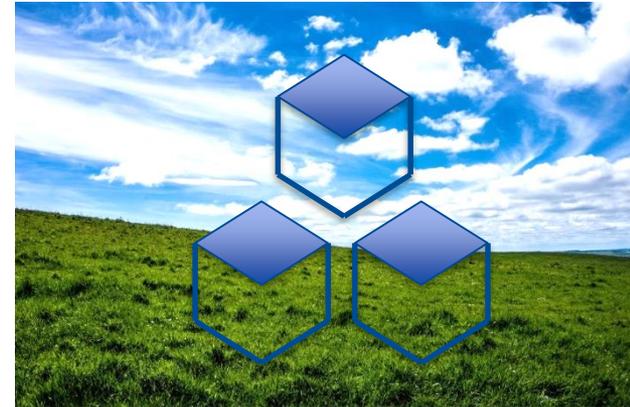
etengo
ACADEMIC EXPERTS

**Wir rekrutieren hochqualifiziertes
Personal aus IT und dem IT-nahen
Umfeld.**



Microservices auf der grünen Wiese

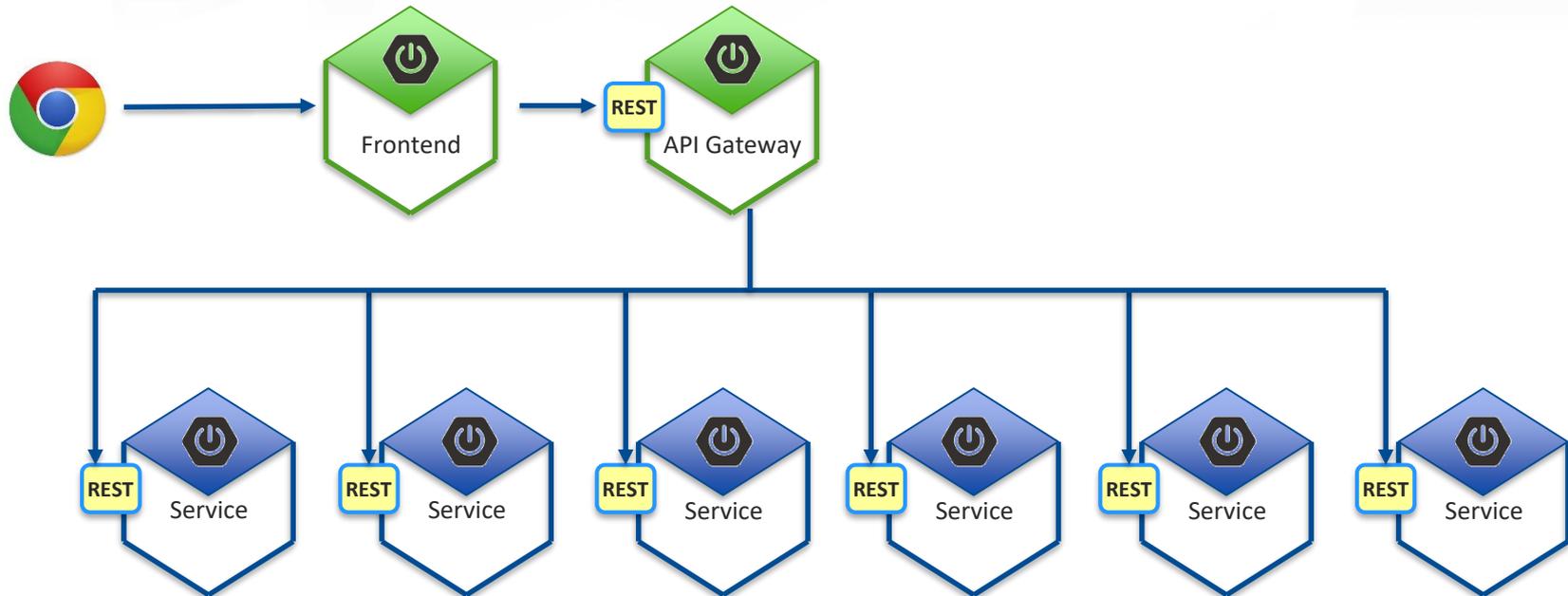
- Neuentwicklung einer bestehenden Anwendung
- Anforderungen des Kunden:
 - Web-Anwendung
 - Modulares Deployment
 - Austauschbarkeit einzelner Komponenten
 - Hohe Verfügbarkeit
 - Externes Anbieten einzelner Services



verteilte Anwendung mit Microservices, zentrales Web-Frontend



Initiale Architekturidee



Was ist für uns ein Microservice?

- In sich **geschlossen**
 - Besitzt die absolute Hoheit über seine Daten
 - Nur ansprechbar über REST/Messaging (DB etc. nicht nach außen verfügbar)
 - Eigenes GIT-Repository
 - Eigene Versionen der Abhängigkeiten
- **Lose Kopplung** zu anderen Services
 - Sollte auch funktionieren, wenn andere Services ausfallen (**Resilienz**)
- **Skalierung** möglich (**Ausfallsicherheit, Performanz**)



Wie schneiden wir Microservices?

- Microservices sollten in sich geschlossenen Bereiche der Anwendung umsetzen
- Wichtig: Lose Kopplung (Performanz, Resilienz)

Domain-driven Design (DDD) ist eine Herangehensweise an die Modellierung komplexer Software. Die Modellierung der Software wird dabei maßgeblich von den umzusetzenden **Fachlichkeiten** der **Anwendungsdomäne** beeinflusst.

(Wikipedia; Eric Evans, 2006)

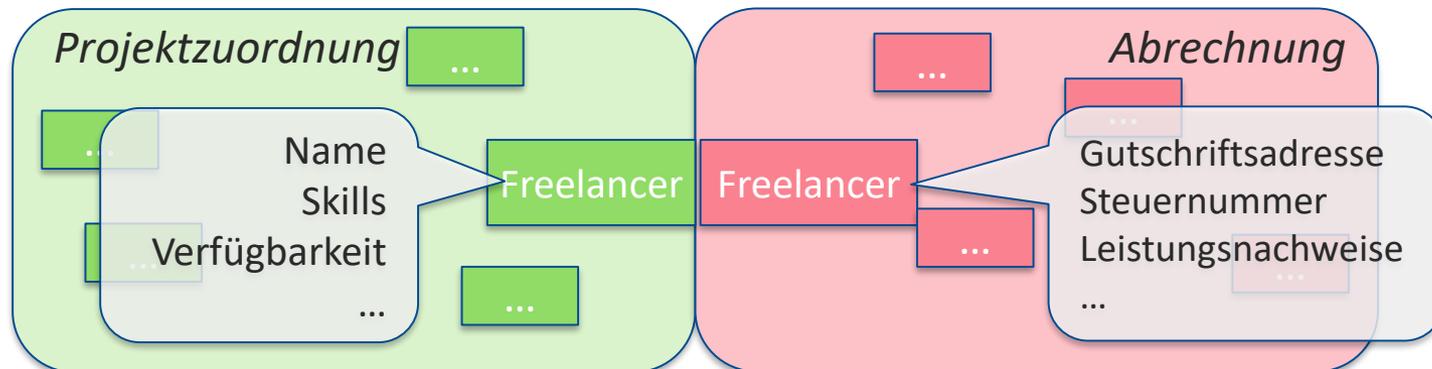


Fachlich: Microservices und DDD



DDD: Bounded Context – Microservice

- Ein fachlicher Begriff kann in verschiedenen Kontexten **verschiedene** Ausprägungen haben
- Beispiel: Personaldienstleister



- **Bounded Context:** Abgeschlossener Ausschnitt aus der Fachdomäne
- Ein **Bounded Context** lässt sich gut auf einen **Microservice** abbilden



Wie finden wir Bounded Contexts?

Event Storming:

- Fachliche Ereignisse identifizieren und gemäß Ablauf anordnen (Beispiele: Freelancer beauftragt, Freelancer erstellt Abrechnung, ...)
- System/Benutzer-Aktionen ableiten
- Beteiligte Entitäten ergänzen
- Gruppieren und Bounded Contexts identifizieren

Indikatoren für guten Schnitt: Starke Kohäsion, geringe Kopplung



Event Storming: Ergebnis (Ausschnitt)



DDD: Unsere Erfahrungen

Vorgehen:

- Anwendung hat mehrere Kerndomänen
- Für jede Kerndomäne ein DDD-Tagesworkshop (Event-Storming) (PO, Fachbereich, Team)

Ergebnis:

- Bounded Contexts als Grundlage für den Schnitt der Microservices

Nebeneffekt:

- Gemeinsames fachliches Verständnis erreicht
- Gut zur Einarbeitung neuer Kollegen



Technisch: Entwicklung einer Microservice-Anwendung



Herausforderungen

- Gemeinsame Daten: Kopplung vs. Redundanz
- Gemeinsamer Code: Kopplung vs. Redundanz
- Architektur
- Deployment
- Monitoring
- Migration
- Testvorgehen



Gemeinsame Daten: Hohe Kopplung vs. Redundanz?

Manche Daten werden in mehreren Services benötigt

- Beispiel: Name des Freelancers

Naiver Ansatz:

- Zentraler Datenservice oder
- Daten in einem fachlichen Service speichern und bei Bedarf direkt abrufen (Direktzugriff)

Vorteile:

- Geringerer Entwicklungsaufwand
- Daten sind immer aktuell

Nachteile:

- Mehr Kommunikation nötig
- Verfügbarkeit des Services mit benötigten Daten ist kritisch
- Zentraler Datenservice: Schnitt ist technisch (fachliche Änderungen betreffen dann mindestens 2 Services)



Gemeinsame Daten: Hohe Kopplung vs. Redundanz?

Alternativer Ansatz:

- mehrfach benötigte Daten per **Messaging** replizieren
- Services halten Kopien (nur Lese-Zugriff!)

Vorteile:

- **Lose Kopplung, Resilienz** (dafür Redundanz)
- **Messaging:** keine Daten gehen verloren
- **Erweiterbarkeit:**
neue Services können sich registrieren
(-> Data Warehouse etc.)

Nachteile:

- Aufwand für Replikation (Entwicklung und Systemlast)



Entscheidung zwischen Direktzugriff und Replikation

Replikation: Wann rechnet sich der erhöhte Aufwand?

- Indikator:
Häufigkeit des Zugriffs abwägen gegenüber
Häufigkeit der Änderung
- Sehr wenige Zugriffe, häufige Änderungen
→ Direktzugriff
- Häufigere Zugriffe, wenige Änderungen
→ Replikation
- Bei Fokus auf Robustheit → Replikation

Generell: Falls häufige Zugriffe vorkommen: passt der Serviceschnitt?



Im Regelfall: Entkopplung und Daten per Messaging replizieren



Gemeinsamer Code: Hohe Kopplung vs. Redundanz

Das Mantra der Software-Entwicklung:
Code-Wiederverwendung, Duplikation vermeiden

Nachteile:

- Wiederverwendung erschwert unabhängige Deployments
- Verteilter Monolith?



Gemeinsamer Code: Redundanz

fachlicher Code: keine Wiederverwendung

- so echte Entkopplung
- auch z.B. bei REST-API/Message-Interfaces

technischer Code: Wiederverwendung (commons-Libs)

- Beispiel: erweiterbare Enums, Validierung, ...
- commons-Libraries sind versioniert
 - Jeder Service entscheidet, wann er die Version aktualisiert
 - man kann nur den Service ausliefern, der die Änderung braucht
- Aber: früher oder später ist Update nötig

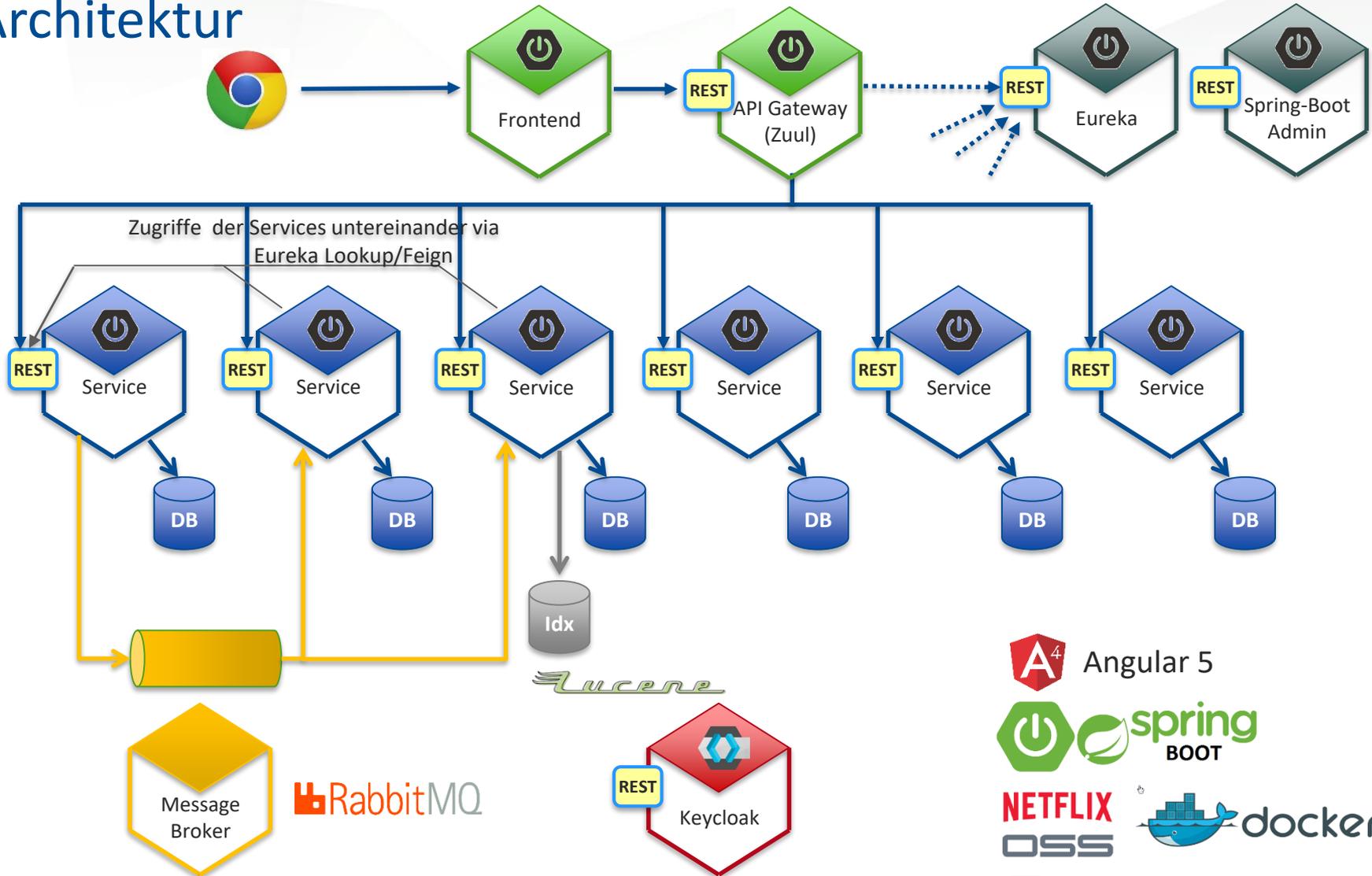


fachlich: Entkopplung durch Duplikation, technisch: Wiederverwendung

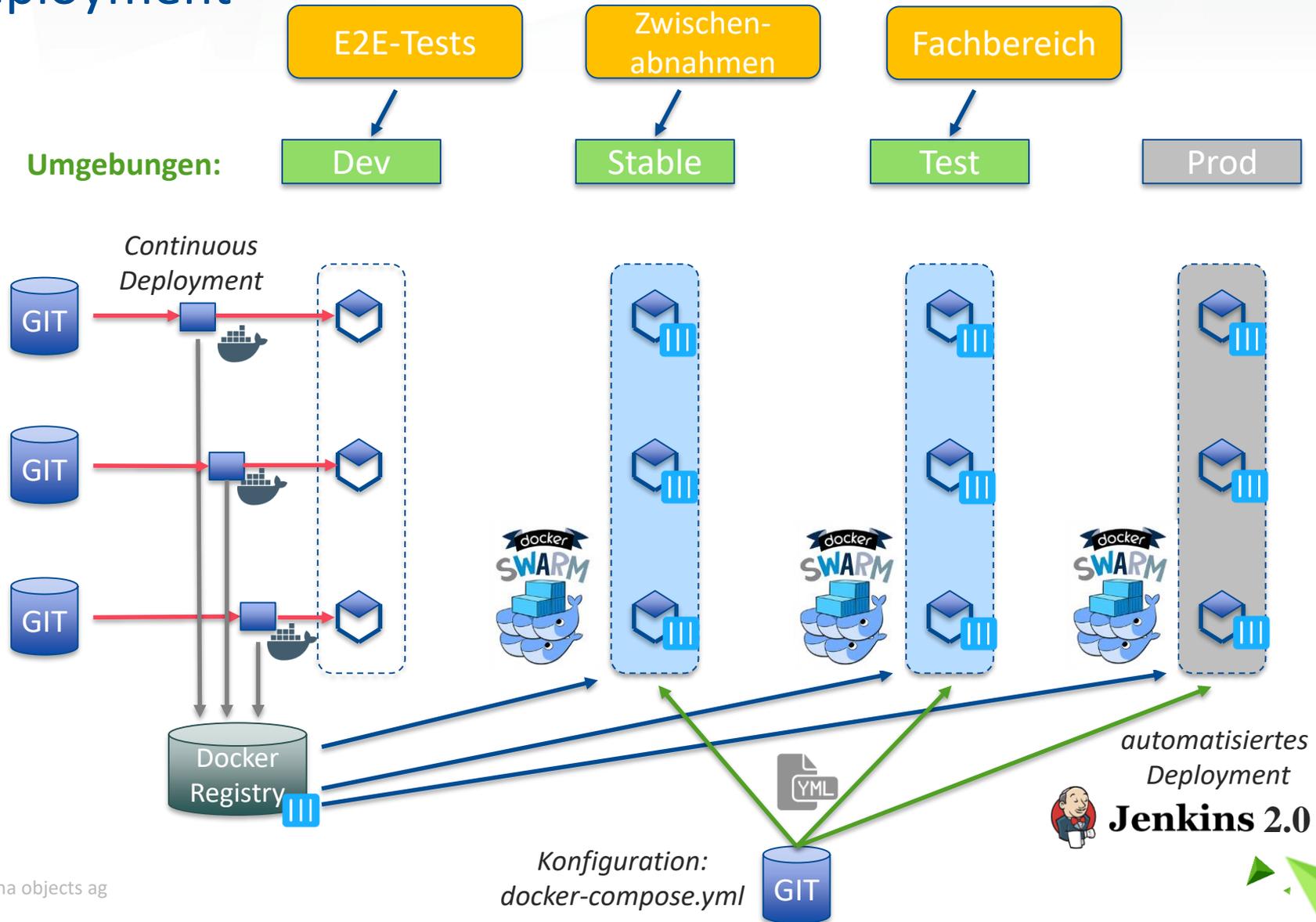


Alle Zugriffe auf Services
via Eureka Lookup

Architektur



Deployment



Monitoring

- Jeder Microservice hat eigenes Log.
- Log-Einträge einer Anfrage können über mehrere Logs verteilt sein.
- Nötig: Aggregation der Logs in einem System zur effizienten Auswertung.
 - Logs von Docker-Containern abgreifen (filebeat)
 - Auswertung mit Elasticsearch/Kibana
- Zur Korrelation einzelner Anfragen: Vorgangs-ID mit im Log speichern (Toolunterstützung: Spring Cloud Sleuth)



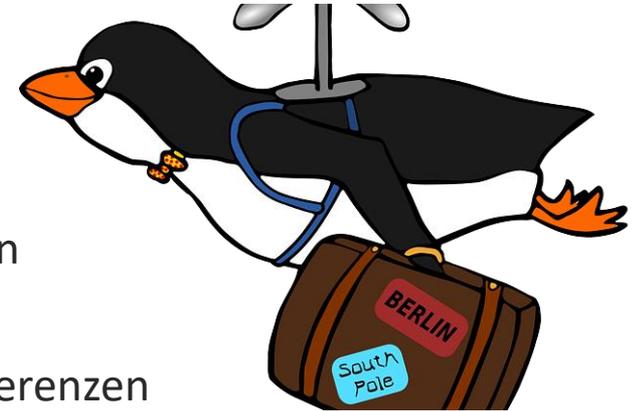
Datenmigration vom Monolithen zum Microservice-System

Altsystem:

- Ein zentrales DB-Schema, das alle Daten enthält
- Daten mit komplexen Objektstrukturen
 - Weitreichende Verknüpfungen zwischen den Daten

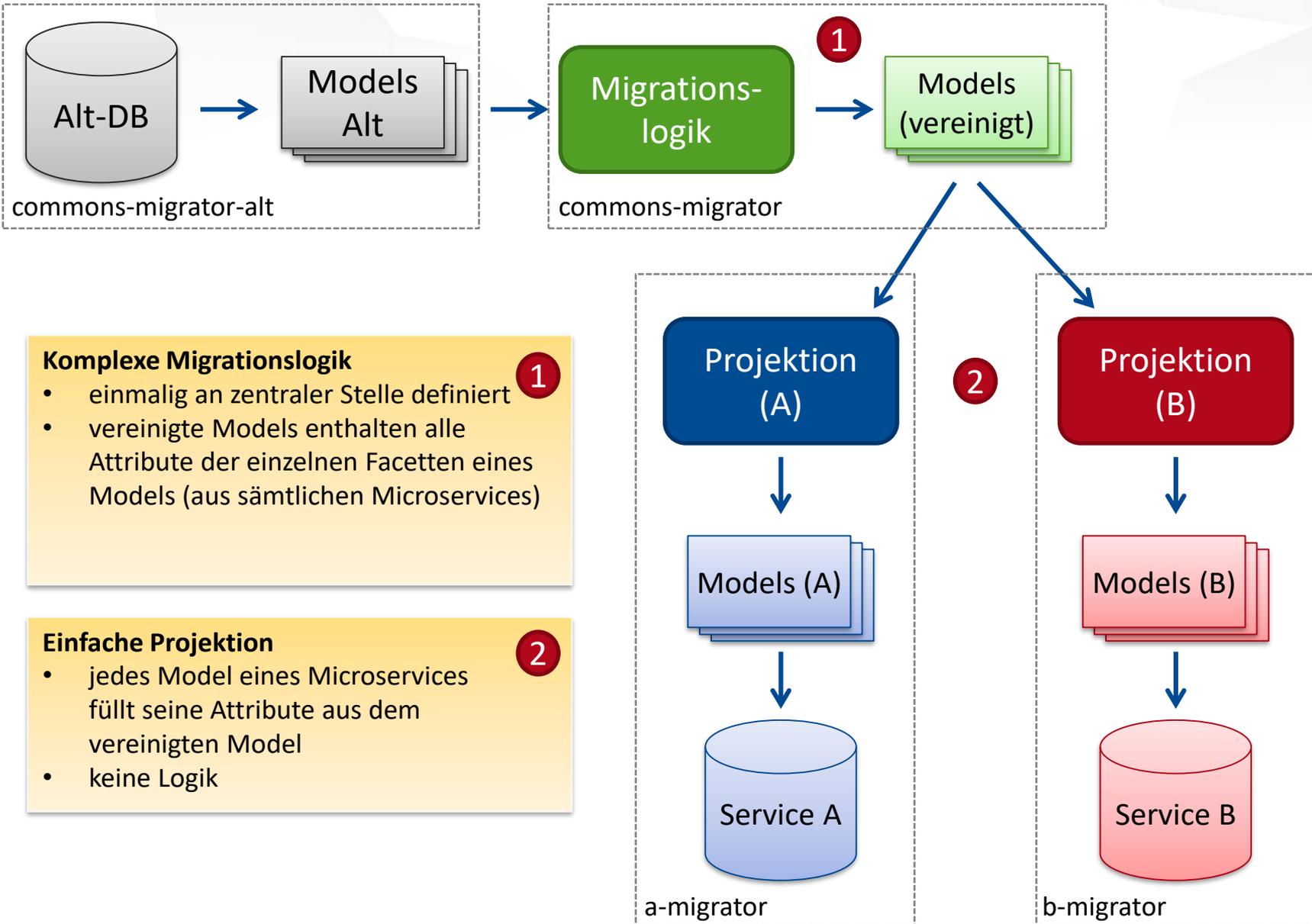
Zielsystem:

- Viele verteilte DB-Schemata, keine übergreifenden Referenzen
- Anderer Schnitt der Objekte
 - Einzelne Services interessieren nur Teilaspekte einzelner Objekte
 - Manche Daten sind mehrfach in verschiedenen Services abgelegt



Herausforderung: Bei der Migration Konsistenz gewährleisten, ohne einen Über-Migrator zu verwenden (der alle Zielobjekte und Zielschemata kennt)





Komplexe Migrationslogik

- einmalig an zentraler Stelle definiert
- vereinigte Models enthalten alle Attribute der einzelnen Facetten eines Models (aus sämtlichen Microservices)

Einfache Projektion

- jedes Model eines Microservices füllt seine Attribute aus dem vereinigten Model
- keine Logik



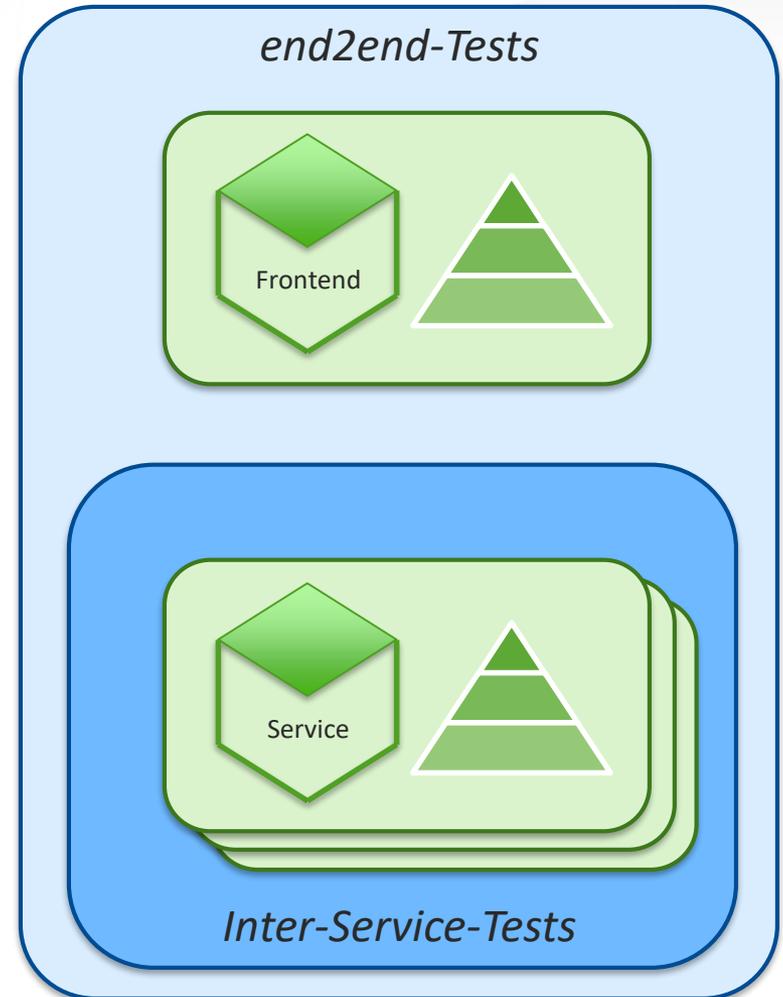
Unser Testvorgehen

Innerhalb eines Services

- Testpyramide

Gesamtsystem

- Inter-Service-Tests (REST, Messaging, Externe Services)
- end2end-Tests mit Selenium (GUI)
- Exploratives Testen (manuell, Timeboxed)



Fazit



Vorteile einer Microservice-Anwendung

- Einzelne Services sind einfach
- Einfachere **Skalierung**
 - Auch von Teilen der Anwendung
- **Ausfallsicherheit/Hochverfügbarkeit**
 - Redundanz / Load Balancing einfach umzusetzen
- **Flexiblere** Architektur
 - Austausch von Teilen möglich
 - geringeres Risiko beim Deployment
- **Lose Kopplung**
 - Entwicklungsarbeit kann gut auf mehrere Teams aufgeteilt werden
 - Unterschiedliche Technologien je nach Anforderungen nutzbar
 - Erweiterung durch neue Services ohne Detailkenntnisse möglich



Lessons Learned

Domain Driven Design-Workshops sehr hilfreich

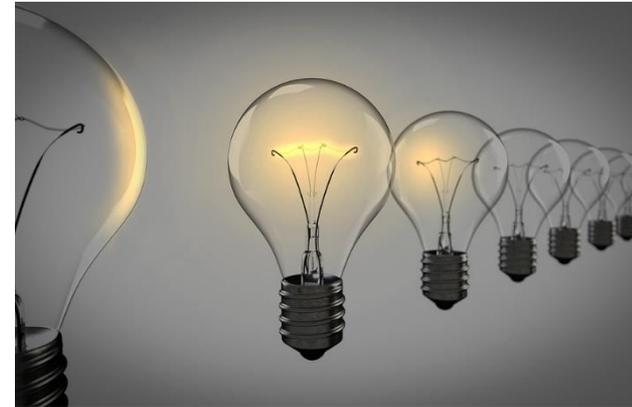
- um guten Schnitt der Microservices zu finden
 - nachträgliche Umstrukturierungen sind teuer
- für gemeinsames Verständnis

Höhere Aufwände (als bei Monolithen)

- Komplexere Infrastruktur
 - Technische Services (Discovery, Gateway, ...)
 - Deployment, Testsysteme → zwingend automatisieren
- Datenabhängigkeiten: Replikation/Messaging
- Codeabhängigkeiten: teilweise Duplikation

Architekturentscheidungen inkrementell klären

- Big Design Up Front nicht sinnvoll/möglich
- Eigene fachliche Stories dafür platzieren (Security, Suche, Migration, Changelog, ...)



Fazit

Vorteile von Microservices:

- Kleine abgeschlossene Einheiten
- Skalierbarkeit
- Ausfallsicherheit
- Flexibilität
- Lose Kopplung



Zu berücksichtigen:

- Gesamtsystem komplexer, höherer Aufwand (Datenhaltung, Kommunikation)
 - Abhängigkeiten zwischen den Services minimieren
- Fachlichen Überblick früh herstellen (DDD-Workshops)
- Architekturentscheidungen inkrementell klären

