DDD und Microservices bei Etengo im Einsatz

04.12.2018

Sebastian Tuttas *sebastian.tuttas@etengo.de*
Bastian Feigl *bastian.feigl@andrena.de*

Agenda

1. Die Mission: Neuentwicklung mit Microservices
2. Fachlich: Microservices und DDD
3. Technisch: Entwicklung einer Microservice-Anwendung
4. Technisch: Datenmigration vom monolithischen Altsystem zur Microservice-Anwendung
5. Fazit

Die Mission: Neuentwicklung mit Microservices

Wir sind Etengo

Der wichtigste Hub für IT-Spezialisten.

etengo
UNTERNEHMENSGRUPPE

Wir rekrutieren freiberufliche IT-Spezialisten
für zeitlich begrenzte Projekteinsätze.

→ SMART FREELANCING

Wir rekrutieren die besten und gefragtesten
IT-Spezialisten zur Festanstellung.

→ ACADEMIC EXPERTS

FREELANCING × HEADHUNTING

Die Etengo Unternehmensgruppe

Fakten im Überblick.

Platz 5

in der deutschen Personaldienstleistung

112 Mio. €

Umsatz in 2017

497.000+

Kontakte zu IT-Spezialisten pro Jahr

160

festangestellte Etengo-Mitarbeiter

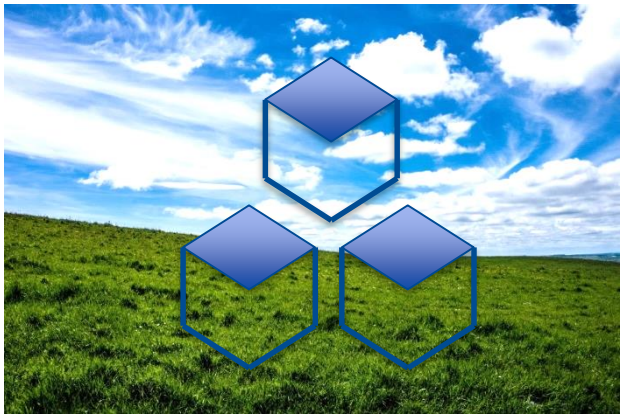
1.000+

Freelancer täglich parallel im Einsatz

FREELANCING × HEADHUNTING

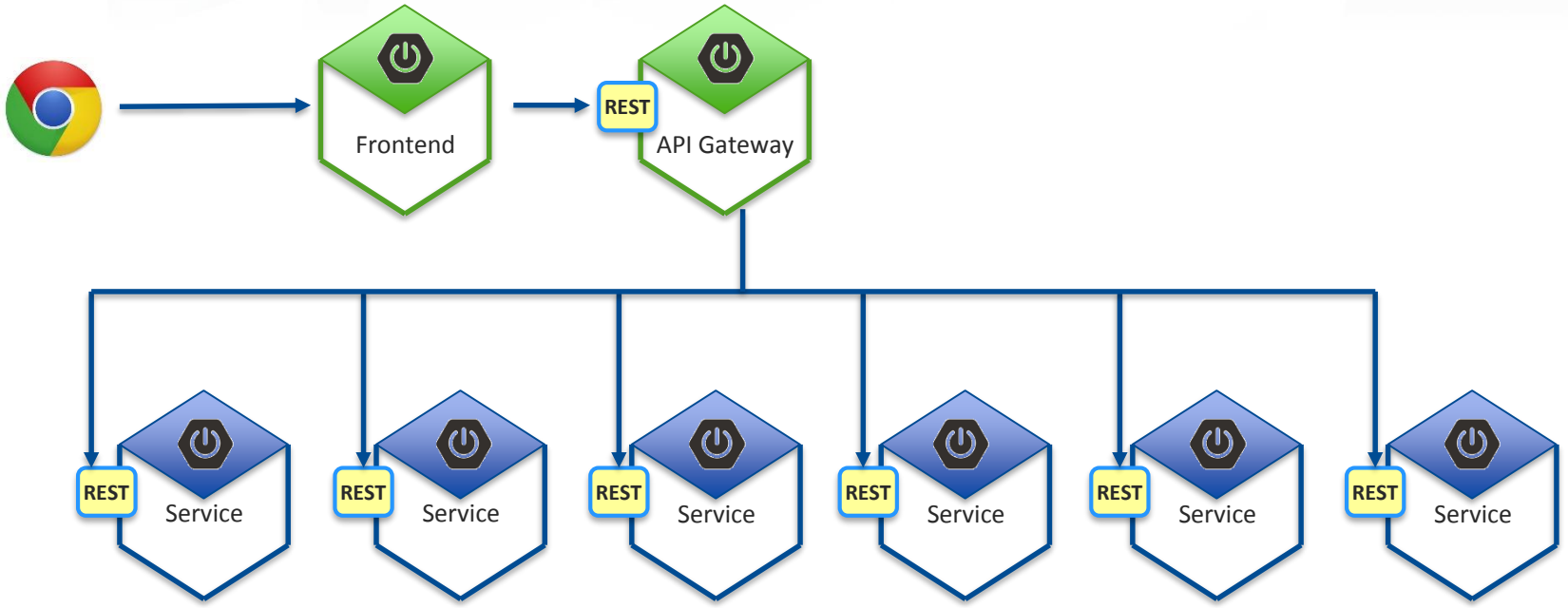
Microservices auf der grünen Wiese

- 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



Angular 2+

spring BOOT

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 geschlossen Bereiche der Anwendung umsetzen
- Wichtig: Lose Kopplung (Performanz, Resilienz)

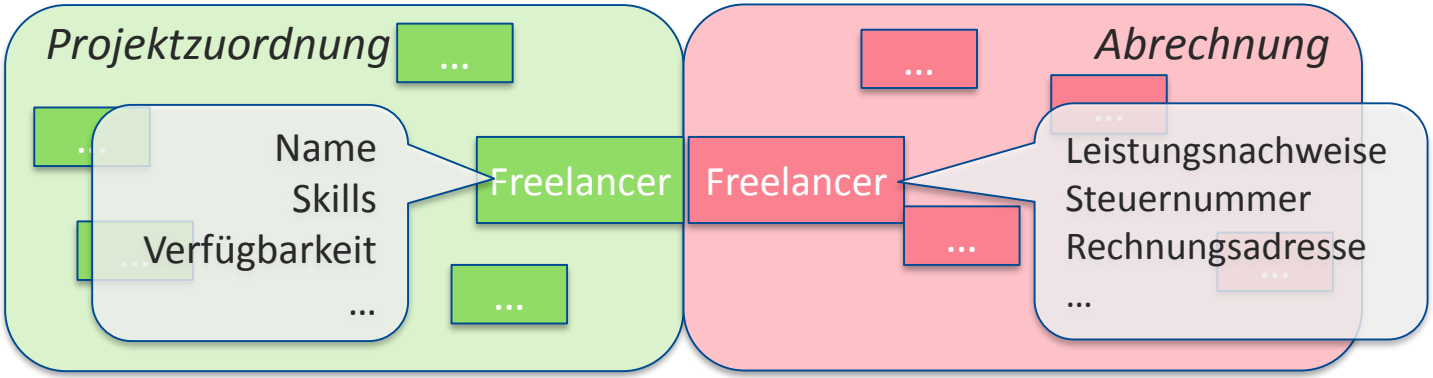
Domain-driven Design (DDD) ist eine Herangehensweise an die Modellierung komplexer Software.
Die Modellierung der Software wird dabei maßgeblich von den umzusetzenden **Fachlichkeiten** der **Anwendungsdomäne** beeinflusst.

(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: Anfrage geht ein, Freelancer wird beauftragt, Freelancer erstellt Abrechnung, …
- System/Benutzer-Aktionen ableiten
- Beteiligte Entitäten ableiten
- 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

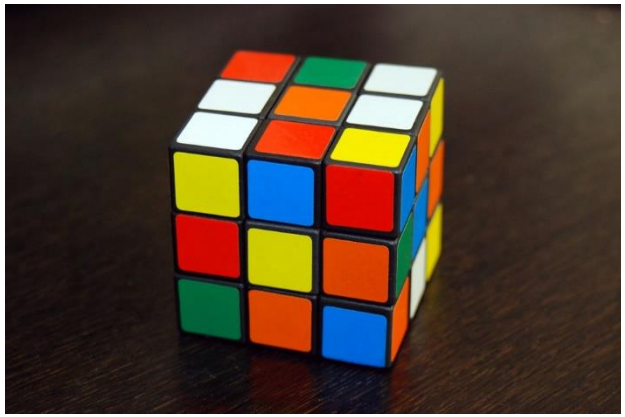
Nebeneffekt:

- 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
- Testvorgehen
- Entwicklungsprozess



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?

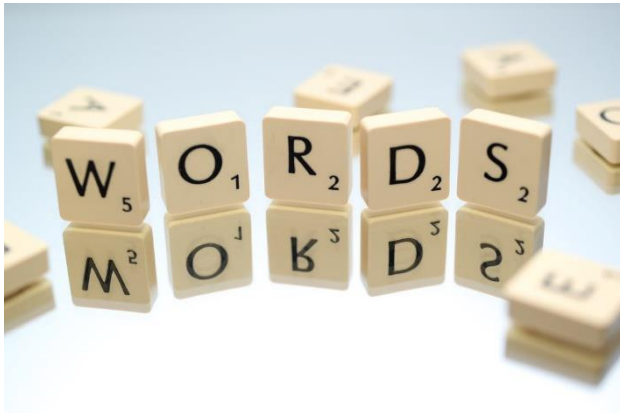
Im Regelfall: Entkopplung und Daten per Messaging replizieren

Gemeinsamer Code: Hohe Kopplung vs. Redundanz

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

Nachteile:

- Wiederverwendung erschwert unabhängige Deployments
- Verteilter Monolith?



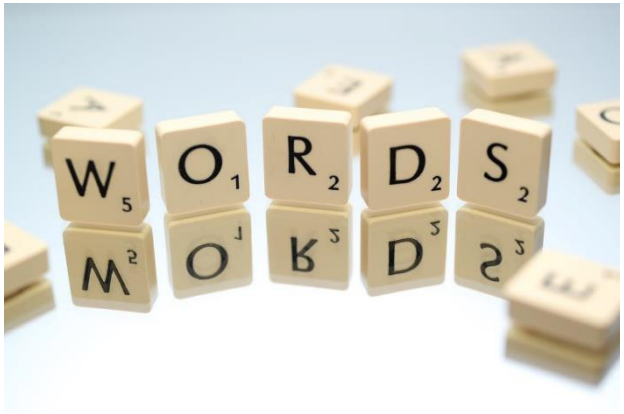
Gemeinsamer Code: Redundanz

fachlicher Code: keine Wiederverwendung

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

technischer Code: Wiederverwendung (commons-Libs)

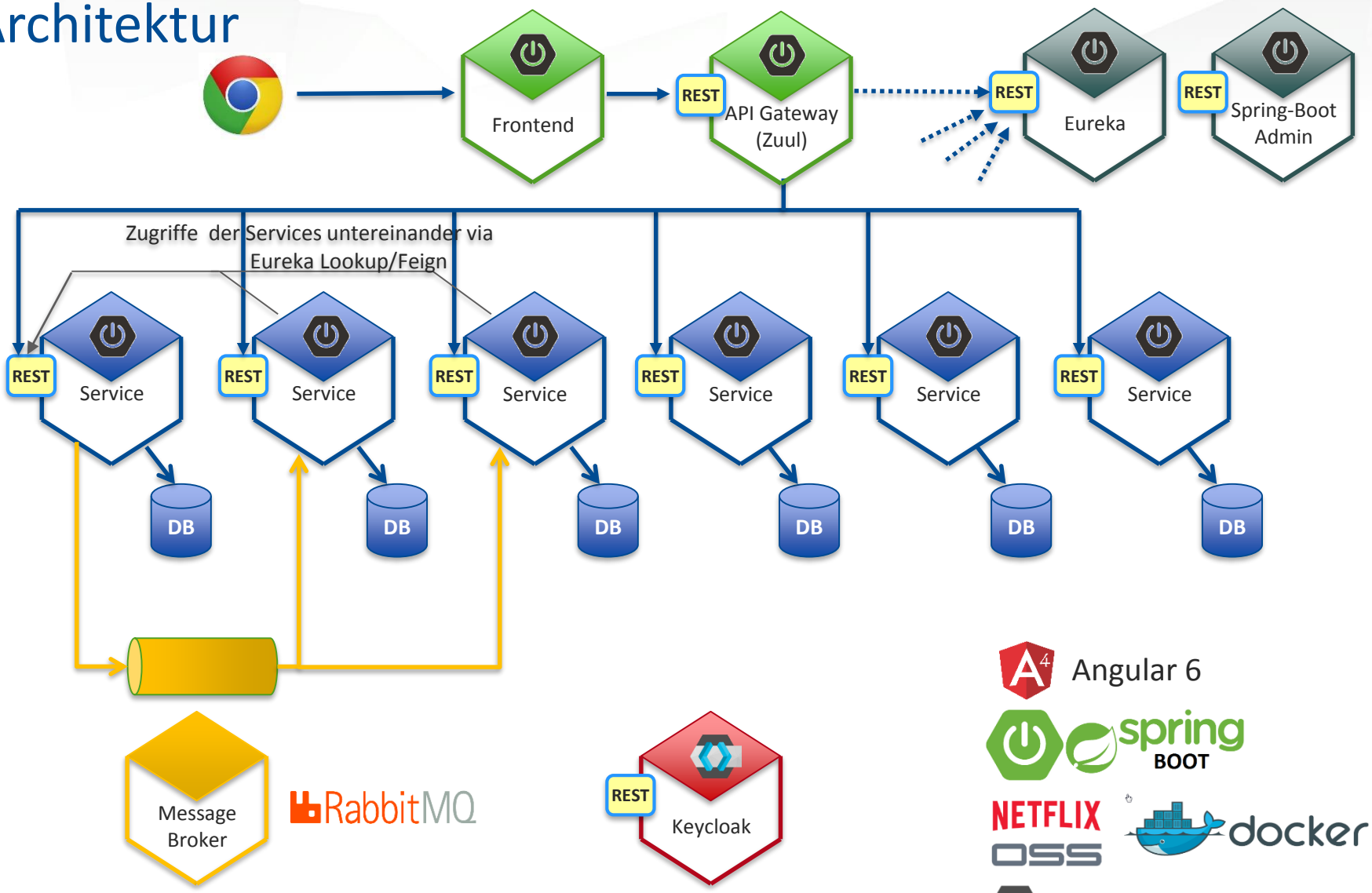
- Beispiel: erweiterbare Enums, Validierung, …
- commons-Libraries sind versioniert
  - Jeder Service entscheidet, wann er die Version aktualisiert
  - man kann nur den Service ausliefern, der die Änderung braucht
- Aber: früher oder später ist Update nötig



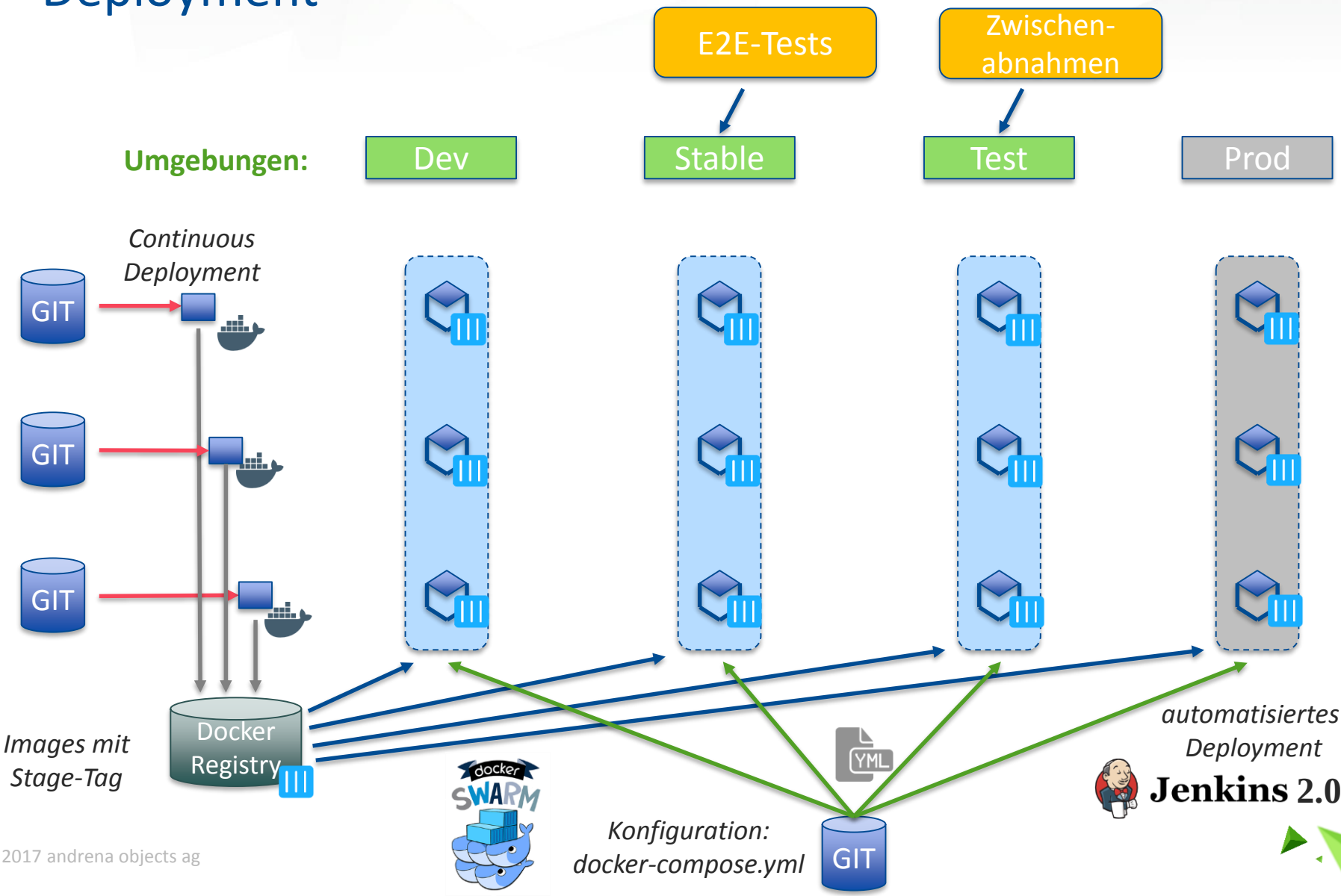
fachlich: Entkopplung durch Duplikation, technisch: Wiederverwendung

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
  - Frontend-Logs: sentry
- Zur Korrelation einzelner Anfragen: Vorgangs-ID mit im Log speichern (Toolunterstützung: Spring Cloud Sleuth)

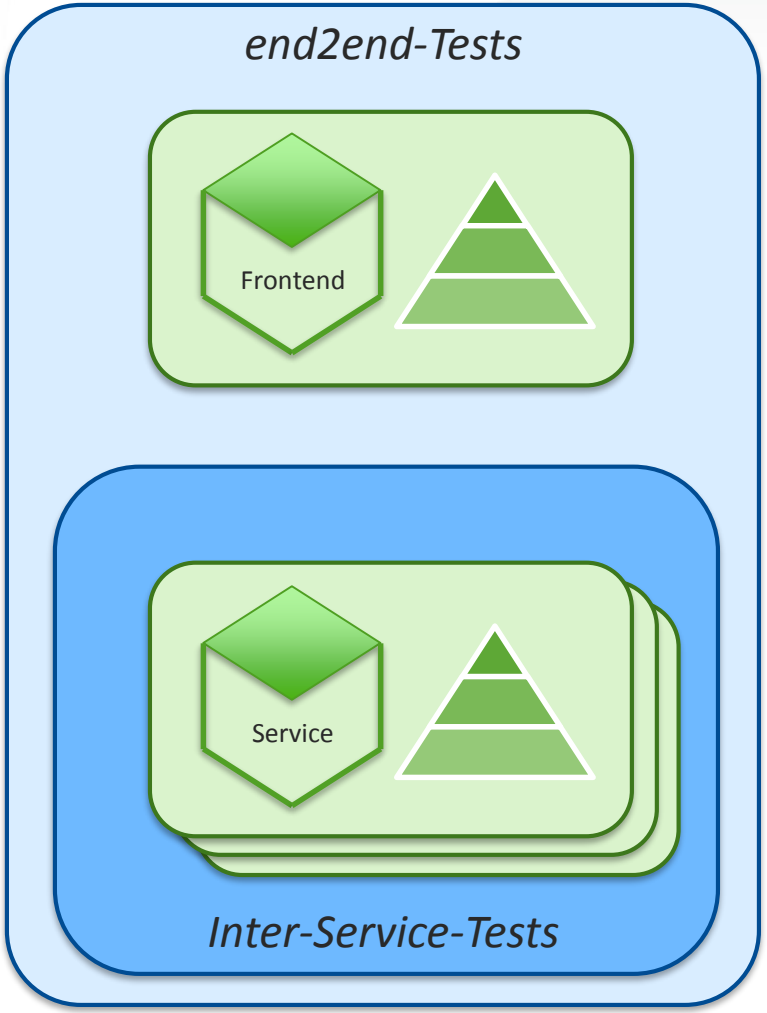
Unser Testvorgehen

Innerhalb eines Services

- Testpyramide

Gesamtsystem

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



Unser Entwicklungsprozess: Scrum

Sprint: 2 Wochen

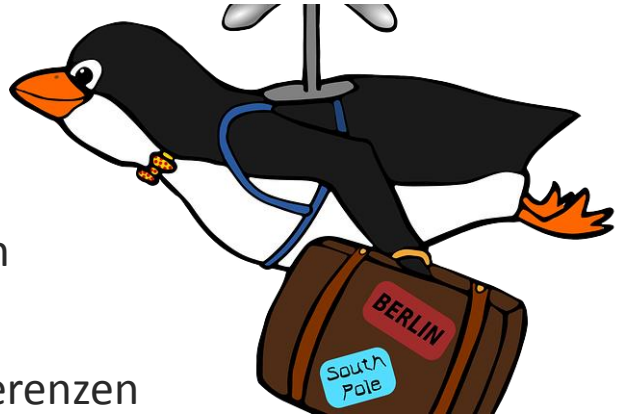
- Konsequent 4AP
- wöchentliche Refinements
- sobald Story fertig: Deployment auf Test-Umgebung
  - dadurch schon im Sprint Feedback vom Fachbereich möglich

Nach jedem Sprint:

- auslieferfähiges Inkrement
- Sprint Review erfolgt auf Test-Umgebung
- regelmäßige SQI-Messung (SQI: 85)

Technisch: Datenmigration vom monolithischen Altsystem zur Microservice-Anwendung

Datenmigration vom Monolithen zum Microservice-System



Altsystem:

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

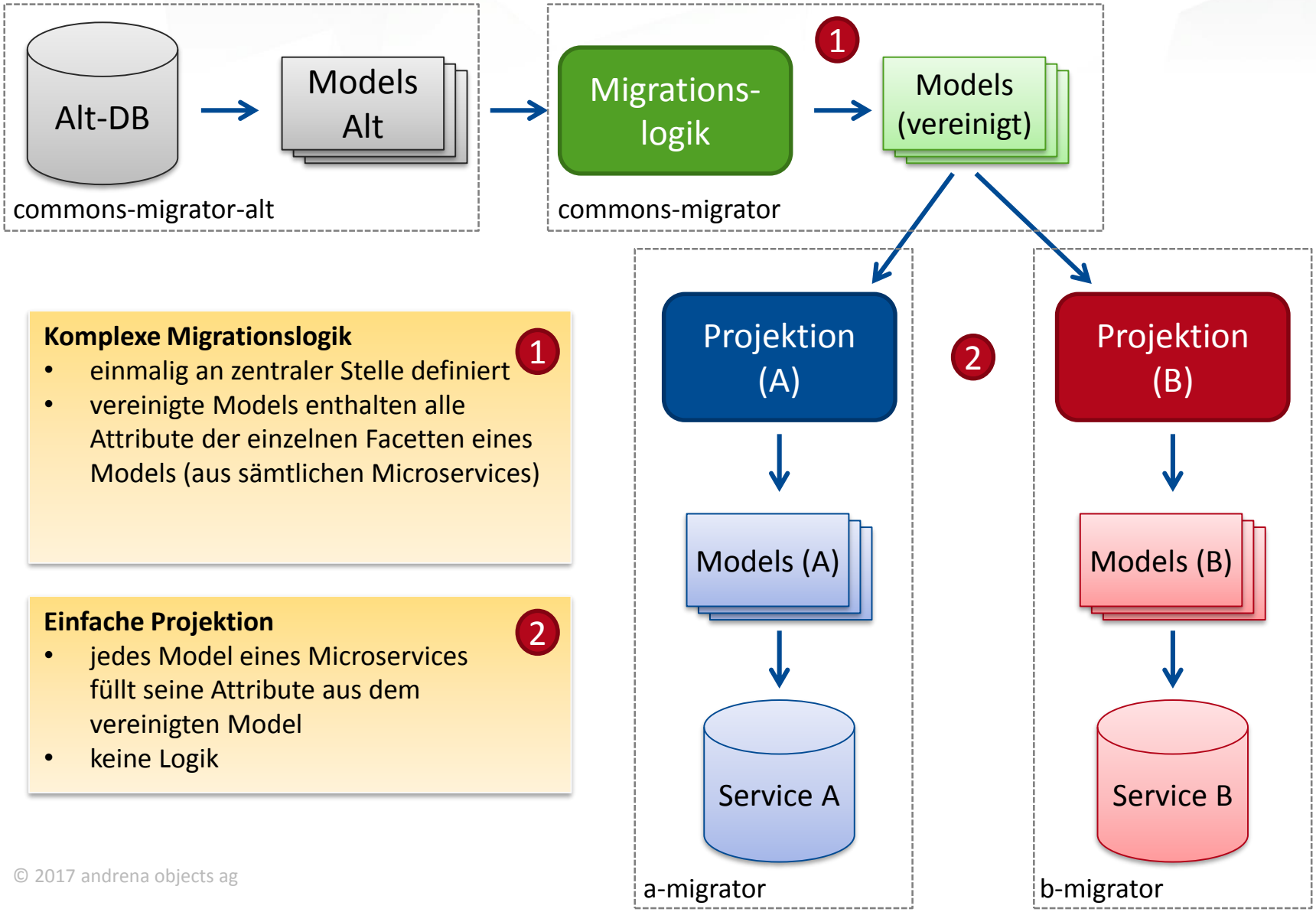
Zielsystem:

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

Herausforderungen:

- Bei der Migration Konsistenz gewährleisten, ohne einen Über-Migrator zu verwenden (der alle Zielobjekte und Zielschemata kennt)
- Zeitfenster für Migration: 1 Wochenende

Architektur der Migratoren



Komplexe Migrationslogik 1

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

Einfache Projektion 2

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

Migration technischer IDs

Altsystem: sequenzielle numerische IDs

- Eigene Sequenz/Nummernkreis pro Objekttyp

Neuanwendung: zufällige UUIDs (Typ 4)

- Erlauben keine Rückschlüsse auf Datenbestand

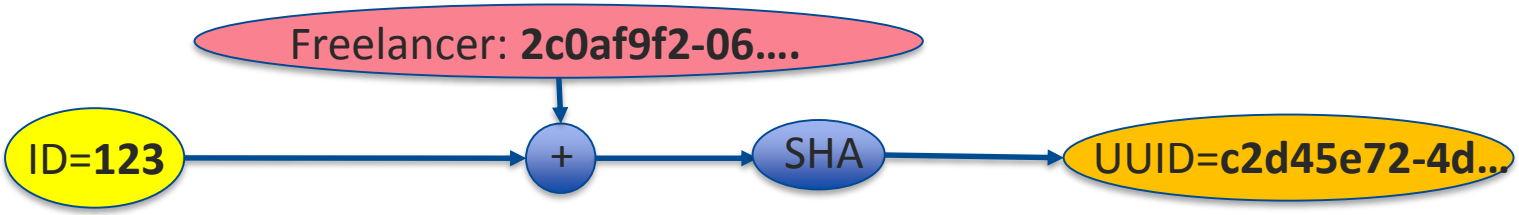
Herausforderung bei Migration:

- Services sollen unabhängig migrierbar sein
- ➢ Brauchen eindeutiges Mapping von alter ID auf UUID um referenzielle Integrität sicherzustellen (Foreign Keys, Referenzen zwischen Services)

Migration technischer IDs

Umsetzung:

- Für migrierte Daten werden namensbasierte UUIDs (Typ 5) verwendet
- commons-migrator enthält feste UUID pro Sequenz als „Namespace“
- „Name“ aus String-Repräsentation der alten ID
- ID + Namespace-UUID werden SHA-1-geshashed und aus diesen Bytes neue UUID erzeugt



- Eindeutige Zuordnung ID → UUID
- Keine Kollision bei gleicher ID verschiedener Sequenzen
- Kein Rückschluss von UUID auf ursprüngliche ID oder „angrenzende“ IDs möglich

Umsetzung der Migration

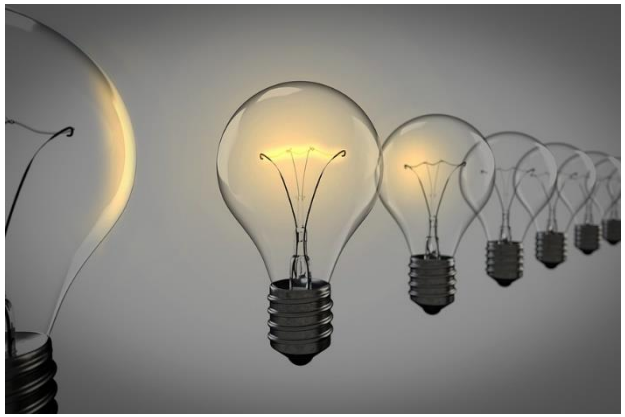
- Je Service eigener Migrator
  als Spring-Boot-Konsolenanwendung
- Eigener Docker-Stack für Migratoren
- Datenhaltung der Services unabhängig
  - ➢ Migratoren können parallel laufen
  - ➢ Zeitfenster (48 Std) kann eingehalten werden
- Ausnahme: Dokumenten-Migrator
  - Es werden sehr große Datenmengen migriert (viele Dateien, einige 100 GB)
  - Zeitbedarf größer als zur Verfügung stehendes Zeitfenster
  - ➢ inkrementelle Migration implementiert

Fazit

Vorteile einer Microservice-Anwendung

- Einzelne Services sind **wenig komplex**
- 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 (API-DTOs + replizierte Daten)

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:

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