FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Masterarbeit in Informatik

Eine prototypische Implementierung zur Erkennung von Architekturänderungen eines verteilten Systems basierend auf unterschiedlichen Monitoring Datenquellen

Patrick Schäfer

FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Masterarbeit in Informatik

Eine prototypische Implementierung zur Erkennung von Architekturänderungen eines verteilten Systems basierend auf unterschiedlichen Monitoring Datenquellen

A prototypical tool to discover architecture changes based on multiple monitoring data sources for a distributed system

Autor:	Patrick Schäfer
Aufgabensteller:	Prof. Dr. Florian Matthes
Betreuer:	M.Sc. Martin Kleehaus

Abgabedatum: 15. November 2017

Ich versichere, dass ich diese Masterarbeit selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, 15.11.2017

Patrick Schäfer

Kurzfassung

Verteilte Architekturen gewinnen von Jahr zu Jahr an Bedeutung, da sie agiles Projektmanagement und horizontale Skalierung moderner IT-Systeme unterstützen. Die verteilten Systeme führen jedoch zu hoher infrastruktureller Komplexität und Intransparenz. Um dem entgegen zu wirken, wurden Architekturerkennungssysteme auf Basis verschiedener Technologien entwickelt. Diese sind spezialisiert auf die Erkennung von Applikationen und IT-Systemen oder auf das Ermitteln von Geschäftsprozessen und Aktivitäten.

Diese Arbeit beschäftigt sich mit der prototypischen Entwicklung eines Architekturerkennungssystems, welches im Unterschied zu bestehenden Systemen sowohl technische Entitäten als auch Geschäftsentitäten einer Architektur berücksichtigt und durch ein Modell Abhängigkeiten und Beziehungen zwischen diesen bereitstellt. Basis des prototypisch implementierten Discovery-Dienstes sind die Datenquellen eines Service Repository sowie die Technologie des Distributed Tracing.

Es wird aufgezeigt, wie die verschiedenen Datenquellen verarbeitet und zusammengeführt werden können, um hieraus ein umfassendes Architekturmodell zu ermitteln. Des Weiteren wird vorgestellt, wie dieses Modell um Entitäten der Geschäftswelt erweitert werden kann und wie Beziehungen etabliert werden können.

Das Ergebnis des Prototyps ist ein umfangreiches Architekturmodell, welches dynamisch erkannt und über ein Interface bereitgestellt wird. Architekturänderungen werden in Echtzeit ermittelt und durch Streaming-Technologien bereitgestellt. Die vorgestellte Modellrepräsentation in Form einer angepassten Adjacency-Matrix stellt eine alternative Form der Architekturvisualisierung dar, welche auch komplexe Beziehungen übersichtlich repräsentieren kann.

Inhaltsverzeichnis

Tabellenverzeichnis III

Abbildungsverzeichnis V

Abkürzungsverzeichnis VII

1. Einleitung 1

2. Enterprise Architecture 5

2.1 Definition einer Enterprise Architecture 5

2.2 Enterprise-Architecture-Ebenen 6

2.2.1 Business-Layer 7

2.2.2 Application-Layer 8

2.2.3 Technology-Layer 9

2.2.4 Beziehungen und Abhängigkeiten 9

3. Microservices 11

3.1 Grundlagen zu Microservices 11

3.2 Microservices im Kontext einer Enterprise Architecture 13

3.2.1 Modellierung von Microservices im EA-Umfeld 13

3.2.2 Microservice-Referenzarchitekturmodell im EA-Kontext 15

3.3 Microservice-Monitoring durch Distributed Tracing 19

3.3.1 Funktionsweise des Distributed-Tracing-Systems Zipkin 21

3.3.2 Entstehung eines Spans des Zipkin-Systems 23

3.3.3 Span-Datenmodell eines Zipkin-Systems 25

4. Stand der Technik 29

4.1 Service Discovery durch Microservice Repositories 29

4.2 Process Discovery durch Business Process Mining 31

4.3 Architekturerkennung über Monitoringdaten 33

4.4 Architekturerkennung durch statische und dynamische Analyse 35

4.5 Abgrenzung 39

5. Prototypische Implementierung Echtzeit-Architekturerkennung 41

5.1 Einsatzumfeld der prototypischen Implementierung....42

5.2 Systemkomponenten der prototypischen Implementierung....44

5.2.1 Architekturerkennungsservice....46

5.2.2 Webanwendung....55

5.2.3 Apache Kafka und Websocket Bridge....62

5.3 Architekturmodell....65

5.3.1 Komponenten....68

5.3.2 Revisionen....70

5.3.3 Relationen....72

5.4 Implementierungsdetails des Architekturerkennungsservice....77

5.4.1 Applikationsstruktur der Java-Applikation....77

5.4.2 Datenmodell....79

5.4.3 Verarbeitung der Distributed-Tracing-Daten....83

5.4.4 Verarbeitung der Daten des Eureka Service Repository....86

6. Evaluation 89

6.1 Evaluations-Umfeld....89

6.2 Durchführung der Evaluation....94

6.2.1 Evaluation der Funktionsweise....94

6.2.2 Evaluation der Performance- und Ressourcenauswirkungen....108

6.3 Limitierungen....115

7. Fazit und Ausblick 119

Literaturverzeichnis 123

Anhang A: XML-Response von /eureka/v2/apps IX

Anhang B: Json-Daten eines Zipkin Spans X

Anhang C: Exemplarische Antwort des Architektur-Endpunkts XI

Anhang D: E-R Diagramm des Architekturerkennsungsservice XII

Tabellenverzeichnis

Tabelle 1: Architektonisches Framework (Jonkers et al., 2003) 6
Tabelle 2: Datenmodell eines Zipkin-Spans 26
Tabelle 3: Auszug eines Event-Logs (van der Aalst, 2015) 32
Tabelle 4: Vergleich der Architekturentitäten ausgewählter Technologien 39
Tabelle 5: Felder einer Json-Meldung zur Architekturänderung 63
Tabelle 6: Implizite Bedeutung einer Relation zwischen Komponententypen 75
Tabelle 7: Konfiguration der Hosts 93
Tabelle 8: Zuordnung von Geschäftsaktivitäten und Services 102
Tabelle 9: Revisionen-Tabelle der Datenbank 107
Tabelle 10: Antwortzeiten instrumentierter versus nicht instrumentierter Host 110
Tabelle 11: CPU-Auslastung (in %) durch Services während der Testzeiträume 112
Tabelle 12: Netzwerklast (in kB/s) durch Services während der Testzeiträume 114

Abbildungsverzeichnis

Abbildung 1: Business-Layer-Modell 7
Abbildung 2: Application-Layer-Modell einer Microservice-Architektur 8
Abbildung 3: Beziehungen zwischen Ebenen einer EA 10
Abbildung 4: Vereinfachte Microservice-Architektur 12
Abbildung 5: MS-Referenzarchitekturmodell (Yale et al., 2016) 15
Abbildung 6: Zipkin-Datenfluss (The OpenZipkin Authors, 2017a) 22
Abbildung 7: Zipkin-UI-Trace-Visualisierung (The OpenZipkin Authors, 2017c) 23
Abbildung 8: Sequenzdiagramm einer Spanerzeugung (The OpenZipkin Authors, 2017a) 24
Abbildung 9: Eureka-Architektur (Netflix Inc., 2017b) 30
Abbildung 10: Abhängigkeitsdiagramm durch Kieker (Hasselbring, 2017) 34
Abbildung 11: Visualisierung einer Anwendungstopologie durch Dynatrace (Dynatrace LLC., 2017b) 35
Abbildung 12: MicroART-Artefakte (Granchelli et al., 2017a) 36
Abbildung 13: Struktur des MicroART-Systems (Granchelli et al., 2017a) 38
Abbildung 14: Referenzumfeld des Prototyps 43
Abbildung 15: Systemkomponenten der prototypischen Implementierung 45
Abbildung 16: Vereinfachtes Schema des Architekturerkennungsservice 46
Abbildung 17: Datenflussdiagramm eines Zipkin-Servers 49
Abbildung 18: Ausschnitt des Datenflussdiagramms der Architekturerkennung 50
Abbildung 19: Abgleich Datenmodell mit Eureka 52
Abbildung 20: Seite „Adjacency Matrix“ der Webanwendung 56
Abbildung 21: Modellierung von Prozessen und Aktivitäten 58
Abbildung 22: Webbasierte Regeldefinition durch manuelles Zuordnen 60
Abbildung 23: Webbasierte Regeldefinition durch reguläre Ausdrücke 61
Abbildung 24: Webbasierte Verwaltung der Regelsammlung 62
Abbildung 25: Funktionsweise der Websocket Bridge 65
Abbildung 26: Schematische Darstellung der Überführung zum Architekturmodell 66
Abbildung 27: Verkürztes Architekturmodell zu zwei Zeitpunkten 67
Abbildung 28: Komponenten des Architekturmodells 68
Abbildung 29: Revisionen im zeitlichen Verlauf 71
Abbildung 30: Gültigkeitszeitraum einer Relation 72

Abbildung 31: Relationenbaum des Architekturmodells 73
Abbildung 32: Strukturierung des Architekturerkennungsservice 78
Abbildung 33: Direkte und indirekte Relationen einer Aktivität 81
Abbildung 34: Klassendiagramm des Datenmodells ohne Relationen 82
Abbildung 35: Veränderung des Datenmodells durch schrittweise Span-Analyse 84
Abbildung 36: Prozessablaufdiagramm der Evaluationsapplikation 90
Abbildung 37: Abhängigkeiten des Microservice-Clusters 91
Abbildung 38: Microservice-Architektur des Evaluationsumfelds 93
Abbildung 39: Adjacency-Matrix nach Start von Host 1 95
Abbildung 40: Adjacency Matrix-nach Start von Host 2 96
Abbildung 41: Adjacency-Matrix nach Start aller Hosts 97
Abbildung 42: Adjacency-Matrix nach Evaluationsschritt 2 98
Abbildung 43: Modellierung des Evaluationsgeschäftsprozesses 100
Abbildung 44: Ausschnitt der Adjacency-Matrix nach Prozessmodellierung 101
Abbildung 45: Angelegte Regeln des Aktivitäts-Mappings 102
Abbildung 46: Adjacency-Matrix nach Evaluationsschritt 3 103
Abbildung 47: Adjacency-Matrix nach Abschaltung des Zuul-Service 105
Abbildung 48: Adjacency-Matrix nach Neustart des Zuul-Service 106
Abbildung 49: Gegenüberstellung der CPU-Auslastung der Testzeiträume 109
Abbildung 50: Durchschnittliche Antwortzeiten nicht instrumentierter versus instrumentierter Host (BlazeMeter, 2017) 110
Abbildung 51: CPU-Auslastung durch Services während der Testzeiträume 112
Abbildung 52: Netzwerklast (in kB/s) durch Services während der Testzeiträume 113

Abkürzungsverzeichnis

API	Application Programming Interface
APM	Application Performance Monitoring
CSS	Cascade Style Sheets
DDD	Domain Driven Design
DNS	Domain Name System
EA	Enterprise Architecture
ERP	Enterprise-Resource-Planing
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
ID	Identifier
IP	Internet Protocol
IT	Informationstechnik
JPA	Java Persistence API
JSON	JavaScript Object Notation
MS	Microservice
REST	Representational State Transfer
SLA	Service Level Agreements
SOA	Service-Orientierte Architektur
SQL	Structured Query Language
UI	User Interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VM	Virtual Machine
XML	Extensible Markup Language

1. Einleitung

Agilität in der Entwicklung von Informationssystemen gewinnt zunehmend an Bedeutung. Durch den Markt werden fortwährend schnellere Reaktionszeiten seiner Akteure auf Veränderungen erwartet, was zu Veränderungen des Projektmanagements bei der Entwicklung von Informationstechnologien (IT) führte. Klassische Managementmethoden, wie das Wasserfallmodell, sind aufgrund ihres langfristigen Planungshorizonts und der damit verbundenen Arbeitsweise ungeeignet, um dynamisch auf Veränderungen zu reagieren. Das agile Projektmanagement hingegen schafft durch einen kurzen Planungshorizont und ständige Involvierung von Marktakteuren die Grundlage, um mit hoher Geschwindigkeit auf Anforderungsänderungen zu reagieren.

Doch agile Methoden erfordern auch neue Ansätze in der Softwarearchitektur. Herkömmliche monolithische Systeme wurden durch Teams mit großem Personalaufwand realisiert. Dies ist im agilen Umfeld mit hohem Aufwand in Kommunikation und Projektmanagement verbunden, da einzelne Subsysteme in Abhängigkeit zueinander stehen und Veränderungen sowie deren Auswirkungen stets teamübergreifend besprochen werden müssen. Daher gewannen in den vergangenen Jahren verteilte Architekturen immer mehr an Bedeutung.

Diese erlauben neben einer horizontalen Systemskalierung das Arbeiten an Teilprojekten, welche durch autarke Teams realisiert werden können. Mit dem Architekturpattern der Microservices durchdringt eine Strategie den Markt, die eine Vielzahl autarker Systeme vorsieht. Diese kommunizieren miteinander durch implementierungsneutrale Schnittstellen und werden jeweils zur Lösung eines klar definierten Problems erzeugt. Die Summe der Einzelsysteme stellt ein gesamtheitliches System dar, welches äquivalent zu den monolithischen Systemen eine übergeordnete Problemstellung löst. Die Gliederung in kleine Einzelsysteme ermöglicht, dass Anforderungsänderungen durch ein dediziertes Team in kurzer Zeit und mit massiv reduziertem Planungs- und Kommunikationsaufwand umgesetzt werden können.

Doch mit den verteilten Systemen entstehen auch neue Herausforderungen. Durch die verteilten Architekturen sind infrastrukturelle Abhängigkeiten sowie die Kom-

plexität der Architektur enorm gestiegen. Das Monitoring der einzelnen Systeme ist mit den Monitoring-Tools der Vergangenheit aufgrund der Vielzahl von Einzelsystemen nur noch beschränkt möglich. Die große Menge an Systemen, welche innerhalb variierender Belastungszeiträume durch automatisches Deployment ständig im Wandel sein kann, erschwert das Erfassen des aktuellen Standes der Infrastruktur und der ihr zugrundeliegenden Architektur.

Aus dieser Problemstellung heraus wurden Systeme entwickelt, welche basierend auf verschiedenen Technologien eine Architekturerkennung durchführen. Diese schaffen jedoch nur für Teilmengen einer Unternehmensarchitektur Transparenz. Während einige Systeme Geschäftsprozesse und deren Verlauf ermitteln, adressieren andere Systeme die Erkennung von Applikationen und Hosts der IT-Ebene. Beziehungen und Abhängigkeiten zwischen Entitäten der Geschäfts- und der IT-Ebene werden durch diese jedoch nicht sichtbar. Würden diese in einem gemeinsamen Modell vorliegen, könnte dies die Basis neuer Analysemöglichkeiten darstellen. Beispielsweise könnten die Auswirkungen einer technischen Störung auf Geschäftsentitäten ermittelt und somit Prozesse auf deren technische Stabilität hin optimiert oder Risikoanalysen durchgeführt werden.

Diese Arbeit befasst sich mit der Erzeugung eines Architekturmodells für verteilte Systeme. Das Architekturmodell soll vollumfänglich im Sinne der Enterprise-Architecture-Layer aufgebaut sein und möglichst in Echtzeit Architekturveränderungen detektieren sowie kommunizieren. Es soll für Architekturen kleinen und großen Umfangs visualisiert werden und durch die Bereitstellung von Schnittstellen als Basis für weitere Analyse- und Visualisierungstools dienen. Zu diesem Zweck wird ein Prototyp konzipiert und entwickelt und in den anschließenden Kapiteln eingehend beschrieben. In diesem Rahmen widmet sich die Arbeit der Beantwortung folgender Forschungsfragen:

- Wie kann eine Microservice-Architektur durch Anwendung von Distributed Tracing ermittelt werden?
- Welche Arten von Beziehungen bestehen zwischen Architekturkomponenten und wie können diese automatisch erkannt werden?
- Wie können Nebenläufigkeit und Synchronität erkannt werden?

- Wie können Veränderungen der Microservice-Architektur entdeckt werden?
- Wie kann ein smartes User Interface bereitgestellt werden, um Business-Services mit Business-Semantik zu erweitern?

Die Arbeit gliedert sich in sieben Kapitel, welche theoretische Grundlagen sowie Konzept, Implementierungsdetails und Evaluation des entwickelten Prototyps vermitteln. In Kapitel 2 wird der Begriff der Enterprise Architecture bündig beleuchtet und anschließend werden die Ebenen einer Enterprise Architecture eingehend beschrieben, um zu vermitteln, welche Entitäten das zu erzeugende Architekturmodell des Prototyps umfassen sollte. In Kapitel 3 wird die Applikationsebene der Enterprise Architecture in Form von Microservices beschrieben. Dies umfasst die Microservice-Architektur in Form eines Referenzmodells sowie das Verfahren des Distributed Tracing, welches zum Monitoring von verteilten Systemen eingesetzt wird. Kapitel 4 stellt verschiedene Möglichkeiten der Architekturerkennung nach dem derzeitigen Stand der Technik vor und führt eine Abgrenzung der vorliegenden Arbeit von den vorgestellten Systemen durch. Nachdem Grundlagen und der Stand der Technik vorgestellt sind, wird in Kapitel 5 die prototypische Implementierung der Echtzeitarchitekturerkennung beschrieben. Dies umfasst die Beschreibung der entwickelten Systeme, des erzeugten Architekturmodells sowie Implementierungsdetails bezüglich der Algorithmen und des Datenmodells. Kapitel 6 basiert auf der Prüfung des Prototyps bezüglich korrekter Funktionalität und seiner Evaluation in einem Testumfeld. Auch die Messung seiner Auswirkungen auf die Performance der zu entdeckenden Infrastruktur, sowie die Limitierungen, welchen der Prototyp unterliegt, werden beschrieben. Die Arbeit schließt mit einem Fazit sowie einem Ausblick bezüglich der angewandten Technologie in Kapitel 7.

2. Enterprise Architecture

Um Änderungen einer Geschäftsarchitektur ganzheitlich zu erfassen, muss eine Architektur über ihre IT-Komponenten hinaus betrachtet werden. Dieses Kapitel dient der Heranführung an die gesamtheitlich abbildende Enterprise Architecture (EA) sowie deren Ebenen, Elemente und Beziehungen.

2.1 Definition einer Enterprise Architecture

Der ISO/IEC/IEEE-Standard 42010 definiert den Begriff der Architektur wie folgt: „fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution" (IEEE, 2011). Demnach beschreibt eine Architektur Konzepte und Eigenschaften eines Systems sowie dessen Elemente und Beziehungen.

Eine EA als Spezialisierung beschreibt folglich ein System in einem Unternehmensumfeld. Durch die Gartner-Gruppe wurde folgende Begriffsdefinition für eine EA definiert:

> „Enterprise architecture is the process of translating business vision and strategy into effective enterprise change by creating, communicating and improving the key requirements, principles and models that describe the enterprise's future state and enable its evolution. The scope of the enterprise architecture includes the people, processes, information and technology of the enterprise, and their relationships to one another and to the external environment. Enterprise architects compose holistic solutions that address the business challenges of the enterprise and support the governance needed to implement them." (Anne Lapkin, 2008)

Der Definition folgend wird eine EA als fortlaufender Prozess verstanden, in dessen Rahmen die Vision und Strategie des Unternehmens in effektive Unternehmensänderungen überführt werden. Die Hauptziele einer EA können wie folgt zusammengefasst werden (Saat, Aier & Gleichauf, 2009):

- Dokumentation des Ist-Zustandes der Unternehmensstruktur mit ihren Business- und IT-Artefakten sowie deren Beziehungen
- Analyse der bestehenden Abhängigkeiten und Beziehungen des Modells im Ist-Zustand
- Planung und Vergleich von Soll-Modellen für Zukunftsszenarien sowie das Erzeugen von hieraus abgeleiteten Transformationsprojekten und Programmen, um die angestrebte EA zu erhalten

2.2 Enterprise-Architecture-Ebenen

Da im Gegensatz zu Architektur-Modellen aus der IT-Landschaft in einer EA neben IT-Artefakten außerdem Business-Artefakte wie Geschäftsprozesse, Unternehmensziele und Performanceindikatoren abgebildet werden, ist durch eine EA eine Vielzahl verschiedener Domänen und Artefakte abzubilden (Winter & Fischer, 2006), was zu einer hohen Komplexität in der Modellierung der Architektur führt.

EA Frameworks adressieren diese Komplexität. Sie gliedern die Architektur-Elemente in strukturierte Ebenen und Dimensionen (Shah & Kourdi, 2007). Diese Strukturierung reduziert die Artefakt-Vielfalt innerhalb einer Ebene und vereinfacht somit die Modellierung der Elemente und ihrer Abhängigkeiten. Nach den Ausführungen von Jonkers et al. (2003) ordnet ein EA Framework zu modellierende Artefakte den Ebenen „Business-Layer“, „Application-Layer“ sowie „Technology-Layer“ zu.

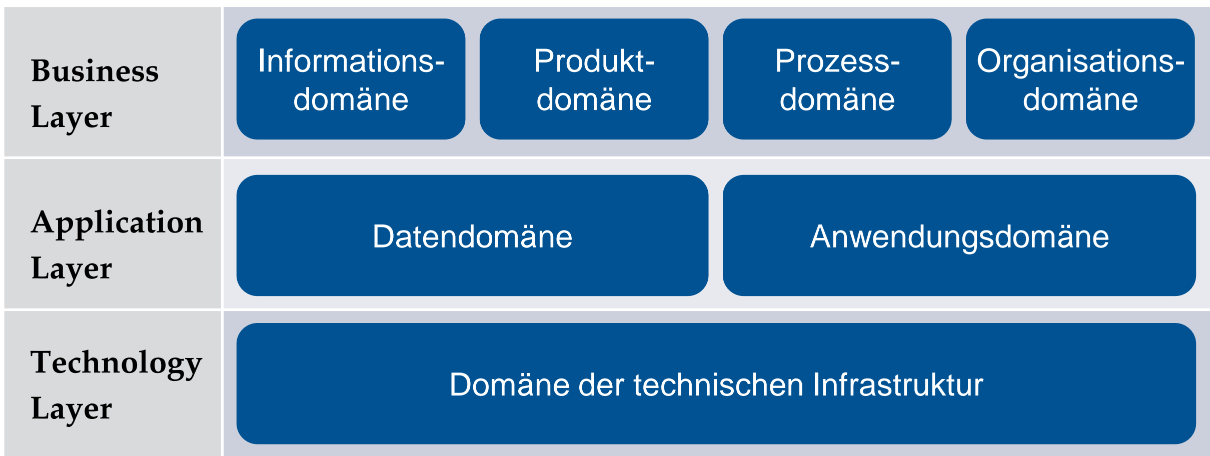


Tabelle 1: Architektonisches Framework (Jonkers et al., 2003)

Die Gesamtheit der Artefakte einer EA kann in verschiedene Domänen gegliedert werden, welche, wie in Tabelle 1 dargestellt, eindeutig in jeweils einer der genannten Ebenen verortet werden können.

Obwohl in der Praxis Architekten für die dargestellten Domänen jeweils eigene Konzepte sowie Modellierungs- und Visualisierungstechniken haben (Jonkers et al., 2003), sind in der EA neben den Domänenmodellen auch die domänenübergreifenden Beziehungen modelliert, so dass für das Unternehmen Architekturänderungen transparent in ihren gesamtheitlichen Auswirkungen werden.

In den folgenden Kapiteln werden die aufgezeigten Ebenen und deren Beziehungen weiterführend beschrieben. Dabei sind die Ausführungen zu den Ebenen und ihren Relationen zweckmäßig vereinfacht und auf diejenigen Aspekte begrenzt, die für das Verständnis dieser Arbeit relevant sind.

2.2.1 Business-Layer

Der Business-Layer bildet die erste Ebene einer EA, welche die darunterliegenden Ebenen direkt beeinflusst. Er dient zur Beschreibung von Unternehmensdaten und -strategien. Hervorzuheben ist die Modellierung von Geschäftsprozessen auf dieser Ebene. Diese werden, wie in Abbildung 1 skizziert, als Abfolge zusammenhängender Aktivitäten oder Operationen dargestellt, die von Akteuren in Form interner oder externer Kunden ausgelöst werden (Haki & Forte, 2010).

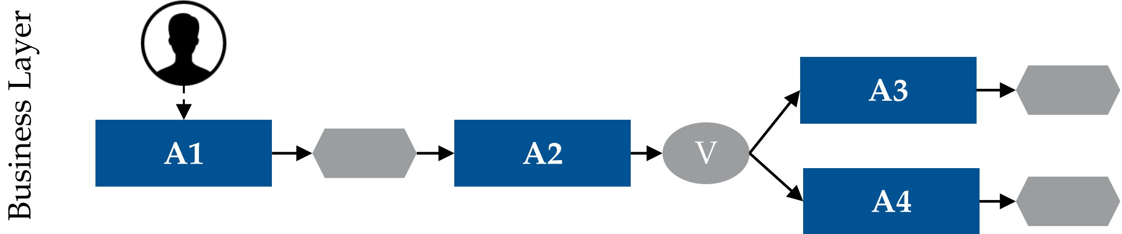


Abbildung 1: Business-Layer-Modell

Der Business-Layer umfasst folgende vier Domänen (Jonkers et al., 2003):

- Die Produktdomäne beschreibt Produkte und Services, welche durch das Unternehmen für Kunden angeboten werden.

- Die Organisationsdomäne definiert Akteure wie Mitarbeiter und Unternehmenseinheiten sowie deren Rollen innerhalb von Prozessen.
- Die Prozessdomäne beschreibt Geschäftsprozesse und -funktionen von Unternehmensprodukten (Produktdomäne).
- Die Informationsdomäne umfasst alle relevanten Informationen der Geschäftsebene.

2.2.2 Application-Layer

Der Application-Layer stellt die zweite Ebene der EA dar und ist anders als der Business-Layer nicht in der Geschäftswelt, sondern in der IT verortet. Er dient der Modellierung von Softwareapplikationen zur Verwaltung von Daten und zur Ausführung von Geschäftsfunktionalitäten. Neben den Applikationen werden ebenfalls deren Beziehungen, Abhängigkeiten und Interaktionen beschrieben (Alonso, Verdún & Caro, 2010).

Wie in Abbildung 2 dargestellt, werden in verteilten Architekturen, z. B. in Microservice-Architekturen, diejenigen Applikationen durch Services repräsentiert, die Geschäftsaktivitäten und -funktionen einzeln oder im Verbund mit in Beziehung stehenden Services technisch unterstützen oder gar vollständig abbilden.

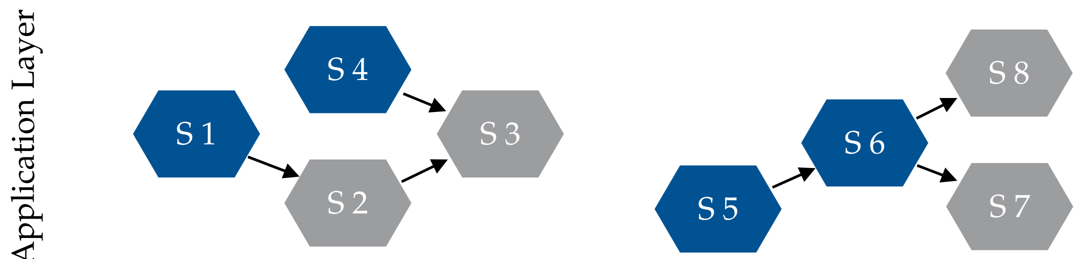


Abbildung 2: Application-Layer-Modell einer Microservice-Architektur

Der Application-Layer umfasst die folgenden zwei Domänen (Jonkers et al., 2003):

- Die Datendomain, welche Informationen beschreibt, die zur maschinellen Verarbeitung geeignet sind.
- Die Anwendungsdomäne, welche Software-Anwendungen beschreibt, die Geschäftsprozesse und -funktionen technisch unterstützen.

2.2.3 Technology-Layer

Der Technology-Layer stellt die unterste Ebene der EA dar und ist ebenfalls eine Ebene des IT-Horizonts. Er beschreibt vorhandene technische Systeme, Infrastruktur und Kommunikationstechnologien, die zum Betrieb der Anwendungen des Application-Layer benötigt werden. Dies umfasst außerdem das Fassungsvermögen der Systeme sowie deren Vernetzung, Netzwerktopologien und eingesetzte Kommunikationsprotokolle (Alonso et al., 2010).

Dem Technology-Layer wird die Domäne der technischen Infrastruktur zugeordnet, welche Hardware-Plattformen sowie die Infrastruktur zur technischen Kommunikation beschreibt (Jonkers et al., 2003).

2.2.4 Beziehungen und Abhängigkeiten

Die Ebenen einer EA bedingen einander zur Erfüllung von Geschäftsprozessen und -funktionen, welche dem übergeordneten Ziel der Produkterzeugung dienen. Wie in Abbildung 3 dargestellt, existieren sowohl zwischen den Elementen einer Ebene als auch Ebenen-übergreifend Beziehungen und Abhängigkeiten. Die dargestellten Intralayer-Beziehungen der Ebenen sind jeweils unterschiedlicher Bedeutung. Auf Basis des Business-Layers sind dies Vorgänger- und Nachfolgerbeziehungen, welche mögliche Pfade zum Durchlaufen eines Prozesses beschreiben. Die Relationen des Application-Layers hingegen stellen im Umfeld der verteilten Systeme keine Verarbeitungsreihenfolge, sondern Abhängigkeiten dar. Eine Anwendung bedingt bei einer bestehenden Beziehung die jeweils andere zur Erfüllung ihrer Aufgabe, steht also mit jener in Kommunikation und im Datenaustausch. Verbindungen des Technology-Layers beschreiben eine „Teil-von-Beziehung". Als Beispiel sei die Verbindung von virtuellen Maschinen und deren Hostsystemen genannt.

Auch den Ebenen-übergreifenden Beziehungen sind je nach verknüpften Ebenen unterschiedliche Bedeutungen beizumessen. Wie in Abbildung 3 erkennbar, bestehen paarweise Relationen ausschließlich zwischen einer Ebene und der direkt darüber- oder darunterliegenden Ebene (Jonkers et al., 2003). Im Konkreten wird von einer Ebene aus nur auf die direkt darunterliegende referenziert.

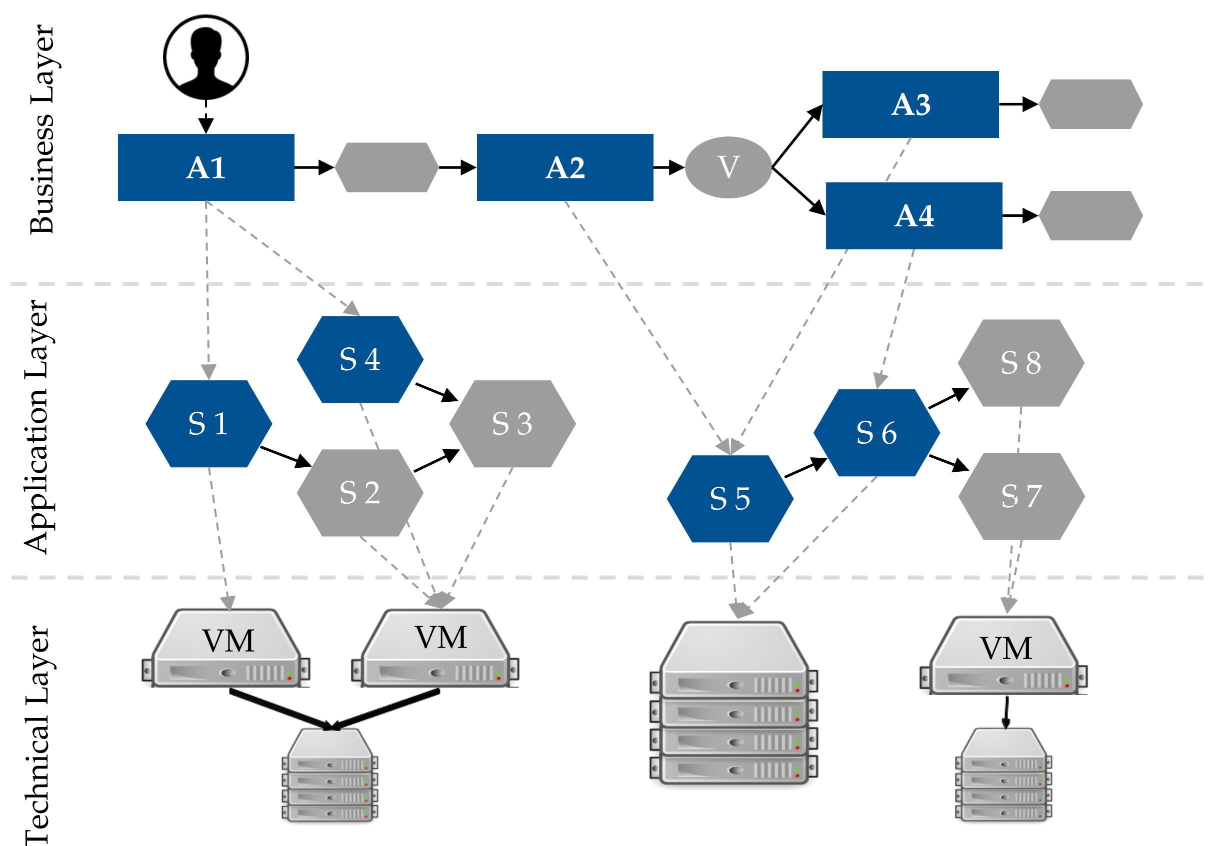


Abbildung 3: Beziehungen zwischen Ebenen einer EA

Somit ergibt sich in der beschriebenen EA eine Verbindung zwischen Business-Layer und Application-Layer. Die Ursache hierfür ist die häufige technische Unterstützung von Geschäftsaktivitäten innerhalb eines Geschäftsprozesses. Dies führt zu einer Abhängigkeit zwischen den modellierten Aktivitäten des Business-Layers und den modellierten Anwendungen oder Services des Application-Layers, welche zur Ausführung der Geschäftsaktivität beitragen.

Weitere Ebenen-übergreifende Verbindungen sind zwischen dem Application-Layer und dem Technology-Layer definiert. Die Anwendungen des Application-Layers bedürfen einer geeigneten Hardware-Umgebung sowie einer infrastrukturellen Anbindung, um betrieben werden zu können. Diese Abhängigkeit wird durch die Verbindung zwischen Applikation und den Hardwarekomponenten des darunterliegenden Technology-Layers definiert. Da zum Betrieb einer Software-Anwendung zwingend ein Hardware-System erforderlich ist, besteht für jede der Anwendungskomponenten eine paarweise Beziehung zu einer Hardware-Komponente.

3. Microservices

Aufgrund verkürzter Entwicklungszyklen und der Notwendigkeit hoher Reaktionsfreude auf Anforderungsänderungen gewinnen agile Entwicklungsmethoden stetig an Bedeutung. Das Konzept der agilen Entwicklung sieht überschaubare Teams vor, welche im Rahmen eines kurzen Planungshorizonts Entwicklungen autark vorantreiben. Dies widerspricht der Entwicklung monolithischer Systeme und begünstigt den fortwährenden Wandel hin zu serviceorientierten Architekturen wie Microservices. Sie erhöhen die Entwicklungsgeschwindigkeit und die Innovationskraft eines Unternehmens. In vielen innovativen Unternehmen wie Netflix (Probst & Becker, 2016), Amazon (Wagner, 2015), Zalando (Schaefer, 2016) und Uber (Reinhold, 2016) haben diese bereits Einzug gefunden.

Dieses Kapitel eröffnet mit der Vermittlung der Grundlagen zu Microservices. Anschließend wird der Einsatz im Bereich EA betrachtet, indem eine Methode zur Herleitung von Microservices aus der Geschäftsdomäne vorgestellt wird und im Rahmen eines Referenzmodells alle Schlüsselkomponenten einer Microservice-Architektur im Kontext einer EA aufgezeigt werden. Die neuen Herausforderungen an das Monitoring eines verteilten Systems im Vergleich zu monolithischen Systemen wird im schließenden Unterkapitel thematisiert.

3.1 Grundlagen zu Microservices

Microservices sind eine besondere Art der serviceorientierten Architekturen (SOA). Sie stellen eine System-Architektur für verteilte Systeme dar. Ein Microservice (MS) ist ein kleiner und autonomer Dienst, der in Kooperation mit weiteren Diensten arbeitet (Newman, 2015).

Während Microservices und SOA das gemeinsame Ziel verfolgen, neue Anforderungen schneller und einfacher zu realisieren, sind sie aufgrund einer abweichenden Herangehensweise voneinander abzugrenzen. SOA geht davon aus, dass Services selten geändert werden müssen. Daher ist auch die Oberfläche von den Services separiert. Microservices hingegen zielen darauf ab, einer fachlichen Gliederung zu unterliegen. Anforderungsänderungen führen hierbei nur zu Anpassungen bei einem Service welcher Logik sowie Oberfläche umfasst. Aufgrund der isolierten Fachlich-

keit und der damit einhergehenden unabhängigen Anpassung eines Service kann dieser auch unabhängig deployt werden (Wolff, 2016).

Ein Verbund von Microservices erlaubt den Betrieb großer und komplexer Software-Anwendungen durch die Komposition der einzelnen Dienste. Wie in Abbildung 4 dargestellt, werden diese häufig hinter einem Gateway (auch als API Proxy bezeichnet) gebündelt, um einen zentralen Anlaufpunkt für Clients zu bieten. Jeder dieser Services hat eine spezifische Aufgabe zu erfüllen, nutzt jedoch unter Umständen zur Erfüllung dieser Aufgabe einen oder mehrere weitere Services. Einige Services dienen somit der direkten Erfüllung einer Geschäftsaktivität. Diese werden als Functional Services (Newman, 2015) bezeichnet. Andere, als Infrastructural Services (Newman, 2015) bezeichnete Dienste bieten lediglich unterstützende Funktionalitäten an. Somit bestehen lose gekoppelte Abhängigkeiten zwischen den Services. Ein Microservice kann jedoch unabhängig von den anderen im Rahmen des Softwareverteilungsprozesses bereitgestellt werden (Bakshi, 2017).

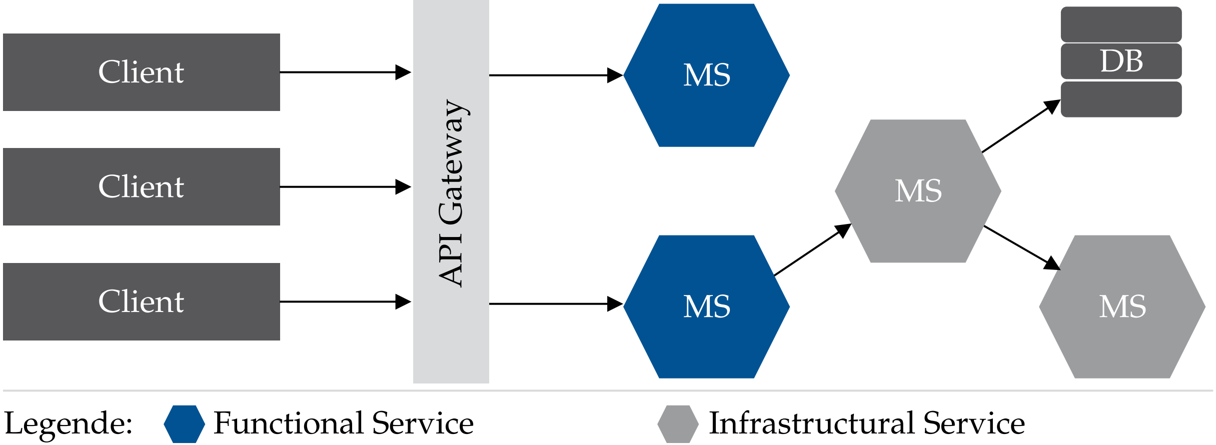


Abbildung 4: Vereinfachte Microservice-Architektur

Dies bedeutet, dass eine Bereitstellung keinerlei Koordination mit Teams anderer Microservices erfordert. Hierdurch wird eine hochfrequentierte Anzahl von Software-Veröffentlichungen und Aktualisierungen innerhalb eines kurzen Zeitraums ermöglicht.

Des Weiteren sind die Services in ihrer Implementierung nicht an bestimmte Programmiersprachen gebunden. Jeder der Services aus dem Verbund kann mit der funktional geeignetsten Technologie realisiert werden, ohne dass dies eine Ein-

schränkung in der Zusammenarbeit mit sich führt. Zu diesem Zweck bieten die Services eine programmiersprachenunabhängige und standardisierte Schnittstelle zur Kommunikation und Bereitstellung ihrer Funktionalitäten an. Üblicherweise wird das Hypertext Transfer Protocol (HTTP) zu Kommunikationszwecken genutzt, um ein ressourcenbasierendes Application Program Interface (API) wie das Representational State Transfer (REST) API zum Funktionsabruf zu nutzen. Die Schnittstellenabstraktion, Autonomie der Services und Programmiersprachenunabhängigkeit bieten die Basis einer verteilten und skalierbaren Systemlandschaft und stellen die Grundlage zur Nutzung agiler Entwicklungsprozesse dar (Bakshi, 2017).

3.2 Microservices im Kontext einer Enterprise Architecture

Im Rahmen des Einsatzes von Microservices in einer Enterprise Architecture stehen Architekten vor der Aufgabe, Microservices entsprechend fachlicher Anforderungen zu definieren. Nachfolgend wird ein Design-Pattern zur Strukturierung des Application-Layers basierend auf der fachlichen Domäne vorgestellt. Das Kapitel schließt mit der Vorstellung einer Referenzarchitektur zum Betrieb einer Microservice-Architektur im Enterprise-Umfeld ab, welche wiederkehrende Schlüsselkomponenten vorstellt und beschreibt.

3.2.1 Modellierung von Microservices im EA-Umfeld

Wie in Kapitel 2.2.2 beschrieben, ist die Microservice-Architektur als Teil der EA auf der Schicht des Application-Layers angesiedelt. Die Microservices dienen der technischen Realisierung fachlicher Geschäftsaktivitäten, welche auf dem Business-Layer einer EA angesiedelt sind. Im Rahmen der Modellierung des Application-Layers stehen Architekten vor der Herausforderung, ein System in Microservices zu strukturieren.

Aufgrund der Beschaffenheit von Microservices ist das Domain Driven Design (DDD) eine geeignete Herangehensweise zur fachlichen Strukturierung des Application-Layers. DDD wurde durch Eric Evans im Jahr 2003 in seinem gleichnamigen Buch (Evans, 2003) vorgestellt und hat zum Ziel, die Domäne der realen Welt zu modellieren (Newman, 2015). Microservices eines durch DDD gegliedertes Systems bilden jeweils eine fachliche Einheit. Das Ziel dieser Strategie ist, dass bei fachlichen Änderungen oder Erweiterungen nur genau ein Microservice angepasst werden

muss. Da dieser von einem Team entwickelt wird, kann hiermit eine hohe Effizienz erreicht werden, da lediglich ein Team involviert ist, welches Änderungen schnell und mit geringem Kommunikationsaufwand umsetzen kann (Wolff, 2016).

Die Modellierung von Microservices unterstützt das DDD unter anderem bei der Lösung von zwei Problemen: beim Verstehen und Beschreiben einer Domäne in einer unmissverständlichen Sprache sowie bei der klaren Definition des Kontexts, in welchem die Modelle Gültigkeit haben und eine eindeutige Terminologie vorliegt. Die durch DDD hierzu bereitgestellten Tools sind Ubiquitous Language sowie der Bounded Context (Martincevic, 2016).

Während der „Bounded Context" dazu dient, Räume zu definieren, in denen ein gemeinsamer Kontext besteht, dient die „Ubiquitous Language" dazu, dass innerhalb jener Kontexte eine Aussage eine spezifische Bedeutung hat. Durch den „Bounded Context" wird sichergestellt, dass Definitionen nicht mehrfach im System vorkommen. Des Weiteren ermöglicht er Architekten die Aufteilung des Systems in Microservices auf Basis der gewonnenen Informationen über die definierten Räume (Martincevic, 2016).

Jede Domäne der Geschäftswelt besteht aus mehreren Bounded Contexts. Innerhalb des jeweiligen Kontexts werden sowohl Elemente modelliert, welche keine Kommunikation über den Kontext hinweg benötigen, als auch Elemente, welche mit anderen „Bounded Contexts" im Austausch stehen. Für den Datenaustausch wird in einem Kontext eine Schnittstelle explizit definiert, welche die für den Datenaustausch zwischen Kontexten zu verwendenden Modelle beschreibt (Newman, 2015).

Wurden jene Kontexte definiert, welche über eigene Modelle verfügen und die Domänen der Geschäftswelt abbilden, so können Geschäftsfunktionalitäten innerhalb der Kontexte durch Microservices modelliert werden. Ein Microservice gehört immer exakt einem Kontext an. Die hieraus entstehende Microservice-Architektur ist stark nach fachlichen Domänen strukturiert, was eine klare Beziehung zwischen Elementen des Business-Layers einer EA und den Microservices des Application-Layers mit sich bringt.

Dem Betrieb einer Microservice-Architektur geht die Organisation der Microservices voraus. Der Betrieb selbst bedarf jedoch noch weiterer Komponenten, um die Microservices nutzbar zu machen. Nachfolgendes Kapitel beschreibt die Infrastruktur zum Betrieb eines Microservice-Clusters in Form eines Referenzmodells.

3.2.2 Microservice-Referenzarchitekturmodell im EA-Kontext

Im Unternehmensumfeld treten wiederkehrende Anforderungen an Microservices und deren Architekturkomponenten auf. Wurden durch einen Schnitt des Systems fachlich orientierte Microservices definiert, sind diese in eine Infrastruktur einzubetten, welche den Einsatz der Microservices ermöglicht. Die Autoren Yale, Silveira und Sundaram (2016) beschreiben diese Infrastruktur in Form eines Referenzarchitekturmodells im Kontext der Enterprise-Architekturen, welches sie 2016 im Rahmen der IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC) vorstellten. Dieses Kapitel widmet sich der Betrachtung dieses Modells und der Beschreibung identifizierter Schlüsselkomponenten.

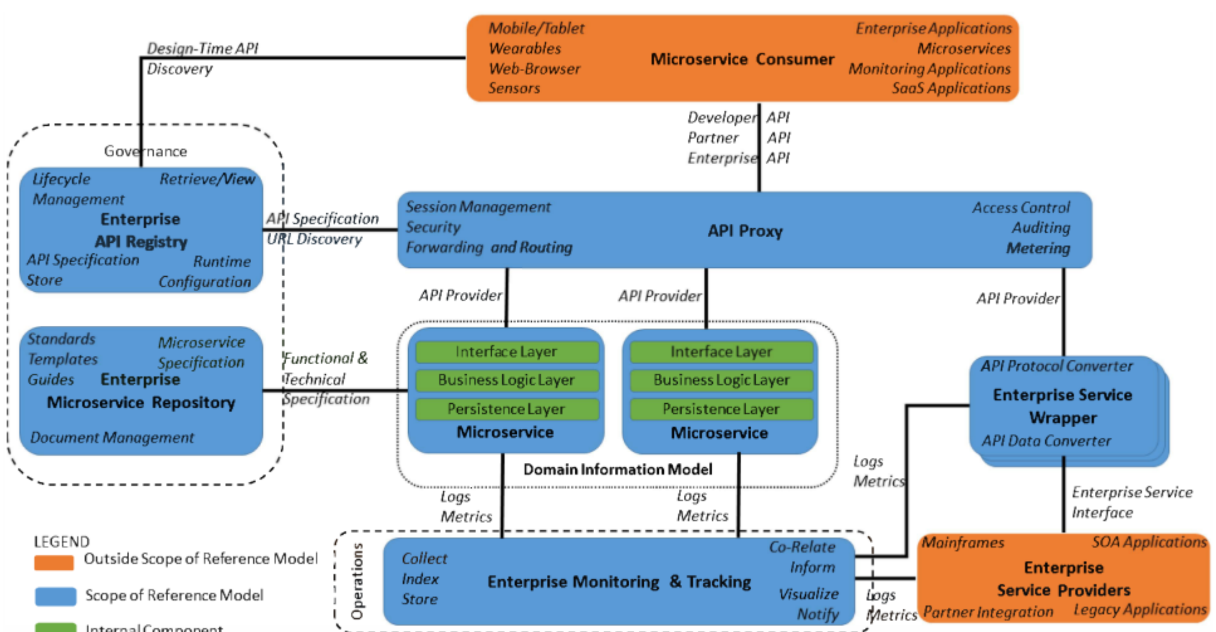


Abbildung 5: MS-Referenzarchitekturmodell (Yale et al., 2016)

Das Modell wird in seiner Gesamtheit durch Abbildung 5 wiedergegeben. Hierbei wird unterschieden zwischen Elementen des Referenzmodells (blau), internen Komponenten eines Elements (grün) und nicht zu dem Modell gehörenden Elementen, welche jedoch aufgrund der Interaktion mit dem Modell von Relevanz sind. Nach-

folgend werden die dem Modell zugehörigen Elemente vertiefend betrachtet. Die Ausführungen haben, insofern nicht anders gekennzeichnet, ihren Ursprung in den Ausführungen von Yale et al. (2016).

Domain Information Model

Microservices sind nach ihren Aufgabenbereichen stark fragmentiert. Jeder der Microservices kann in eigenen Technologien und Tools und mit einem eigenen Sprachgebrauch umgesetzt werden. Dies erschwert das Kommunizieren zwischen den verantwortlichen Teams sowie die Arbeit mit mehreren Microservices. Das Modell sieht eine Gruppierung von Microservices vor, welche einer gemeinsamen Domäne der Geschäftsebene zuordenbar ist und in ihrer Ordnung der Kommunikationsstruktur des Unternehmens gleicht. Diese als Microservice Domain bezeichnete Einteilung dient als Informationsschicht, welche Standards, einen gemeinsamen Sprachgebrauch sowie verwendete Technologien und Tools für die der Domäne zugehörigen Microservices definiert.

Dies hat den Zweck, die Lücke zwischen dem Enterprise Information Model als unternehmensweiter Beschreibung und dem Microservice Information Model als zu granularer Microservice-Beschreibung zu schließen, indem ein Information Model für eine logisch gruppierte Menge von Microservices etabliert wird. Hieraus ergibt sich der Vorteil, dass ein Nutzer einen gemeinsamen Sprachgebrauch sowie Standards innerhalb der gleichen Domäne von Microservices vorfindet, was zu einer klaren Kommunikation und standardisierten Arbeitsweise mit diesen Microservices führt.

Das hier so bezeichnete Domain Information Model kann ebenfalls als Bounded Context, wie im vorangegangenen Kapitel 3.2.1 beschrieben, verstanden werden.

API Proxy

Ein Proxy stellt einen Vertreter für ein Objekt dar. In diesem Kontext vertritt der API Proxy die APIs der EA-Microservices. Als zentrale Anlaufstelle eingehender Schnittstellenanfragen leitet er diese weiter an die jeweiligen Microservices. Der Proxy bietet Funktionalitäten wie den Schutz der zugrundeliegenden Ressourcen vor direktem Zugriff, er ist eine hinlänglich bekannte zentrale Anlaufstelle zum Schnittstellenzu-

griff, sorgt für die Drosslung des Datenflusses etc. Der Einsatz des Proxys führt zur Entkopplung von Microservices und ihren Konsumenten, was Lokalitätsveränderungen und die Skalierung eines Services vereinfacht.

Das Unternehmen stellt die Interfaces der Microservices verschiedenen Nutzergruppen wie unternehmensinternen Konsumenten, Partnern und öffentlichen Nutzern zur Verfügung. Die Microservices, welche diesen Nutzergruppen zugänglich sind, unterscheiden sich in ihren Service Level Agreements (SLA) sowie den Anforderungen an Sicherheit und Zugriffsschutz. Durch den Einsatz des Proxies wird der Zugriff auf die jeweiligen Ressourcen der unterschiedlichen Anwendergruppen sichergestellt.

Als Beispiele für API Proxys sind unter anderem Zuul (Netflix inc, 2017) oder Kong (Kong inc., 2017) zu finden.

Enterprise API Registry

Das API Registry ist einer von zwei Steuerungsdiensten innerhalb einer Microservice-Architektur eines Unternehmens. Es dient dem Aufdecken bereitgestellter Microservice-Schnittstellen innerhalb und außerhalb des Unternehmensumfelds. Es ist als zentrale Komponente zu betrachten, welche innerhalb des gesamten Unternehmens Verwendung findet und deren Lokalität hinlänglich bekannt sein sollte.

Das Registry stellt seine Schnittstelleninformationen in einem Standardformat wie der JavaScript Object Notation (JSON) oder der Extensible Markup Language (XML) bereit, die Konsistenz aufweisen und für Menschen einfach lesbar sind. Der Zugriff und der Umfang der bereitgestellten Informationen unterliegt einer Zugriffskontrolle, sodass interne und externe Anfragen mit unterschiedlichem Informationsgehalt beantwortet werden können.

Das Registry bietet Funktionalitäten, welche es dem Nutzer erlauben, nach detaillierten Informationen verfügbarer API-Spezifikationen zu suchen und diese abzurufen. Die bereitgestellten Informationen umfassen die möglichen Operationen, welche durch die jeweilige API bereitgestellt werden, sowie den Kontext ihrer Verwendung, Zugriffsbeschränkungen und Sicherheitsanforderungen, wie etwa zu verwendende Protokolle zur Nutzung der API.

API-Registry-Funktionalitäten werden beispielsweise durch die Software Swagger (SmartBear Software, 2017b) geleistet.

Enterprise Microservice Repository

Der zweite der Steuerung dienliche Service ist das Enterprise Microservice Repository, welches wie auch das Enterprise API Registry eine unternehmensweite Komponente ist. Entgegen des API Registry hält es keine Schnittstelleninformationen, sondern Metainformationen der Microservices bereit. Dies umfasst deren Lifecycle-Status, ihre Version, Besitzer aus Entwicklungs- und Geschäftsperspektive sowie eine detaillierte Beschreibung des Einsatzzwecks der jeweiligen Microservices. Außerdem stellt es Informationen über eingesetzte Tools, Implementierungsdetails wie Technologien und Architektur, konsumierende APIs sowie nichtfunktionale Anforderungen der Microservices zur Verfügung. Als Implementierungen eines Microservice Repository sind beispielsweise Eureka (Netflix Inc., 2017a) oder Consul (HashiCorp, 2017) zu nennen. In Kapitel 4.1 wird der Einsatz von Eureka zum Zweck des Service Discovery fortführend beschrieben.

Enterprise Service Wrappers

Etablierte Unternehmen haben vor dem Aufkommen von Microservice-Architekturen häufig bereits in andere Architekturen wie SOA investiert. Somit ist bereits ein großer Teil der Business-Logik des Unternehmens in einer anderen Softwaretechnologie vorhanden. Die Transformation hin zu Microservices kann häufig aufgrund des Volumens zu transformierender Geschäftslogik nicht innerhalb eines kurzen Zeitraums durchgeführt werden. Um diesem Missstand zu begegnen, sieht das Referenzmodell Enterprise Service Wrappers vor, welche den Zugriff auf jene Ressourcen über APIs ermöglichen. Diese Wrappers stellen eine API bereit, welche die Funktionalitäten der bisherigen Software bietet. An die API gestellte Anfragen transformiert der Wrapper zu Anfragen an die Schnittstellen der bisherigen Softwaretechnologie. Umgekehrt verfährt er mit Antworten, welche durch jene Software erzeugt werden.

Somit stehen jene veralteten Softwarekomponenten für den Zugriff durch Microservices über den standardisierten Weg der API-Technologien zur Verfügung. Eine Microservice-artige Dekomposition der Software kann ebenfalls durch den Einsatz

mehrerer Wrappers erzielt werden. Mit dem hierdurch ermöglichten Zugriff auf bestehende Business-Logik kann eine Transformation zu einer Microservice-Architektur zügig und unter Verwendung existierender Ressourcen erfolgen.

Service Wrappers sind aufgrund der Vielfalt möglicher Technologien und Schnittstellen der Legacy-Software in Eigenentwicklung zu erzeugen.

Enterprise Monitoring and Tracking Manager

Für Unternehmen stellen operative Daten eine wichtige Grundlage zur Entscheidungsfindung dar. Entsprechende Analysetools sind abhängig von der Datenverfügbarkeit. Während jene Daten in monolithischen Systemen an einer zentralen Stelle verfügbar waren, sind diese in verteilten Systemen ebenfalls verteilt. Dies bedeutet, dass aufgrund der verteilten Architektur mehrere Datenquellen bereitstehen, welche auf verschiedenen Implementierungen von Microservices basieren und jeweils eine Teilmenge der Systeminformationen bereitstellen. Der Enterprise Monitoring and Tracking Manager hat zum Ziel, die Daten jener verteilten Systeme zu sammeln und zentral zu speichern.

Zu diesem Zweck müssen Microservices und andere in der Architektur vorhandene Applikationen standardisierte Events wie Logs und Metriken veröffentlichen. Diese werden durch den Manager in zusammengeführter Form zu Analysezwecken bereitgestellt. Operative Teams können den Datenspeicher nach einzelnen Transaktionen innerhalb eines Microservice oder auch nach unternehmensweiten Interaktionen abfragen. Der Manager bietet neben dem Monitoring der Events außerdem Reporting-Funktionalitäten sowie ein Dashboard zur Visualisierung erhobener Daten der Vergangenheit und Gegenwart.

3.3 Microservice-Monitoring durch Distributed Tracing

Mit dem Wandel von monolithischen Systemen hin zu Microservices steigt die Komplexität von IT-Infrastrukturen. In verteilten Systemen werden verschiedenste Microservices betrieben, welche durch unterschiedliche Teams mit eigenen Programmiersprachen und Tools entwickelt und gepflegt werden (Burrows et al., 2010).

Dies führt zu neuen Herausforderungen im Monitoring des Gesamtsystems. Während das Monitoring in monolithischen Systemen auf zentral erzeugten Logdateien

basieren konnte, sind für verteilte Umgebungen neue Lösungsansätze notwendig. Ein Microservice kann aufgrund seiner Abhängigkeiten von weiteren Microservices im Rahmen des Monitorings nicht isoliert betrachtet werden, da Normabweichungen eines Microservices negative Effekte auf abhängige Services haben können. Vielmehr ist eine Betrachtung aller Services notwendig, die in die Verarbeitung einer Nutzeranfrage involviert sind. Dies ermöglicht, zu erkennen, welcher der Services der Ursprung einer Anomalie ist und wie sich diese auf folgende Services auswirkt.

Um ein transaktionsabhängiges Monitoring mit klassischen Monitoring-Tools zu erreichen, müssten Log-Files aller Microservices ausgewertet und manuell miteinander in Bezug gebracht werden. Das Identifizieren involvierter Microservices einer Transaktion bringt hierbei große Herausforderungen mit sich, da eine eindeutige Transaktionszuordnung nicht zwingend erkennbar ist. Dieser Problematik stellt sich das Distributed Tracing, indem es den Verlauf der Verarbeitung einer Nutzeranfrage dezentral durch die involvierten Microservices dokumentiert und auf einer zentralen Plattform automatisiert zusammenführt. Dies wird durch den Einsatz eines gemeinsamen Trace-Identifikators (ID) aller involvierten Microservices der Verarbeitung eines Requests ermöglicht (Hall, 2016).

Die Monitoring-Daten umfassen neben den involvierten Microservices, deren Ausführungsreihenfolge, Zeitstempel der Verarbeitung, Request-Methode, Uniform Resource Locator (URL), Internet-Protocol-Adresse (IP), Port und weiteren auch Performance-Informationen wie die Verarbeitungsdauer einer Anfrage durch die jeweiligen Services.

Gegenwärtig existieren verschiedene Distributed-Tracing-Systeme wie Zipkin (The OpenZipkin Authors, 2017c), Dapper (Burrows et al., 2010), HTrace (The Apache Software Foundation, 2017) und X-Trace (X-Trace, 2017), welche eine Instrumentierung auf Applikationsebene durchführen. Diese Systeme verwenden untereinander inkompatible APIs, was zu einer Standardisierung des Distributed Tracing durch die OpenTracing-Initiative führte (OpenTracing, 2017).

Im Rahmen der prototypischen Umsetzung der vorliegenden Arbeit wird das Distributed-Tracing-System Zipkin eingesetzt, da es die Clientinstrumentierung einer Vielzahl verschiedener Technologien unterstützt und die zentrale Server-

Komponente des Zipkin-Systems aufgrund seiner modularen Softwarearchitektur als modifizierbare Basis des Prototyps dient. Nachfolgende Ausführungen beziehen sich daher stets auf jenes System.

3.3.1 Funktionsweise des Distributed-Tracing-Systems Zipkin

Das Distributed Tracing wird auf Applikationsebene realisiert, indem die einzelnen Microservices um sogenannte Tracer erweitert werden. Diese werden gemeinsam mit den MS-Applikationen ausgeführt und zeichnen Zeit- und Metadaten verschiedener Operationen der Applikation auf. Ein Beispiel einer solchen Operation ist das Entgegennehmen oder Beantworten eines Requests. Die aufgezeichneten Daten werden als Span bezeichnet (The OpenZipkin Authors, 2017a).

Zipkin-Instrumentierungen werden in den Technologien C#, Go, Java, JavaScript, Ruby und Scala bereitgestellt (The OpenZipkin Authors, 2017b). Ergänzt wird die Liste der Instrumentierungen durch von der Community entwickelte und gepflegte Bibliotheken. Die Instrumentierung wurde mit dem Ziel entwickelt, den produktiven Betrieb nicht zu stören und nur geringfügigen Mehraufwand zu erzeugen. Durch das Distributed Tracing entstandener Overhead wird auch im Rahmen der Evaluation dieser Arbeit untersucht (siehe Kapitel 6.2.2).

Die Instrumentierung ergänzt ausgehende Requests um zusätzliche ID-Header, um die Reihenfolge der Verarbeitung und die Zugehörigkeit zu einem gemeinsamen Trace zu verbreiten. Zu diesem Zweck wird bei einem neu erzeugten Request die ID des Trace sowie die ID des eigenen Spans angefügt, um den Vorgänger zu dokumentieren. Liegt der Applikation keine Trace-ID vor, da sie die erste instrumentierte Applikation in der Kette der verarbeitenden Applikationen ist, erzeugt sie eine Trace-ID identisch mit ihrer Span-ID.

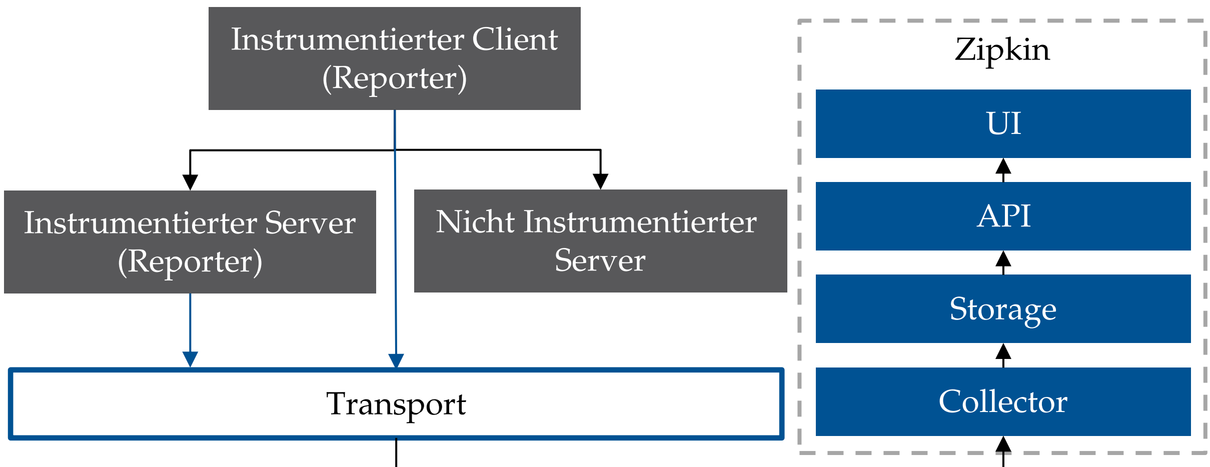


Abbildung 6: Zipkin-Datenfluss (The OpenZipkin Authors, 2017a)

Wurde durch eine instrumentierte Anwendung ein Span erzeugt, so wird dieser anschließend, wie in Abbildung 6 dargestellt, an das Zipkin-System übermittelt. Der für den Versand verantwortliche Teil der Instrumentierung wird als Reporter bezeichnet. Er übermittelt den Span durch eine von drei unterstützten Transportmethoden (The OpenZipkin Authors, 2017a):

- HTTP(s)
- Apache Kafka
- Scribe

Der empfangende Collector validiert den Span und persistiert diesen anschließend in der Storage-Komponente. Das System wurde derart konzipiert, dass die Storage-Komponente durch eine eigene oder eine der mitgelieferten Implementierungen (In-Memory, Cassandra, Elastic Search, MySql) realisiert werden kann.

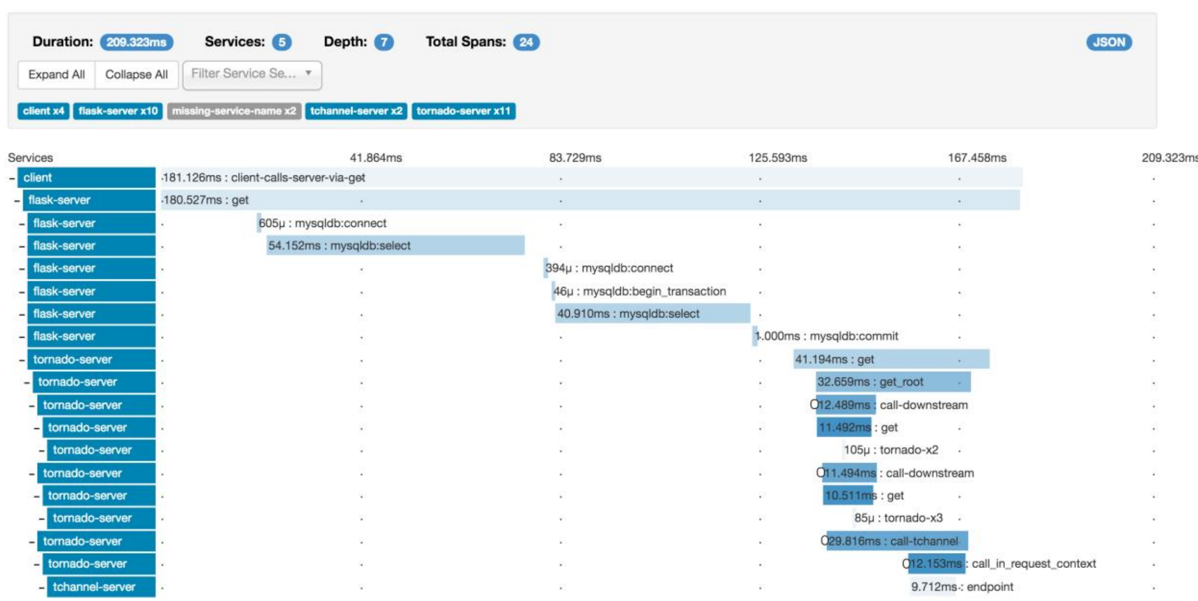


Abbildung 7: Zipkin-UI-Trace-Visualisierung (The OpenZipkin Authors, 2017c)

Persistierte Spans werden durch die API des Zipkin-Systems bereitgestellt. Diese wird vorrangig vom Zipkin-eigenen User Interface (UI) genutzt, welches unter anderem Spans eines Trace in ihrem zeitlichen Verlauf und die ermittelte Architektur visualisiert.

3.3.2 Entstehung eines Spans des Zipkin-Systems

Für die Erzeugung valider Spans trägt die Applikationsinstrumentierung Verantwortung. Beispielsweise wird durch sie die Gleichheit der an den Server asynchron verschickten Trace- und Span-ID mit den weitergegebenen IDs an nachfolgende Services sichergestellt. Das in Abbildung 8 dargestellte Sequenzdiagramm demonstriert die Erzeugung eines Spans im Rahmen eines HTTP-Requests. Der Client sendet in diesem Szenario einen HTTP-Request „GET /foo“ an einen Microservice.

Wie aus dem Sequenzdiagramm ersichtlich, wird der Request, welcher durch den Anwender im Applikationscode ausgelöst wird, an die Instrumentierung adressiert. Diese analysiert den Request und zeichnet dessen Tags als Metadaten des zu erzeugenden Spans auf. Anschließend wird der Request um Header der Trace-ID und der Span-ID erweitert. Anschließend erfolgt eine Aufzeichnung der aktuellen Zeit, um die Verarbeitungszeit berechnen zu können.

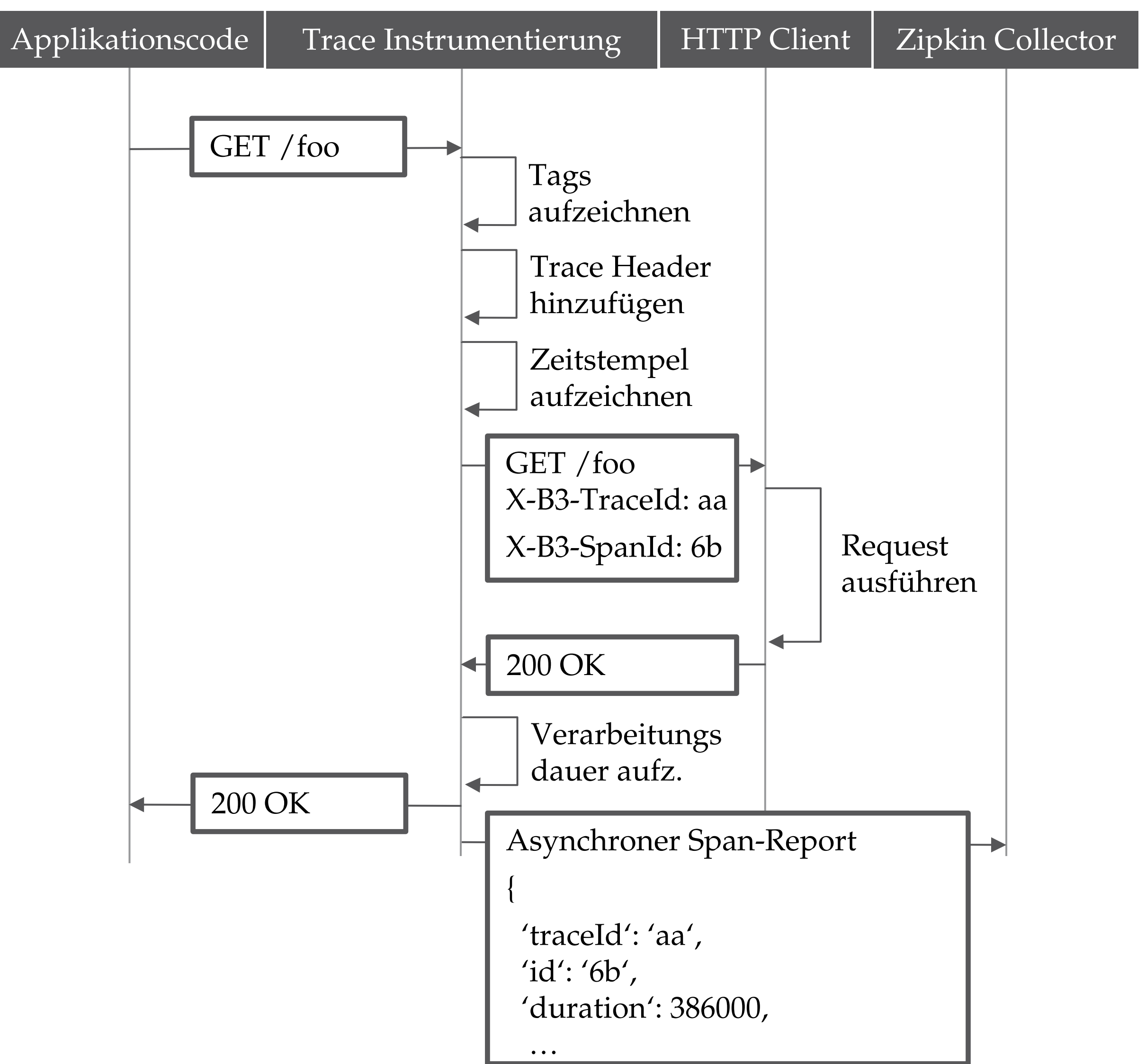


Abbildung 8: Sequenzdiagramm einer Spanerzeugung (The OpenZipkin Authors, 2017a)

Der um die Header erweiterte Request wird anschließend an den HTTP-Client der jeweiligen Technologie weitergeleitet, welcher den Versand des Requests verantwortet. Nachdem der HTTP-Client eine Response vom adressierten Microservice erhalten hat, leitet er diese an die Trace-Instrumentierung weiter. Diese misst den Zeitpunkt des Eintreffens und berechnet aus dem Delta der zwei gemessenen Zeitpunkte die Verarbeitungsdauer des Requests. Anschließend wird die Response an den Applikationscode weitergeleitet und ein Span erzeugt, welcher die aufgezeichneten Werte umfasst. Dieser wird asynchron an den Collector des Zipkin-Servers gesendet.

Neben der Erzeugung von Spans bei der Erzeugung oder Beantwortung eines Requests können durch die Instrumentierungssoftware zu beliebigen weiteren Zeitpunkten Spans erzeugt werden, um Ereignisse zu dokumentieren. So kann beispielsweise der Aufruf einer Methode oder die Erzeugung asynchroner Verbindungen dokumentiert werden.

3.3.3 Span-Datenmodell eines Zipkin-Systems

Das Datenmodell eines Spans des Distributed Tracing Systems Zipkin umfasst die in Tabelle 2 aufgeführten Felder.

Datum	Beschreibung
traceId	Eine gemeinsame ID aller Spans, welche im Rahmen der Verarbeitung einer externen Anfrage erzeugt wurden.
name	Durch den Instrumentierungsservice festgelegter Zeichenketten-Identifier des Spans. Der Inhalt variiert nach Entstehungsursache des Spans. Bei eingehenden externen Anfragen nimmt dieses Feld den Unified Resource Identifier (URI) des Endpunkts als Wert an.
id	Identifier eines Spans. Sind beide Instanzen einer paarweisen Kommunikation instrumentiert, werden zwei Spans gleicher ID erzeugt, wodurch deren Zusammengehörigkeit erkennbar wird.
parentId	ID des vorangegangenen Spans mit gleicher traceId. Der erste erzeugte Span einer externen Anfrage verfügt nicht über eine parentId. Die parentId erlaubt die Rekonstruktion der Reihenfolge erzeugter Spans.
timestamp	Zeitpunkt, zu welchem der Span erzeugt wurde.
Duration	Wenn der Span von der anfragenden Instanz stammt, beschreibt dieses Feld die vergangene Zeit zwischen Anfrageversand und Erhalt der Antwort. Stammt er hingegen vom angefragten Service, beschreibt es die Dauer zwischen Anfrageempfang und Versand der Antwort. Die Dauer wird in Millisekunden angegeben.

annotations	Dieses Feld beinhaltet ein Array von Annotationen, welche den zeitlichen Verlauf der Kommunikation dokumentieren.
binaryAnnotations	Dieses Feld beinhaltet ein Array von Annotationen, welche keinen zeitlichen Aspekt umfassen, den Span jedoch um weitere Informationen in Form von Key-Value Paaren ergänzen.

Tabelle 2: Datenmodell eines Zipkin-Spans

Abhängig vom auslösenden Ereignis der Spanerzeugung wird das Datenmodell mit variierenden Daten belegt. So wird bei einem Span, welcher keinen zeitlichen Verlauf dokumentiert, das „annotations“-Feld nicht genutzt. Ein Beispiel hierfür ist die Dokumentation eines Funktionsaufrufs, welcher einen konkreten Zeitpunkt und keinen Verlauf beschreibt. Nachfolgend werden für den Prototyp relevante Ursachen der Spanerzeugung betrachtet und die Daten dieser Spans beschrieben.

Versand von Spans im Rahmen einer paarweisen Kommunikation

Im Rahmen einer paarweisen Kommunikation wird ein Span durch die anfragende und ein Span durch die antwortende Instanz versendet, wobei zur Identifikation der Zusammengehörigkeit jeweils die selbe Span-ID zugewiesen wird.

Das „annotations“-Feld ist abhängig vom Span-Erzeuger mit folgenden Zeitangaben belegt:

- „cs“ – Zeitpunkt, zu welchem die anfragende Instanz die Anfrage versendet hat sowie der Endpunkt der anfragenden Instanz.
- „sr“ – Zeitpunkt, zu welchem die angefragte Instanz die Anfrage erhalten hat sowie der Endpunkt der angefragten Komponente.
- „ss“ – Zeitpunkt, zu welchem die angefragte Instanz die Anfrage beantwortet hat sowie der Endpunkt der angefragten Instanz.
- „cr“ – Zeitpunkt, zu welchem die anfragende Instanz die Antwort erhalten hat sowie der Endpunkt der anfragenden Instanz.

Der Span der anfragenden Instanz beinhaltet die Annotationen „cs“ und „cr“, der Span der antwortenden Instanz „sr“ und „ss“. Neben der Zeitangabe erfolgt die Angabe des relevanten Endpunkts. Ein Endpunkt wird beschrieben durch seine IP-

Adresse den Port und die Bezeichnung des Service, der unter dem Port betrieben wird. Mithilfe dieser Informationen können die zwei involvierten Endpunkte eines Spans eindeutig identifiziert werden.

Versand eines Spans zum Informationszugewinn

Spans können auch während der Verarbeitung durch eine Service-Instanz aufgrund eingetretener Ereignisse erzeugt werden. Diese beschreiben nicht die Kommunikation zwischen zwei Instanzen, sondern jene Aktivität innerhalb der Instanz. Welche Ereignisse zur Erzeugung eines Spans führen, ist abhängig von der Implementierung der Instrumentierungssoftware und weder standardisiert noch durch den Zipkin-Server limitiert.

Ein Beispiel sei die Dokumentation der ersten aufgerufenen Methode durch eine eingehende Anfrage. Mit Aufruf der Methode wird durch die Komponente ein Span versendet, bei welchem das „annotations"-Feld nicht belegt ist. Jedoch wird ein Wert innerhalb der „binaryAnnotations" gesetzt, der die Bezeichnung der aufgerufenen Methode beinhaltet. Der Span erhält als „parentId" die ID des Spans, welcher im Rahmen der vorangegangenen paarweisen Kommunikation durch die Instanz erzeugt wurde. Somit kann der Anfrage die aufgerufene Methode zugeordnet werden.

Ein weiteres Beispiel sei die Dokumentation asynchroner Requests. Wird ein Request asynchron durchgeführt, dann wird für dessen Absenden ein neuer Thread erzeugt. Durch die Instrumentierung wird die Erzeugung eines Threads mit einem separaten Span dokumentiert. Dieser umfasst eine „binaryAnnotation" namens „lc" für „local component" sowie eine „binaryAnnotation" namens „thread", welche die Asynchronität kennzeichnen. Wird ein Request innerhalb dieses Threads erzeugt, so wird dem hierbei erzeugten Span als Vorgängerelement der zuvor erzeugte Span des Threads zugewiesen. Mithilfe dieser Kennzeichnungen wird die Asynchronität der Kommunikation festgehalten.

4. Stand der Technik

Es bestehen verschiedene Technologien und Lösungen zur Erkennung einer Architektur und zum Aufbau eines Architekturmodells. Diese unterscheiden sich unter anderem nach statischer und dynamischer Datenanalyse, der Echtzeitnähe des Architekturmodells und dessen Bereitstellung sowie im Umfang erkannter Entitäten und Beziehungen einer EA. Dieses Kapitel widmet sich vier ausgewählten Lösungen zur Erkennung von Architekturen, welche den derzeitigen Stand der Technik repräsentieren, und schließt mit einer Abgrenzung der vorliegenden Arbeit von diesen.

4.1 Service Discovery durch Microservice Repositories

Service Discovery ist kein neues Thema der Microservice-Architekturen, gewinnt jedoch aufgrund der extremen Zunahme von Services im Rahmen verteilter Architekturen stark an Bedeutung. Eines der ältesten Service-Discovery-Systeme ist das Domain Name System (DNS). Dieses dient der Umwandlung von Hostnamen eines Netzwerks in IP-Adressen und findet bis heute Verwendung (Rotter et al., 2017).

Diese Art des Service Discovery ist jedoch für komplexe verteilte Systeme unzureichend, denn diese benötigen Funktionalitäten wie Metainformationen eines Service oder seinen Zustand (Rotter et al., 2017). Wie in der in Kapitel 3.2.2 vorgestellten Referenzarchitektur beschrieben, umfassen Microservice-Architekturen häufig bereits Microservice Repositories, welche Service Discovery betreiben, indem in der Infrastruktur vorhandene Services ihren Status an eine Instanz des Repository melden. Diese sammelt die Informationen, tauscht sie mit weiteren vorhandenen Instanzen von Service Repositories aus und stellt sie über eine Schnittstelle bereit.

Die Architekturerkennung beschränkt sich beim Service Discovery ausschließlich auf den Application-Layer der EA und umfasst keine Relationen bzw. Abhängigkeiten zwischen den ermittelten Services. Das primäre Anliegen der Microservice Repositories ist nicht die Schaffung höherer Architekturtransparenz, sondern das Unterstützen technischer Prozesse wie Loadbalancing und Proxy Gateway Services, welche die durch das Repository bereitgestellten Informationen erfordern.

Nachfolgend wird das im Rahmen dieser Arbeit verwendete Microservice Repository Eureka eingehend betrachtet. Eureka ist ein durch den Streaming-Anbieter Netflix Inc. realisiertes Open-Source-Projekt (Netflix Inc., 2017b), welches Server- und Client-seitige Software umfasst und frei zugänglich ist. Es erlaubt eine einfache Implementierung in Systeme, welche auf weit verbreiteten Microservice Frameworks wie Spring Cloud basieren.

Wie in Abbildung 9 dargestellt, können sich Microservice-Instanzen (hier „Application Service“) an einem Eureka-Server anmelden, um Anwesenheit und Status mitzuteilen. Durch ein „Renew“ wird durch die Clients zyklisch die Anwesenheit aktualisiert. Durch ein „Cancel“ oder das Ausbleiben der „Renew“-Meldungen wird der Service abgemeldet. Um Microservices über mehrere Rechenzentren verteilen zu können, können mehrere Eureka-Server eingesetzt und miteinander verbunden werden. Diese besitzen jeweils Statusinformationen ihres Rechenzentrums, welche sie mit den weiteren Eureka-Servern austauschen, um ein Gesamtabbild der vorhandenen Service-Instanzen zu erzeugen.

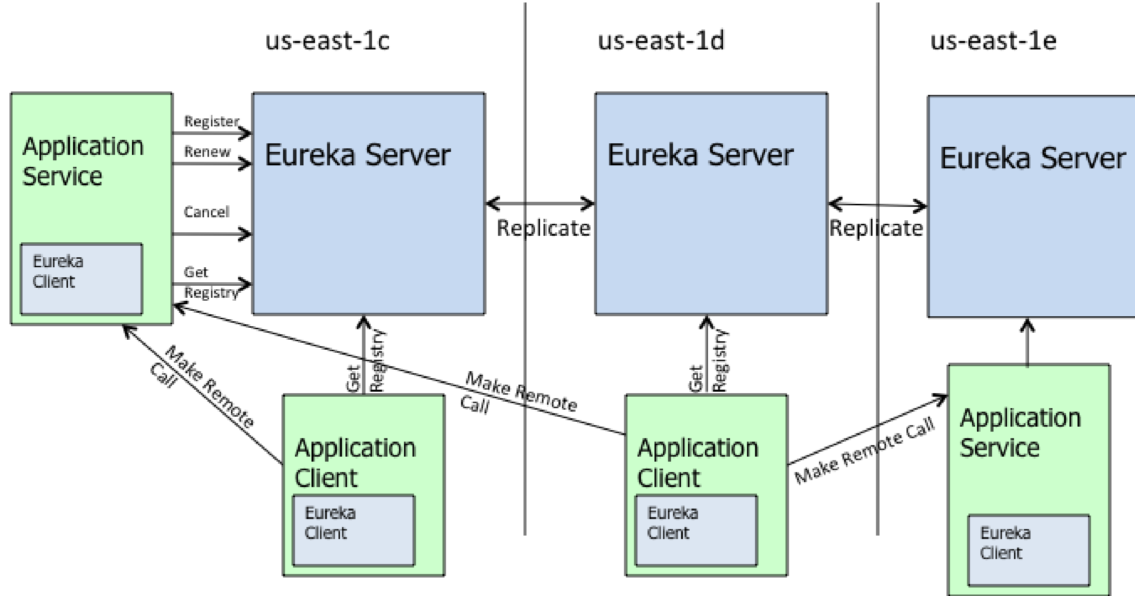


Abbildung 9: Eureka-Architektur (Netflix Inc., 2017b)

Ein Eureka-Server stellt eine REST-API bereit, von welcher durch HTTP-Requests oder den Eureka Client Service-Statusinformationen der Microservices konsumiert werden können. Die Eureka-REST-API stellt diverse Endpunkte zur Statusaktualisie-

rung sowie zum Statusabruf von Services und Instanzen zur Verfügung (siehe Netflix Inc., 2017c). Beispielsweise stellt der Endpunkt „GET /eureka/v2/apps“ alle ihm bekannten Services (dort als „Applications“ bezeichnet) sowie ausführende Instanzen jener Services bereit. Zu einer Instanz werden unter anderem IP-Adresse, Port, Service-Bezeichnung und Status zur Verfügung gestellt. Der Status weist einen der folgenden Werte auf:

- Verfügbar („UP“)
- Nicht erreichbar („DOWN“)
- Startend („STARTING“)
- Außer Betrieb („OUT_OF_SERVICE“)
- Unbekannt („UNKNOWN“)

Anhang A bildet beispielhaft eine vollständige XML-Antwort des Eureka-Servers für den genannten Endpunkt ab.

Während das Service Discovery der Unterstützung technischer Prozesse dient, ist es in der Repräsentation der Architektur stark limitiert. Da es weder Beziehungen zwischen Services noch zu anderen Ebenen der EA entdeckt, trägt es zur Transparenz einer Architektur nur in sehr begrenztem Maß bei.

4.2 Process Discovery durch Business Process Mining

Eine relativ junge Technologie stellt das Business Process Mining dar, welches sich der Ebene des Business-Layers einer EA zuwendet. Das Business Process Mining besteht aus drei Disziplinen:

- Discovery von Geschäftsprozessen
- Überprüfung der Konformität eines Geschäftsprozesses
- Verbesserung des Geschäftsprozesses

Realisiert werden diese durch das Extrahieren von Informationen aus Event-Logs ohnehin im Enterprise-Umfeld bestehender Informationssysteme (van der Aalst et al., 2012). Im Rahmen dieser Arbeit ist die Teildisziplin Discovery von besonderem Interesse. Das Ziel von Process Discovery ist es, aus den Event-Logs, Enterprise-Resource-Planning-Systemen (ERP) und Dokumenten-Management-Systemen In-

formationen zu extrahieren, zusammenzuführen und daraus ein Process Model zu erzeugen. Die Herausforderung liegt hierbei oft darin, dass die zusammenzuführenden Daten der verschiedenen Systeme unterschiedlicher Syntax und Semantik sind (van der Aalst, 2011).

Case id	Timestamp	Activity	Resource	Cost
654423	30-04-2014:11.02	Register request	John	300
654423	30-04-2014:11.06	Check completeness of documents	Ann	400
655526	30-04-2014:16.10	Register request	John	200
655526	30-04-2014:16.14	Make appointment	Ann	450
654423	30-04-2014:11.12	Ask for second opinion	Pete	100
654423	30-04-2014:11.18	Prepare decision	Pete	400
654423	30-04-2014:11.19	Pay fine	Pete	400
655526	30-04-2014:16.26	Check completeness of documents	Sue	150
655526	30-04-2014:16.36	Reject claim	Sue	100
...	...	...	...	...

Tabelle 3: Auszug eines Event-Logs (van der Aalst, 2015)

Wie in Tabelle 3 beispielhaft dargestellt, bezieht sich jeder Eintrag des Event-Logs auf eine ausgeführte Aktivität, also einen definierten Prozessschritt. Außerdem ist jeder Eintrag einem Case zugeordnet, welcher z. B. die Zuordnung zu einem Prozessdurchlauf beschreibt. Um die Reihenfolge der Aktivitätsausführung erkennen zu können, muss außerdem der zeitliche Verlauf erkennbar sein. Daher umfasst ein Event-Log zumindest folgende drei Informationen (van der Aalst, 2015):

- eine Case-ID, welche den übergeordneten Case jedes Eintrags kennzeichnet
- die durchgeführte Aktivität
- einen Zeitstempel zur Erfassung der Chronologie

Aufgrund der Zuordnung zu einer Case-ID können die einzelnen Aktivitäten eines Prozesses ermittelt werden. Durch die Zeitstempel können diese chronologisch geordnet und somit eine Ausführungsreihenfolge ermittelt werden.

Im Gegensatz zu herkömmlichen Verfahren zur Erfassung von Geschäftsprozessen, wie Interviews, wird durch Process Discovery nicht der Sollzustand, sondern der tatsächliche Istzustand der Unternehmensprozesse erfasst.

Doch Process Discovery ist nicht beschränkt auf die Erkennung der Prozesse und ihrer Ausführungsreihenfolge. Beispielsweise umfasst es ebenfalls das Erkennen organisationsbezogener oder zeitlicher Perspektiven (van der Aalst et al., 2012).

Business Process Mining wird bereits durch verschiedenste Werkzeuge wie das Open-Source-Tool ProM (Process Mining Group, 2017) oder das kommerziell ausgerichtete Produkt Celonis PI (Celonis SE, 2017) ermöglicht.

4.3 Architekturerkennung über Monitoringdaten

Das Erzeugen und Sammeln von Monitoringdaten dient nicht nur der Überwachung eines Systems, sondern die hierdurch erzeugten Daten können ebenfalls zur Analyse der zugrundeliegenden Architektur verwendet werden. Durch van Hoorn et al. (2009) wurde ein solches System mit dem Namen Kieker publiziert, welches im Folgenden beschrieben wird. Kieker wurde entwickelt, um folgende Forschungsthemen zu adressieren (van Hoorn et al., 2009):

- Lokalisation von Störungen auf Basis der Erkennung von Anomalien im zeitlichen Verlauf
- Architekturbasierte Umgebungsanpassung/Rekonfiguration
- Visualisierung des Softwareverhaltens zur Laufzeit
- Intrusion Detection auf Applikationsebene
- Performance-Analyse auf Basis von Traces

Kieker kann im Umfeld von Java-Applikationen zum Einsatz kommen und setzt zur dynamischen Analyse auf das Verfahren des Distributed Tracing (siehe Kapitel 3.3), durch welches Monitoring-Daten dezentral erzeugt und zentral analysiert werden (van Hoorn, Waller & Hasselbring, 2012). Die Datenerhebung und -analyse erfolgt in

einem fortwährenden Prozess. Um die dynamische Analyse zu ermöglichen, müssen alle Java-Applikationen mit einer im Umfang von Kieker enthaltenen Software instrumentiert werden. Neben dem Application Performance Monitoring (APM) werden die eingehenden Monitoring-Daten zur Architekturerkennung verwendet. Die Architekturerkennung umfasst (Hasselbring, 2017):

- Entitäten der Architektur
- Verteilte Control Flows
- Nebenläufigkeit und Synchronität
- Visualisierungen

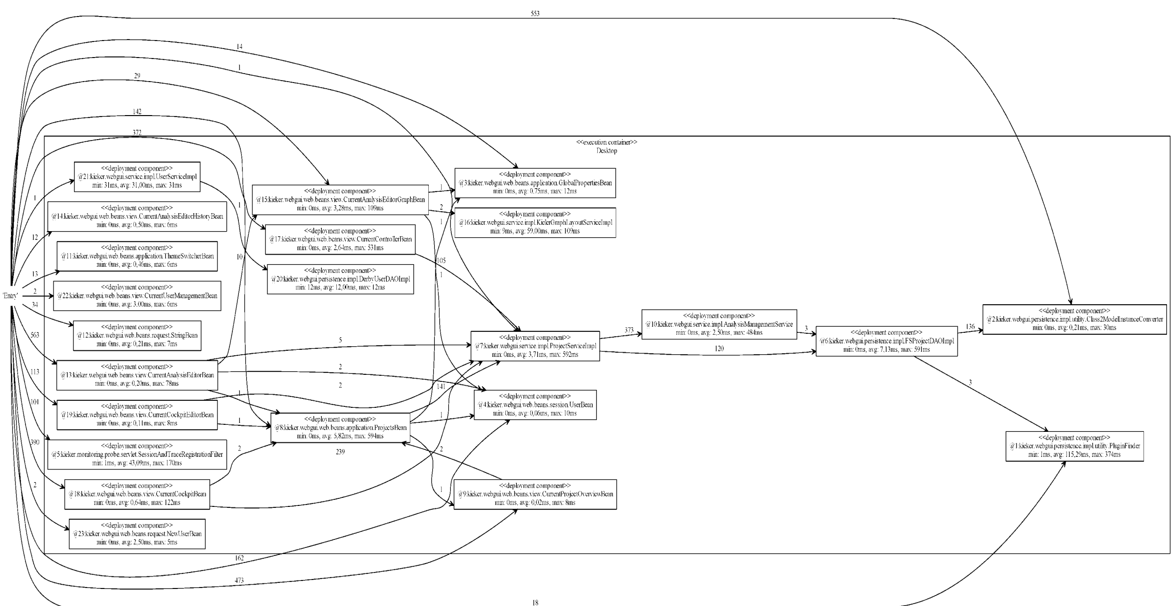


Abbildung 10: Abhängigkeitsdiagramm durch Kieker (Hasselbring, 2017)

Die durch das System erkannten Entitäten umfassen Komponenten wie Klassen und Typen, Operationen sowie Ausführungscontainer wie die Server, auf welchen die Applikationen betrieben werden. Die Visualisierung ermittelter Entitäten kann, wie in Abbildung 10 dargestellt, in Form von Abhängigkeitsgraphen erfolgen. Des Weiteren werden Call Trees oder Sequenzdiagramme zur visuellen Darstellung der Architekturinformationen bereitgestellt.

Neben dem Einsatz von Distributed Tracing existieren weitere Technologien zur Erzeugung von Monitoring-Daten, etwa agentenbasierte Systeme. Diese erfordern keine Instrumentierung einer Applikation, sondern des ausführenden Containers, z. B. eines Servers. Der Agent sammelt fortlaufend Monitoring-Daten und überwacht Logdateien, um relevante Informationen zu extrahieren. In einem definierten Zeitintervall sendet er diese an eine zentrale Instanz zur Datenanalyse und Datenzusammenführung der einzelnen Hosts. Dieses Verfahren wird beispielsweise durch das kommerzielle APM-System Dynatrace (Dynatrace LLC., 2017a) angewandt.

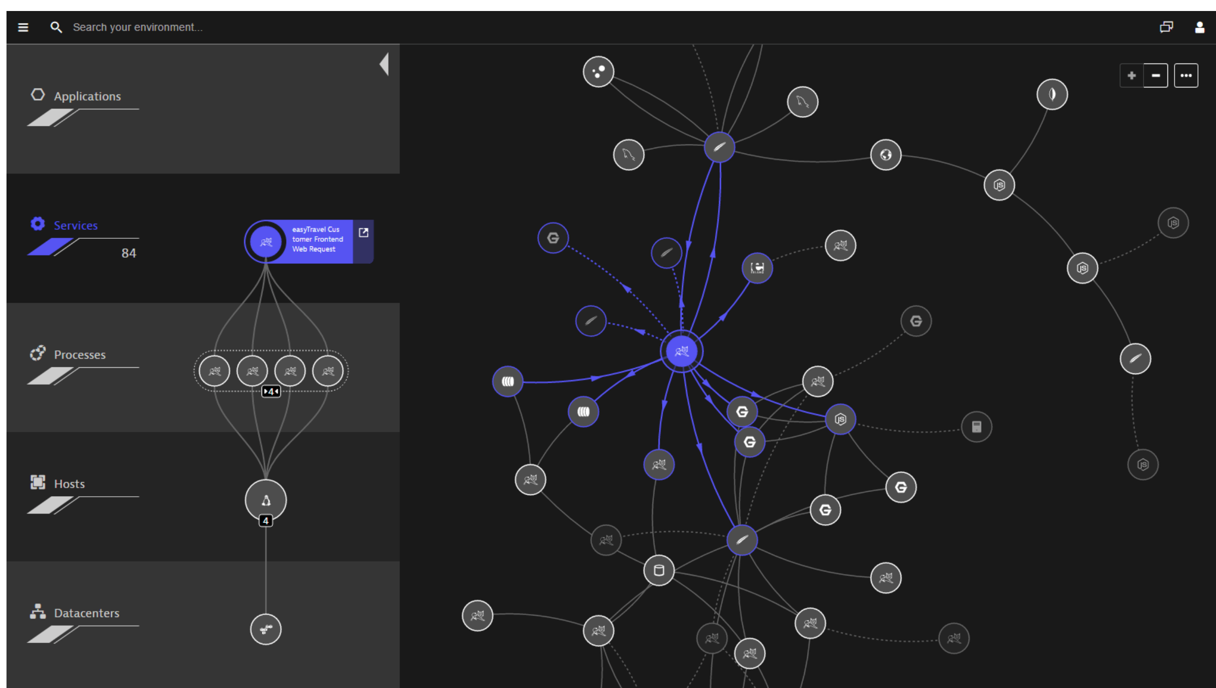


Abbildung 11: Visualisierung einer Anwendungstopologie durch Dynatrace (Dynatrace LLC., 2017b)

Dynatrace ist auf Basis dieser Daten in der Lage, den vollständigen Technologie-Stack eines Systems inklusive seiner Abhängigkeiten zu erkennen und wie in Abbildung 11 dargestellt zu visualisieren (Dynatrace LLC., 2017b).

4.4 Architekturerkennung durch statische und dynamische Analyse

Im Rahmen der IEEE International Conference on Software Architecture Workshops stellten Granchelli et al. (2017b) das Architekturerkennungssystem MicroART vor, welches als Eingabeparameter lediglich ein GitHub Repository (GitHub Inc, 2017)

des Quellcodes, der Logfiles von Microservice-Kommunikationen sowie eine Referenz zur Docker Container Engine benötigt (Granchelli et al., 2017b). MicroART verwendet sowohl statische als auch dynamische Analyse zur Architekturerkennung. Die dynamische Analyse erfordert im Gegensatz zur statischen, dass die einzelnen Applikationen (bzw. Microservices) im ausgeführten Zustand sind.

Das System besteht, wie in Abbildung 12 aufgezeigt, aus den drei Phasen „Physical Architecture Recovery“, „Service Discovery Identification“ und „Logical Architecture Recovery“. Während „Physical Architecture Recovery“ und „Logical Architecture Recovery“ technische, automatisierte Phasen sind, wird die „Service Discovery Identification“ semiautomatisiert durch einen Architekten durchgeführt.

Die Physical-Architecture-Recovery-Phase dient der Erzeugung eines Architekturmodells. Dies erfolgt durch statische Analyse des GitHub Repository sowie der Konfigurationsbeschreibung des Container-Management-Systems vereint mit dynamischer Analyse der Microservice-Interaktionen über deren Logfiles.

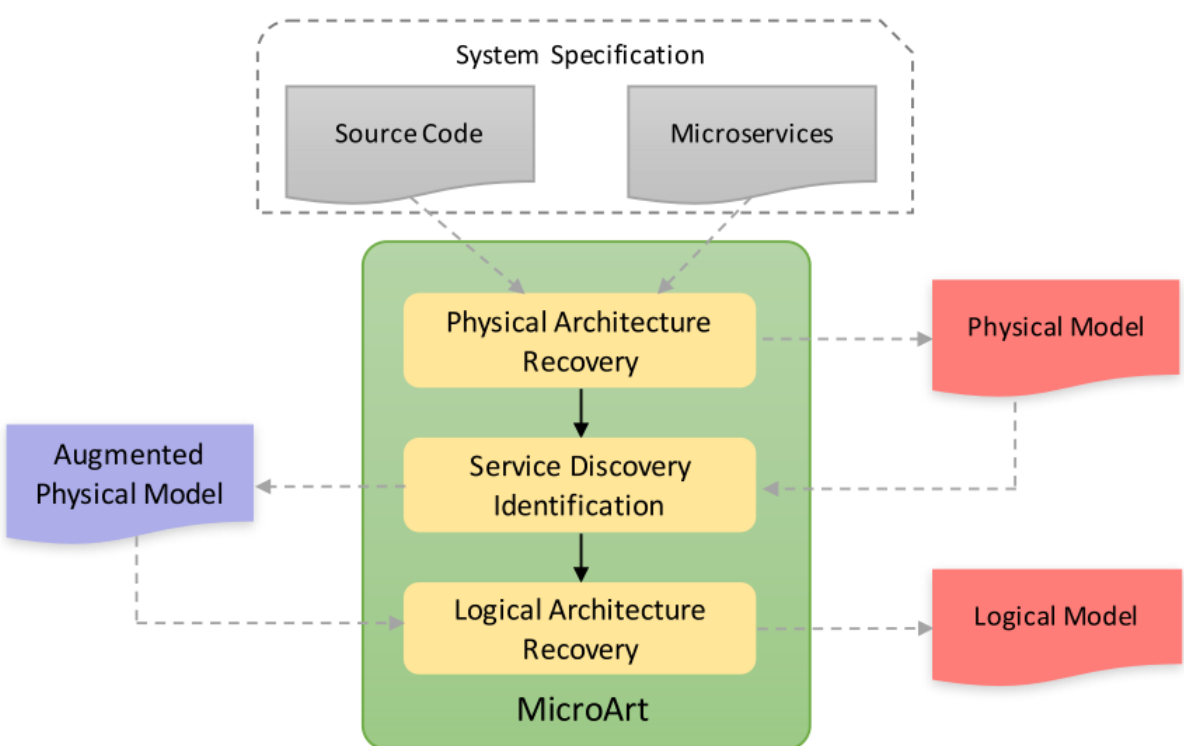


Abbildung 12: MicroART-Artefakte (Granchelli et al., 2017a)

Im Rahmen der Service-Discovery-Identification-Phase wird dem Architekten das aus der ersten Phase ermittelte „Physical Model“ visualisiert. Um die Korrektheit erkannter Relationen sicherzustellen, muss durch den Architekten jener Service ausgewählt werden, welcher das Service Discovery innerhalb der Infrastruktur ermöglicht.

Das sich hieraus ergebende erweiterte „Physical Model“ wird in der Phase des „Logical Architecture Recovery“ zusammen mit dem Input des Architekten verwendet, um Service Recovery Services aufzulösen und das „Logical Model“ zu erzeugen, welches vom Architekten für Analysen, Dokumentationen etc. weiterverwendet werden kann.

MicroART besteht aus vier Komponenten, welche der Realisierung der beschriebenen Phasen dienen und jeweils eine gesonderte Aufgabe erfüllen. Wie in Abbildung 13 dargestellt, bildet der GitHub Analyzer die erste Komponente ab. Dieser erhält als Input die URL des Repository, dessen Quellcode auf das lokale System geklont und dort analysiert wird. Dabei werden folgende Informationen extrahiert:

- Bezeichnung und Beschreibung des Systems
- Die Entwickler, welche über den Entwicklungszeitraum hinweg zur Entwicklung beigetragen haben

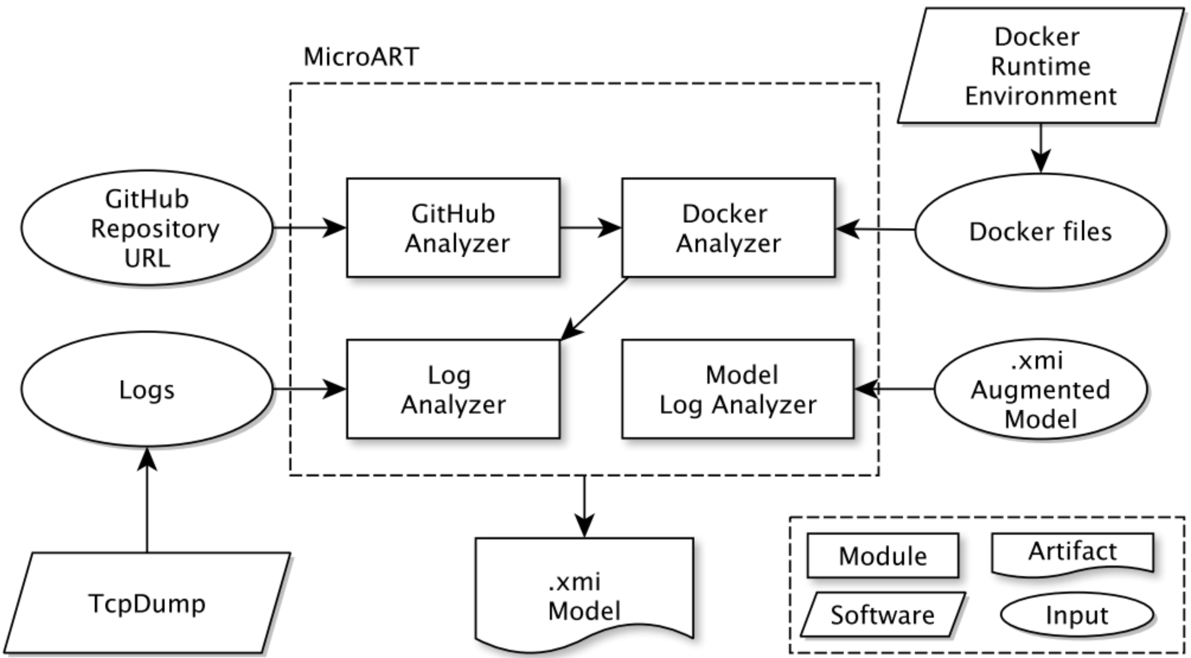


Abbildung 13: Struktur des MicroART-Systems (Granchelli et al., 2017a)

Die zweite Komponente ist der Docker Analyzer, welcher die Docker-Ausführungsumgebung abruft, um Network Interfaces und IP-Adressen der einzelnen Microservices zu ermitteln. Da diese Deployment-Informationen nicht im Repository enthalten sind, ist diese Komponente zwingend notwendig und erlaubt das Mapping der Services zum jeweiligen Netzwerk-Identifier.

Die dritte Komponente des Log Analyzer dient der dynamischen Analyse von durch die Microservices erzeugten Logdateien, welche die Interaktion zwischen den Services dokumentieren. Hiermit können die Beziehungen zwischen den Services erkannt werden.

Die vierte Komponente ist der Model Log Analyzer. Dieser verarbeitet das erweiterte „Physical Model" aus Phase zwei um anschließend die beschriebenen Logdateien nach relevanten Daten zu filtern. Hieraus resultiert schlussendlich das „Logical Model", welches die erkannte Architektur repräsentiert.

4.5 Abgrenzung

Alle der bisher beschriebenen Technologien und Systeme dieses Kapitels tragen zur Erkennung einer Architektur bei. Dies erfolgt in unterschiedlicher Form und variierendem Umfang. Ordnet man die erkannten Architekturentitäten den Ebenen einer EA (siehe Kapitel 2.2) zu, so kommt man, wie in Tabelle 4 erkennbar, zu der Erkenntnis, dass keine der vorgestellten Technologien die Ebenen einer EA vollständig abbildet.

Technologie	Business-Layer	Application-Layer	Technology-Layer	Intralayer-Relationen	Interlayer-Relationen
Eureka		x			
Process Mining	x			x	
Kieker/Dynatrace		x	x	x	x
MicroART		x	x	x	x
Prototyp	x	x	x	x	x

Tabelle 4: Vergleich der Architekturentitäten ausgewählter Technologien

Zu erkennen ist ebenfalls, dass die Technologien entweder den Application- und Technology-Layer oder den Business-Layer umfassen. Keine der Lösungen adressiert die Zusammenführung von Geschäfts- und Technologieebenen.

An dieser Stelle grenzt sich der im Rahmen dieser Arbeit entwickelte Prototyp ab. Dieser hat zum Ziel, alle Ebenen inklusive ihrer Intra- und Interlayer-Beziehungen in einem gemeinsamen Architekturmodell zusammenzuführen. Ein solches Modell ermöglicht eine gesamtheitliche Architekturbetrachtung sowie neue Möglichkeiten der Modellanalyse. Beispielsweise erlaubt ein vollständiges Modell die Ermittlung von Einflüssen einer technischen Störung auf Business-Entitäten wie hierdurch beeinträchtigte Prozesse und somit hieraus resultierende Kosten. Die neu geschaffene Transparenz der Abhängigkeiten von Businessentitäten und technischen Komponenten ermöglicht des Weiteren eine Grundlage zur Bewertung der Relevanz einzelner Services, welche für Skalierungsentscheidungen der Infrastruktur herangezogen werden können.

Des Weiteren grenzt sich die prototypische Implementierung in ihren Echtzeitanforderungen von den vorgestellten Systemen ab. Keines der Systeme stellt einen vollständig echtzeitfähigen Datenfluss von der Architekturerkennung bis zur Ergebnisbereitstellung zur Verfügung. Das Microservice Repository Eureka besitzt ein echtzeitnahes Abbild ausgeführter Services, stellt diese jedoch lediglich über eine REST-API bereit, welche keine Echtzeitverarbeitung von Änderungen zulässt. Das Business Process Mining basiert auf einer zyklisch durchgeführten Datenanalyse, welche die Erkennung von Änderungen in Echtzeit nicht gewährleistet. Das Dynatrace-System erfasst Änderungen in Echtzeit, analysiert jene aufgrund des Agentensystems jedoch verzögert. Das MicroART-System besteht aus drei Phasen, von welchen die zweite menschlichen Input benötigt, was eine Echtzeiterkennung verbietet. Das Kieker-System erkennt aufgrund des gewählten Distributed-Tracing-Ansatzes Änderungen in Echtzeit, stellt Ergebnisse jedoch nicht über eine für Echtzeitverarbeitung geeignete Schnittstelle zur Verfügung.

Die prototypische Implementierung erkennt Architekturerkennungen durch die Verwendung von Distributed Tracing mit der ersten Verwendung der geänderten Komponenten. Aufgrund der Verwendung von Streaming-Technologien werden gewonnene Informationen stets in Echtzeit bereitgestellt und können im Push-Verfahren von angebundenen Systemen verarbeitet werden. Dies ermöglicht die Visualisierung und Kommunikation von Architekturänderungen in Echtzeit. Deren Dokumentation stellt eine Grundlage für weitere Analysen dar. Beispielsweise ermöglicht sie das Erkennen von Korrelationen zwischen Architekturänderungen und Systemstörungen.

Auch in der Repräsentation des ermittelten Architekturmodells erfolgt eine Abgrenzung zu den vorgestellten Lösungen. Eureka verfügt über keine grafische Aufbereitung. Die weiteren Systeme bieten Darstellungen wie Abhängigkeitsgraphen und Sequenzdiagramme zur Visualisierung ihrer Modelle. Diese Formen der Darstellung führen bei einer hohen Dichte von Beziehungen zu komplexen und unübersichtlichen Graphen. Daher wird im Rahmen des Prototyps eine Repräsentation in Form einer Matrix gewählt. Während eine Matrix nicht in gleichem Maße intuitiv gelesen werden kann wie ein Graph, bleibt sie auch bei einer hohen Anzahl zu visualisierender Entitäten und Beziehungen leicht lesbar und verständlich.

5. Prototypische Implementierung Echtzeit-Architekturerkennung

Verteilte Systeme in Form von Microservices finden zunehmend Einzug in die Infrastruktur innovativer Unternehmen wie Netflix (Probst & Becker, 2016), Amazon (Wagner, 2015), Zalando (Schaefer, 2016) und Uber (Reinhold, 2016), wodurch Microservices vermehrt an Bedeutung gewinnen. Während sie aufgrund ihres abgegrenzten Aufgabenspektrums gut zu entwickeln und zu warten sind, steigt die Komplexität zugrundeliegender System-Architekturen aufgrund einer Vielzahl von miteinander kommunizierenden Microservices. Durch automatisierte und lastenabhängige Skalierung von einzelnen Services können Systeme im ständigen Wandel sein und aufgrund der Vielzahl von einzelnen Services schnell zu Intransparenz der realen technischen Strukturen führen.

Der im Rahmen dieser Arbeit entstandene Prototyp nimmt sich dieser Herausforderungen an. Er hat zum Ziel, eine verteilte Microservice-Architektur fortlaufend und in Echtzeit zu ermitteln und bereitzustellen. Dies umfasst sowohl die einzelnen Microservices und deren Hardwarebasis als auch deren Kommunikation und technische Abhängigkeiten. Die Verfügbarkeit eines in Echtzeit ermittelten Architekturmodells führt zur Transparenz des aktuellen System-Deployments und ermöglicht den Abgleich von Soll- und Istzustand des verteilten Systems. Der Prototyp erhebt den Anspruch, eine Architektur nicht auf ihre technischen Komponenten reduziert zu betrachten, sondern eine möglichst ganzheitliche Architektur im Sinne der Ebenen einer EA (Kapitel 2.2) abzubilden. Das Vorhandensein einer solchen Architektur bietet die Grundlage für die Analyse von Wechselwirkungen zwischen Geschäfts- und IT-Welt.

Die prototypische Implementierung wurde für den Einsatz in einem Systemumfeld konzipiert, welches dem in Kapitel 3.2.2 vorgestellten Referenzmodell entspricht. Einige der Komponenten des Referenzmodells sind zwingend zum Betrieb des Prototyps erforderlich, da diese als zu verarbeitende Datenquellen verwendet werden. Das Einsatzumfeld sowie notwendige Infrastruktur-Komponenten zum Betrieb des Prototyps werden in Kapitel 5.1 beschrieben.

Das prototypische Architekturerkennungssystem basiert auf der Analyse mehrerer Datenquellen in Kombination mit der Anreicherung von Geschäftssemantik durch

Anwender. Um mithilfe dieser Informationen die zugrundeliegende Microservice-Architektur zu ermitteln, wurden mehrere Systemkomponenten entwickelt. Kapitel 5.2 vermittelt einen Überblick über jene Komponenten, deren Einsatzzweck und Wirkungsweise.

Das Zusammenwirken der vorangegangenen Systemkomponenten des Prototyps führt zu der Bereitstellung eines Architekturmodells, welches das Ergebnis der Datenverarbeitung und -zusammenführung darstellt. Es beschreibt die Komponenten und Beziehungen der ermittelten Microservice-Architektur in einer maschinell verwertbaren Form und ermöglicht dadurch anschließenden Systemen die Verwendung und Analyse der ermittelten Architektur. Struktur und Elemente des Architekturmodells werden in Kapitel 5.3 eingehend erläutert.

Nachdem Wirkungsweise und Ergebnis der prototypischen Implementierung aufgezeigt sind, können Kapitel 5.4 technische Details der Implementierung des Prototyps entnommen werden. Dies umfasst die Beschreibung der Applikationsstruktur, des zugrundeliegenden Datenmodells sowie der Algorithmen zur Verarbeitung der genutzten Datenquellen, aus welchen Architekturinformationen gewonnen und zusammengeführt werden.

5.1 Einsatzumfeld der prototypischen Implementierung

Die Definition eines Einsatzumfelds, welches die Funktionsfähigkeit des Prototyps gewährleistet, wurde mit dem Ziel möglichst geringfügiger Abhängigkeiten zu konkreten Systemen getroffen. Die Auswahl genutzter Datenquellen und unterstützter Architekturformen basiert auf der hohen Verbreitung jener Systeme im Microservice-Umfeld sowie der in Kapitel 3.2.2 beschriebenen Referenzarchitektur eines verteilten Systems.

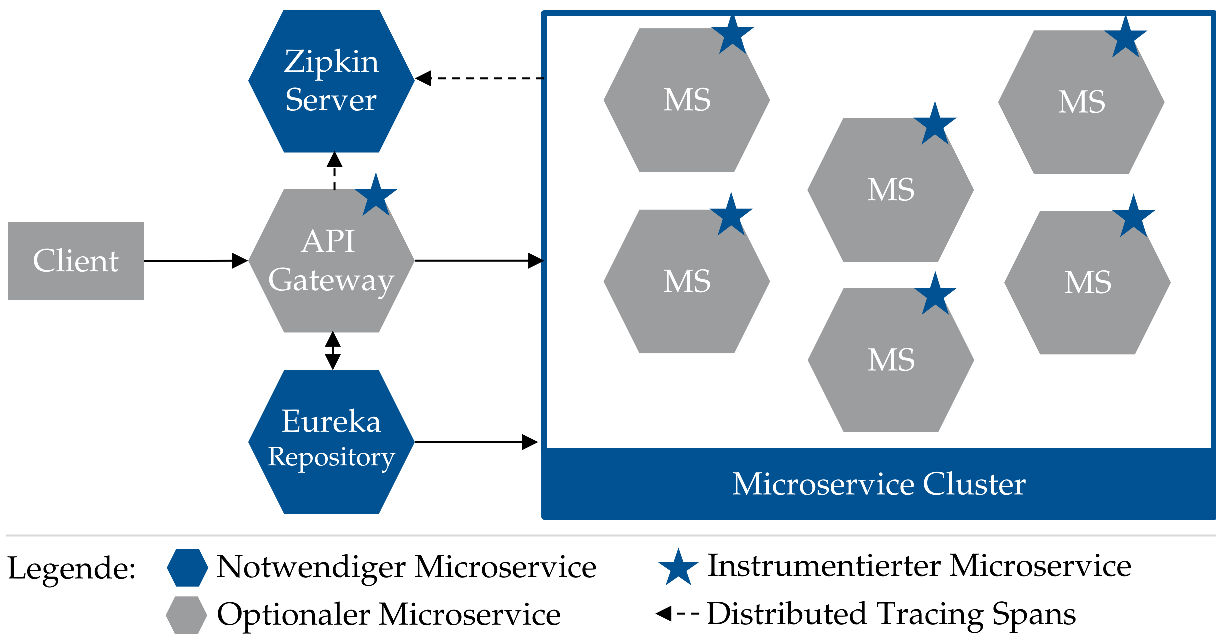


Abbildung 14: Referenzumfeld des Prototyps

Abbildung 14 stellt ein Referenzumfeld dar, welches durch die prototypische Implementierung unterstützt wird. Die Einsatzfähigkeit wird aufgrund der in der Abbildung blau markierten Elemente ermöglicht, welche die Datenbasis für die Algorithmen der Architekturerkennung bereitstellen.

Der Prototyp wurde zur Ermittlung von verteilten Systemen in Form von Microservice-Clustern entwickelt. Es ist erforderlich, dass jeder Microservice jenes Clusters über eine Instrumentierung durch ein Distributed-Tracing-System verfügt, welche die Kommunikation zwischen Microservices dokumentiert und zur Analyse bereitstellt. Des Weiteren müssen jene Microservices derart konfiguriert sein, dass sie ihre Verfügbarkeit an ein zentrales Service Repository melden.

Durch die prototypische Implementierung werden Monitoring-Daten sowie bereitgestellte Architekturinformationen eines Service Repository zur Architekturerkennung verarbeitet. Während das zugrundeliegende Konzept der Datenverarbeitung unabhängig von einer konkreten Implementierung jener Datenquellen ist, sind die im Rahmen des Prototyps unterstützten Implementierungen auf die Distributed-Tracing-Technologie Zipkin (siehe Kapitel 3.3.1) sowie die Service-Repository-Technologie Eureka (siehe Kapitel 4.1) beschränkt. Diese Abhängigkeit bedingt das

Vorhandensein jener Technologien, um die Einsatzfähigkeit des Prototyps zu gewährleisten.

Die im Zuge des Distributed Tracing instrumentierten Clients des Microservice-Clusters vermitteln die durch sie erzeugten Spans an den zentralen Zipkin-Server. Dies erfordert nicht, dass die Services zwingend durch Zipkin-Clients instrumentiert werden müssen. Die Instrumentierung muss jedoch Spans in einem durch den Zipkin-Server unterstützten Format bereitstellen.

In realen Anwendungsszenarien sind häufig Gateway-Services im Einsatz, welche eine zentrale Schnittstelle zur Nutzung eines Microservice-Clusters für Konsumenten darstellen. Gateway-Services erzeugen eine besondere Charakteristik des Datenflusses, welche von einem direkten Zugriff auf die Microservices stark abweicht. Der Architekturerkennungsservice ist zu einer Gateway-Architektur kompatibel, ohne diese zu bedingen.

5.2 Systemkomponenten der prototypischen Implementierung

Die prototypische Implementierung erzeugt und pflegt auf Basis der Datenquellen Zipkin und Eureka ein Datenmodell, welches den analysierten Ist-Zustand der Architektur wiedergibt und der Erkennung von Architekturänderungen dient. Durch die Verarbeitung der Datenquellen werden Architekturinformationen extrahiert und mit dem derzeitigen Datenmodell verglichen. Ergibt sich hieraus die Erkenntnis, dass eine Architekturkomponente, z. B. ein Service oder eine Abhängigkeit zwischen Services, hinzugekommen oder entfallen ist, wird diese Erkenntnis durch den Prototyp dokumentiert und bereitgestellt. Anschließend wird das Datenmodell den neuen Erkenntnissen entsprechend angepasst.

Da sich aus der Datenanalyse lediglich technische Ebenen einer EA ermitteln lassen, der Prototyp jedoch den Anspruch erhebt, die Architektur weitreichend abzubilden, umfasst die prototypische Implementierung des Weiteren eine REST-API sowie eine Weboberfläche. Diese finden unter anderem Verwendung bei der Modellierung von Geschäftsobjekten des Business-Layers. Des Weiteren dient die Weboberfläche der Pflege eines Regelwerks, welches die Abhängigkeiten zwischen Komponenten des Business-Layers und des Application-Layers beschreibt.

Durch die Verknüpfung analysierter sowie über die Weboberfläche modellierter Daten kann ein Architekturabbild entsprechend des Konzepts der Enterprise-Architecture-Layer (siehe Kapitel 2.2) erzeugt werden. Architekturänderungen können zeitnah erkannt und in Echtzeit bereitgestellt werden. Dies bedingt technische Systeme zur Dateneingabe, Datenverarbeitung und Datenausgabe. Abbildung 15 stellt alle Komponenten des prototypischen Systems und deren Kommunikationswege dar. Die Art der Kommunikation wird unterschieden in durch den Empfänger beauftragten Datenversand (Pull, durchgängige Pfeile) sowie durch den Versender unaufgefordert initiierten Datenversand (Push, gestrichelte Pfeile).

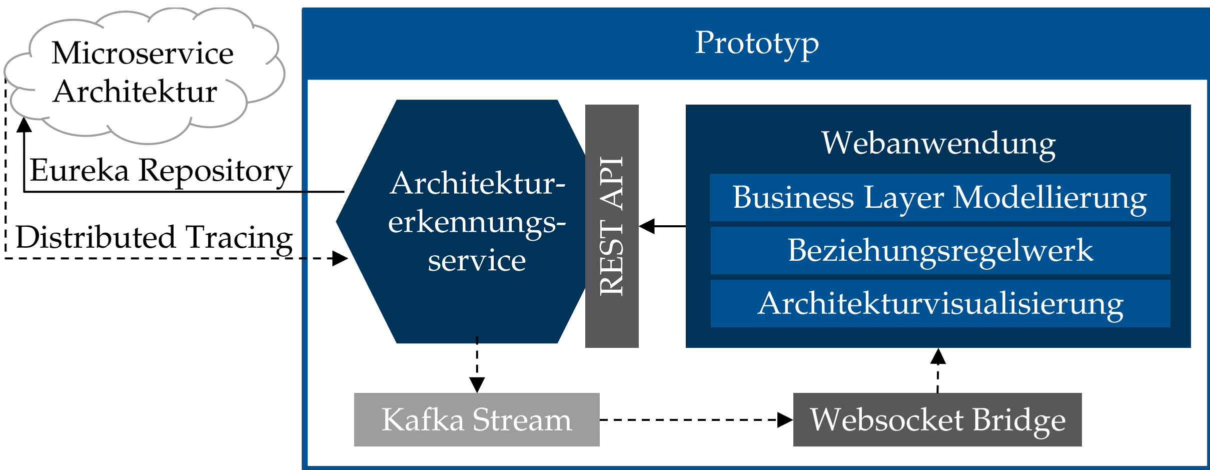


Abbildung 15: Systemkomponenten der prototypischen Implementierung

Die zentrale Komponente des Prototyps bildet der Architekturerkennungsservice. Dieser pflegt das Datenmodell, erzeugt das zur Ausgabe verwendete Architekturmodell und detektiert Architekturveränderungen anhand eingehender Daten. Die Webanwendung dient unter Verwendung der durch den Erkennungsservice bereitgestellten REST-API der Modellierung des Business-Layers, der Pflege des Regelwerks zur Erzeugung von Relationen zwischen Geschäftsaktivitäten und Services sowie der Visualisierung des vom Architekturerkennungsservice bereitgestellten Architekturmodells. Der Anforderung an Echtzeitdatenverarbeitung wird der Prototyp durch das Streaming von Architekturänderungen über die Streaming-Technologie Apache Kafka sowie Websockets gerecht. In den folgenden Unterkapiteln werden diese Systemkomponenten eingehend beschrieben.

5.2.1 Architekturerkennungsservice

Der Architekturerkennungsservice stellt die zentrale Applikation des Architekturerkennungssystems dar. Es handelt sich hierbei um einen in der Java-Technologie angesiedelten Applikationsserver, welcher unter Einsatz des Spring Cloud Framework (Pivotal Software Inc., 2017a) implementiert wurde.

Der Service agiert auf Basis der zwei Eingangsdatenquellen Eureka Service Repository (Kapitel 4.1) und Distributed Tracing (Kapitel 3.3). Diese Datenquellen dienen der kontinuierlichen Gewinnung von Architekturinformationen, welche mit dem Datenmodell des Erkennungsservice abgeglichen werden, um Architekturveränderungen zu detektieren und das Datenmodell anschließend zu aktualisieren. Durch die REST-API des Service können Teile des Datenmodells modifiziert oder Informationen dessen bereitgestellt werden.

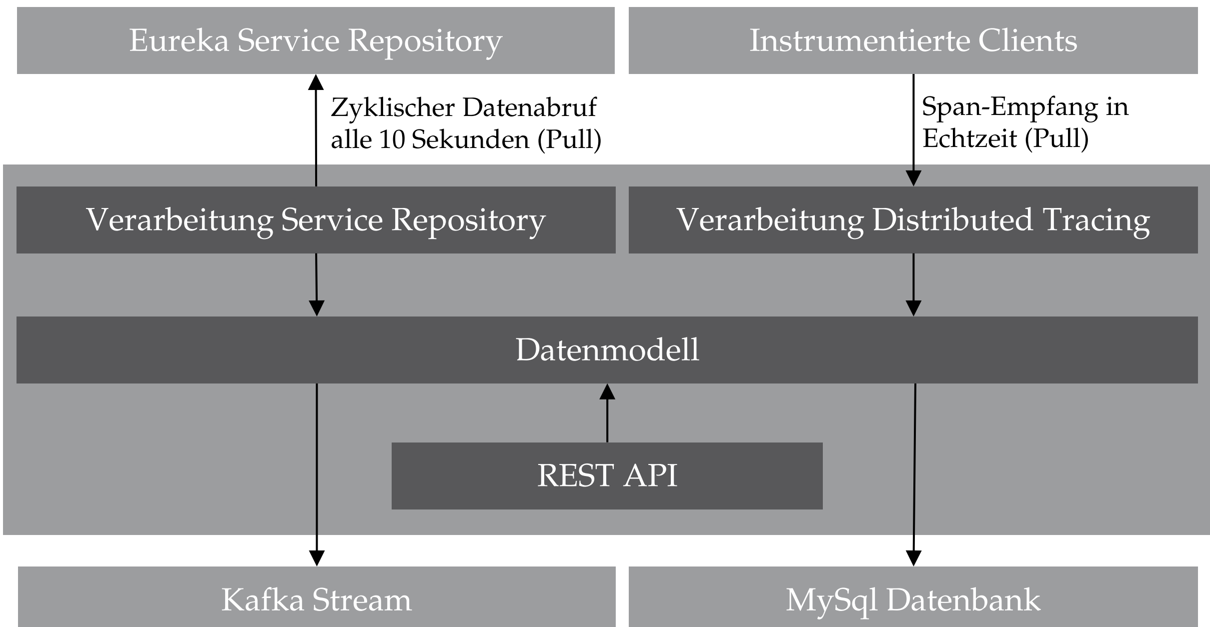


Abbildung 16: Vereinfachtes Schema des Architekturerkennungsservice

Der Architekturerkennungsservice lässt sich entsprechend in drei Module einteilen, über welche in diesem Kapitel ein Überblick gegeben wird:

- Verarbeitung der Service-Repository-Daten
- Verarbeitung der Distributed-Tracing-Daten

- Datenbereitstellung und Servicekonfiguration durch REST-API

All diese Module werden im Parallelbetrieb in eigenen dedizierten Threads ausgeführt. Dies ist notwendig, um eingehende Daten zur Auswertung von Architekturänderungen in Echtzeit verbreiten zu können. Eine besondere Herausforderung ist hierbei die Implementierung des Datenmodells. Denn wie in Abbildung 16 dargestellt, greifen alle drei Module auf das Datenmodell lesend und schreibend zu. Zur Absicherung des Parallelbetriebs wurde dieses daher ausschließlich mit Thread-sicheren Datenstrukturen realisiert, welche gleichzeitiges Lesen und Schreiben sowie das modifikationsfreie Iterieren durch dessen Datencontainer sicherstellen. Weiterführende Implementierungsdetails des Datenmodells können Kapitel 5.4 entnommen werden.

Führt eines der Module aufgrund einer Architekturänderung zur Modifikation des Datenmodells, wird diese Veränderung historisiert und durch den Service in Echtzeit über seine Streaming-Schnittstelle kommuniziert. Während der aktuellste Stand des Datenmodells durch die Applikation im Arbeitsspeicher verfügbar gehalten wird, um performante Soll-Ist-Abgleiche durchführen zu können, werden ältere Zustände des Modells in einer MySQL-Datenbank abgespeichert. Hierbei wird nicht der gesamte Zustand, sondern lediglich die Änderung am Modell persistent gesichert. Wird ein Architekturabbild zu einem vergangenen Zeitpunkt über die REST-API abgerufen, so wird das angeforderte Modell auf Basis der in der Datenbank hinterlegten Informationen erzeugt.

Konzept zur Verarbeitung der Distributed-Tracing-Daten

Das Distributed Tracing wird als Datenquelle verwendet, um kommunizierende Microservices in Echtzeit zu erkennen und deren Kommunikation mit anderen Microservices zu analysieren. Die Analyse dieser Datenquelle führt zu folgenden Veränderungen des Datenmodells:

- Hinzufügen ermittelter Architekturkomponenten (Services, Serviceinstanzen, Hardware), durch welche Spans versendet wurden.
- Hinzufügen von Intralayer-Relationen der Service-Komponenten, welche aus der Analyse paarweiser Kommunikationen abgeleitet wurden.

- Hinzufügen von Interlayer-Relationen zwischen Aktivitäten und Services, welche aus Spaninformationen in Kombination mit einem Regelwerk (siehe Kapitel 5.2.2) ermittelt werden konnten.

Weiterführende Informationen zu den Datenmodellmodifikationen und deren Ursprung können den Implementierungsdetails des Moduls in Kapitel 5.4.3 entnommen werden.

Der Algorithmus zur Verarbeitung der Distributed-Tracing-Datenquelle nutzt zur effizienten Auswertung eingehender Spans einen Zipkin-Server (siehe The OpenZipkin Authors, 2017d), welcher um die prototypische Implementierung zur Spananalyse erweitert wird. Dies erlaubt dem Algorithmus die Nutzung der durch die Server-Implementierung mitgebrachten Funktionen wie der Entgegennahme, Validierung und Persistierung eingehender Spans ohne zusätzlichen Implementierungsaufwand.

Wie in Abbildung 17 dargestellt, verfügt ein Zipkin-Server zum Empfang eingehender Spans über verschiedene Kollektoren. Dies ermöglicht die Span-Übermittlung über diverse Transporttechnologien, wie beispielsweise die Streaming-Technologie Apache Kafka oder eine Übertragung via http an eine REST-API. Empfangene Spans werden durch die Kollektoren in ein uniformes Java-Objekt überführt, wodurch sie von der zum Transport verwendeten Datenstruktur abstrahiert werden. In einer anschließenden Validierung stellt der Zipkin-Server sicher, dass weiterverarbeitete Span-Objekte inhaltlich plausibel sind. Ist dies der Fall, erfolgt eine Übergabe der Java-Objekte an ein Storage-Modul. Dieses verantwortet die persistente Datenspeicherung und -bereitstellung. Das Zipkin-Server-Projekt verfügt von Haus aus über mehrere Implementierungen eines Storage-Moduls, wie unter anderem eine Arbeitsspeicher-basierte oder eine MySQL-Datenbank-gestützte Variante.

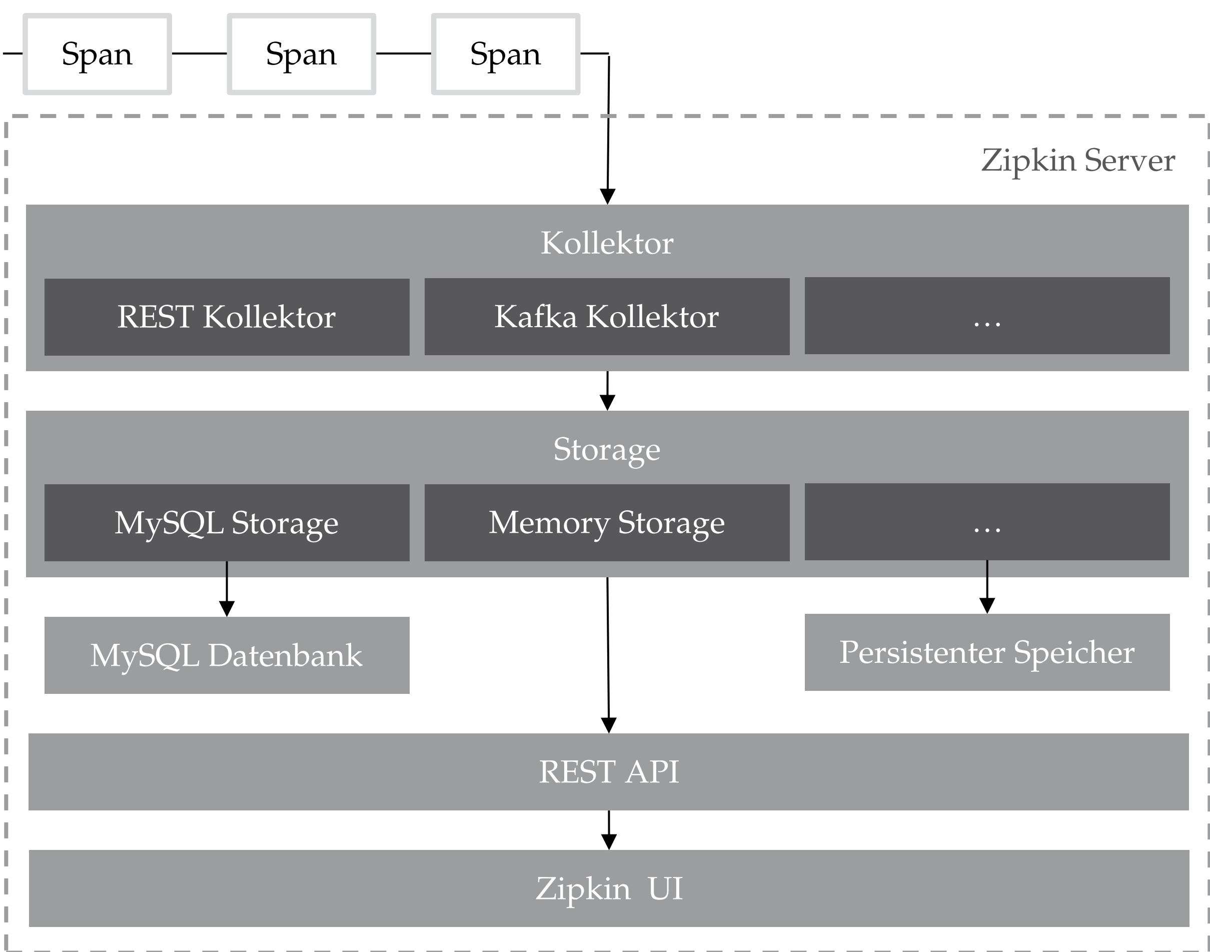


Abbildung 17: Datenflussdiagramm eines Zipkin-Servers

Um die gegebenen Funktionalitäten der verzögerungsfreien Eingangsverarbeitung, Abstraktion und Validierung von Spans durch einen Zipkin-Server für den Architekturerkennungsservice nutzbar zu machen, wird durch den Prototyp in den Datenfluss der Zipkin-Implementierung eingegriffen. Das Storage-Modul des Servers wird durch eine Eigenimplementierung ersetzt. Diese übernimmt jedoch nicht selbst die Aufgabe der Speicherung, sondern tritt als Proxy in Erscheinung. Sie instanziiert ein Standard-Zipkin-MySQL-Storage-Objekt und leitet alle Funktionsaufrufe des eigenimplementierten Storage-Objekts an dieses weiter, um die Datenspeicherung analog dem direkten Einsatz eines MySql Storage zu leisten. Wie in Abbildung 18 zu erkennen, werden nun jedoch eingehende Spans zusätzlich an den Datenanalysealgorithmus des Prototyps weitergereicht, wodurch diesem validierte Spans in Form uniformer Java-Objekte zur Datenanalyse bereitgestellt werden.

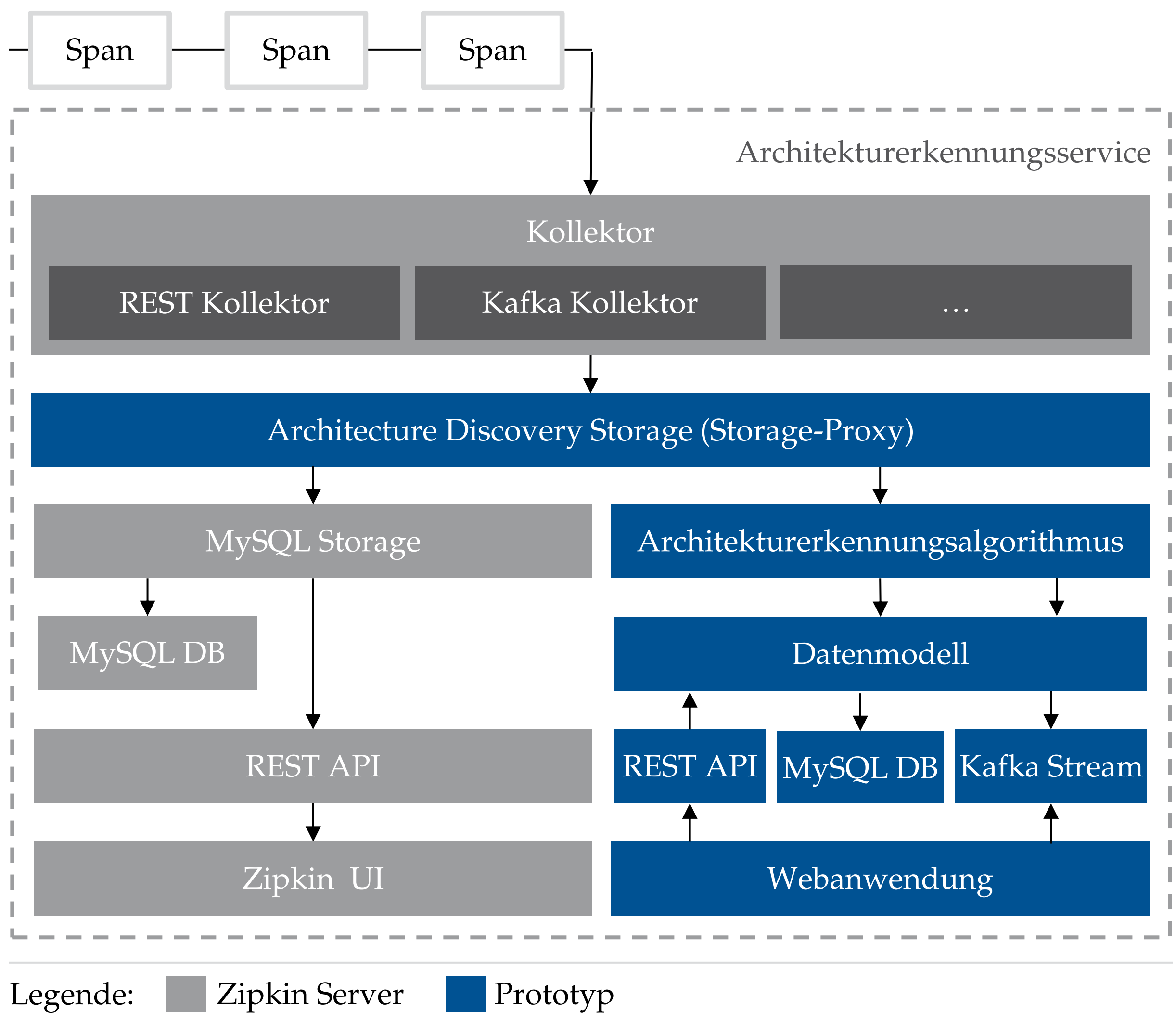


Abbildung 18: Ausschnitt des Datenflussdiagramms der Architekturerkennung

Durch den Einsatz der Proxy-Storage-Komponente bleiben alle nativen Funktionalitäten des Zipkin-Servers intakt und werden somit auch durch den Architekturerkennungsservice selbst bereitgestellt. Dies bedeutet, dass in einer bestehenden instrumentierten Microservice-Architektur der Zipkin-Server ohne Funktionsverlust oder Neukonfiguration durch den Architekturerkennungsservice substituiert werden kann.

Verarbeitung der Service-Repository-Daten

Die Verarbeitung des Service Repository ist essentiell, um Unzulänglichkeiten des Distributed-Tracing-Verfahrens zu ergänzen. Das Modul zur Analyse des Distribu-

ted Tracing ist in der Lage, Architekturinformationen aus bestehenden Datenströmen zu extrahieren. Es ist jedoch nicht in der Lage, das Entfallen in der Vergangenheit erkannter Elemente festzustellen. Des Weiteren bedarf die Erkennung durch Distributed Tracing das Eingehen von externen Anfragen, um die Span-Erzeugung durch die verarbeitenden Architekturkomponenten auszulösen. Sind Komponenten selten oder nie in eine Anfrageverarbeitung involviert, bleibt die Erzeugung von Spans aus, wodurch die Komponente für das Distributed-Tracing-System nicht sichtbar wird und somit nicht detektiert werden kann.

Diese Einschränkungen werden durch das Modul zur Verarbeitung der Service-Repository-Daten aufgehoben. Die Analyse der Datenquelle führt zu folgenden Veränderungen des Datenmodells:

- Hinzufügen ermittelter Architekturkomponenten (Services, Serviceinstanzen, Hardware), welche an dem Repository angemeldet sind.
- Entfernen von Architekturkomponenten (Services, Serviceinstanzen, Hardware), welche im Datenmodell vorhanden, jedoch nicht (mehr) am Service Repository angemeldet sind.

Wie in Kapitel 4.1 beschrieben, verfügt das Service Repository zu jeder Zeit über ein Abbild aller vorhandenen Services des Systems. Es stellt diese Informationen über eine REST-API bereit, was ein aktives Abfragen (Pull) dieser Informationen erfordert. Über Datenänderungen wird der Erkennungsservice nicht informiert, wodurch eine zyklische Abfrage der REST-API erforderlich ist. Das Modul führt in einem Intervall von zehn Sekunden die Datenverarbeitung durch, welche durch den Endpunkt „GET /eureka/v2/apps“ des Eureka Service Repository ein Gesamtabbild registrierter Serviceinstanzen analysiert.

Die aus diesen Informationen extrahierten Architekturinformationen werden mit dem Datenmodell auf drei Weisen abgeglichen. Abbildung 19 stellt die ersten beiden Abgleiche dar, bei welchen das Datenmodell um neue Elemente bereichert und um entfallene reduziert wird, ohne dabei die Schnittmenge der beiden Datenmengen zu modifizieren.

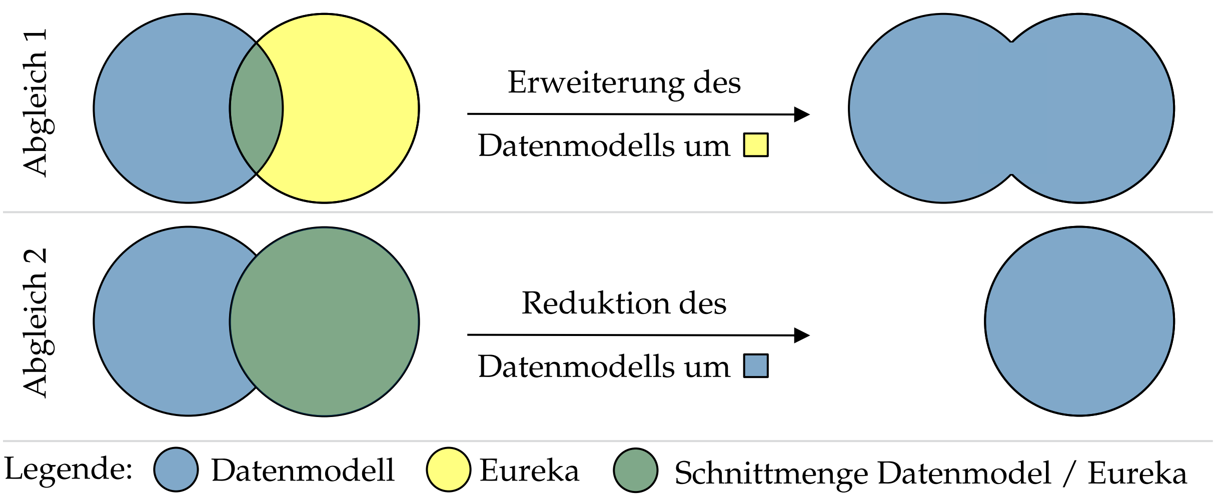


Abbildung 19: Abgleich Datenmodell mit Eureka

Durch den ersten Abgleich werden dem Datenmodell alle Komponenten und Relationen hinzugefügt, welche seit dem letzten Abgleich neu bei Eureka angemeldet wurden. Im zweiten Abgleich werden aus dem Datenmodell alle Komponenten und Relationen entfernt (bzw. es wird deren Gültigkeitszeitraum beendet), welche nicht mehr bei Eureka angemeldet sind und entsprechend nicht mehr zur Architektur gehören. Neben der Information über die Anwesenheit eines Architekturelements umfassen die bereitgestellten Daten auch den Zeitpunkt der Anmeldung am Repository. In einem dritten Abgleich wird überprüft, ob der letzte Anmeldungszeitpunkt am Repository vor oder nach der Erzeugung einer Revision im Datenmodell erfolgte. Eine Anmeldung nachher ist der Fall, wenn sich ein Element nach der Entdeckung ab- und wieder anmeldete, z. B. aufgrund eines neuen Softwarereleases. Dies führt ebenfalls zu einer Veränderung des Datenmodells (Revisionswechsel, siehe Kapitel 5.4.4).

REST-API

Durch die beschriebenen Module werden Architekturkomponenten der IT-Infrastruktur erkannt und im Datenmodell manifestiert. Architekturkomponenten der Geschäftsebene können durch diese nicht erkannt werden. Das Ziel einer umfassenden Architekturrepräsentation nach dem Konzept der EA-Layer erfordert somit eine weitere Datenquelle zur Erfassung von Architekturkomponenten der Geschäftsebene. Dem dient die REST-API des Architekturerkennungsservice. Sie bietet Endpunkte zur Erzeugung von Geschäftsprozessen und -aktivitäten an. Die zum Prototyp gehörende Webanwendung nutzt jene Endpunkte in Verbindung mit einer

Prozessmodellierungsoberfläche, um das Datenmodell des Architekturerkennungs-service um Geschäftsobjekte zu bereichern und somit den Business-Layer der EA zu modellieren. Dies führt zu folgenden Änderungen am Datenmodell des Erkennungs-service:

- Hinzufügen/Entfernen modellierter Architekturkomponenten (Prozesse, Aktivitäten).
- Hinzufügen/Entfernen von Intralayer-Relationen zwischen Aktivitäten.
- Hinzufügen/Entfernen von Interlayer-Relationen zwischen Prozessen und Aktivitäten.

Des Weiteren bietet die REST-API Endpunkte zur Pflege eines Regelwerks. Dieses wird durch das Distributed-Tracing-Modul verwendet, um Relationen zwischen Aktivitäten der Geschäftswelt und Services der IT-Welt zu erzeugen. Das Regelwerk wird im Rahmen des Kapitels 5.2.2 eingehend beschrieben. Außerdem verfügt die API über eine umfangreiche Sammlung von Endpunkten zur Datenbereitstellung auf Basis des internen Datenmodells. Dies umfasst die Bereitstellung eines vollständigen Architekturmodells, einzelner Architekturelemente sowie Metainformationen zu jenen Elementen.

Der Endpunkt zur Abfrage einer vollumfänglichen Architektur („GET /api/v1/ad/architecture/“) ist von besonderer Bedeutung, da er das Berechnungsergebnis des Service bereitstellt und als Schnittstelle weiterverarbeitender Applikationen fungiert. Das durch den Endpunkt zurückgelieferte und als Architekturmodell bezeichnete Datenschema umfasst die einzelnen Komponenten, Revisionen und Beziehungen eines Architekturabbilds, welche in Kapitel 5.3 in ihrer Vielfalt beleuchtet werden. Anhang C kann ein durch den Endpunkt bereitgestelltes Architekturmodell in Auszügen entnommen werden.

Der Endpunkt dient verschiedensten Anwendungsszenarien, wie einer Architektur-visualisierung, der als Impact Analysis bezeichneten Analyse des Einflusses einer Störung auf andere Architekturkomponenten oder der als Rootcause Analysis bezeichneten Ursachenanalyse auftretender Anomalien innerhalb einer Architektur. Somit wurde der Endpunkt mit dem Ziel konzipiert, durch verschiedene Filterungs-parameter ein breites Anwendungsspektrum zu bedienen. Da ein Architekturmodell

in einer breiten Systemlandschaft mit einer Vielzahl an Metadaten zu großen Datenmengen führen kann, erlauben die Parameter das Steuern der Datenmenge entsprechend des Anwendungsfalls. Folgende Parameter können zur Konfiguration des zurückgegebenen Architekturmodells verwendet werden:

snapshot

Der Parameter „snapshot“ dient der Angabe eines Zeitstempels in Millisekunden, um den Zustandszeitpunkt des angeforderten Architekturmodells zu definieren. Dies ermöglicht, einen historisierten Architekturzustand der Vergangenheit abzurufen. Wird der Parameter nicht übergeben, wird der Zeitpunkt des Eintreffens der REST-Anfrage verwendet.

annotation-filter

Die Metadaten einer Revision, Komponente oder Relation werden als Annotationen bezeichnet. Diese Key-Value-Paare reichern die Elemente mit aus der Datenanalyse oder manuellen Erfassung erhobenen Zusatzinformationen an. Der Parameter „annotation-filter“ kann zur Definition einer komma-separierten Liste von Keys genutzt werden, um gewünschte Metainformationen zu spezifizieren. Ist der Parameter undefiniert, werden alle vorhandenen Annotationen angefügt, was zu großen Datenmengen führen kann.

component-type-filter

Wie in Kapitel 5.3.1 dargelegt, ist eine Komponente einem von fünf Typen zugeordnet. Dieser Parameter erlaubt die Definition einer komma-separierten Liste von Komponententypen, auf welche sich das Architekturmodell beschränken soll. Somit können irrelevante Layer der EA in einem Anwendungsszenario unberücksichtigt bleiben. Erlaubte Werte sind: „process“, „activity“, „service“, „service-instance“ und „hardware“. Erfolgt keine Definition, umfasst das ausgelieferte Architekturmodell alle Komponententypen.

relation-filter

Zwischen Revisionen bestehende Beziehungen können im zurückgegebenen Architekturmodell auf verschiedene Weisen repräsentiert werden. Folgende Optionen stehen zur Verfügung:

- „PARENTS“: Einer Revision sind alle Relationen zu Elternelementen angefügt.
- „CHILDREN“: Einer Revision sind alle Relationen zu Kindelementen angefügt.
- „ALL“: Einer Revision sind sowohl Relationen zu Elternelementen als auch zu Kindelementen angefügt.
- „NONE“: Relationen sind nicht im Architekturmodell vermerkt.

Wird der Parameter nicht gesetzt, kommt die Konfiguration „ALL“ zum Einsatz. Die Verwendung dieses Parameters ist besonders sinnvoll, wenn durch das Architekturmodell in nur eine Richtung (Bottom-Up, Top-Down) traversiert wird und somit Relationen in die entgegengesetzte Richtung ohne Verwendung bleiben.

5.2.2 Webanwendung

Die Webanwendung ist eine auf JavaScript basierende Applikation, welche auf dem Framework Angular4 der Google Inc. (Goolge Inc., 2017) basiert. Zur Gestaltung und Verwendung auf Geräten verschiedener Auflösungen wurde ein Template verwendet, welches auf der Hypertext Markup Language (HTML), auf Cascading Style Sheets (CSS) sowie auf der JavaScript-Bibliothek Bootstrap (Bootstrap, 2017) basiert. Die Applikation nutzt sowohl den Apache Kafka Stream unter Verwendung einer Websocket Bridge (siehe Kapitel 5.2.3), um den Anwender in Echtzeit über Änderungen am Datenmodell zu informieren, als auch die REST-API des Architekturerkennungsservice, um das Architekturmodell abzurufen oder den Architekturerkennungsservice zu konfigurieren. Die Webanwendung untergliedert sich in drei Hauptseiten, welche im Folgenden beschrieben werden.

Adjacency-Matrix

Die Seite „Adjacency-Matrix“ dient der Repräsentation einer durch den Architekturerkennungsservice bereitgestellten Architektur. Die Datenbasis bietet der Endpunkt zur Abfrage von Architekturmodellen (siehe Kapitel 5.2.1). Ein Architekturmodell kann aus einigen hundert Komponenten bestehen. Während die baumförmige Darstellung der Architektur aufgrund der hierarchischen Struktur naheliegend wäre, führt sie bei großen Datenmengen zu unüberschaubaren Ergebnissen. Außerdem ist mit einem solchen Graphen die Visualisierung indirekter Relatio-

nen (Kapitel 5.3.3) nur schwer realisierbar. Daher wurde im Rahmen dieser Arbeit die Datenrepräsentation in Form einer Adjacency-Matrix gewählt. Diese führt sowohl in ihren Zeilen als auch in den Spalten alle vorhandenen Komponenten einmal auf. Besteht eine Relation zwischen zwei Komponenten, so ist dies durch ein X in dem Feld, in welchem sich Zeile und Spalte der relevanten Komponenten schneiden, gekennzeichnet. Da Beziehungen des Architekturmodells stets gerichtet sind, ist die Matrix nach folgender Maßgabe zu lesen: Den Beginn einer Relation stellen die Zeilen dar, während die Spalten das Ziel der gerichteten Relation darstellen.

Abweichend von üblichen Adjacency-Matrizen erfolgt des Weiteren eine semantische Gruppierung und Ordnung der Zeilen und Spalten entsprechend dem Komponententyp. Komponenten gleichen Typs werden mit einer gemeinsamen Farbe eingefärbt. Die folgenden Typen werden unterschieden: „Process“, „Activity“, „Service“, „Instance“ und „Hardware“.

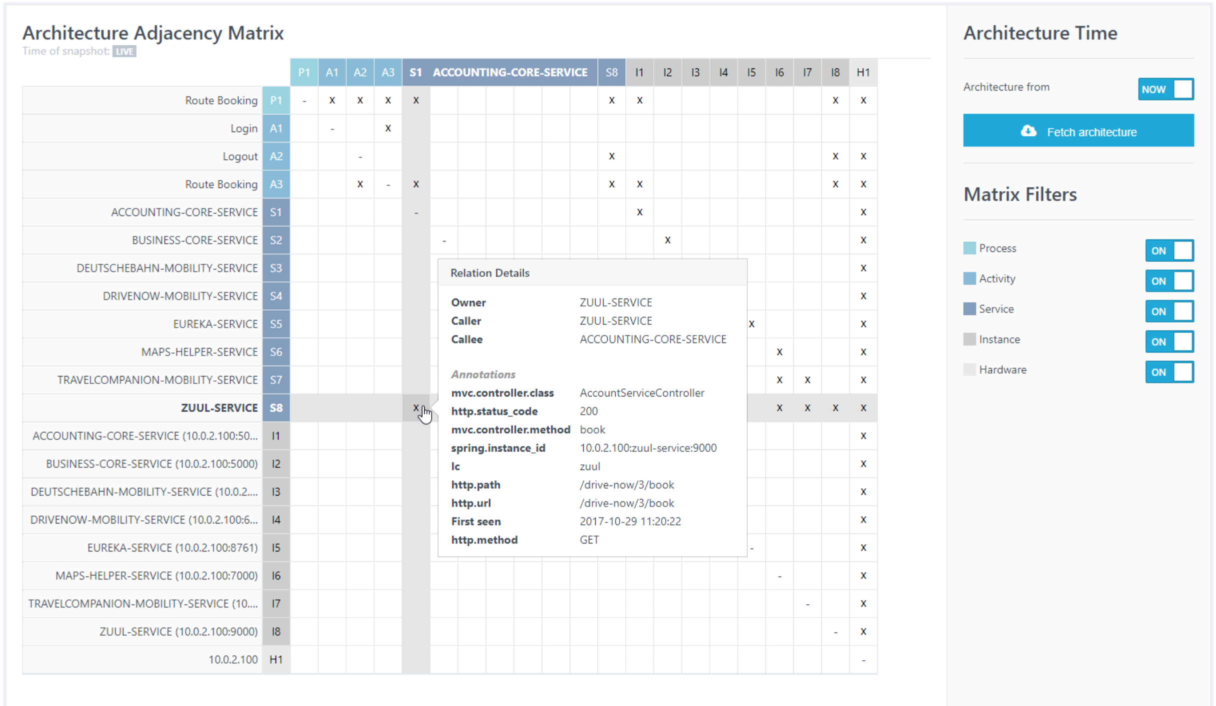


Abbildung 20: Seite „Adjacency Matrix“ der Webanwendung

Eine Relation kann über zusätzliche Informationen, sog. Annotationen verfügen, welche durch das Architekturmodell bereitgestellt werden. Ein Klick auf das mit X

markierte Feld öffnet ein weiteres Fenster, welches die Relationen-Annotationen sowie Detailinformationen bereithält.

Die Anwendung bietet zwei Modi zur Auswahl eines Architekturmodells. Zum einen kann das Architekturmodell in seinem derzeit bekannten Zustand abgerufen werden. Wird dies ausgewählt, erhält der Anwender einen Hinweis, sobald die angezeigte Repräsentation aufgrund einer Modelländerung nicht mehr dem letzten Modellstand entspricht. Zu diesem Zweck wird der Websocket-Stream auf Nachrichten über Modellveränderungen abgehört. Der zweite Modus ermöglicht die Angabe eines Zeitpunkts in der Vergangenheit, um einen historisierten Architekturmodellstand zu visualisieren.

Eine große Herausforderung in der Visualisierung des Architekturmodells besteht in der potentiellen Vielzahl an Architekturkomponenten und der entsprechend großen zu visualisierenden Datenmenge. Zu diesem Zweck sind verschiedene Mechanismen implementiert. Ist die Adjacency-Matrix größer als der verfügbare Raum des Browsers, so kann sie gescrollt werden. Beim Scrollen auf der jeweiligen Achse werden die jeweiligen Titelspalten nicht mitgescrollt, so dass jederzeit sichtbar ist, welcher Komponente eine Zeile oder Spalte zuzuordnen ist. Um die einzelnen Spalten in ihrer Breite schmal zu halten, werden anstatt der Komponententitel lediglich Nummerierungen wiedergegeben. Wird die Maus über einem Feld positioniert, so werden Zeile und Spalte farbig markiert und der Titel der Spalte überlagernd eingeblendet. Hierdurch werden Informationen selektiv bei Bedarf bereitgestellt, was zu einer besseren Übersicht bei großen Datenmengen führt.

Um die Menge der zu visualisierenden Daten weiter zu reduzieren, erlaubt die Anwendung die Filterung von Komponenten. So besteht die Möglichkeit, die Visualisierung eines jeden Komponententyps zu- oder abzuschalten. Ist ein Komponententyp nicht aktiviert, wird keine der zugehörigen Komponenten mehr in der Adjacency-Matrix aufgeführt.

Process Modeller

Die Seite „Process Modeller“ dient der Modellierung und Bearbeitung modellierter Geschäftsprozesse und deren Aktivitäten. Die Webanwendung erlaubt die Modellie-

rung in Graphenform, wozu das Diagram Framework JointJs (client IO s.r.o., 2017) eingesetzt wird.

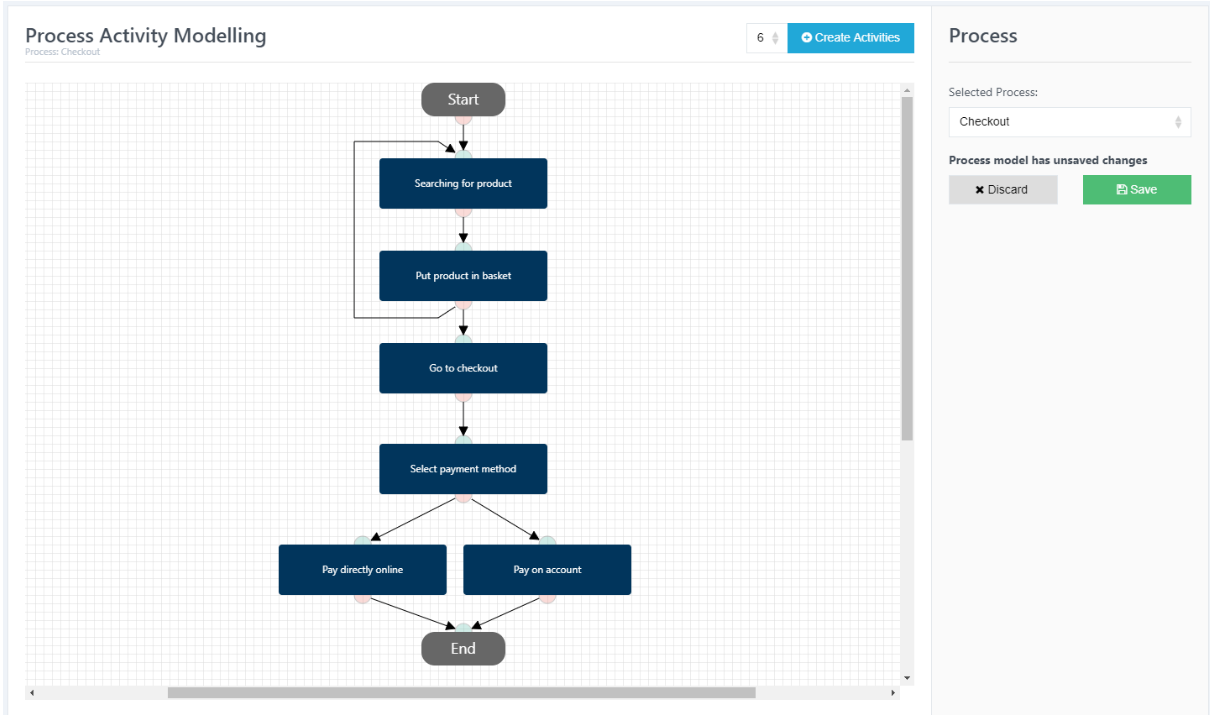


Abbildung 21: Modellierung von Prozessen und Aktivitäten

Nachdem der Anwender einen neuen Prozess erzeugt oder einen bestehenden Prozess zur Bearbeitung auswählt, wird eine gitterartige Modellierungsoberfläche mit einem Start- sowie einem Endelement bereitgestellt. Der Anwender hat anschließend die Möglichkeit, neue Aktivitäten zu erzeugen, welche durch rechteckige Objekte repräsentiert werden. Diese können auf dem Gitter frei angeordnet werden. Der Prozessfluss wird mithilfe von Drag and Drop modelliert, indem Relationen zwischen der Startkomponente, den Aktivitäten und der Endkomponente erzeugt werden.

Die Modellierung wird mit dem Speichern beendet. Die Speicherung ist nur dann möglich, wenn jede Aktivität eine Vorgänger- und eine Nachfolgerrelation aufweist und sowohl das Start- als auch das Endelement mindestens über eine Relation zu einer Aktivität verfügen. Ist dies der Fall, wird durch den erzeugten Graphen iteriert und es werden unter Verwendung der Architekturerkennungsservice-API entsprechende Komponentenobjekte erzeugt, zwischen welchen anschließend unter Verwendung eines weiteren Endpunkts die modellierten Relationen etabliert werden.

Die graphische Repräsentation des Prozessverlaufs wird in einem Json-Format der zugehörigen Prozesskomponente als Annotation hinzugefügt, was das erneute Laden und Bearbeiten der visuellen Prozessrepräsentation ermöglicht.

Activity Mapper

Wird eine technisch unterstützte Geschäftsaktivität ausgeführt, so bedeutet dies den Aufruf eines oder mehrerer Services der Microservice-Architektur durch den Anwender. Die Geschäftslogik einer Aktivität wird also unter Verwendung von Microservices realisiert. Somit besteht eine Abhängigkeit zwischen Aktivitätskomponenten und Servicekomponenten des internen Datenmodells. Das Architekturerkennungssystem wertet die eingehenden Anfragen im Rahmen der Distributed-Tracing-Datenanalyse aus und betrachtet hierbei insbesondere den Pfad und die HTTP-Methode (GET, PUT, POST, DELETE) der in das System eingehenden Requests. Dem Erkennungsservice steht eine Regelsammlung zur Verfügung, welche einen regulären Ausdruck zur Beschreibung eines gültigen Pfades, eine Auflistung gültiger HTTP-Methoden dieser Regel sowie eine zugeordnete Aktivitätskomponente umfasst. Der Architekturerkennungsservice durchläuft das gesamte Regelwerk und wendet jede darin enthaltene Regel auf den Request an. Entsprechen Pfad und HTTP-Methode des Requests den Regeldefinitionen, werden alle im Rahmen des Requests erzeugten Spans, welche die gleiche Trace-ID umfassen, ausgewertet, um involvierte Services zu ermitteln. Anschließend werden Beziehungen zwischen der in der Regel hinterlegten Aktivität und den ermittelten Services erzeugt. Dieser Vorgang wird für jede weitere gültige Regel wiederholt. Eine genaue Beschreibung des Vorgangs kann den Implementierungsdetails aus Kapitel 5.4.3 entnommen werden.

Das zugrundeliegende Regelwerk wird durch den Anwender definiert. Durch die API des Architekturerkennungsservice werden Endpunkte zur Erzeugung und Löschung von Regeln bereitgestellt. Ein weiterer Endpunkt stellt Pfade von Requests bereit, welche bisher in das System eingegangen sind, jedoch durch keine Regel einer Geschäftsaktivität zugeordnet werden konnten. Die Webanwendung verwendet diese, um dem Anwender eine Oberfläche zur Regelverwaltung zur Verfügung zu stellen. Diese besteht aus den im Folgenden beschriebenen drei Unterseiten.

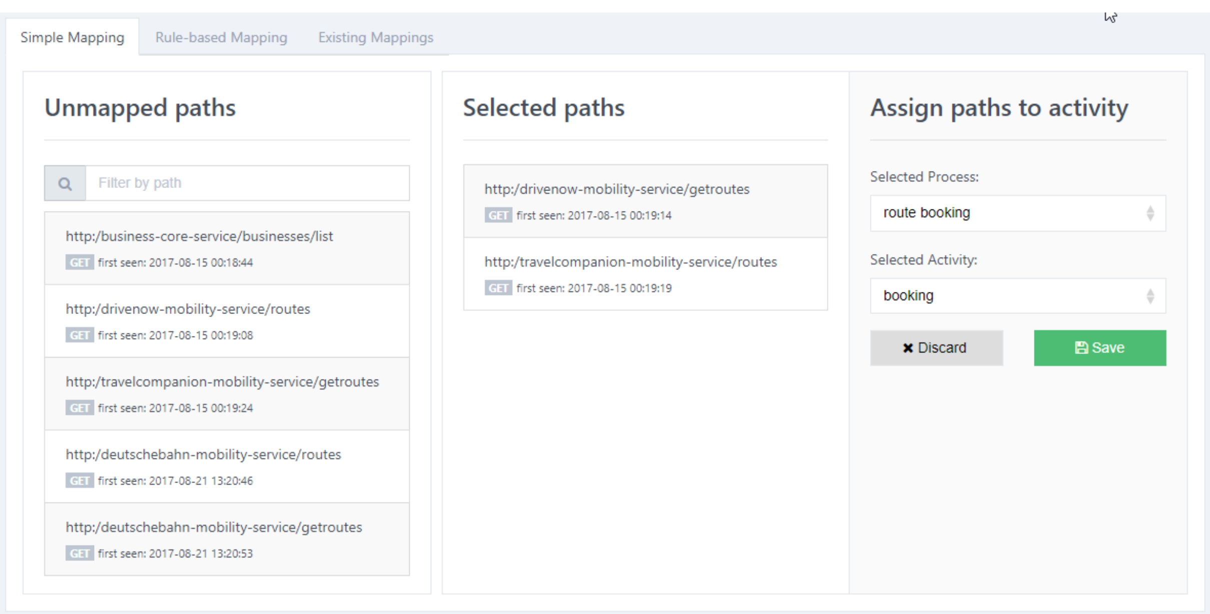


Abbildung 22: Webbasierte Regeldefinition durch manuelles Zuordnen

Die Unterseite „Simple Mapping" erlaubt ein einfaches Zuordnen von Pfaden zu Aktivitäten. In der linken Spalte der Anwendung werden die Pfade und die HTTP-Methode bisher nicht zugeordneter Requests aufgelistet. Da dies in großen Systemen eine Vielzahl von Pfaden sein kann, besteht die Möglichkeit der Filterung. Das hierzu bereitgestellte Eingabefeld durchsucht die Pfade nach eingegebenen Stichworten oder validiert sie gegen einen eingegebenen regulären Ausdruck. Durch einen Klick auf einen der Pfade wird dieser selektiert und somit in die mittlere Spalte verschoben. Es können beliebig viele Pfade gleichzeitig selektiert werden. Anschließend wird in der rechten Spalte eine Geschäftsaktivität eines modellierten Prozesses ausgewählt. Durch das Speichern der Auswahl wird für jeden selektierten Pfad eine Regel erzeugt, welche die HTTP-Methode der Auswahl sowie einen regulären Ausdruck umfasst, der exakt dem Auswahlpfad entspricht.

Die Oberfläche ermöglicht eine einfache Zuordnung von Requests zu Aktivitätskomponenten, ohne mit regulären Ausdrücken vertraut sein zu müssen, allerdings mit dem Nachteil, dynamische Anteile innerhalb eines Pfades nicht als variabel kennzeichnen zu können und eine Vielzahl von Einzelselektionen durchführen zu müssen. Außerdem können hiermit keine Regeln erzeugt werden, welche Pfade umfassen, die bisher nicht durch das System beobachtet wurden. Aufgrund der Erzeugung einer neuen Regel für jeden selektierten Pfad kann ein großes Regelwerk ent-

stehen, welches negative Auswirkungen auf die Verarbeitungszeit der Distributed-Tracing-Daten zur Folge hat.

Um dem vor allem für große Architekturen vorzubeugen, wird die Unterseite „Rule-based Mapping“ bereitgestellt, welche es Nutzern ermöglicht, eine Regel detailliert und losgelöst von bisher beobachteten Requests zu definieren. Dies erfordert jedoch, dass dem Nutzer die Syntax regulärer Ausdrücke geläufig ist. Aufgrund des notwendigen tiefgreifenden Wissens über die Systemarchitektur ist jedoch davon auszugehen, dass der Anwender im Regelfall über diese Fähigkeit verfügt.

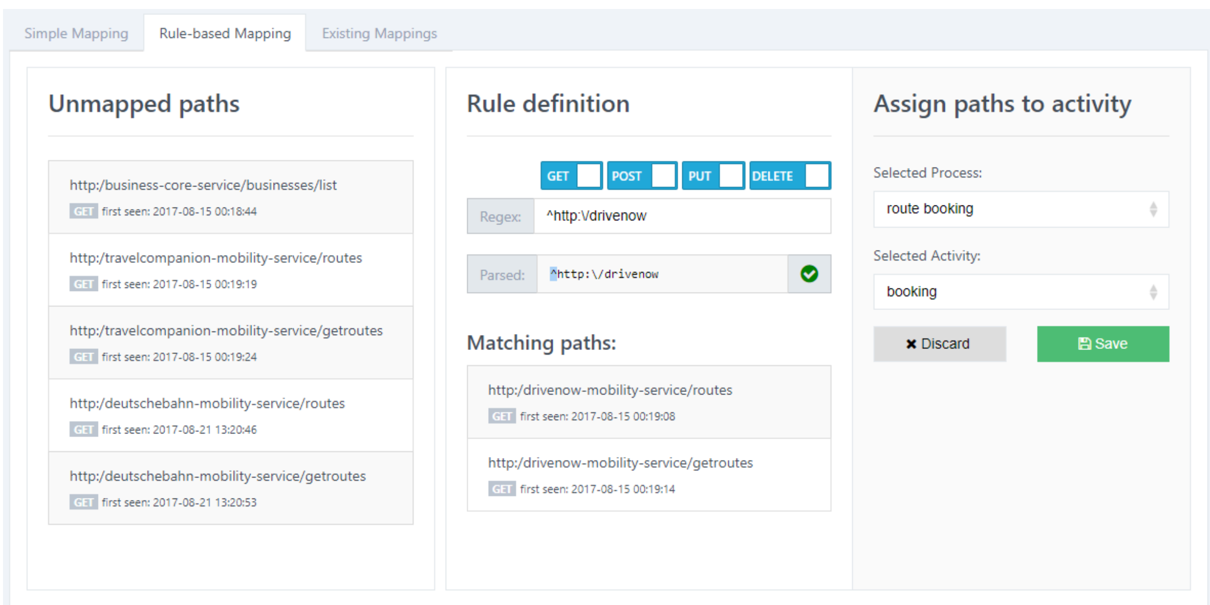


Abbildung 23: Webbasierte Regeldefinition durch reguläre Ausdrücke

Wie in Abbildung 23 dargestellt, werden in der linken Spalte weiterhin beobachtete Requests aufgelistet. Die mittlere Spalte dient nun der Regeldefinition. Diese umfasst die Auswahl gültiger HTTP-Methoden, wobei beliebig viele, mindestens jedoch eine Methode selektiert sein muss. Außerdem wird durch den Anwender der reguläre Ausdruck zur Validierung des Pfades selbst eingegeben. Während der Eingabe wird der Ausdruck auf syntaktische Validität überprüft und zur besseren Lesbarkeit syntaktisch koloriert. Syntaxfehler werden rot eingefärbt. Ist die Syntax valide, wird die Regel auf die Requests der linken Spalte angewandt und zutreffende Requests werden in die mittlere Spalte verschoben. Somit kann der Nutzer in Echtzeit überprüfen, welche der Anfragen durch seine derzeitige Regeldefinition berücksichtigt werden. Mithilfe der rechten Spalte wird der Regel eine Geschäftsaktivität zugeordnet. Mit

dem Speichern der Regel wird diese nun wie definiert abgelegt. Im Gegensatz zum Simple Mapping wird nun also nicht für jeden umfassenden Pfad eine eigene spezifische Regel erzeugt. Eine Regel kann selbst dann erzeugt werden, wenn sie derzeit auf keinen der gelisteten Pfade zutrifft. Somit können auch Regeln für bisher unbeobachtete Requests definiert werden. Durch die Verwendung von Platzhaltern innerhalb des regulären Ausdrucks können außerdem variable Anteile eines Pfades berücksichtigt werden.

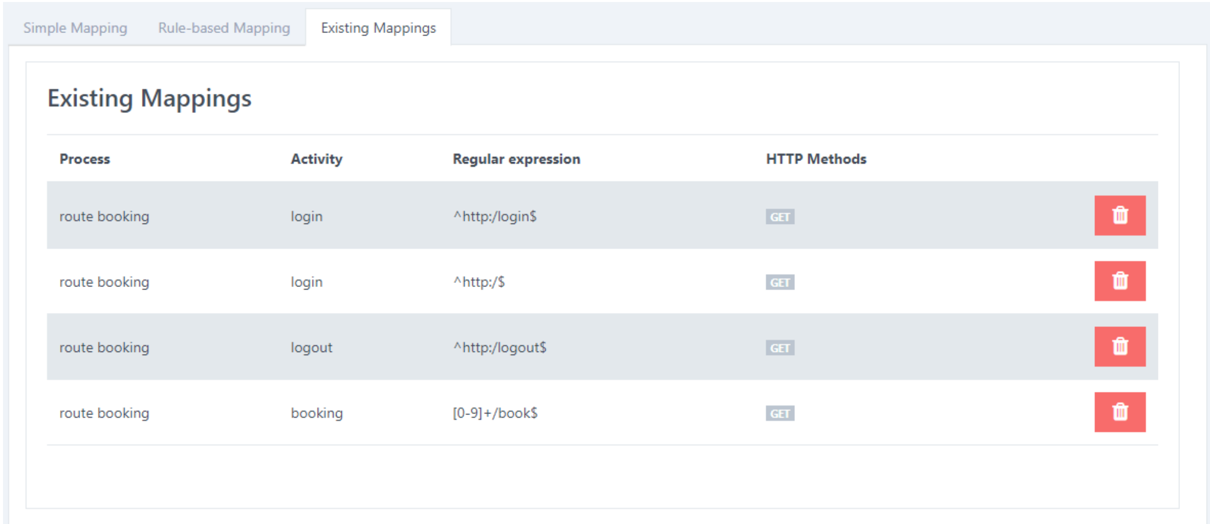


Abbildung 24: Webbasierte Verwaltung der Regelsammlung

Die dritte Unterseite dient der Einsicht in das bestehende Regelwerk. Wie in Abbildung 24 aufgezeigt, werden alle Regeln des Regelwerks inklusive ihrer Definitionen aufgelistet. Ist eine Regel veraltet, so kann sie über diese Seite aus dem Regelwerk entfernt werden.

5.2.3 Apache Kafka und Websocket Bridge

Besonderes Augenmerk wird bei der Konzeption des Prototyps darauf gelegt, eine durchgängige Datenverarbeitung in Echtzeit zu ermöglichen. Begonnen bei der Echtzeitanalyse der Distributed-Tracing-Daten wird diese Strategie bei der Erzeugung von Output durch den Architekturerkennungsservice fortgesetzt. Weiterverarbeitende Systeme werden mithilfe eines Push-Verfahrens über detektierte Architekturänderungen in Echtzeit informiert, was ihnen die Implementierung eines reaktiven Verhalten auf Architekturveränderungen ermöglicht.

Realisiert wird dies durch den Einsatz der Message-Streaming-Plattform Apache Kafka (Apache Foundation, 2017a), die den Datenaustausch zwischen Applikationen in Echtzeit ermöglicht. Die Plattform setzt das Prinzip der Datenerzeuger und -konsumenten konsequent um. Wird eine Message erzeugt, erhalten Konsumenten diese in Echtzeit und ohne explizite Aufforderung. Daten werden unter definierten Topics erzeugt, welche es erlauben, applikationsgebundene Informationen zu gruppieren.

Der Architekturerkennungsservice etabliert eine Verbindung zum Kafka-Stream in der Rolle eines Datenerzeugers. Wird eine Architekturveränderung erkannt, so erzeugt der Service ein Json-Objekt, welches in der Standardkonfiguration unter dem Topic „adTopic“ veröffentlicht wird. Sowohl Topic als auch Adresse des Kafka-Streams sind innerhalb des Architekturerkennungsservice konfigurierbar. Eine erzeugte Json-Nachricht besteht aus den in Tabelle 5 aufgezeigten Feldern.

Feldbezeichnung	Beschreibung
referenceId	ID des Objekts, auf welchem eine Operation ausgeführt wurde.
referenceType	Typ des Objekts, auf welchem eine Operation ausgeführt wurde. Mögliche Werte: „COMPONENT“, „REVISION“, „RELATION“.
operation	Operation, welche auf dem referenzierten Objekt ausgeführt wurde. Mögliche Werte: „CREATED“, „UPDATED“, „DELETED“.
time	Zeitstempel des Zeitpunkts, zu dem die Architekturänderung erkannt und verbreitet wurde.

Tabelle 5: Felder einer Json-Meldung zur Architekturänderung

Mithilfe der referenceId können unter Zuhilfenahme der Service-API weitere Details wie Annotationen und Name zu dem Objekt abgerufen werden, insofern die Änderung für die konsumierende Applikation von Relevanz ist.

Das Feld referenceType erlaubt Rückschlüsse auf den Typ des veränderten Objekts im Architekturmodell. Die Veränderung kann eine Komponente („COMPONENT“), eine Revision („REVISION“) oder eine Beziehung zwischen zwei Revisionen („RELATION“) betreffen.

Das operation-Feld ermöglicht die Identifizierung der Art der Architekturänderung. Mögliche Architekturänderungen sind, dass jenes Element hinzugefügt wurde („CREATED“), dass es verändert wurde („UPDATED“) oder dass es entfernt wurde („DELETED“). Entfernt bedeutet in diesem Kontext, dass z. B. der Gültigkeitszeitraum einer Revision oder einer modellierten Relation geschlossen wurde. Ein physisches Löschen von Daten findet zu keinem Zeitpunkt statt.

Das time-Feld dokumentiert den Zeitpunkt, zu welchem die Änderung durch das System erkannt wurde.

Durch jene Daten können an Kafka angebundene Konsumenten Veränderungen der Architektur in Echtzeit verarbeiten und Reaktionsstrategien implementieren. Um als Kafka-Konsument von den Echtzeitdaten profitieren zu können, benötigt eine Anwendung eine clientseitige Implementierung zum Verbindungsaufbau. Während diese in Technologien wie Node.js, C++, .Net , Python, Java und vielen weiteren (siehe Apache Foundation (2017b) für eine vollständige Auflistung) verfügbar ist, werden JavaScript-basierte Webanwendungen aufgrund der technologischen Einschränkungen nicht unterstützt.

Da eine Verwendung mit einer JavaScript-Applikation jedoch naheliegend ist, um beispielsweise Architekturen und deren Änderungen im Browser in Echtzeit zu visualisieren, wurde eine weitere Komponente zur Überwindung der Kafka-Abhängigkeit entwickelt. Während moderne clientbasierte Webtechnologien nicht auf Kafka zugreifen können, verfügen sie mit Websockets über eine andere Technologie zur Stream-basierten Kommunikation. Um sich dies zunutze zu machen, wurde, wie in Abbildung 25 dargestellt, eine Websocket Bridge entwickelt, welche Messages des Kafka-Streams an über Websockets angebundene Clients weiterreicht.

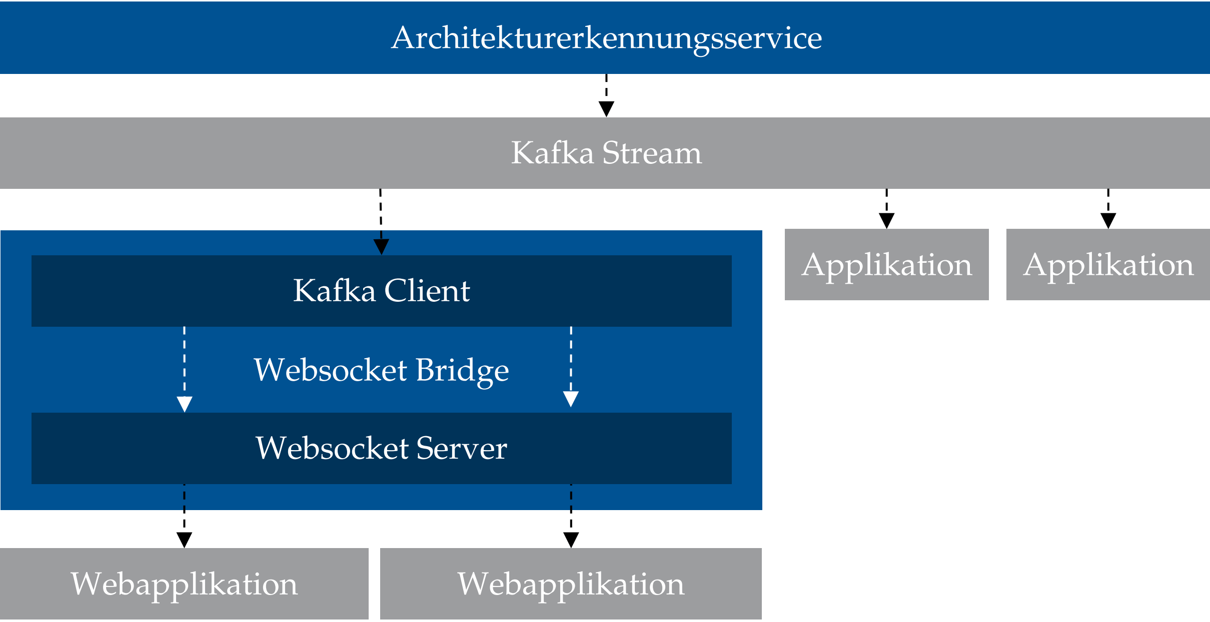


Abbildung 25: Funktionsweise der Websocket Bridge

Die Websocket Bridge ist eine in Node.js entwickelte Server-Applikation, welche einen Websocket-Server ausführt. Verbindet sich ein Websocket-Client, überträgt dieser in der Verbindungsanfrage das Kafka-Topic, von welchem er Nachrichten abonnieren möchte. Daraufhin wird durch die Websocket Bridge unter Verwendung des Node.js-Moduls „kafka-proxy“ (siehe Microsoft, 2017) eine Verbindung zum Kafka-Stream aufgebaut und es werden neue Nachrichten in diesem Topic abgehört. Trifft eine neue Nachricht ein, wird sie durch den Kafka-Client der Bridge empfangen und anschließend an alle Websocket-Clients versendet, welche das jeweilige Topic abonniert haben. Die Websocket Bridge kann auf diese Weise mehreren Webapplikationen parallel den Zugriff auf ein oder mehrere Kafka-Topics gewähren.

5.3 Architekturmodell

Das Architekturmodell ist ein temporäres Datenkonstrukt, welches eine Architektur zu einem konkreten Zeitpunkt repräsentiert. Dem Modell kommt als Austauschformat für weiterverarbeitende Applikationen eine besondere Bedeutung zu, weshalb seine Elemente in den nachfolgenden Kapiteln eingehend beschrieben werden.

Die Erzeugung des Architekturmodells erfolgt, wie in Abbildung 26 skizziert, unter Verwendung einer Serializer-Klasse, welche das Modell abhängig vom auslösenden

REST-Request erzeugt und anschließend in Form eines Json-Objekts an den API-Konsumenten ausliefert.

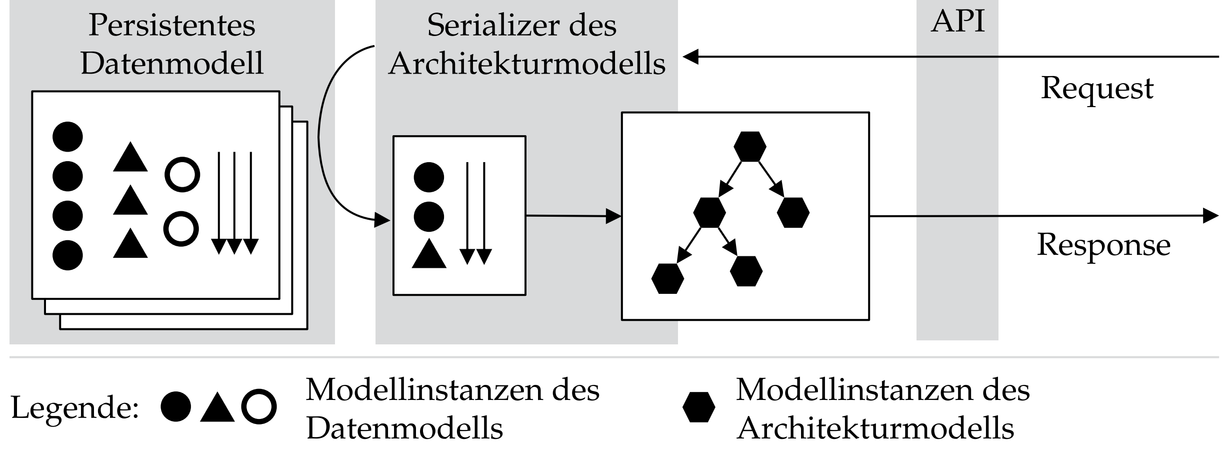


Abbildung 26: Schematische Darstellung der Überführung zum Architekturmodell

Der Serializer überführt zu diesem Zweck eine Teilmenge des Datenmodells in das Architekturmodell. Der Umfang jener Teilmenge wird durch die dem Request beigefügten Parameter definiert (siehe Kapitel 5.2.1). Die Überführung erfolgt, indem Instanzen der Modelle des Datenmodells ausgelesen werden. Auf deren Basis werden Modellinstanzen der API-eigenen Modelle temporär erzeugt. Die Gesamtheit der neu instanziierten Modelle ergibt das von der API zurückgelieferte Architekturmodell, welches dem API-Konsumenten eine einfache maschinelle Weiterverarbeitung, geringfügige Datenredundanz und schnelles Traversieren durch die hierarchische Datenstruktur ermöglicht.

Das Architekturmodell wird durch Instanzen der folgenden drei Objekttypen realisiert:

- Komponente
- Revision einer Komponente
- Relation zwischen Komponenten bzw. deren Revisionen

Diese werden in den nachfolgenden Unterkapiteln ausführlich beschrieben. Kern des Modells bilden die Komponenten, wie Geschäftsaktivitäten, Services und Hardware,

sowie deren Relationen innerhalb eines Komponententyps (Intralayer) als auch typübergreifend (Interlayer).

Das Architekturmodell dient dem Ziel, eine EA, wie in Kapitel 5 ausgeführt, möglichst gesamtheitlich zu repräsentieren. Dies bedeutet, entgegen bisheriger Architekturerkennungssysteme, sowohl Business-Layer als auch die technischen Ebenen in einem gemeinsamen Modell zu vereinen. Die definierten Komponententypen bilden alle Schichten der EA (Kapitel 2.2) ab und ermöglichen somit eine weitreichende Architekturrepräsentation.

Um den Zustand des Architekturmodells zu historisieren und somit auch Architekturzustände der Vergangenheit verfügbar zu machen, werden Beziehungen nicht direkt zwischen Komponenten, sondern zwischen deren Revisionen gepflegt. Revisionen stellen einen zeitlichen Gültigkeitsbereich einer Komponente dar. Die Summe paarweiser Beziehungen zwischen Revisionen bildet einen baumförmigen Beziehungsgraphen innerhalb des Architekturmodels ab, wie in Abbildung 27 skizziert.

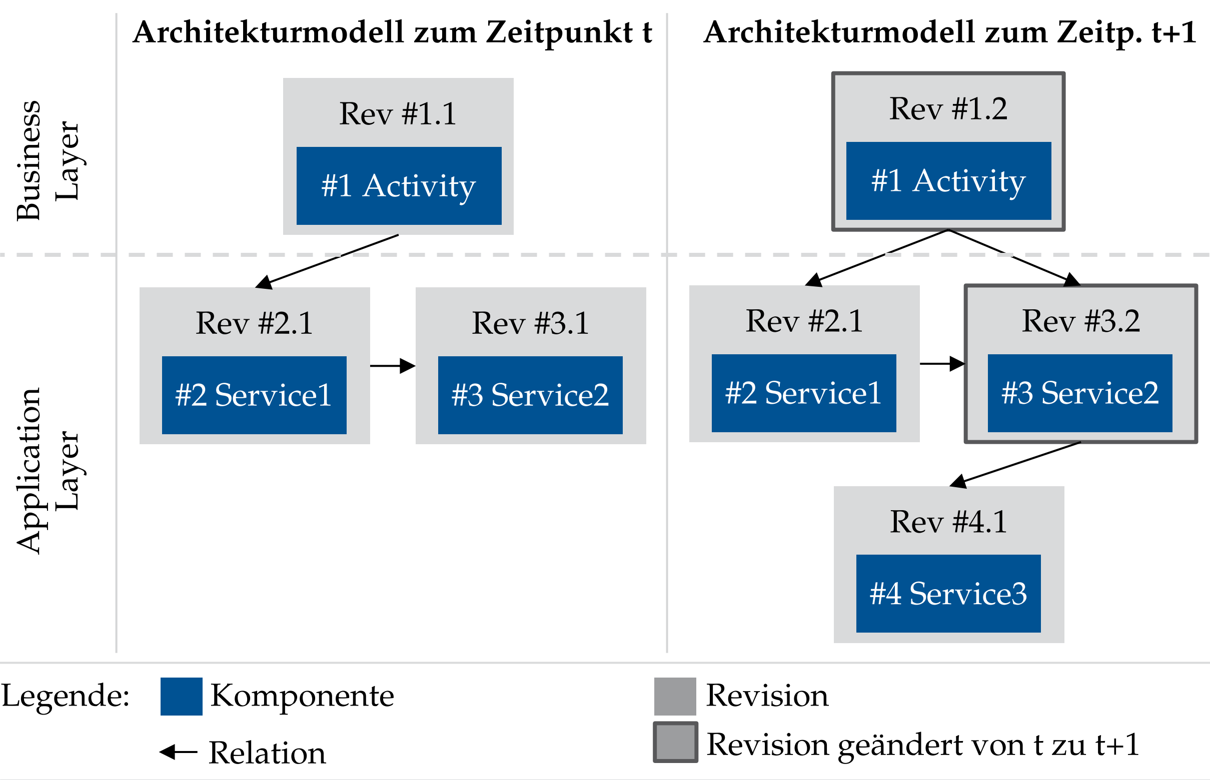


Abbildung 27: Verkürztes Architekturmodell zu zwei Zeitpunkten

Neben direkten Abhängigkeiten zwischen Revisionen umfasst das Architekturmodell außerdem indirekte Abhängigkeiten, welche nicht direkt in den analysierten Daten beobachtet werden können, jedoch aufgrund transitiver Zusammenhänge bestehen. Zu jedem der Objekttypen (Komponente, Revision, Relation) umfasst das Architekturmodell als Annotationen bezeichnete Zusatzinformationen in Form von Key-Value-Paaren, welche im Rahmen der Datenanalyse gewonnen und angefügt werden.

5.3.1 Komponenten

Wie in Abbildung 28 dargestellt, finden sich im Architekturmodell die folgenden fünf Komponenten wieder: Prozess, Aktivität, Service, Service-Instanz und Hardware. Während die Komponenten Prozess und Aktivität dem Business-Layer und somit der Geschäftsebene zugeordnet sind, sind Service und Service-Instanz (Application-Layer) sowie Hardware (Technology-Layer) der IT zuzuordnen.

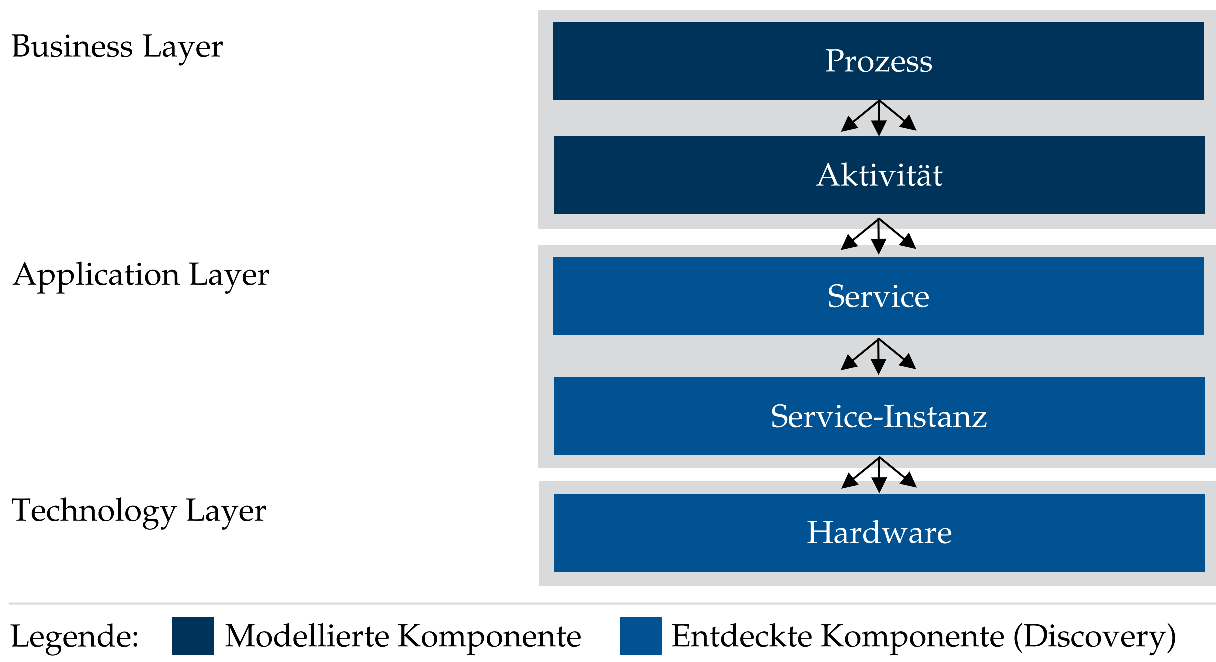


Abbildung 28: Komponenten des Architekturmodells

Die der IT zuzuordnenden Komponenten sind Resultat der Datenquellenverarbeitung (siehe Kapitel 5.2.1). Zur Ermittlung von Komponenten der Geschäftswelt hingegen ist zusätzliche Semantik erforderlich, welche durch die Webapplikation mo-

delliert wird (siehe Kapitel 5.2.2). Im Folgenden werden die einzelnen Komponenten näher erläutert:

Prozess

Eine Prozess-Komponente ist ein Synonym für einen realen Geschäftsprozess, den ein Anwender durchläuft. System-Prozesse werden hierbei nicht repräsentiert. Einem Prozess sind in der Regel mehrere Geschäftsaktivitäten zugeordnet, welche den Prozessverlauf beschreiben. Im Architekturmodell wird dies durch eine 1:n-Beziehung zwischen Prozessen und Aktivitäten beschrieben.

Aktivität

Eine Aktivitätskomponente repräsentiert eine Geschäftsaktivität, welche durch einen Anwender im Rahmen eines Geschäftsprozesses durchlaufen wird. Sie kann als Arbeitsschritt innerhalb einer Kette von Arbeitsschritten zur Erfüllung eines Prozesses angesehen werden. In der prototypischen Implementierung wird jede Anfrage eines Anwenders an das Gateway der Microservice-Architektur als eine Aktivität betrachtet.

Service

Eine Service-Komponente ist ein logisches Konstrukt, welches die Applikation eines Microservices darstellt. Wird zumindest eine Instanz einer Microservice-Applikation betrieben, wird eine Service-Komponente als logische Repräsentation dieser Applikation erzeugt. Dies abstrahiert Abhängigkeiten zu einem Service von dessen konkreter Instanziierung auf einer Hardware.

Service-Instanz

Im Gegensatz zur Service-Komponente ist die Komponente der Service-Instanz der Repräsentant genau einer Applikationsinstanz eines Microservice. Mehrere Service-Instanzen können die gleiche Applikation repräsentieren, z. B. im Falle der verteilten Lastverarbeitung. Die Eindeutigkeit einer Service-Instanz wird durch die Servicebezeichnung in Kombination mit der verwendeten IP-Adresse sowie des Service-Ports definiert.

Hardware

Die Hardware-Komponente repräsentiert ein Hardware-System, auf welchem ein oder mehrere Service-Instanzen betrieben werden. In der prototypischen Implementierung wird eine Hardware durch ihre IP-Adresse identifiziert. Dies beruht auf der Annahme, dass eine Hardware mit lediglich einer IP-Adresse ausgestattet ist und keine Virtualisierungstechnologien verwendet werden. In Fällen, in welchen diese Annahme nicht zutrifft, kann ein und dasselbe physische System fälschlicherweise durch mehrere Hardware-Komponenten repräsentiert werden. Eine eindeutige Identifizierung eines Hardware-Systems wäre über eine weitere Datenquelle direkt von den jeweiligen Systemen möglich, wird jedoch im Rahmen dieser Arbeit nicht umgesetzt.

5.3.2 Revisionen

Eine Revision ist immer einer Komponente zugeordnet und beschreibt diese für einen bestimmten Gültigkeitszeitraum. Verändert sich der Status der Komponente, so wird mit dem Zeitpunkt der Statusänderung eine neue Revision erzeugt und der Gültigkeitszeitraum der bisherigen Revision beendet. Einer Komponente ist somit zu keinem Zeitpunkt mehr als einer Revision zugeordnet, kann über den zeitlichen Verlauf allerdings mehrere Revisionen besitzen.

Als Beispiel einer Statusänderung sei die Veröffentlichung einer neuen Software-Version einer Microservice-Applikation genannt. Mit der Verteilung der neuen Version erhalten alle Service-Instanz-Komponenten eine neue Revision aufgrund ihres veränderten Status, welcher potentiell dazu geführt haben könnte, dass sich Beziehungen zwischen den Komponenten verändert haben. Denn mit der veränderten Software-Version können zusätzliche Service-Interaktionen hinzugekommen oder bisherige Interaktionen entfallen sein.

Da Beziehungen nicht zwischen Komponenten direkt, sondern zwischen deren Revisionen gepflegt werden, sind ab dem Zeitpunkt der Erzeugung einer neuen Revision alle bisherigen Beziehungen, an welchen die zugehörige Komponente beteiligt ist, ungültig. Besteht eine bisherige Beziehung auch für eine nachfolgende Revision, wird diese durch das System bei erneuter Beobachtung für diese Revision angelegt.

Dieses Vorgehen ermöglicht neben der Historisierung des Architekturmodells das „Vergessen“ von Beziehungen, welche zu einem früheren Zeitpunkt bestanden, zum aktuellen Zeitpunkt jedoch aufgrund von z. B. Software- oder Architekturänderungen nicht mehr vorhanden sind.

Der in Abbildung 29 dargestellte zeitliche Verlauf der Gültigkeit von Revisionen verdeutlicht, wie ein Architekturmodell für einen bestimmten Zeitpunkt mit Hilfe der Revisionen gebildet werden kann.

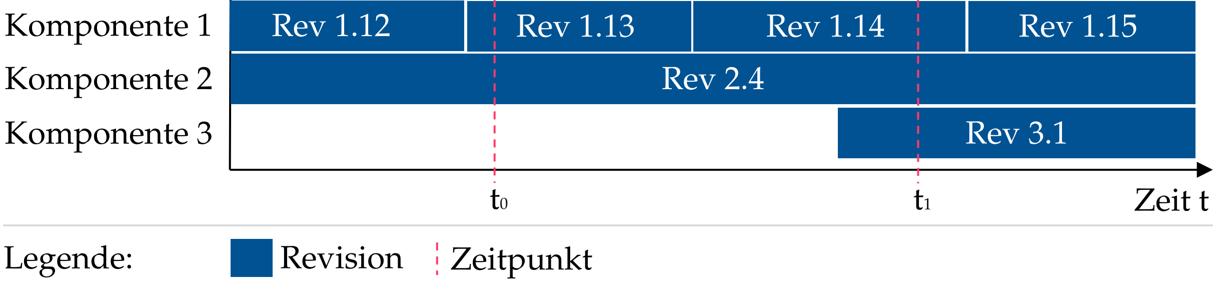


Abbildung 29: Revisionen im zeitlichen Verlauf

Soll für den Zeitpunkt t das Architekturmodell geliefert werden, werden alle Revisionen, welche zu diesem Zeitpunkt Gültigkeit haben, selektiert und die Beziehungen der selektierten Revisionen auf deren Komponenten übertragen.

Im abgebildeten Beispiel führt dies zum Zeitpunkt t_0 zur Selektion der Revisionen 1.13 sowie 2.4. Das Architekturmodell besteht somit aus Komponente 1 und 2 sowie den Beziehungen, welche zwischen Revisionen 1.13 und 2.4 bestanden. Zum Zeitpunkt t_1 werden teilweise andere Revisionen selektiert. Zum einen ist Revision 3.1 hinzugekommen, wodurch das Datenmodell um Komponente 3 erweitert wird, zum anderen hat Komponente 1 eine neue Revision 1.14 erhalten, wodurch im Vergleich zu t_0 Beziehungen hinzugekommen oder entfallen sein könnten.

In der prototypischen Realisierung wird eine auf Annahmen basierende Strategie zur Erzeugung von Revisionen implementiert. Wird eine Komponente angelegt, wird zeitgleich eine Revision erzeugt. Wird durch Auswertung der Service-Repository-Datenquelle (siehe Kapitel 4.1) festgestellt, dass eine Komponente nicht mehr vorhanden ist oder sie nach Gültigkeitsbeginn der Revision registriert wurde, so wird die aktuelle Revision geschlossen. Dies basiert auf der Annahme, dass ein Software-Update einen Neustart und somit eine (De-)Registrierung des Service an der Service

Registry mit sich führt. Wird die Komponente erneut durch die Auswertung einer der Datenquellen gesichtet, löst dies die Erzeugung einer neuen Revision aus.

5.3.3 Relationen

Beziehungen innerhalb des Architekturmodells werden zwischen Revisionen zweier Komponenten definiert. Da Revisionen im zeitlichen Verlauf einen Gültigkeitsbereich haben, ergibt sich für deren Relationen ebenso ein Zeitraum, in welchem diese Gültigkeit haben. Abbildung 30 zeigt dies beispielhaft für zwei Komponenten mit mehreren Revisionen auf. Annahme der Darstellung ist, dass die Relation zwischen zwei Komponenten trotz neuer Revisionen Bestand hat und erneut durch das Architekturerkennungssystem beobachtet werden konnte.

Abbildung 30: Gültigkeitszeitraum einer Relation

Relationen werden unter Verarbeitung der Distributed-Tracing-Daten aufgedeckt und dem Datenmodell hinzugefügt. Mit dem Hinzufügen der Relation bekommt diese eine Gültigkeit, die mit dem Start der später begonnenen Revision beginnt. Dieses Vorgehen birgt die Annahme in sich, dass die Beziehung zwischen den Revisionen bereits besteht, seit diese Revisionen existieren und nicht erst seit der erstmaligen Beobachtung der Relation.

Wird eine Relation nach einem Revisionswechsel nicht erneut beobachtet, wird sie auch nicht dem Datenmodell hinzugefügt. Die diesem Verhalten zugrundeliegende Idee ist, dass im Rahmen einer neuen Revision Veränderungen an einer Komponente vorgenommen worden sein könnten, wodurch eine Relation zwischen den Komponenten nicht mehr existiert. Wird die Relation erneut durch das Architekturerkennungssystem entdeckt, wird sie für die neuen Revisionen der Komponenten angelegt.

Eine Beziehung ist stets paarweise und gerichtet. Sie kann zwischen Revisionen zweier Komponenten gleichen oder auch unterschiedlichen Typs existieren. Jedoch kann eine Beziehung nur zwischen zwei Komponenten derselben hierarchischen Ordnung oder von einer übergeordneten zu einer untergeordneten Ordnung bestehen. Die hierarchische Ordnung der Komponententypen lautet wie folgt: Prozess, Aktivität, Service, Service-Instanz, Hardware. Das Architekturmodell bildet direkte sowie indirekte Beziehungen ab. Eine direkte Beziehung kann, wie in Abbildung 31 dargestellt, lediglich innerhalb der gleichen Ordnung oder zwischen zwei direkt aufeinanderfolgenden Ordnungen bestehen.

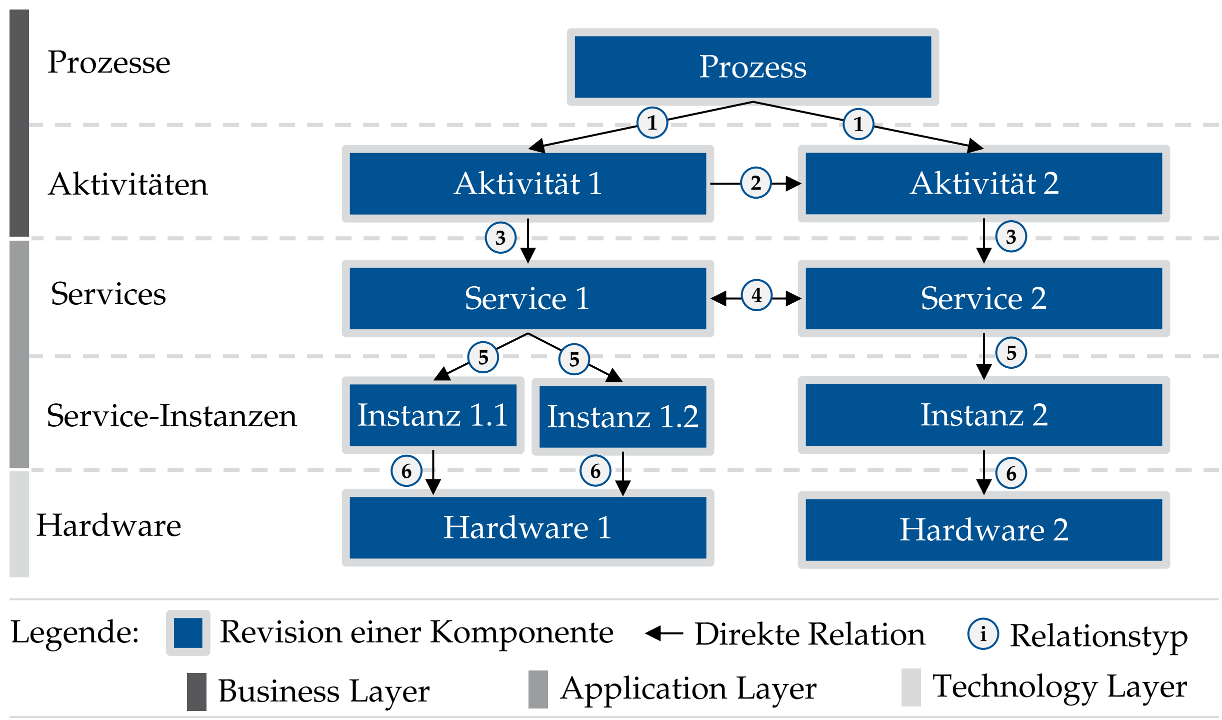


Abbildung 31: Relationenbaum des Architekturmodells

Des Weiteren wird in Abbildung 31 aufgezeigt, dass verschiedene Relationstypen entsprechend dem Typenpaar von Komponenten bestehen, welchen folgende Semantik zugewiesen wurde:

1) Prozess **besteht aus** Aktivität
2) Aktivität 2 **folgt auf** Aktivität 1
3) Aktivität **wird verarbeitet von** Service
4) Service 1 **verwendet** Service 2
5) Service **besteht aus** Instanz

6) Instanz **läuft auf** Hardware

Indirekte Relationen hingegen können von einer übergeordneten zu einer beliebigen untergeordneten Komponente bestehen. Sie ergeben sich aus der Transitivität der direkten Beziehungen. Dies bedeutet, wenn eine Service-Instanz in direkter Relation zu einer Hardware steht, auf welcher sie betrieben wird, so steht eine Aktivität, welche den Service verwendet, in indirekter Beziehung zu jener Hardware, da sie von der Service-Instanz und somit indirekt von der Hardware abhängig ist. Das Konzept der indirekten Beziehungen ermöglicht, tiefgreifende Abhängigkeiten zu erkennen, zu visualisieren und ihre Konsequenzen zu analysieren. Dies ermöglicht beispielsweise im Fall eines Hardwareausfalls, mithilfe des Architekturmodells zu ermitteln, welche Geschäftsaktivitäten hierdurch beeinflusst werden.

Je nach Typenkombination ist einer Beziehung eine andere Bedeutung zuzuordnen. Tabelle 6 zeigt auf, welche Bedeutung eine Beziehung zweier Komponenten impliziert. Die tabellarische Darstellung beschränkt sich hierbei auf die praktisch möglichen Beziehungen.

Relation von Komp. 1	Relation zu Komp. 2	Kardi-nalität	Implizite Bedeutung
Prozess	Aktivität	1:n	Aktivität ist Teil des Prozesses
Aktivität	Aktivität	n:m	Aktivität 2 kann auf Aktivität 1 folgen
Aktivität	Service	n:m	Service wurde zur Aktivitätsdurchführung verwendet
Aktivität	Service-Instanz	n:m	Service-Instanz wurde (indirekt) zur Aktivitätsdurchführung verwendet
Aktivität	Hardware	n:m	Hardware wurde (indirekt) zur Aktivitätsdurchführung verwendet
Service	Service	n:m	Service 1 nutzt Service 2
Service	Service-Instanz	1:n	Service-Instanz ist Instanz von Service
Service	Hardware	N:m	Service wird (indirekt) auf Hardware betrieben
Service-Instanz	Hardware	n:1	Service-Instanz wird auf Hardware betrieben

Tabelle 6: Implizite Bedeutung einer Relation zwischen Komponententypen

Während indirekte Relationen aufgrund der Transitivität direkter Relationen automatisiert und ohne zusätzliche Informationsquellen außer den direkten Relationen selbst erzeugt werden, erfordern direkte Relationen je nach Paarung der Komponententypen verschiedene Verfahren zur Relations-Detektion. Nachfolgende Aufzählung beschreibt, unter Einsatz welcher Methodiken direkte Relationen detektiert werden.

Relation zwischen Prozess und Aktivität sowie zwischen zwei Aktivitäten

Diese Art der direkten Relationen wird durch den Anwender modelliert. Da sie einzig auf Modellen des Business-Layers basieren und somit nicht mit den verfügbaren Daten automatisiert ermittelt werden können, steht dem Anwender über die Webanwendung des Prototyps eine Oberfläche zur Prozessmodellierung zur Verfügung, in welcher er Aktivitäten anlegt, sie Prozessen zuordnet und den Prozessverlauf in Form von Relationen zwischen Aktivitäten beschreibt (siehe Kapitel 5.2.2).

Relation zwischen Aktivität und Service

Relationen zwischen Aktivitäten und Services bilden die Schnittstelle zwischen Geschäfts- und IT-Welt. Sie werden teilautomatisiert erkannt, indem durch den Anwender ein Regelwerk definiert wird, welches URI-Pfade den einzelnen Aktivitäten zuordnet. Dieses Regelwerk wird automatisiert auf eingehende Distributed Tracing Spans angewandt und ermöglicht somit die Erkennung von Relationen zwischen Aktivitäten und Services.

Relation zwischen zwei Services

Relationen zwischen Services können im Rahmen der Analyse von Distributed-Tracing-Daten vollautomatisiert ermittelt werden. Spans enthalten, wie in Kapitel 3.3.3 beschrieben, Informationen über die involvierten Endpunkte der Kommunikation. Ein Endpunkt stellt eine Service-Instanz dar. Aufgrund der n:1-Beziehung zwischen Service-Instanz und Service selbst können die zugehörigen Services ermittelt und kann eine Relation zwischen ihnen etabliert werden.

Relation zwischen Service und Instanz sowie zwischen Instanz und Hardware

Diese Art der Relation kann auf zwei Wegen ermittelt werden, welche parallel eingesetzt werden. Zum einen werden die in Distributed Tracing Spans vorhandenen Endpoints ausgewertet. Sie beinhalten Informationen über eine Service-Instanz und die Bezeichnung des Service sowie die IP-Adresse der Hardware, auf welcher die Service-Instanz betrieben wurde. Das Eureka Service Repository stellt diese Informationen ebenfalls für alle registrierten Services zur Verfügung. Da Eureka zyklisch und nicht in Echtzeit verarbeitet wird, werden beide Methoden zur Relationserkennung herangezogen, um eine möglichst frühe Relationserkennung zu realisieren.

Das Architekturmodell sieht vor, dass für jede Relation zusätzliche Annotationen abgelegt werden können. So werden bei einer Relation zwischen Services alle Annotationen hinterlegt, die durch die Instrumentierung der Service-Instanzen an die Spans angefügt wurden. Dies ermöglicht z. B. die Kennzeichnung einer Beziehung als synchron oder asynchron, denn insofern dies durch die Instrumentierungssoftware unterstützt wird, erhalten Spans eine Annotation, wenn für die Kommunikation zwischen zwei Services ein neuer Thread erzeugt wurde und somit eine asynchrone Relation zwischen diesen besteht.

5.4 Implementierungsdetails des Architekturerkennungsservice

Der Architekturerkennungsservice bildet die essentielle Komponente des Architekturerkennungssystems. Aufgrund seiner hohen Relevanz dient dieses Kapitel einer vertiefenden Beschreibung angewandter Implementierungsdetails und -konzepte. Im Fokus der Betrachtung stehen die Softwarearchitektur des Service, die Algorithmen zur Verarbeitung der zwei Datenquellen sowie das zugrundeliegende Datenmodell. Die Applikation ist ein RESTful Webservice, welcher mit der Technologie Java realisiert wurde. Sie basiert auf den Projekten „Spring Cloud" sowie „Zipkin-Server". Zur persistenten Datenspeicherung ist eine MySql-Datenbank an die Applikation angebunden.

5.4.1 Applikationsstruktur der Java-Applikation

Die Klassen der Java-Applikation sind in Packages gegliedert. Wie in Abbildung 32 aufgezeigt, besteht die Gliederung in ihrer ersten Ebene aus vier Packages: config, service, jpa und controller. Die Packages sind derart strukturiert, dass eine klare Definition von Zugriffswegen zwischen den Klassen der jeweiligen Packages definiert ist. Dies stellt Datenkonsistenz sicher, ermöglicht die Absicherung paralleler Verarbeitung sowie die Implementierung von Caching-Strategien. Eine Klasse des controller-Package benötigt zum Verrichten ihrer Arbeit Informationen des Datenmodells, welches im jpa-Package abgelegt ist. Sie darf jedoch nicht direkt auf dieses zugreifen, da ansonsten Caching-Mechanismen umgangen werden und Konflikte bei der Parallelverarbeitung der verschiedenen Controller auftreten. Lediglich die Klassen des service-Package dürfen auf die Informationen des jpa-Package zugreifen und diese den Controllern bereitstellen, da diese Caching-Strategien implementieren und Daten in Strukturen vorhalten, welche parallele Verarbeitung zulassen. Im Folgenden werden die vier Packages der ersten Ebene fortführend beschrieben.

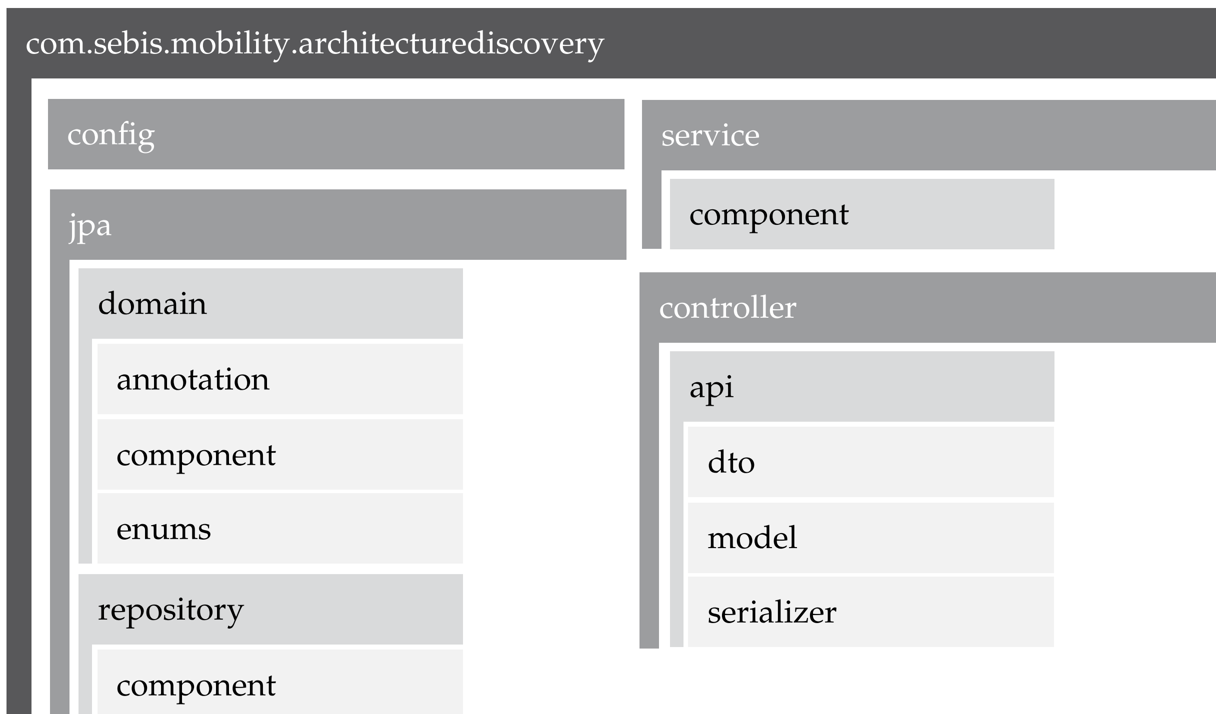


Abbildung 32: Strukturierung des Architekturerkennungsservice

config

Das config-Package beinhaltet Klassen zur Applikationskonfiguration. Im Konkreten werden hier von der Standardkonfiguration abweichende Einstellungen der REST-API sowie des Swagger-Frontends definiert. Swagger ist ein API-Framework, welches zur Dokumentation der API eine Benutzeroberfläche bietet, durch welche die Endpunkte in einer Weboberfläche betrachtet und getestet werden können (SmartBear Software, 2017a).

service

Das service-Package beinhaltet Klassen zur Verwaltung von Objekten des Datenmodells. Es wird durch die Controller angesprochen, um Modelle zu laden oder zu modifizieren, welche es selbst aus dem jpa-Package bezieht. Die Service-Klassen haben Caching-Strategien implementiert, welche je nach Notwendigkeit definieren, wann ein Modell aus der Datenbank gelesen oder in die Datenbank geschrieben wird. Des Weiteren definieren jene Strategien, ob ein Objekt des Datenmodells im Arbeitsspeicher gehalten oder aus diesem verdrängt wird. Die im Rahmen des Prototyps verfolgte Strategie umfasst, dass alle Modelle des aktuell bekannten Architekturzu-

stands im Arbeitsspeicher gehalten werden, Modelle vergangener Zustände aus dem Arbeitsspeicher vertrieben werden. Jede Änderung an einem Modell wird unverzüglich in der Datenbank persistiert.

controller

Innerhalb des controller-Packages sind Klassen zur Ausführung der Anwendungslogik untergebracht. Dies umfasst zum einen die Datenquellenverarbeitungsklassen, zum anderen die Implementierung der REST-API. Innerhalb des untergeordneten api-Packages werden Modelle zur Beantwortung von API-Anfragen und von der Standardserialisierung abweichende Serializer deklariert. Die Serializer-Klassen dienen der Umwandlung einer Modell-Instanz in ein JSON-Objekt, welches zur API-Kommunikation verwendet wird.

jpa

Das jpa-Package umfasst Klassen und Packages, welche Relevanz für die Java-Persistence-API (JPA) haben. Dies umfasst zum einen die Klassen des Datenmodells, welche in dem Package domain definiert wurden, zum anderen die im repository-Package abgelegten Repositories. Für jede der Modellklassen ist eine Repository-Klasse deklariert, in welcher der Datenbankzugriff für das Modell definiert wird. Das Repository stellt Methoden zur persistenten Speicherung des Java-Objektes in der Datenbank sowie den Objektabruf aus der persistenten Datenschicht zur Verfügung.

5.4.2 Datenmodell

Das Datenmodell des Architekturerkennungsservice dient der internen Strukturierung und Ablage von Architektur- und Steuerungsdaten. Seine Daten werden je nach Caching-Strategie langfristig im Arbeitsspeicher gehalten und vollständig in der angebundenen MySql-Datenbank persistiert, deren Datenschema Anhang D entnommen werden kann. An das Datenmodell besteht die Anforderung hoher Flexibilität. Es wird ständig und parallel durch mehrere Threads auf Basis eingehender Daten verändert, muss jedoch gleichzeitig den Abruf eines Zustands bzw. die Erzeugung eines Architekturmodells zu einem bestimmten Zeitpunkt ermöglichen.

Wie in Abbildung 34 dargestellt, besteht das Datenmodell aus Instanzen von 18 Modell-Klassen, von welchen die drei Klassen „AbstractPersistable“, „Compo-

nent“ sowie „Annotation“ rein abstrakte Klassen sind, welche nicht selbst instanziiert werden und lediglich die Basis für erbende Klassen darstellen. Im Folgenden wird der Einsatzzweck der Klassen bündig beschrieben.

Die Klasse „AbstractPersistable“ bildet die Basis jeder anderen Klasse des Datenmodells. Alle Klassen des Datenmodells haben etwas gemein. Sie sollen in einer Datenbank persistiert werden können und sie sollen für die Verwendung mit der REST-API serialisierbar sein. Diesem Zweck dient „AbstractPersistable“, welches entsprechende Dependencies herstellt und Basisfunktionalitäten wie die Verwaltung der ObjektID bereitstellt.

Die abstrakte Klasse „Component“ stellt die Basis aller Komponentenklassen dar. Sie implementiert die Attribute der Komponenten und verknüpft die Komponente mit Funktionalitäten zum Management von Annotationen des Typs „ComponentAnnotation“. In den einzelnen Spezialisierungen der „Component“-Klasse können abweichende Implementierungen durchgeführt werden. Im Rahmen der prototypischen Implementierung geschieht dies nicht. Die spezialisierten Klassen werden hier lediglich zur typsicheren Verarbeitung von Komponenten eingesetzt, ohne dass diese eine weitere Implementierung umfassen.

Eine Komponente hat, wie in Kapitel 5.3.2 beschrieben, ein oder mehrere Zeitfenster, in welchen sie gültig ist. Gültigkeit beschreibt, dass die Komponente in diesem Zeitraum vermutlich in einem aus Architektursicht konstanten Status existiert. Diese Zeiträume werden durch die Klasse „Revision“ wiedergegeben, welche zu einem Komponentenobjekt einen Zeitraum speichert. Für ein Komponentenobjekt gibt es keine Überschneidungen in den Gültigkeitszeiträumen der Revisionen.

Zwischen Komponenten bestehen Relationen, wie diese bereits in Kapitel 5.3.3 beschrieben wurden. Relationen werden jedoch nicht zwischen den Komponenten direkt, sondern deren Revisionen aufgespannt. Entsprechend umfasst ein Relation-Objekt die Revisions-Objekte Caller und Callee. Um indirekte Relationen zu ermöglichen, umfasst es des Weiteren das Revisions-Objekt Owner.

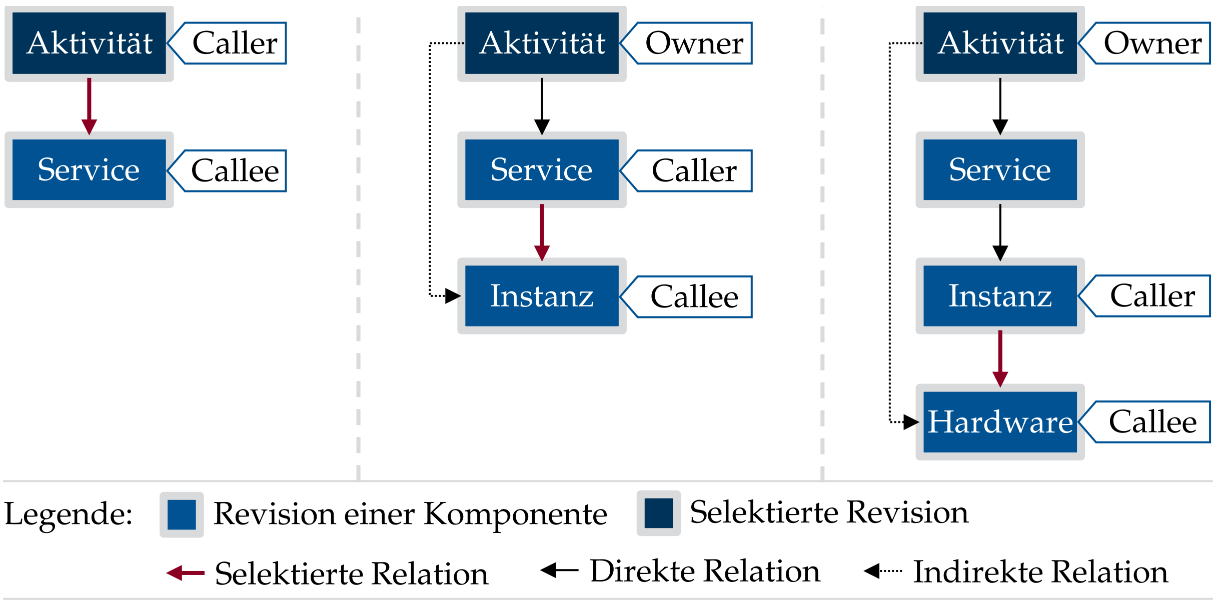


Abbildung 33: Direkte und indirekte Relationen einer Aktivität

Caller Ist das Revisionsobjekt, von welchem die Relation ausgeht. Callee ist das Gegenstück der gerichteten paarweisen Relation. Eine Relation wird zwar immer zwischen zwei Relationen aufgespannt, sie kann jedoch einer dritten Revision zugeordnet werden, was im Rahmen dieser Arbeit als indirekte Relation bezeichnet wird. Die Zuordnung einer Relation zu einer dritten Revision bedeutet, dass diese Revision Abhängigkeiten zu der Relation hat. Wenn zum Beispiel eine Geschäftsaktivität (Owner) einen Service (Caller) zur Ausführung benötigt, welcher auf einer Instanz (Callee) betrieben wird, so ist die Aktivität nicht nur vom Service, sondern auch von der Instanz indirekt abhängig, da sie den Service, dieser zur Ausführung jedoch seine Instanz benötigt. Abbildung 33 verdeutlicht dieses Prinzip, indem es beispielhaft alle direkten und indirekten Relationen einer Aktivität schrittweise darstellt.

Im Gegensatz zum Architekturmodell erfolgt im Datenmodell eine Unterscheidung zwischen automatisiert erkannten Relationen und modellierten Relationen, welche durch die Klasse „ModeledRelation“ verkörpert werden. Modellierte Relationen besitzen entgegen der entdeckten Relationen einen Gültigkeitszeitraum. Bei entdeckten Relationen ergibt sich dieser Zeitraum aus der Gültigkeit der hierdurch verbundenen Revisionen. Da jedoch bei der Modellierung eine Relation durch den Anwender entfernt werden kann, ohne dass sich die hierdurch verbundenen Revisionen ändern, ist es erforderlich, im Datenmodell zusätzlich den Gültigkeitszeitraum der modellierten Relation zu speichern.

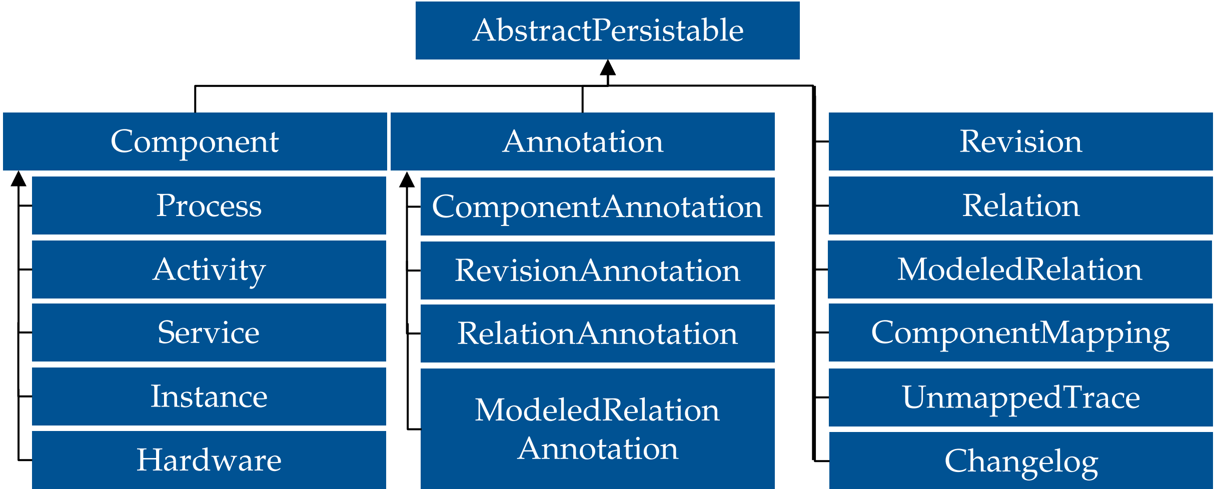


Abbildung 34: Klassendiagramm des Datenmodells ohne Relationen

Wie im Architekturmodell in Kapitel 5.3 beschrieben, können Metainformationen in Form von Annotationen zu allen Typen von Komponenten, Revisionen sowie modellierten und entdeckten Relationen zugewiesen werden. Zu diesem Zweck dient die abstrakte Klasse „Annotation", welche die Verwaltung der Key-Value-Paare implementiert. Hiervon abgeleitet werden die vier spezialisierten Klassen der voran genannten Modelltypen. Ein Objekt dieser Modelltypen besitzt jeweils eine Kollektion von Annotationen der für sie spezialisierten Klasse.

Neben den Klassen zur Architekturbeschreibung umfasst das Datenmodell weitere Klassen, um die Verarbeitung von Eingabeinformationen sowie den Modellierungsprozess (siehe Kapitel 5.2.2) zu ermöglichen. Objekte der Klasse „UnmappedTrace" werden erzeugt, wenn Pfad und Methode eines an das Microservice-System gesendeten Requests nicht über das etablierte Regelwerk einer Geschäftsaktivität zugeordnet werden können. Für jede Kombination von Pfad und HTTP-Methode wird lediglich ein Objekt erzeugt, welches außerdem die ID des Trace beinhaltet, aus welchem es abgeleitet wurde, um diesen nach Erweiterung des Regelwerks erneut zu prozessieren. Die „UnmappedTrace"-Objekte werden außerdem genutzt, um in der Webanwendung bisher nicht gemappte Pfade listen zu können.

Die Objekte der Klasse „ComponentMapping" repräsentieren die Regeln des etablierten Regelwerks. Sie umfassen den regulären Ausdruck des Pfades, die erlaubten HTTP-Methoden sowie eine Referenz auf eine Komponente, auf welche die Regel mappen soll.

Der Architekturerkennungsservice stellt nicht nur Architekturen in einem Gesamtzustand dar, sondern informiert außerdem über getroffene Änderungen am aktuellen Architekturmodell und ermöglicht somit Service-Konsumenten eine Echtzeitreaktion auf Veränderungen. Detektierte Änderungen werden durch Objekte der Klasse „Changelog“ zum Ausdruck gebracht. Ein solches Objekt umfasst die Referenz auf die ID des geänderten Objekts, dessen Art (Komponente, Revision oder Relation) sowie eine Beschreibung der auf das Objekt durchgeführten Operation (Objekt wurde erstellt, aktualisiert oder gelöscht). Diese Objekte werden neben der Bereitstellung in der REST-API außerdem über den Kafka-Stream in serialisierter Form in Echtzeit vermittelt.

5.4.3 Verarbeitung der Distributed-Tracing-Daten

Die Distributed-Tracing-Daten spielen eine essenzielle Rolle zur Erzeugung eines Architekturmodells. Aus ihnen werden Komponenten extrahiert und Beziehungen zwischen diesen erkannt. Im Folgenden wird der Algorithmus zur Verarbeitung der eingehenden Daten eingehend beschrieben.

Der Algorithmus ist in der Klasse „SpanController“ des „Controller“-Packages definiert. Er stellt die Methode „void proceedSpans(List<Span> spans)“ bereit, um eingehende Spans zu verarbeiten. Diese Methode wird, wie in Kapitel 5.2.1 beschrieben, durch die eigenimplementierte Storage-Komponente aufgerufen, nachdem eingehende Spans durch die Zipkin-Server-Standardimplementierung validiert und in einer MySql-Datenbank persistiert wurden. Der Methode wird eine Liste in Java-Objekte überführter Spans des Distributed-Tracing-Systems übergeben. Vier besondere Herausforderungen sind bei der Datenverarbeitung zu berücksichtigen:

- Die Span-Liste entspricht in ihrer Reihenfolge nicht zwingend dem zeitlichen Verlauf der Span-Erzeugung.
- Mehrere Spans innerhalb einer Liste können unterschiedlichen Traces zugeordnet sein.
- Die Summe aller Spans eines Traces kann über mehrere Span-Listen und somit Funktionsaufrufe verteilt bereitgestellt werden.
- Dem System ist zu jeder Zeit unbekannt, wie viele Spans ihm zur Vollständigkeit des Traces fehlen.

Der Algorithmus durchläuft die Liste und analysiert jeden der Spans. Die Analyse eines Spans wird in einer Folge von Verarbeitungsschritten durchgeführt. Abbildung 35 zeigt die möglichen Veränderungen des Datenmodells durch jeden dieser Schritte.

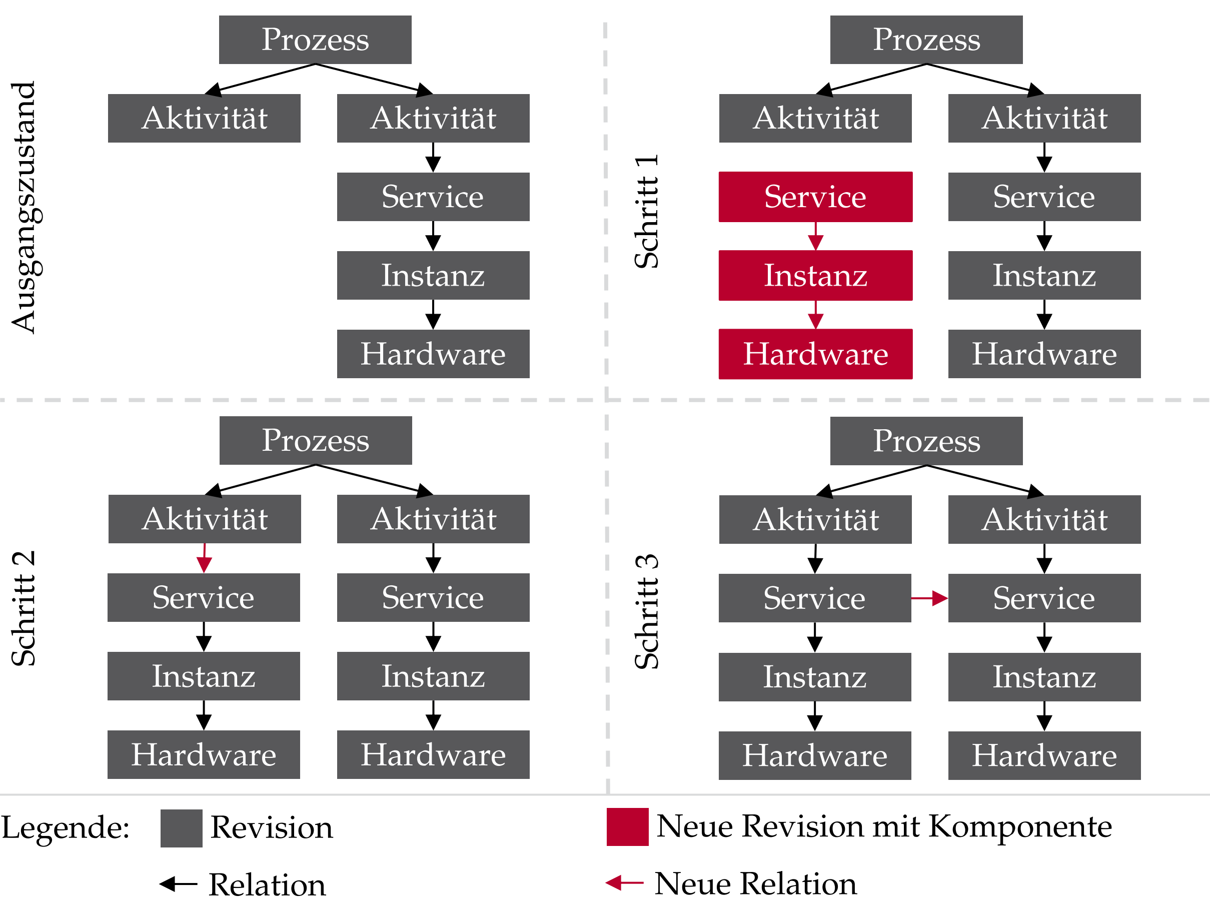


Abbildung 35: Veränderung des Datenmodells durch schrittweise Span-Analyse

Schritt 1: Extraktion von Service-, Instanz- und Hardwarekomponenten

Wie in Kapitel 3.3.3 beschrieben, verfügen Spans über Annotationen, welche Metainformationen beinhalten. Jeder Annotation ist ein Endpunkt angefügt, welcher beschreibt, welchem Microservice die Annotation zuzuordnen ist. Im ersten Schritt werden alle vorhandenen Annotationen des Spans durchlaufen und deren Endpunkte ausgewertet. Ein Endpunkt wird definiert durch Servicename, IP-Adresse und Port. Diese Informationen ermöglichen Rückschluss auf:

- Service-Komponente auf Basis des Servicenamens

- Instanz-Komponente durch Kombination von Servicename, IP-Adresse und Port
- Hardwarekomponente unter Verwendung der IP-Adresse.

Nach der Ermittlung dieser drei Komponenten wird deren Vorhandensein im Datenmodell geprüft. Ist dies für eine oder mehrere Komponenten nicht der Fall, wird sie inklusive einer Revision erzeugt. Zwischen den drei Komponenten bestehen außerdem Beziehungen: der Service wird ausgeführt durch die Instanz, welche auf der Hardware betrieben wird. Diese werden ebenfalls bei Nicht-Vorhandensein dem Datenmodell hinzugefügt.

Schritt 2: Erzeugung der Relation zwischen Aktivität und Service

Schritt 2 wird nur dann ausgeführt, wenn der Span der Eingangsspan in das Microservice-System ist, also vom ersten Server des Microservice-Clusters erzeugt wurde. Dies ist der Fall, wenn der Span keinen Vorgänger-Span definiert hat (parentId = null) und über eine „sr"-Annotation verfügt. Dem „name"-Attribut des Spans ist in dem Fall der Pfad des Requests zugewiesen. Außerdem ist die HTTP-Methode des Requests in der binären Annotation „http.method" an den Span angefügt. Der Algorithmus iteriert durch das über die Webanwendung erzeugte Regelwerk (siehe Kapitel 5.2.2) und validiert die regulären Ausdrücke gegen den Pfad, insofern die HTTP-Methode durch die Regel berücksichtigt wird. Bei jeder erfolgreichen Validierung wird zwischen der Aktivitätskomponente der Regel und der Service-Komponente des „sr"-Endpunkts eine direkte Relation erzeugt. Denn nach Regelwerk wird die Aktivität durch diesen Service technisch realisiert. Außerdem wird zwischen der Aktivitätskomponente und der Instanz-Komponente sowie der Hardware-Komponente des Endpunkts eine indirekte Beziehung etabliert. Sollte keine der Regeln zutreffen, so wird geprüft, ob für die gegebene Kombination aus Pfad und Methode im Datenmodell bereits ein Objekt der Klasse „UnmappedTrace" existiert. Ist dies nicht der Fall, so wird eines erzeugt und ihm neben Pfad und Methode auch die Trace-ID des aktuellen Spans zugewiesen. Wird künftig eine Regel angelegt, welche auf den Span zutrifft, so wird der gesamte Trace mithilfe der Trace-ID erneut prozessiert und es werden die beschriebenen Relationen erzeugt.

Schritt 3: Erkennung von Relationen zwischen Services

Eine Relation zwischen zwei Services macht sich durch zwei Spans kenntlich. Die aufrufende Service-Instanz erzeugt einen Span mit einer „cs"-Annotation und die aufgerufene Service-Instanz erstellt einen Span mit einer „sr"-Annotation. Besitzt der aktuell verarbeitete Span eine dieser beiden Annotationen, so wird für ihn Schritt 3 ausgeführt, den beide Spans zur Erkennung der Relation durchlaufen haben müssen. Denn jeder Span beinhaltet nur die Information zum Endpunkt seines Erzeugers, nicht jedoch zu dem des Kommunikationspartners. Zusammengehörige Spans können dadurch identifiziert werden, dass sie eine gemeinsame Span-ID besitzen.

Um das Span-Paar zu ermitteln, existiert eine Map, welche als Key Span-IDs beinhaltet und als Value einen Span. Bei der Verarbeitung eines Spans wird geprüft, ob die Span-ID bereits in der Map vorhanden ist. Ist dem nicht so, so ist der andere Span noch nicht verarbeitet. Der aktuelle Span wird der Map hinzugefügt und die Span-Analyse beendet. Ist die ID allerdings bereits in der Map vorhanden, dann ist der zweite Span bereits verarbeitet und es liegen nun beide Spans vor. Der Eintrag wird aus der Map entfernt und die Endpunkte der „sr"- und „cs"-Annotationen der Spans ausgewertet, um die zugehörigen Services zu ermitteln. Anschließend wird eine Relation zwischen den Revisionen des aufrufenden und des aufgerufenen Service erzeugt, insofern diese nicht bereits im Datenmodell vorhanden ist. Außerdem wird eine indirekte Beziehung zwischen der Revision des aufrufenden Service und den Revisionen aller Komponenten hergestellt, mit welchen die Revision des aufgerufenen Service in Beziehung steht. Dies unterliegt der Annahme, dass der aufrufende Service den aufgerufenen Service und somit auch dessen Instanz und Hardware zur Ausführung seiner eigenen Logik benötigt und somit auch mit diesen in einer Relation steht.

5.4.4 Verarbeitung der Daten des Eureka Service Repository

Die Verarbeitung der Daten des Service Repository dient der Steuerung von Gültigkeitszeiträumen von Revisionen sowie der Erkennung von Architekturkomponenten, welche keine Spans erzeugen.

Die Implementierung der Repository-Verarbeitung erfolgt in der Klasse „RegistryController" des Controller-Packages. Sie verfügt über eine Methode „void

eurekaCall()“, welche zur Verarbeitung der Repository-Daten aufgerufen werden kann. Der Aufruf erfolgt durch den Architekturerkennungsservice in einem Intervall von zehn Sekunden. Die Methode wird in einem eigenen Thread ausgeführt und kann somit parallel zur Verarbeitung der Distributed Traces aufgerufen werden, um eine Echtzeitauswertung zu realisieren. In jeder Intervallausführung werden die nachfolgend beschriebenen Schritte zur Verarbeitung durchlaufen.

Schritt 1: Abruf des Endpunkts und Ermittlung der Komponenten

Im ersten Schritt wird durch die API des Eureka-Repository ein XML-Dokument abgerufen, welches den aktuellen Stand der Services beschreibt. Ein solches Dokument kann Anhang A entnommen werden. Hierzu wird der Endpunkt „GET /eureka/v2/apps“ über einen HTTP-Request kontaktiert. Das erhaltene XML-Dokument umfasst alle Services und deren Instanzen, welche sich bei dem Repository an- oder abgemeldet haben. Dem Aufbau des Dokuments entsprechend folgt dem einleitenden <applications>-Tag ein <application>-Tag für jeden der Services. Dieser hat wiederum zwei Kindelemente, einen <name>-Tag, welcher die Bezeichnung des Service beinhaltet, sowie einen <instance>-Tag, welcher eine Service-Instanz repräsentiert und Detailinformationen wie Status der Instanz, Port und IP-Adresse umfasst.

Der Algorithmus durchläuft alle Service-Instanzen und verarbeitet jene, welche den Status „UP“ aufweisen, also derzeit verfügbar sind. In drei separaten Listen werden Servicename, Instanzbezeichnung (Kombination aus Servicename sowie IP-Adresse und Port) und IP-Adresse abgelegt, was jeweils den Namen der Service-, Instanz- und Hardwarekomponenten des Datenmodells entspricht. In einer vierten Liste wird der Zeitpunkt der Instanzanmeldung an dem Repository für jede Instanz abgelegt.

Schritt 2: Erzeugung neu entdeckter Komponenten

Im zweiten Schritt werden aus einer Kopie der Liste der Servicenamen all jene Einträge entfernt, für welche im Datenmodell eine aktuell gültige Revision existiert, deren Service-Komponente den gleichen Servicenamen trägt wie der Listeneintrag. In der Liste verbleiben somit bisher nicht im Datenmodell vorhandene Services oder solche Services, welche derzeit keine gültige Revision besitzen, da sie beispielsweise für einen Zeitraum unerreichbar waren. Existiert der Service der verbleibenden Listeneinträge bereits, wird eine neue Revision erzeugt, existiert er noch nicht, so wird

eine neue Servicekomponente inklusive Revision erzeugt. Dieser Vorgang wird äquivalent für die Hardware- und Instanzliste durchgeführt. Bei der Erzeugung einer Instanzrevision wird zusätzlich eine Beziehung zur zugehörigen Hardware-Revision sowie zur zugehörigen Service-Revision etabliert und die Information über Port und IP-Adresse als Annotation an die Instanz-Revision angefügt.

Schritt 3: Schließung von Revisionen

Im dritten Schritt wird eine Liste der derzeit im Datenmodell vorhandenen Service-Revisionen erzeugt und es werden aus dieser alle Elemente entfernt, welche durch die Verarbeitung des Eureka Repository bekannt sind. In der Liste verbleiben Elemente, welche im Datenmodell gültige Revisionen haben, jedoch nicht am Service Repository angemeldet sind. Die Liste wird anschließend durchlaufen und der Gültigkeitszeitraum der zugehörigen Service-Revision zum Verarbeitungszeitpunkt terminiert. Gleiches geschieht für die Instanz- und Hardwarekomponenten. Somit werden nicht mehr verfügbare Komponenten aus dem aktuellen Datenmodell verdrängt.

Schritt 4: Revisionserneuerung

Innerhalb des Intervalls von zehn Sekunden ist es möglich, dass ein Service offline und wieder online war. Dies kann den Grund haben, dass er aufgrund einer Softwareänderung neu gestartet wurde. Von daher ist es notwendig, aufgrund des potentiell veränderten Status eine neue Revision zu erzeugen. Zu diesem Zweck wird die in Schritt 1 angelegte vierte Liste der Instanzanmeldungen durchlaufen. Anschließend wird die zu der Instanz gehörende Revision aus dem Datenmodell bezogen. Liegt der Beginn des Gültigkeitszeitraums der Revision vor dem Zeitpunkt der Instanzanmeldung, bedeutet dies, dass die Revision vor dem Neustart der Instanz erzeugt wurde. Ist dies der Fall, so wird der Gültigkeitszeitraum der Revision terminiert und eine neue Revision für die Instanzkomponente erzeugt. Auch die Relationen zu Service- und Hardware-Revision werden erneut etabliert.

6. Evaluation

Dieses Kapitel beschreibt die Evaluation der prototypischen Implementierung. Sie gliedert sich in die Beschreibung der erzeugten Testumgebung sowie des zugrundeliegenden Business-Szenarios. Im anschließenden Kapitel wird die Evaluation der Anwendung im zuvor beschriebenen Umfeld dokumentiert. Neben der Evaluation der korrekten Funktionalität des Architekturerkennungsservice werden außerdem die Auswirkungen der Instrumentierung einer Microservice-Architektur bezüglich Performance und Ressourcennutzung analysiert. Dieses Kapitel schließt mit der Benennung der Limitierungen, welchen der entwickelte Prototyp unterliegt.

6.1 Evaluations-Umfeld

Die Evaluation des Prototyps wird in einer hierzu erzeugten Testumgebung durchgeführt. Die Testumgebung umfasst eine Business-Applikation, die notwendige Infrastruktur zum Applikationsbetrieb sowie die prototypische Implementierung des Architekturerkennungsservice. Nachfolgend werden Applikation und Infrastruktur eingehend beschrieben.

Business-Applikation

Die Business-Applikation mit dem Namen Travelcompanion richtet sich an Reisende, welchen ermöglicht werden soll, sich mit Reisegruppen zu verbinden, die ein gemeinsames Reiseziel mit dem Reisenden haben. Zu diesem Zweck stehen Reisedaten der Anbieter Deutsche Bahn und des Dienstes DriveNow der BMW AG zur Verfügung. Der Travelcompanion-Service verarbeitet diese Daten und bietet Vorschläge für eine gewählte Route unter Berücksichtigung der genannten Anbieter. Bestehen bei den Anbietern Reisegruppen mit freien Plätzen, werden diese gelistet und ein Platz kann gebucht werden. Aufgrund der Aufteilung der Gruppenpauschale können die Reisekosten der Gruppe als auch des einzelnen reduziert werden.

Vor der Nutzung der Anwendung hat ein Login zu erfolgen. Die Anmeldesession kann nach der Applikationsnutzung durch einen Logout beendet werden. Das in Abbildung 36 dargestellte Prozessablaufdiagramm zeigt den vollständigen Prozess der Applikationsnutzung auf.

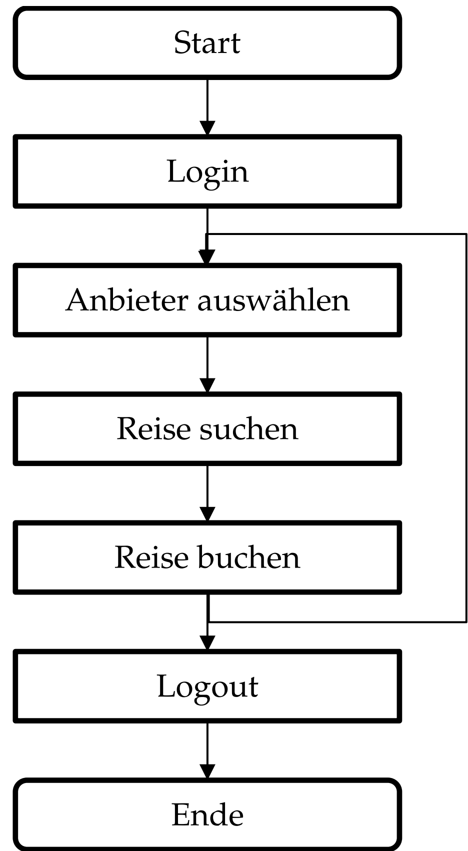


Abbildung 36: Prozessablaufdiagramm der Evaluationsapplikation

Die Applikation wird durch ein Cluster von Microservices bereitgestellt, welche im Folgenden beschrieben werden.

Microservice-Cluster

Das Microservice-Cluster umfasst sieben in Spring Boot implementierte Microservices, welche zur Ausführung der Businesslogik Anwendung finden:

- Den Zuul-Service, welcher als API-Proxy sowie zum Login und Logout genutzt wird.
- Den Business-Core-Service, welcher verfügbare Anbieter zur Auswahl stellt.
- Den DeutscheBahn-Mobility-Service, welcher die Reisesuche für den Anbieter Deutsche Bahn bereitstellt.
- Den DriveNow-Mobility Service, welcher die Reisesuche für den Anbieter DriveNow liefert.

- Den Travelcompanion-Service, welcher die kombinierte Suche über alle Anbieter ermöglicht.
- Den Maps-Helper-Service, welcher die Berechnung der Distanz einer Route ermöglicht.
- Den Accounting-Core-Service, welcher die Buchung von Reisen ermöglicht.

Wie in Abbildung 37 dargestellt, stehen die zwei Services des Zuul API Proxy sowie der Travelcompanion-Service in Abhängigkeit zu anderen Microservices.

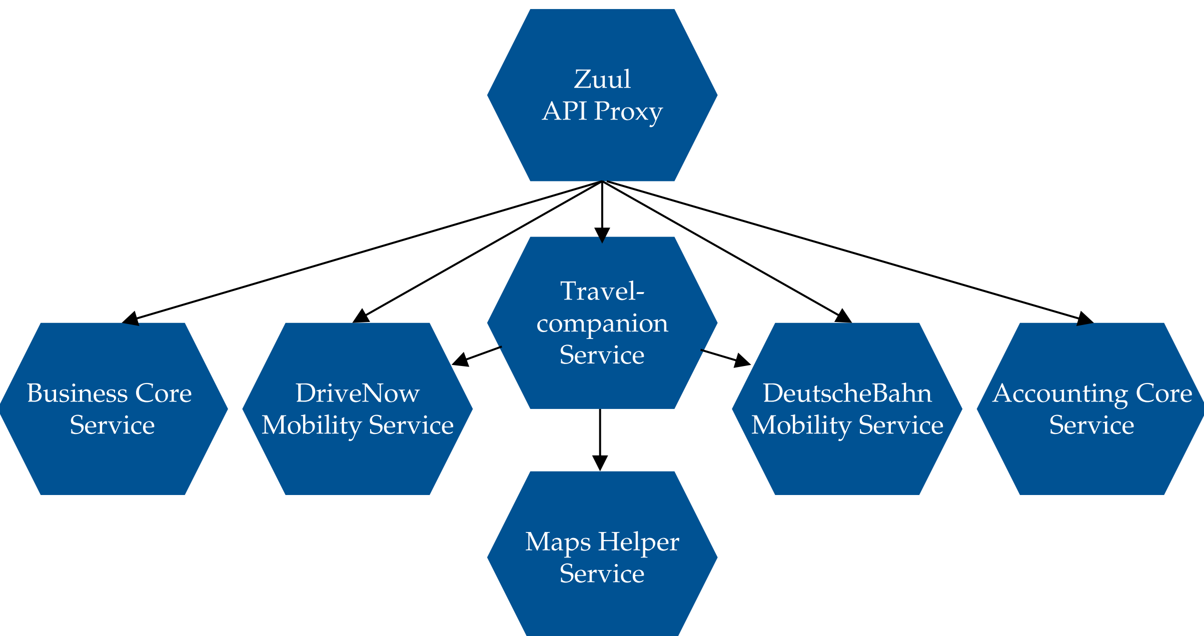


Abbildung 37: Abhängigkeiten des Microservice-Clusters

Der Zuul-Service steht aufgrund seiner Funktion als API Proxy in Abhängigkeit zu allen genannten Microservices, da jeder in das System eingehende Request durch ihn verarbeitet und anschließend an die weiteren Services gereicht wird. Ist einer jener Services gestört, kann dies zu Störungen des Zuul-Service führen.

Durch den Travelcompanion-Service werden weitere Services zur Verarbeitung von Suchanfragen aufgerufen, was zur Abhängigkeiten von jenen Services führt. Er ist abhängig vom DeutscheBahn-Mobility Service sowie vom DriveNow-Mobility Service, um die angebotenen Reisen jener Anbieter zu erfragen. Zur Ermittlung der Distanz einer Reise besteht des Weiteren eine Abhängigkeit zum Maps-Helper-Service.

Abhängigkeiten zwischen Business-Applikation und Microservice-Cluster

Das beschriebene Microservice-Cluster dient der Ausführung des vorangegangenen Geschäftsprozesses. Jede Aktivität des Prozesses wird durch einen oder mehrere Microservices technisch realisiert:

- Die Aktivitäten „Login" und „Logout" werden durch den Zuul-Service ausgeführt.
- Die Aktivität „Anbieter auswählen" wird durch den Business-Core-Service ermöglicht.
- Die Aktivität „Reise suchen" wird abhängig vom gewählten Anbieter durch den DeutscheBahn-Mobility Service, den DriveNow-Mobility Service oder den Travelcompanion-Service realisiert.
- Die Aktivität „Reise buchen" ermöglicht der Accounting-Core-Service

Die Aktivitäten haben entsprechend der Auflistung Abhängigkeiten zu diesen Services. Diese sind aufgrund des Einsatzes eines API Proxy indirekt, da eine direkte Abhängigkeit lediglich zu jenem Proxy besteht. Weitere indirekte Abhängigkeiten der Aktivitäten ergeben sich zwischen der Aktivität „Reise suchen" und den durch den Travelcompanion genutzten Maps-Helper-Service.

Microservice-Architektur

Die zur Evaluation erzeugte Microservice-Architektur ist an der in Kapitel 3.2.2 beschriebenen Referenzarchitektur orientiert und umfasst neben dem beschriebenen Cluster von Microservices zur Verarbeitung von Geschäftslogik weitere Services zum Betrieb der Infrastruktur. Dies umfasst neben dem bereits beschriebenen Zuul-Service als API Proxy einen Eureka-Service, welcher als Service Repository dient, sowie den Architekturerkennungsservice und einen Kafka-Stream zum Transport von Spans des Distributed Tracing. Abbildung 38 zeigt die gesamte Architektur, welche insgesamt neun Microservices umfasst, noch einmal grafisch auf.

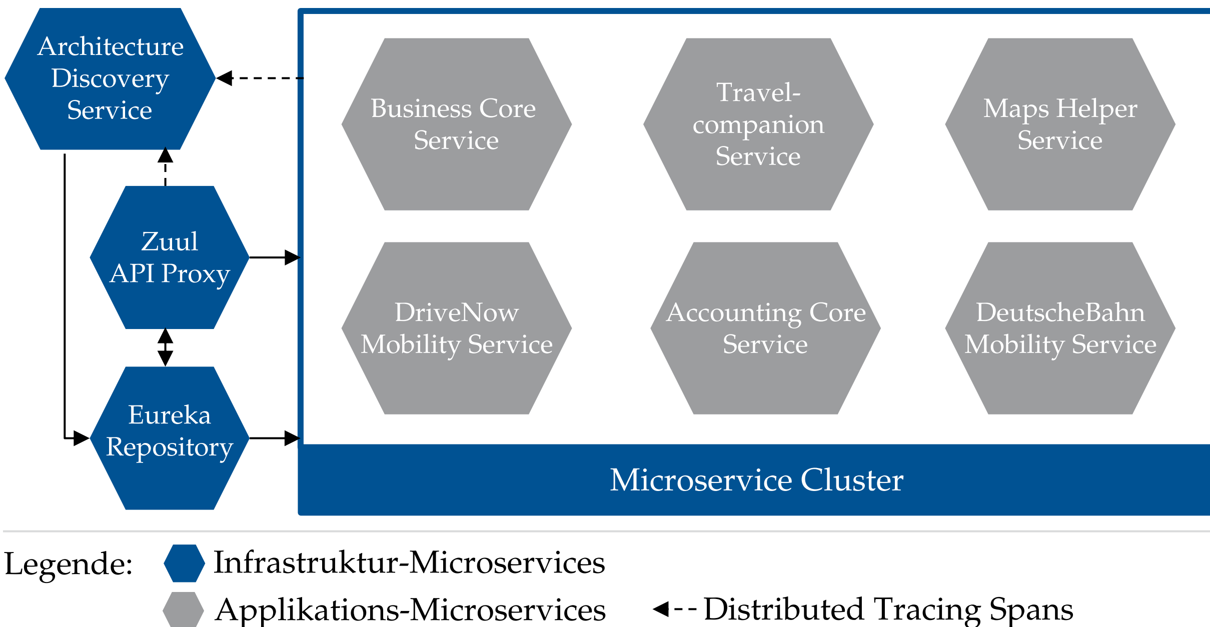


Abbildung 38: Microservice-Architektur des Evaluationsumfelds

Alle Microservices des Clusters sowie der Zuul-Service sind mit Eureka-Clients ausgestattet, durch welche eine An- und Abmeldung am Repository erfolgt. Des Weiteren sind sie durch die Instrumentierung Sleuth (Pivotal Software Inc., 2017b) für das Distributed Tracing auf Basis von Zipkin ausgestattet.

Deployment der Microservice-Architektur

Die Architektur wird auf vier separaten Hosts betrieben, welche jeweils virtualisiert, jedoch mit dedizierten Hardwareressourcen ausgestattet sind. Alle Hosts werden mit dem Linux-Derivat Debian 8.8 im 64-Bit-Modus ausgeführt. Tabelle 7 stellt weitere Konfigurationsdetails der betriebenen Hosts bereit.

	Host 1	**Host 2**	**Host 3**	**Host 4**
IP-Adresse	10.0.2.100	10.0.2.110	10.0.2.120	10.0.2.121
CPU-Kerne	2	1	1	1
Arbeitsspeicher in MB	4096	2048	2048	2048

Tabelle 7: Konfiguration der Hosts

Host 1 dient dem Betrieb des Architecture-Discovery-Service sowie seinen Abhängigkeiten. Dies führt zu folgendem Deployment auf Host 1:

- Architecture Discovery Service
- Eureka Microservice Repository
- Apache Kafka (als Distributed Tracing Transport)

Host 2 wird zur Ausführung der elementaren Applikations-Microservices verwendet:

- Business-Core-Service
- Accounting-Core-Service
- Zuul-Service

Host 3 und 4 werden jeweils zum Betrieb der Microservices der Reiseanbieter verwendet. Der redundante Betrieb wird durch den Eureka-Service zum Loadbalancing genutzt, wodurch die Services beider Hosts Verwendung finden:

- DeutscheBahn-Mobility-Service
- DriveNow-Mobility-Service
- Travelcompanion-Service
- Maps-Helper-Service

6.2 Durchführung der Evaluation

Die Evaluation des Prototyps wird auf zwei Weisen durchgeführt. Zum einen wird die Funktionsweise auf ihre Korrektheit geprüft, zum anderen Auswirkungen des Prototyps auf die Performance des Gesamtsystems analysiert.

6.2.1 Evaluation der Funktionsweise

Die Funktionsweise des Architekturerkennungsservice wird durch vier Schritte evaluiert. Im ersten Schritt werden die einzelnen Hosts nacheinander mit den auf ihnen deployten Applikationen gestartet, um die Erkennung von Service, Instanzen und Hardware zu prüfen.

Der zweite Evaluationsschritt umfasst die Erkennung von Relationen zwischen Services. Dies erfolgt durch die Verarbeitung des Distributed Tracing. Zu diesem Zweck

wird die Applikation aufgerufen und der beschriebene Prozess durchlaufen. Das zu erwartende Ergebnis ist eine vollständig erkannte Architektur auf Applikations- und Infrastrukturebene inklusive ihrer Beziehungen.

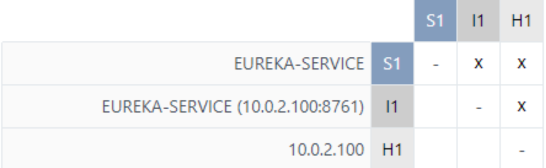
Der dritte Evaluationsschritt dient dem Hinzufügen des Business-Layers zum Architekturmodell. Zu diesem Zweck wird der beschriebene Prozess in der Webanwendung modelliert und anschließend mithilfe der Webanwendung ein Mapping zwischen den modellierten Aktivitäten und den Services hergestellt. Im Ergebnis sollten der Prozess und seine Aktivitäten in der Adjacency-Matrix sichtbar werden und Beziehungen entsprechend der Beschreibung in Kapitel 6.1 dargestellt werden.

Im vierten Evaluationsschritt wird das Reaktionsvermögen auf Veränderungen der Architektur geprüft, indem ein Service für einen Zeitraum abgeschaltet und anschließend neu gestartet wird. Es wird erwartet, dass die abgeschaltete Komponente durch den Architekturerkennungsservice bemerkt und die zugehörige Revision geschlossen wird. Nach Neustart der Komponente sollte eine neue Revision erzeugt worden sein.

Evaluationsschritt 1: Erkennung von Services, Instanzen und Hardware

Ziel des ersten Evaluationsschrittes ist es, die Funktionsweise zur Erkennung von Microservices durch die Verarbeitung von Eureka-Daten zu überprüfen. Dies umfasst Services, deren Instanzen und Hardware sowie die Interlayer-Beziehungen jener Komponententypen. Die Durchführung erfolgt stufenweise, indem ein Host nach dem anderen inklusive seiner Applikationen gestartet wird und nach dem Start eines jeden Hosts die Veränderungen des Architekturmodells über die Adjacency-Matrix des Frontend evaluiert werden.

Da keinerlei Traffic innerhalb der Anwendung produziert wird, werden keine Spans erzeugt. Der Architekturerkennungsprozess beschränkt sich damit auf die Resultate der Verarbeitung des Eureka-Service (siehe Kapitel 5.4.4).



		S1	I1	H1
EUREKA-SERVICE	S1	-	x	x
EUREKA-SERVICE (10.0.2.100:8761)	I1		-	x
10.0.2.100	H1			-

Abbildung 39: Adjacency-Matrix nach Start von Host 1

Die Adjacency-Matrix entspricht nach dem Start dem Host 1, s. Abbildung 39. Sie umfasst den Eureka-Service, eine Instanz dessen sowie ein Hardware-System. Der Service steht in Relation zu seiner Instanz sowie indirekt zu der Hardware, auf welcher die Instanz betrieben wird, was durch die zwei „x“ in der Zeile „EUREKA-SERVICE“ deutlich wird. Die Instanz selbst steht in Abhängigkeit zu ihrer Hardware. Die einzige auf Host 1 mit einem Eureka-Client ausgestattete Applikation ist der Eureka-Service selbst. Da dieser auf Host 1 mit der IP 10.0.2.100 betrieben wird, wird der Service, seine Instanz und die Hardware korrekt auf Basis der Eureka-Datenauswertung erkannt. Weitere Services neben dem Eureka-Service werden nicht ausgeführt und folgerichtig nicht gelistet.

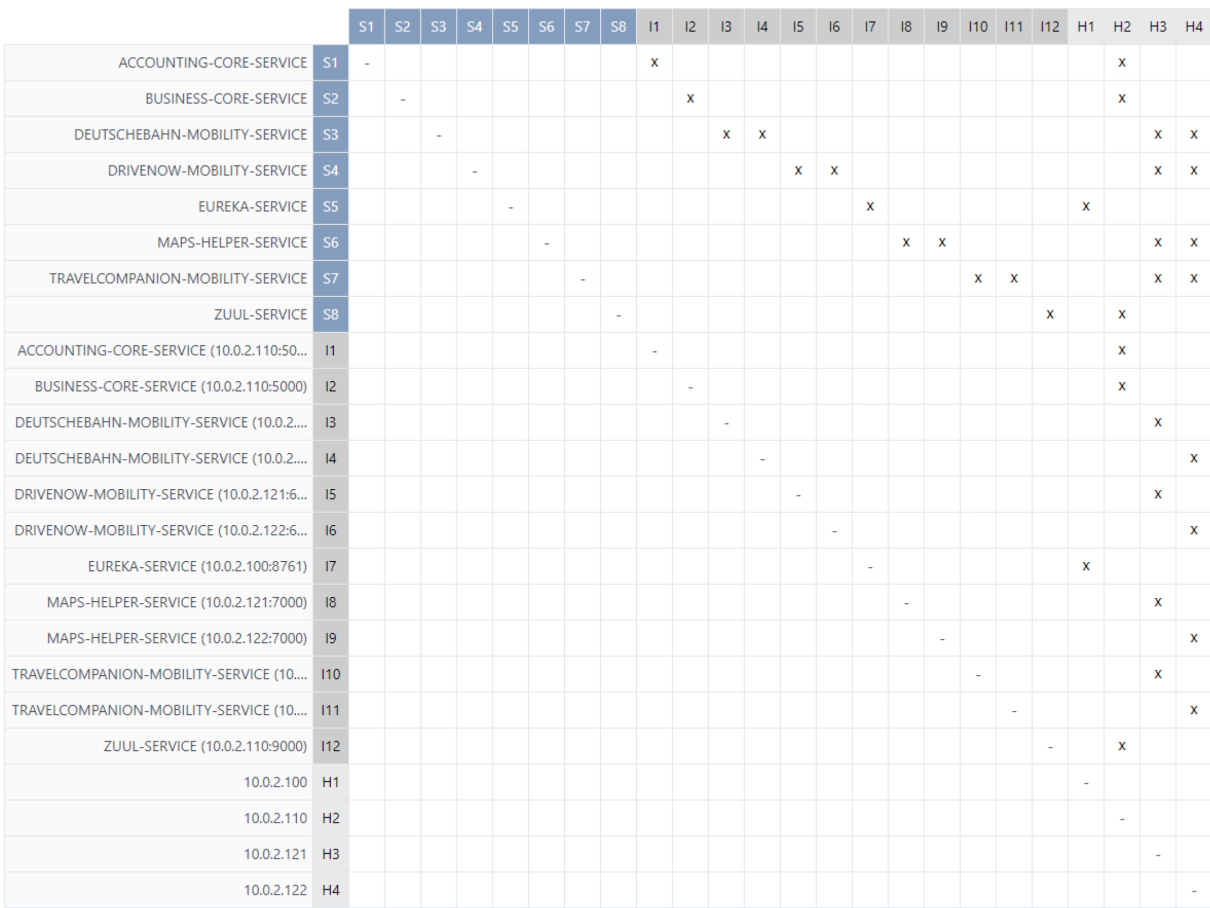
Nach Start des zweiten Hosts entspricht die Adjacency-Matrix Abbildung 40.

		S1	S2	S3	S4	I1	I2	I3	I4	H1	H2
ACCOUNTING-CORE-SERVICE	S1	-				x					x
BUSINESS-CORE-SERVICE	S2		-				x				x
EUREKA-SERVICE	S3			-				x		x	
ZUUL-SERVICE	S4				-				x		x
ACCOUNTING-CORE-SERVICE (10.0.2.110:50...	I1					-					x
BUSINESS-CORE-SERVICE (10.0.2.110:5000)	I2						-				x
EUREKA-SERVICE (10.0.2.100:8761)	I3							-		x	
ZUUL-SERVICE (10.0.2.110:9000)	I4								-		x
10.0.2.100	H1									-	
10.0.2.110	H2										-

Abbildung 40: Adjacency Matrix-nach Start von Host 2

Wie erwartet, werden die drei hierauf ausgeführten Serviceinstanzen erkannt und Services sowie die zugehörige Hardware abgeleitet. Die drei Instanzen stehen in Abhängigkeit zur Hardware „10.0.2.110“, was der IP des Hosts 2 entspricht. Jegliche Interlayer-Beziehungen sind korrekt erfasst.

Host 3 und 4 werden parallel gestartet. Da beide Hosts die gleichen Microservices betreiben, ist zu erwarten, dass je Servicekomponente zwei Instanzen erkannt werden, jedoch der Service selbst einmalig gelistet wird. Abbildung 41 zeigt den Zustand des Datenmodells nach dem Start der genannten Hosts.



		S1	S2	S3	S4	S5	S6	S7	S8	I1	I2	I3	I4	I5	I6	I7	I8	I9	I10	I11	I12	H1	H2	H3	H4
ACCOUNTING-CORE-SERVICE	S1	-								x													x		
BUSINESS-CORE-SERVICE	S2		-								x												x		
DEUTSCHEBAHN-MOBILITY-SERVICE	S3			-								x	x											x	x
DRIVENOW-MOBILITY-SERVICE	S4				-									x	x									x	x
EUREKA-SERVICE	S5					-										x						x			
MAPS-HELPER-SERVICE	S6						-										x	x						x	x
TRAVELCOMPANION-MOBILITY-SERVICE	S7							-											x	x				x	x
ZUUL-SERVICE	S8								-												x		x		
ACCOUNTING-CORE-SERVICE (10.0.2.110:50...	I1									-													x		
BUSINESS-CORE-SERVICE (10.0.2.110:5000)	I2										-												x		
DEUTSCHEBAHN-MOBILITY-SERVICE (10.0.2....	I3											-												x	
DEUTSCHEBAHN-MOBILITY-SERVICE (10.0.2....	I4												-												x
DRIVENOW-MOBILITY-SERVICE (10.0.2.121:6...	I5													-										x	
DRIVENOW-MOBILITY-SERVICE (10.0.2.122:6...	I6														-										x
EUREKA-SERVICE (10.0.2.100:8761)	I7															-						x			
MAPS-HELPER-SERVICE (10.0.2.121:7000)	I8																-							x	
MAPS-HELPER-SERVICE (10.0.2.122:7000)	I9																	-							x
TRAVELCOMPANION-MOBILITY-SERVICE (10....	I10																		-					x	
TRAVELCOMPANION-MOBILITY-SERVICE (10....	I11																			-					x
ZUUL-SERVICE (10.0.2.110:9000)	I12																				-		x		
10.0.2.100	H1																					-			
10.0.2.110	H2																						-		
10.0.2.121	H3																							-	
10.0.2.122	H4																								-

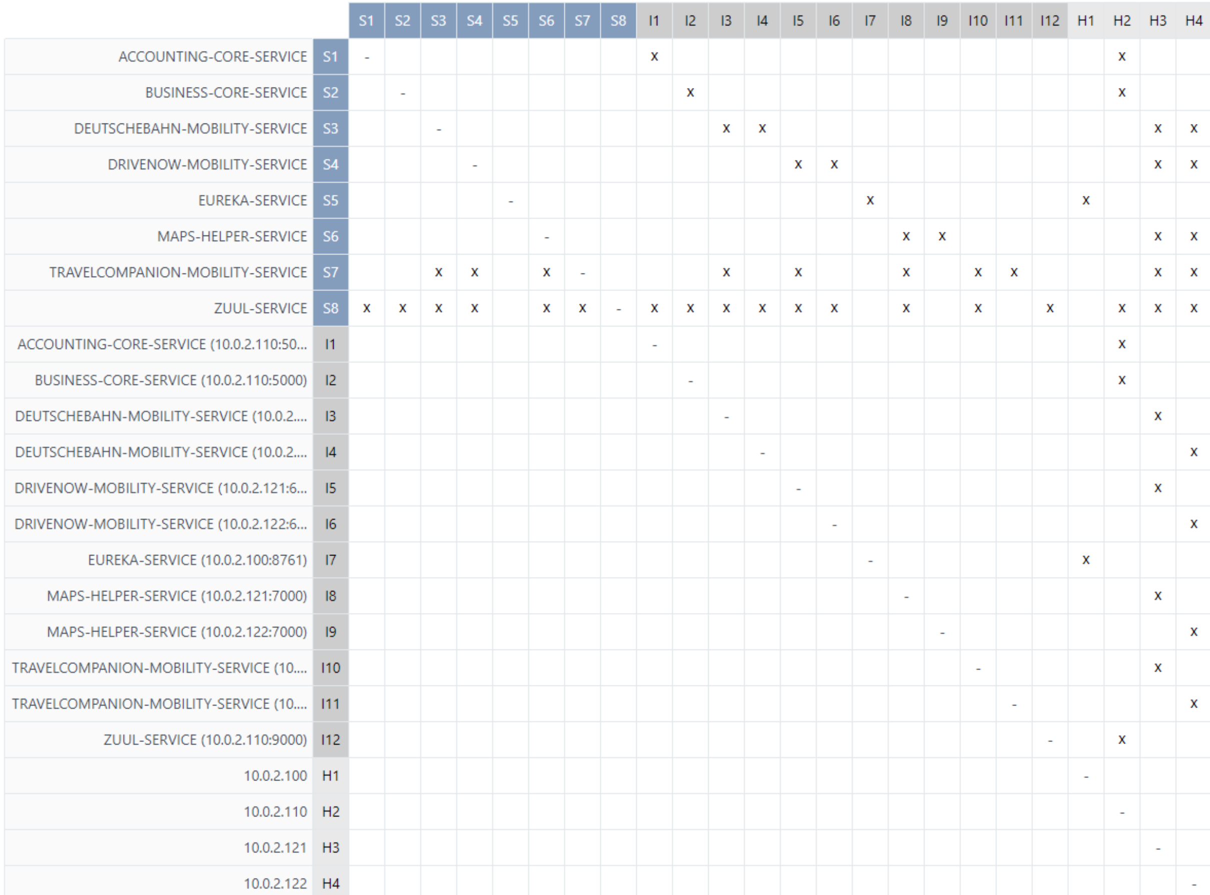
Abbildung 41: Adjacency-Matrix nach Start aller Hosts

Zu erkennen ist, dass die vier auf den Hosts betriebenen Services nun in der Matrix erscheinen. Jeder dieser Services steht in Relation zu je zwei Instanzen sowie zu zwei die Hosts repräsentierenden Hardwaresystemen. Dies entspricht den Erwartungen an die Repräsentation des Loadbalancings jener Services durch zwei Instanzen. Die Matrix zeigt nach Start aller Hosts fehlerfrei alle Services, Instanzen und Hardware sowie deren durch die Auswertung von Eureka erkennbaren Interlayer-Beziehungen an.

Evaluationsschritt 2: Erkennung von Relationen

Während durch den ersten Evaluationsschritt lediglich Relationen zwischen einem Service, dessen Instanzen und der zu den Instanzen gehörenden Hardware ermittelt werden konnten, wird im zweiten Schritt die Verarbeitung des Distributed Tracing evaluiert, welche jegliche direkte und indirekte Inter- und Intralayer-Beziehungen der Komponenten des Application- und Technology-Layers ermittelt.

Zu diesem Zweck wird die Geschäftsapplikation entsprechend dem beschriebenen Geschäftsprozess über die Anwendungsoberfläche verwendet, wodurch die involvierten Services Distributed Tracing Spans erzeugten. Mit jedem der drei Reiseanbieter wird jeweils eine Reise gesucht und gebucht.



		S1	S2	S3	S4	S5	S6	S7	S8	I1	I2	I3	I4	I5	I6	I7	I8	I9	I10	I11	I12	H1	H2	H3	H4
ACCOUNTING-CORE-SERVICE	S1	-								x													x		
BUSINESS-CORE-SERVICE	S2		-								x												x		
DEUTSCHEBAHN-MOBILITY-SERVICE	S3			-								x	x											x	x
DRIVENOW-MOBILITY-SERVICE	S4				-									x	x									x	x
EUREKA-SERVICE	S5					-										x						x			
MAPS-HELPER-SERVICE	S6						-										x	x						x	x
TRAVELCOMPANION-MOBILITY-SERVICE	S7			x	x		x	-				x		x			x		x	x				x	x
ZUUL-SERVICE	S8	x	x	x	x		x	x	-	x	x	x	x	x	x		x		x		x		x	x	x
ACCOUNTING-CORE-SERVICE (10.0.2.110:50...	I1									-													x		
BUSINESS-CORE-SERVICE (10.0.2.110:5000)	I2										-												x		
DEUTSCHEBAHN-MOBILITY-SERVICE (10.0.2....	I3											-												x	
DEUTSCHEBAHN-MOBILITY-SERVICE (10.0.2....	I4												-												x
DRIVENOW-MOBILITY-SERVICE (10.0.2.121:6...	I5													-										x	
DRIVENOW-MOBILITY-SERVICE (10.0.2.122:6...	I6														-										x
EUREKA-SERVICE (10.0.2.100:8761)	I7															-						x			
MAPS-HELPER-SERVICE (10.0.2.121:7000)	I8																-							x	
MAPS-HELPER-SERVICE (10.0.2.122:7000)	I9																	-							x
TRAVELCOMPANION-MOBILITY-SERVICE (10....	I10																		-					x	
TRAVELCOMPANION-MOBILITY-SERVICE (10....	I11																			-					x
ZUUL-SERVICE (10.0.2.110:9000)	I12																				-		x		
10.0.2.100	H1																					-			
10.0.2.110	H2																						-		
10.0.2.121	H3																							-	
10.0.2.122	H4																								-

Abbildung 42: Adjacency-Matrix nach Evaluationsschritt 2

In der resultierenden Matrix ist der Zuul Gateway Proxy durch seine Abhängigkeiten zu allen Services gut identifizierbar. Lediglich zu Service S5 (Eureka) wird keine Abhängigkeit erkannt, was schlüssig ist, da dies einer der Infrastrukturservices ist und nicht über den Zuul Proxy von außen aufgerufen wird. Zu sehen ist außerdem in der Zeile des Zuul-Service, dass keine Abhängigkeit zu H1 (Host 1) besteht, da darauf lediglich Infrastruktur-Services betrieben werden. Außerdem können fehlende Abhängigkeiten zu den Instanzen I9 (Maps-Helper-Service) und I11 (Travelcompanion-Mobility-Service) beobachtet werden, was darauf zurückzuführen ist, dass bei

der Ausführung des Travelcompanion-Reiseanbieters die jeweils andere der zwei verfügbaren Instanzen durch das Loadbalancing ausgewählt wurde.

Die Abhängigkeiten des Travelcompanion-Service (S7) bestätigen dies. Der Service zeigt Abhängigkeiten zum DeutscheBahn-Mobility-Service (S3) und dem DriveNow-Mobility-Service (S4), welche bestehen, um Reisen der beiden Anbieter abzurufen. Des Weiteren besteht eine Abhängigkeit zum Maps-Helper-Service (S6), um Reisedistanzen zu ermitteln. Die Abhängigkeiten zu den Instanzen I8 und I10 bestätigen die Loadbalancing-Selektion während der Ausführung des Travelcompanion-Anbieters.

Die in der Matrix abgebildeten Relationen sind valide, schlüssig und unter Berücksichtigung der erzeugten Menge an Spans vollständig. Lediglich Abhängigkeiten zu nicht durch den Loadbalancer selektierten Instanzen fehlen im Architekturmodell und würden im zeitlichen Verlauf durch weitere Anwendungsnutzung sichtbar werden.

Evaluationsschritt 3: Anreicherung des Architekturmodells mit Businesssemantik

Um die Elemente des Business-Layers zu erzeugen, wird der in Kapitel 6.1 beschriebene Geschäftsprozess über das Modellierungswerkzeug der Webanwendung erstellt. Anschließend wird das Mapper-Werkzeug der Webanwendung verwendet, um Beziehungen zwischen modellierten Aktivitäten des Geschäftsprozesses und erkannten Microservices zu etablieren.

Im Ergebnis der Prozessmodellierung sollten in der Adjacency-Matrix der Prozess, alle modellierten Aktivitäten und ihre Nachfolger-Abhängigkeiten sichtbar werden.

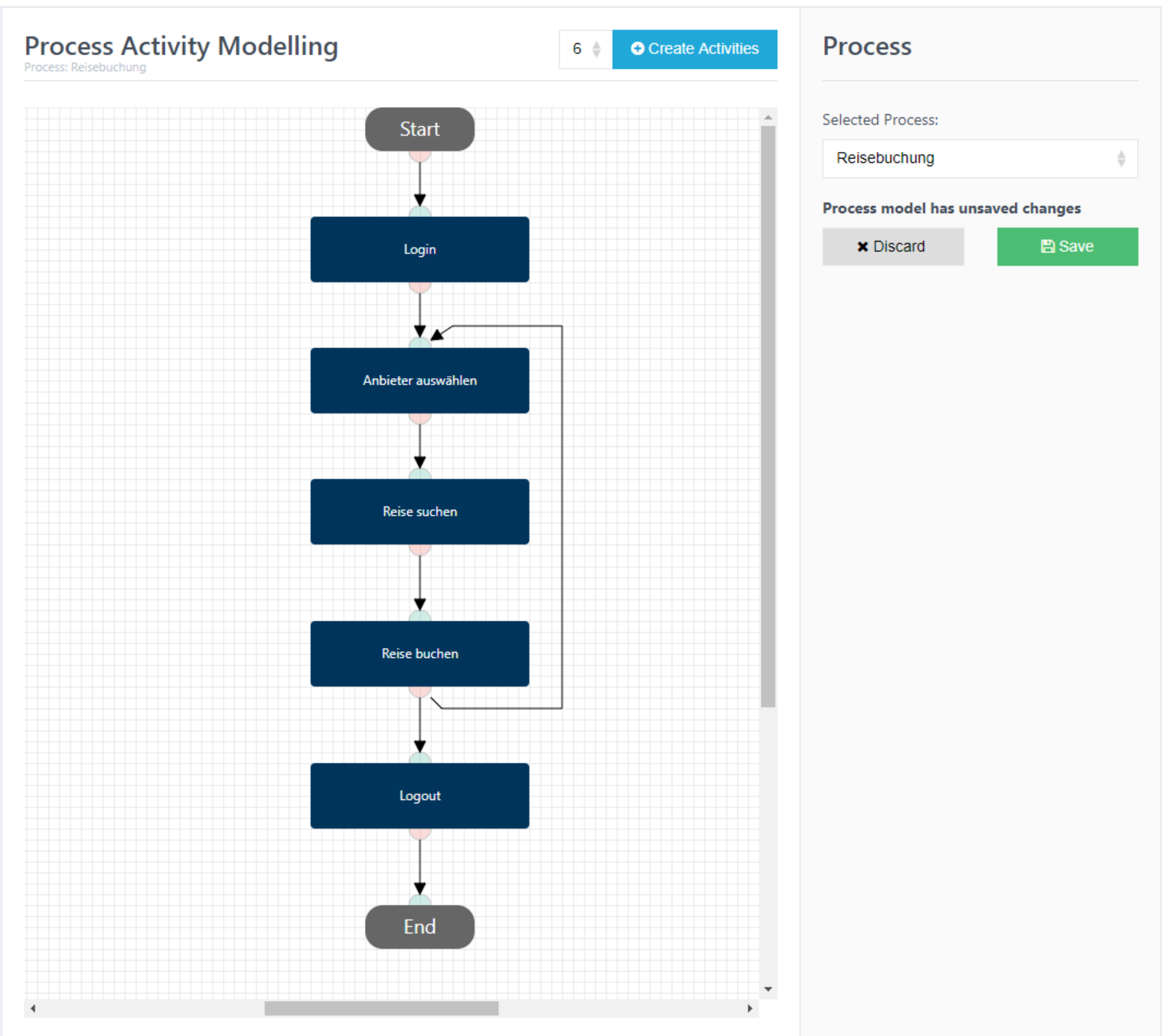
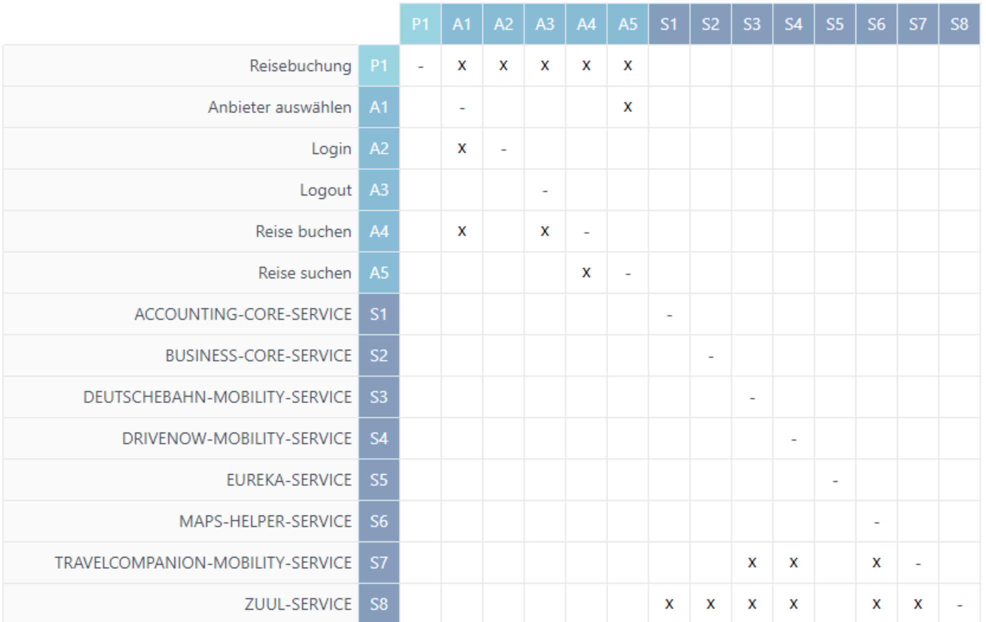


Abbildung 43: Modellierung des Evaluationsgeschäftsprozesses

Wie aus Abbildung 43 ersichtlich, ist es möglich, mit dem Modellierungstool den gegebenen Geschäftsprozess zu modellieren. Auch der zyklische Aktivitätsverlauf zwischen den Aktivitäten „Anbieter Auswählen“, „Reise suchen“, „Reise Buchen“, „Anbieter Auswählen“ kann durch das Tool abgebildet werden.

Mit einem Klick auf den Button „Save“ wird das Datenmodell erweitert und es werden entsprechende Benachrichtigungen über neu erzeugte Relationen und Revisionen der Aktivitäts- und Prozesskomponenten angezeigt, welche über die Websocket Bridge in Echtzeit empfangen werden.



		P1	A1	A2	A3	A4	A5	S1	S2	S3	S4	S5	S6	S7	S8
Reisebuchung	P1	-	x	x	x	x	x								
Anbieter auswählen	A1		-				x								
Login	A2		x	-											
Logout	A3				-										
Reise buchen	A4		x		x	-									
Reise suchen	A5					x	-								
ACCOUNTING-CORE-SERVICE	S1							-							
BUSINESS-CORE-SERVICE	S2								-						
DEUTSCHEBAHN-MOBILITY-SERVICE	S3									-					
DRIVENOW-MOBILITY-SERVICE	S4										-				
EUREKA-SERVICE	S5											-			
MAPS-HELPER-SERVICE	S6												-		
TRAVELCOMPANION-MOBILITY-SERVICE	S7									x	x		x	-	
ZUUL-SERVICE	S8							x	x	x	x		x	x	-

Abbildung 44: Ausschnitt der Adjacency-Matrix nach Prozessmodellierung

Die Adjacency-Matrix entspricht nach der Prozessmodellierung Abbildung 44 (Instanzen und Hardware sind ausgeblendet). Für den Prozess sowie all seine Aktivitäten ist entsprechend der Erwartungen ein Eintrag vorhanden. In der Zeile des Prozesses (P1) ist zu erkennen, dass diesem die modellierten Aktivitäten zugeordnet sind und dieser somit in Abhängigkeit zu den Aktivitäten steht. In den Zeilen der Aktivitäten (A1–A5) können durch die mit einem „x“ gekennzeichneten Felder die Nachfolger-Beziehungen abgelesen werden. Sind zwei Felder einer Zeile mit einem „x“ gekennzeichnet, so hat eine Aktivität zwei potentielle Nachfolger in einer Oder-Beziehung. Die Startaktivität Login (A2) ist dadurch erkennbar, dass keine der Aktivitäten eine Nachfolgerelation zu dieser aufweist. Die Endaktivität Logout (A3) ist erkennbar, da sie keinerlei Nachfolgerbeziehungen aufweist.

Die in der Matrix aufgezeigten Aktivitäten und Prozesse sind entsprechend der Modellierung vollständig und die aufgezeigten Nachfolgerbeziehungen der Aktivitäten sind den Erwartungen entsprechend.

Um Beziehungen zwischen den hinzugefügten Aktivitäten und den technisch ermittelten Komponenten zu etablieren, wird ein Mapping von Aktivitäten zu Pfaden durchgeführt, wie in Kapitel 5.2.2 beschrieben. Das Ergebnis dieses Mappings sollten Relationen zwischen den Aktivitäten und den Services sowie indirekte Abhängigkeiten zu genutzten Instanzen und verwendeter Hardware sein. Die erwarteten Bezie-

hungen zwischen Aktivitäten und Services ergeben sich daraus, welcher Microservice für die technische Realisierung welcher Aktivität zuständig ist. Tabelle 8 zeigt die ausführenden Services der Geschäftsaktivitäten auf.

Geschäftsaktivität	Ausführender Service
Login	Zuul-Service
Anbieter auswählen	Business-Core-Service
Reise suchen	Je nach ausgewähltem Anbieter: DeutscheBahn-Mobility-Service DriveNow-Mobility-Service Travelcompanion-Service
Reise buchen	Accounting-Core-Service
Logout	Zuul-Service

Tabelle 8: Zuordnung von Geschäftsaktivitäten und Services

Das Mapping erfolgt durch die Zuordnung von regulären Ausdrücken zu Aktivitäten. Die regulären Ausdrücke werden auf Aufrufpfade angewandt. Somit kann ein Aufruf wie „http://hostname/login" beispielsweise der Aktivität Login zugeordnet werden.

Existing Mappings

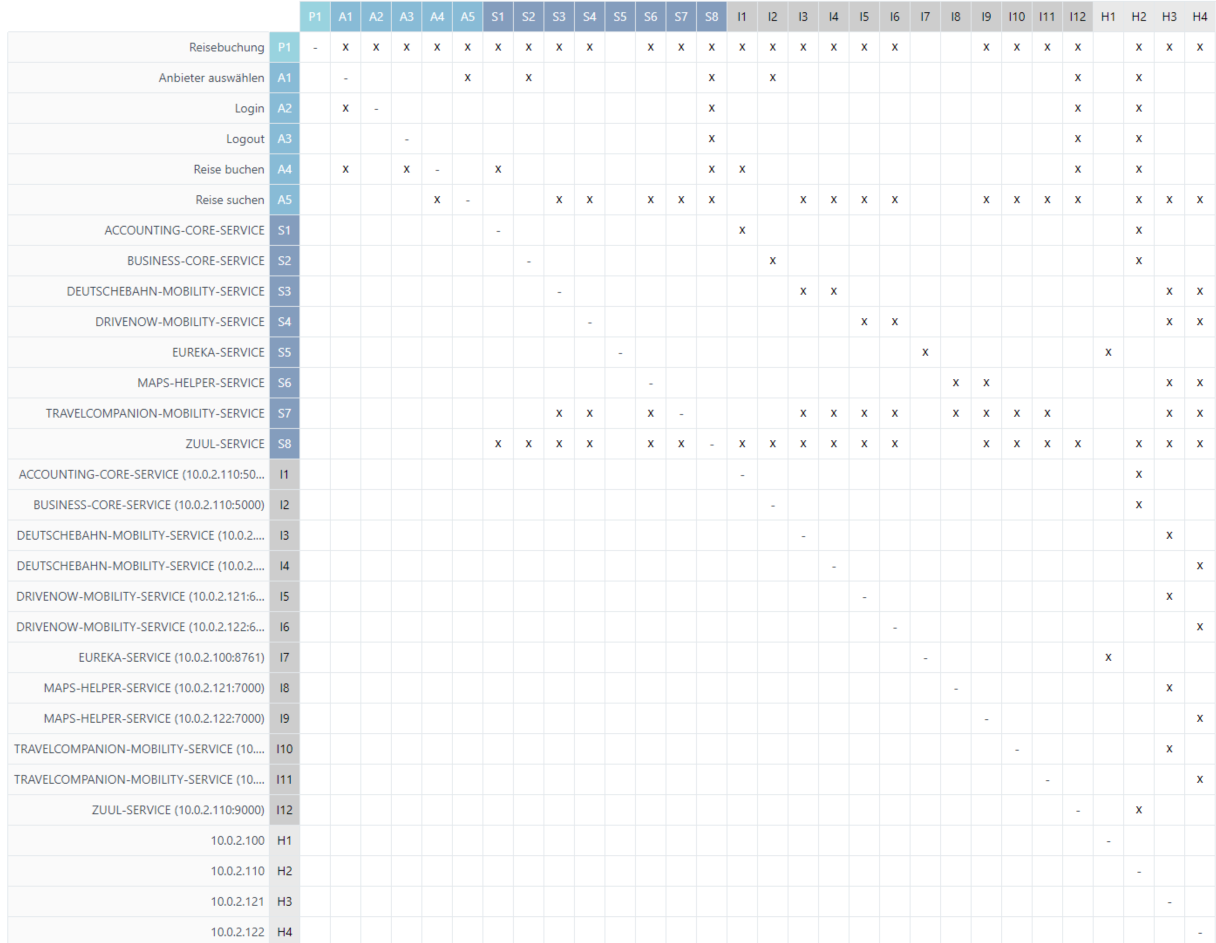
Process	Activity	Regular expression	HTTP Methods
Reisebuchung	Login	login$	GET PUT POST DELETE
Reisebuchung	Logout	logout$	GET PUT POST DELETE
Reisebuchung	Reise buchen	/[0-9]+/book$	GET PUT POST DELETE
Reisebuchung	Reise suchen	routes$	GET PUT POST DELETE
Reisebuchung	Anbieter auswählen	list$	GET PUT POST DELETE

Abbildung 45: Angelegte Regeln des Aktivitäts-Mappings

Wird der Aufruf im System beobachtet, werden automatisch Beziehungen zwischen den involvierten technischen Komponenten (Services, Instanzen, Hardware) des

Aufrufs und der zugeordneten Geschäftsaktivität hergestellt. Zur Evaluation werden den einzelnen Aktivitäten die in Abbildung 45 aufgezeigten Regeln zugeordnet.

Durch das Anlegen der Regeln sollte sich das Architekturmodell um Relationen zwischen Aktivitäten und Elementen des Application-Layers sowie des Technology-Layers ergänzen. Da Aufrufe, für welche kein Mapping existiert, zwischengespeichert und erneut verarbeitet werden, sobald ein für sie gültiges Mapping erzeugt ist, werden die Abhängigkeiten aufgrund der in Evaluationsschritt 2 erzeugten Aufrufe sofort und ohne weitere Traffic-Erzeugung angelegt. Die resultierende Adjacency-Matrix ist in Abbildung 46 dargestellt und umfasst nun die gesamte Evaluationsumgebung inklusive all ihrer Abhängigkeiten.



		P1	A1	A2	A3	A4	A5	S1	S2	S3	S4	S5	S6	S7	S8	I1	I2	I3	I4	I5	I6	I7	I8	I9	I10	I11	I12	H1	H2	H3	H4
Reisebuchung	P1	-	x	x	x	x	x	x	x	x	x		x	x	x	x	x	x	x	x	x			x	x	x	x		x	x	x
Anbieter auswählen	A1		-				x		x						x		x										x		x		
Login	A2		x	-											x												x		x		
Logout	A3				-										x												x		x		
Reise buchen	A4		x		x	-		x							x	x											x		x		
Reise suchen	A5					x	-			x	x		x	x	x			x	x	x	x			x	x	x	x		x	x	x
ACCOUNTING-CORE-SERVICE	S1							-								x													x		
BUSINESS-CORE-SERVICE	S2								-								x												x		
DEUTSCHEBAHN-MOBILITY-SERVICE	S3									-								x	x											x	x
DRIVENOW-MOBILITY-SERVICE	S4										-									x	x									x	x
EUREKA-SERVICE	S5											-										x						x			
MAPS-HELPER-SERVICE	S6												-										x	x						x	x
TRAVELCOMPANION-MOBILITY-SERVICE	S7									x	x		x	-				x	x	x	x		x	x	x	x				x	x
ZUUL-SERVICE	S8							x	x	x	x		x	x	-	x	x	x	x	x	x			x	x	x	x		x	x	x
ACCOUNTING-CORE-SERVICE (10.0.2.110:50...	I1															-													x		
BUSINESS-CORE-SERVICE (10.0.2.110:5000)	I2																-												x		
DEUTSCHEBAHN-MOBILITY-SERVICE (10.0.2....	I3																	-												x	
DEUTSCHEBAHN-MOBILITY-SERVICE (10.0.2....	I4																		-												x
DRIVENOW-MOBILITY-SERVICE (10.0.2.121:6...	I5																			-										x	
DRIVENOW-MOBILITY-SERVICE (10.0.2.122:6...	I6																				-										x
EUREKA-SERVICE (10.0.2.100:8761)	I7																					-						x			
MAPS-HELPER-SERVICE (10.0.2.121:7000)	I8																						-							x	
MAPS-HELPER-SERVICE (10.0.2.122:7000)	I9																							-							x
TRAVELCOMPANION-MOBILITY-SERVICE (10....	I10																								-					x	
TRAVELCOMPANION-MOBILITY-SERVICE (10....	I11																									-					x
ZUUL-SERVICE (10.0.2.110:9000)	I12																										-		x		
10.0.2.100	H1																											-			
10.0.2.110	H2																												-		
10.0.2.121	H3																													-	
10.0.2.122	H4																														-

Abbildung 46: Adjacency-Matrix nach Evaluationsschritt 3

Zu erkennen ist, dass alle Aktivitäten nun Abhängigkeiten zum Zuul-Service sowie seiner Instanz und Hardware haben. Da jeder andere Service durch das Zuul-

Gateway aufgerufen wird, ist dies plausibel. Die Aktivitäten „Login“ (A2) und „Logout“ (A3) zeigen keine weiteren Abhängigkeiten, was den erwarteten Service-Zuordnungen (siehe Kapitel 6.1) entspricht. „Anbieter auswählen“ (A1) zeigt des Weiteren korrekterweise eine Abhängigkeit zum Business-Core-Service auf. Die Aktivität „Reise suchen“ (A5) zeigt Abhängigkeiten zu den Services DeutscheBahn-Mobility-Service sowie DriveNow-Mobility-Service. Außerdem besteht eine Abhängigkeit zum Travelcompanion-Service als dritter Anbieter und eine indirekte Abhängigkeit zum Maps-Helper-Service, welcher durch den Travelcompanion-Service verwendet wird. Die erkannten Relationen sind plausibel und vollständig. Auffällig ist, dass die Aktivität „Reise suchen“ (A5) trotz ihrer Abhängigkeit zum Travelcompanion-Service keine indirekte Abhängigkeit zu der von ihm verwendeten Instanz des Maps-Helper-Service I8 aufweist. Die Ursache hierfür liegt darin, dass der Maps-Helper-Service lediglich einmalig aufgerufen und hierbei durch das Loadbalancing die zweite Instanz I9 verwendet wird. Die Relation zu I8 würde durch Folgeaufrufe erkannt werden.

Durch die ersten drei Evaluationsschritte kann das Erfassen einer Architektur erfolgreich überprüft werden. Alle Architekturkomponenten sind erfasst und Relationen zwischen kommunizierenden Komponenten sind erkannt. Der Geschäftsprozess kann dem Architekturmodell hinzugefügt und Abhängigkeiten zwischen den verschiedenen Ebenen können etabliert werden.

Evaluationsschritt 4: Reaktionsvermögen auf Architekturänderungen

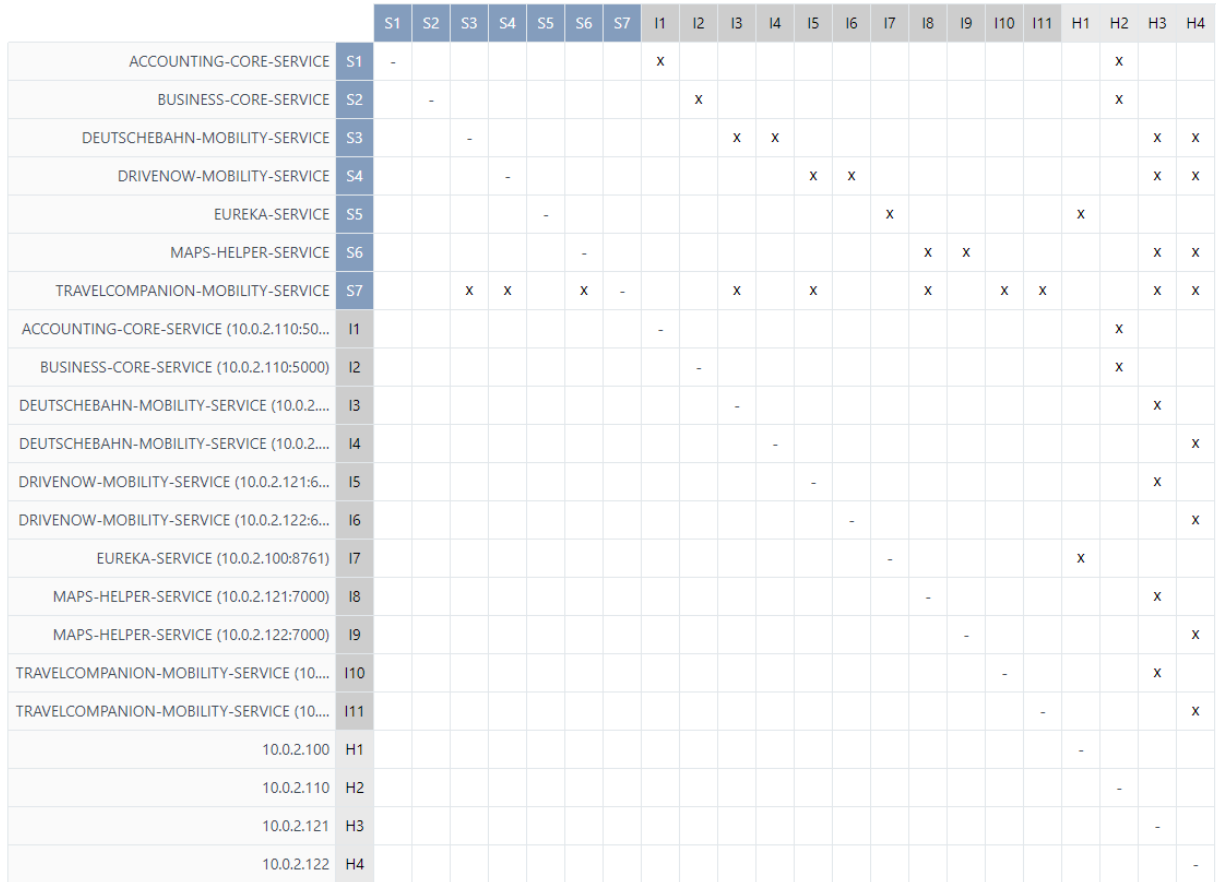
Nachdem das Erkennen von Architekturkomponenten evaluiert ist, wird das Erkennen von Änderungen innerhalb der Architektur evaluiert. Es existieren zwei relevante Szenarien. Ein Szenario ist die Entfernung einer Komponente aus einer Architektur, das zweite ist die Modifikation der Komponente. Wird die Software eines Microservice aktualisiert, so wird er gestoppt und neu gestartet. Da sich hierdurch die Relationen geändert haben können, sind bisher erkannte Relationen als nicht mehr gültig zu definieren, bis sie erneut durch das System beobachtet werden.

Im Rahmen des vierten Evaluationsschritts wird der Zuul-Service gestoppt und evaluiert, dass dieser nicht mehr Bestandteil des Architekturmodells ist. Anschließend

wird der Service wieder gestartet und geprüft, ob der Service dem Architekturmodell erneut hinzugefügt und vorherige Relationen entfernt worden sind.

Durch den vierten Evaluationsschritt wird das Reaktionsvermögen auf Veränderungen der Architektur geprüft, indem ein Service für einen Zeitraum abgeschaltet und anschließend neu gestartet wird. Es wird erwartet, dass die abgeschaltete Komponente durch den Architekturerkennungsservice bemerkt und die zugehörige Revision geschlossen wird. Nach Neustart der Komponente sollte eine neue Revision erzeugt worden sein.

Der Prozess des Zuul-Service wird durch Zusenden eines SIGTERM-Signals beendet, was einem geordneten Abschalten entspricht. Das Frontend zeigt nach fünf Sekunden das Entfallen der Komponente an. Dies liegt im erwarteten Zeitrahmen von etwa zehn Sekunden, welcher dem Intervall der Eureka-Verarbeitung entspricht.

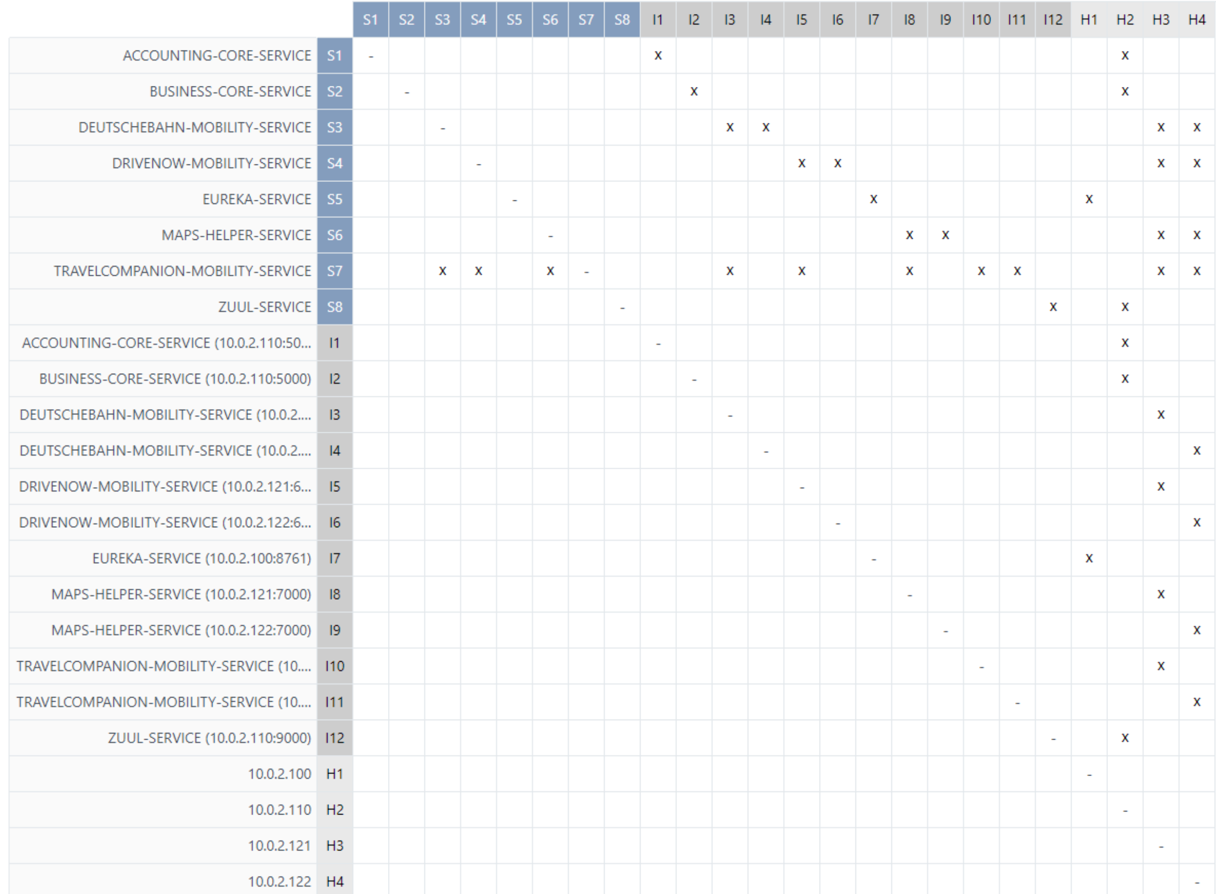


		S1	S2	S3	S4	S5	S6	S7	I1	I2	I3	I4	I5	I6	I7	I8	I9	I10	I11	H1	H2	H3	H4
ACCOUNTING-CORE-SERVICE	S1	-							x												x		
BUSINESS-CORE-SERVICE	S2		-							x											x		
DEUTSCHEBAHN-MOBILITY-SERVICE	S3			-							x	x										x	x
DRIVENOW-MOBILITY-SERVICE	S4				-								x	x								x	x
EUREKA-SERVICE	S5					-									x					x			
MAPS-HELPER-SERVICE	S6						-									x	x					x	x
TRAVELCOMPANION-MOBILITY-SERVICE	S7			x	x		x	-			x		x			x		x	x			x	x
ACCOUNTING-CORE-SERVICE (10.0.2.110:50...	I1								-												x		
BUSINESS-CORE-SERVICE (10.0.2.110:5000)	I2									-											x		
DEUTSCHEBAHN-MOBILITY-SERVICE (10.0.2....	I3										-											x	
DEUTSCHEBAHN-MOBILITY-SERVICE (10.0.2....	I4											-											x
DRIVENOW-MOBILITY-SERVICE (10.0.2.121:6...	I5												-									x	
DRIVENOW-MOBILITY-SERVICE (10.0.2.122:6...	I6													-									x
EUREKA-SERVICE (10.0.2.100:8761)	I7														-					x			
MAPS-HELPER-SERVICE (10.0.2.121:7000)	I8															-						x	
MAPS-HELPER-SERVICE (10.0.2.122:7000)	I9																-						x
TRAVELCOMPANION-MOBILITY-SERVICE (10....	I10																	-				x	
TRAVELCOMPANION-MOBILITY-SERVICE (10....	I11																		-				x
10.0.2.100	H1																			-			
10.0.2.110	H2																				-		
10.0.2.121	H3																					-	
10.0.2.122	H4																						-

Abbildung 47: Adjacency-Matrix nach Abschaltung des Zuul-Service

Wie in Abbildung 47 erkennbar, ist der Zuul-Service sowie seine Instanz anschließend nicht mehr Teil der Matrix. Die zugehörige Hardware-Komponente bleibt aufgrund anderer auf ihr betriebener Services erhalten.

Anschließend wird der Zuul-Service erneut gestartet. Da durch den Discovery-Service nicht sichergestellt wird, ob sich die Version des Service verändert hat, wird die Annahme getroffen, dass bisherige Relationen nicht mehr gültig sind, bis sie erneut erkannt werden. Nach dem Start des Service hat sich die Matrix, wie in Abbildung 48 sichtbar, verändert.



		S1	S2	S3	S4	S5	S6	S7	S8	I1	I2	I3	I4	I5	I6	I7	I8	I9	I10	I11	I12	H1	H2	H3	H4
ACCOUNTING-CORE-SERVICE	S1	-								x													x		
BUSINESS-CORE-SERVICE	S2		-								x												x		
DEUTSCHEBAHN-MOBILITY-SERVICE	S3			-								x	x											x	x
DRIVENOW-MOBILITY-SERVICE	S4				-									x	x									x	x
EUREKA-SERVICE	S5					-										x						x			
MAPS-HELPER-SERVICE	S6						-										x	x						x	x
TRAVELCOMPANION-MOBILITY-SERVICE	S7			x	x		x	-				x		x			x		x	x				x	x
ZUUL-SERVICE	S8								-												x		x		
ACCOUNTING-CORE-SERVICE (10.0.2.110:50...	I1									-													x		
BUSINESS-CORE-SERVICE (10.0.2.110:5000)	I2										-												x		
DEUTSCHEBAHN-MOBILITY-SERVICE (10.0.2....	I3											-												x	
DEUTSCHEBAHN-MOBILITY-SERVICE (10.0.2....	I4												-												x
DRIVENOW-MOBILITY-SERVICE (10.0.2.121:6...	I5													-										x	
DRIVENOW-MOBILITY-SERVICE (10.0.2.122:6...	I6														-										x
EUREKA-SERVICE (10.0.2.100:8761)	I7															-						x			
MAPS-HELPER-SERVICE (10.0.2.121:7000)	I8																-							x	
MAPS-HELPER-SERVICE (10.0.2.122:7000)	I9																	-							x
TRAVELCOMPANION-MOBILITY-SERVICE (10....	I10																		-					x	
TRAVELCOMPANION-MOBILITY-SERVICE (10....	I11																			-					x
ZUUL-SERVICE (10.0.2.110:9000)	I12																				-		x		
10.0.2.100	H1																					-			
10.0.2.110	H2																						-		
10.0.2.121	H3																							-	
10.0.2.122	H4																								-

Abbildung 48: Adjacency-Matrix nach Neustart des Zuul-Service

Es ist erkennbar, dass der Zuul-Service (S8) erneut in der Matrix gelistet wird, in der zugehörigen Zeile jedoch nur Abhängigkeiten zur ebenfalls wieder hinzugefügten Instanz und der Hardware bestehen. Diese Informationen werden aus der Eureka-Auswertung ermittelt und sind somit bereits erneut bestätigte Abhängigkeiten. Alle

weiteren Abhängigkeiten sind nicht erneut bestätigt und erscheinen erst durch erneute Anfragen an das System und die damit einhergehende Verarbeitung des Distributed Tracing.

Die Erzeugung der Revisionen wird anschließend in der Datenbank überprüft. Tabelle 9 zeigt alle bis zu dem beschriebenen Zeitpunkt erzeugten Revisionen auf.

ID ▲ 1	COMPONENT_ID	VALID_FROM	VALID_TO	NAME	DTYPE
1	1	2017-10-21 17:21:14	2017-10-21 17:24:04	ZUUL-SERVICE	Service
2	2	2017-10-21 17:21:14	NULL	ACCOUNTING-CORE-SERVICE	Service
3	3	2017-10-21 17:21:14	NULL	MAPS-HELPER-SERVICE	Service
4	4	2017-10-21 17:21:14	NULL	BUSINESS-CORE-SERVICE	Service
5	5	2017-10-21 17:21:14	NULL	DEUTSCHEBAHN-MOBILITY-SERVICE	Service
6	6	2017-10-21 17:21:14	NULL	DRIVENOW-MOBILITY-SERVICE	Service
7	7	2017-10-21 17:21:14	NULL	EUREKA-SERVICE	Service
8	8	2017-10-21 17:21:14	NULL	TRAVELCOMPANION-MOBILITY-SERVICE	Service
9	9	2017-10-21 17:21:14	NULL	10.0.2.122	Hardware
10	10	2017-10-21 17:21:14	NULL	10.0.2.100	Hardware
11	11	2017-10-21 17:21:14	NULL	10.0.2.110	Hardware
12	12	2017-10-21 17:21:14	NULL	10.0.2.121	Hardware
13	13	2017-10-21 17:21:14	NULL	10.0.2.121:DRIVENOW-MOBILITY-SERVICE:6001	Instance
14	14	2017-10-21 17:21:15	NULL	10.0.2.100:EUREKA-SERVICE:8761	Instance
15	15	2017-10-21 17:21:15	NULL	10.0.2.122:TRAVELCOMPANION-MOBILITY-SERVICE:6002	Instance
16	16	2017-10-21 17:21:15	NULL	10.0.2.121:MAPS-HELPER-SERVICE:7000	Instance
17	17	2017-10-21 17:21:15	NULL	10.0.2.121:TRAVELCOMPANION-MOBILITY-SERVICE:6002	Instance
18	18	2017-10-21 17:21:15	NULL	10.0.2.110:BUSINESS-CORE-SERVICE:5000	Instance
19	19	2017-10-21 17:21:15	NULL	10.0.2.121:DEUTSCHEBAHN-MOBILITY-SERVICE:6003	Instance
20	20	2017-10-21 17:21:15	NULL	10.0.2.122:DEUTSCHEBAHN-MOBILITY-SERVICE:6003	Instance
21	21	2017-10-21 17:21:15	2017-10-21 17:24:04	10.0.2.110:ZUUL-SERVICE:9000	Instance
22	22	2017-10-21 17:21:15	NULL	10.0.2.122:DRIVENOW-MOBILITY-SERVICE:6001	Instance
23	23	2017-10-21 17:21:15	NULL	10.0.2.122:MAPS-HELPER-SERVICE:7000	Instance
24	24	2017-10-21 17:21:15	NULL	10.0.2.110:ACCOUNTING-CORE-SERVICE:5001	Instance
25	1	2017-10-21 17:30:34	NULL	ZUUL-SERVICE	Service
26	21	2017-10-21 17:30:34	NULL	10.0.2.110:ZUUL-SERVICE:9000	Instance

Tabelle 9: Revisionen-Tabelle der Datenbank

Es ist ersichtlich, dass die Revisionen des Zuul-Service (ID 1) und seiner Instanz (ID 21) zeitgleich nach dem Abschalten des Service geschlossen werden, da das Feld VALID_TO mit einem Zeitwert versehen wird und somit aus dem Architekturmodell späterer Zeitpunkte entfällt. Sechs Minuten später werden mit dem Neustart des Service neue Revisionen für die beiden Komponenten erzeugt (ID 25 und 26). Durch das System werden die jeweiligen Komponenten korrekt wiedererkannt, was an der gemeinsamen COMPONENT_ID der Revisionen (1 und 25, 21 und 26) sichtbar wird.

6.2.2 Evaluation der Performance- und Ressourcenauswirkungen

Jeder Service muss zur korrekten Funktionsweise des Architekturerkennungsdienstes um eine Distributed-Tracing-Instrumentierung erweitert werden. Die Auswirkungen und Ressourcenkosten jener Instrumentierung sind Fokus dieses Kapitels.

Zur Aufzeichnung von System- und Prozessmetriken wird Host 2 mit Monitoring-Agents der Application-Performance-Monitoring-Lösung Dynatrace (Dynatrace LLC., 2017a) ausgerüstet. Anschließend werden mithilfe der Applikation Apache JMeter (Apache Software Foundation, 2017) systematisch Aufrufe in der Microservice-Architektur erzeugt. Der Testplan sieht einen Zeitraum von fünf Minuten vor, in welchem mit einer Frequenz von zehn Aufrufen/Sekunde Last auf dem Host erzeugt wird. Hierzu wird der Pfad „/business-core-service/businesses/list" verwendet, welcher den Zuul-Service sowie den Business-Core-Service involviert und verfügbare Anbieter auflistet. Bei einem Aufruf werden vier Spans erzeugt:

- beim Eintreffen der Anfrage vom Client an den Zuul-Service
- beim Weiterleiten der Anfrage von Zuul-Service zu Business-Code-Service
- beim Antworten des Business-Core-Service an den Zuul-Service
- Beim Antworten des Zuul-Service an den Client

Anschließend wird dieselbe Last mit deaktivierter Instrumentierung der benannten Services erzeugt und es werden die Metriken der Services und des Hosts aufgezeichnet. In einer darauffolgenden Analyse werden die aufgezeichneten Metriken (CPU, Netzwerklast und Antwortzeiten) den getesteten Zeiträume gegenübergestellt, um die Auswirkungen der Instrumentierung auf jene Metriken zu überprüfen. Der Analysezeitraum umfängt nicht den gesamten Testzeitraum, sondern lediglich den Zeitraum von Minute eins bis vier, um Start- und Stop-Anomalien nicht auszuwerten und geringfügige Abweichungen der Startzeitpunkte zwischen JMeter und Dynatrace zu entkräften.

Um sicherzustellen, dass das Aktivieren und Deaktivieren der Instrumentierung erfolgreich ist, werden CPU-Daten von Host 1 und Host 2 für beide Testzeiträume gegenübergestellt. Zu erwarten ist eine Korrelation der CPU beider Hosts bei aktivierter Instrumentierung aufgrund der von Host 2 erzeugten Spans, welche durch den

Architekturerkennungsservice auf Host 1 verarbeitet werden. Bei deaktivierter Instrumentierung ist zu erwarten, dass die CPU-Werte von Host 1 durch den Test nicht beeinflusst werden.

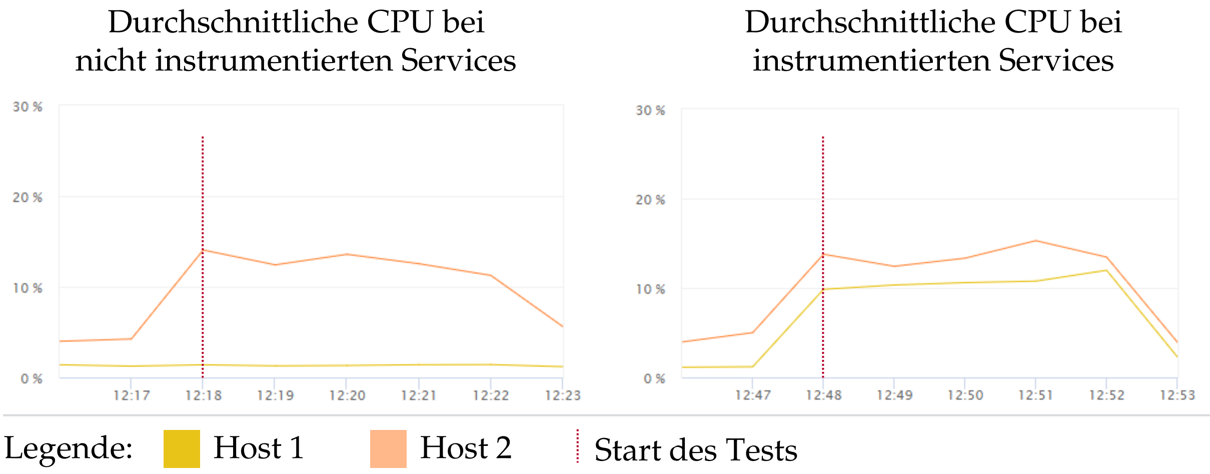


Abbildung 49: Gegenüberstellung der CPU-Auslastung der Testzeiträume

Wie in Abbildung 49 zu erkennen, entspricht das CPU-Verhalten den Erwartungen des jeweiligen Testzeitraums. Im linken Graphen ist kein Ausschlag der CPU von Host 1 zu erkennen, während der CPU-Verlauf von Host 1 im rechten Diagramm eine starke Korrelation zum CPU-Verlauf von Host 2 aufzeigt.

Aufgrund der beobachteten CPU-Metriken kann darauf geschlossen werden, dass die Testfälle korrekt konfiguriert ist und eine Verbreitung von Spans nur im instrumentierten Testfall erfolgt.

Auswertung der Antwortzeiten

Die Antwortzeiten auf die strukturiert generierten Anfragen werden mit JMeter in Form von Textdateien erfasst und anschließend über das Tool BlazeMeter Sense (BlazeMeter, 2017) visualisiert und analysiert. Abbildung 50 stellt das durchschnittliche Antwortverhalten des Testplans für instrumentierte und nicht instrumentierte Microservices gegenüber.

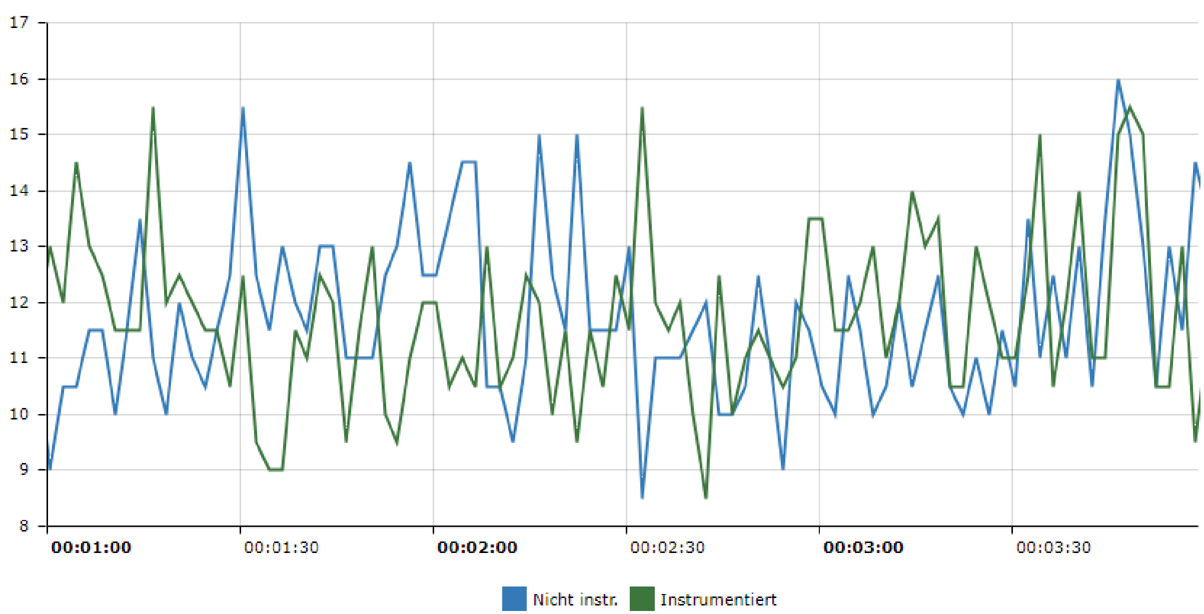


Abbildung 50: Durchschnittliche Antwortzeiten nicht instrumentierter versus instrumentierter Host (BlazeMeter, 2017)

Es ist zu erkennen, dass sich beide Konfigurationen im Durchschnitt (Aggregation von je zehn Sekunden) in einem Bereich zwischen 8 und 16 Millisekunden bewegen. Auch die Verteilung der Antwortzeiten zeigt keine besondere Abweichung auf. Tabelle 10 zeigt eine Zusammenfassung des Antwortverhaltens beider Konfigurationen.

Antwortzeit	Nicht instrumentiert	Instrumentiert
Minimal	5 ms	6 ms
Maximal	38 ms	44 ms
Durchschnittlich	12 ms	12 ms
Median	10 ms	10 ms
Standardabweichung	6.36	6.17

Tabelle 10: Antwortzeiten instrumentierter versus nicht instrumentierter Host

Auch die in der Tabelle erhobenen Daten zeigen keine signifikanten Abweichungen auf. Lediglich leicht erhöhte Minimal- und Maximalantwortzeiten werden festgestellt, welche jedoch in der Summe der Abfragen keine Signifikanz darstellen, wie die äquivalenten Durchschnitts- und Medianwerte aufzeigen.

Dass zwischen den instrumentierten und nicht instrumentierten Systemen keine signifikante Veränderung der Antwortzeiten festzustellen ist, ist plausibel. Wie in Kapitel 3.3.2 beschrieben, werden Spans asynchron und nach Versand einer Antwort verschickt. Somit erzeugen diese kaum Verzögerungen im antwortenden Prozess. Lediglich aufgrund der zusätzlichen Operationszeit zur Spanerzeugung entstehen Verzögerungen bei der Erzeugung einer Antwort auf eine Anfrage. Diese sind jedoch so gering, dass sie mit den gewählten Messmethoden nicht sichtbar werden. Aufgrund der ohnehin schwankenden und vergleichsweise hohen Latenzen in verteilten Systemen haben jene im Mikrosekundenbereich liegenden zusätzlichen Verzögerungen keine Relevanz.

Auswertung der Ressourcennutzung

Neben der Performance bei der Beantwortung von Anfragen sind die Ressourcenanforderungen einer verteilten Applikation relevant, da sie einen Kostenfaktor zum Betrieb der Architektur darstellen. Daher werden die Microservices im Rahmen der beschriebenen Testpläne auf Differenzen in der Ressourcennutzung analysiert. Basis der nachfolgenden Analysen sind die aufgezeichneten Daten des Dynatrace-Monitorings. Im Nachfolgenden werden die aufgezeichneten Prozesses-Metriken der Prozesse Business-Core-Service und Zuul-Service ausgewertet.

Abbildung 51 zeigt die CPU-Auslastung im zeitlichen Verlauf der Testphasen. Während in dem Graphen des nicht instrumentierten Systems zu sehen ist, wie die Last innerhalb des Testzeitraums (12:19-12:22) ca. 10 % beträgt, ist im Graphen der instrumentierten Services ein ansteigender Verlauf zu erkennen.

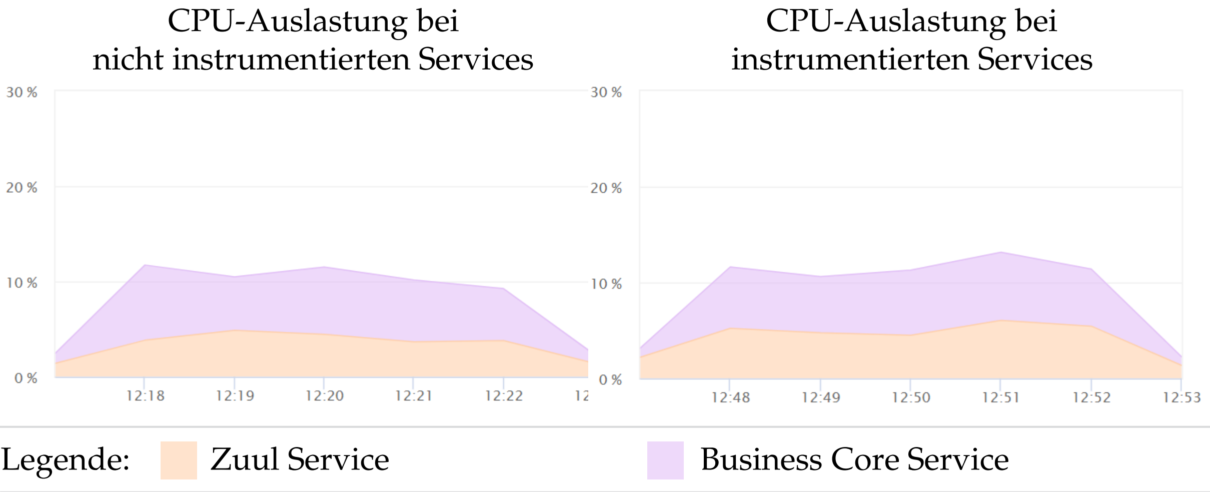


Abbildung 51: CPU-Auslastung durch Services während der Testzeiträume

Die CPU-Last beginnt etwa auf gleichem Niveau wie im nicht instrumentierten Zustand, nimmt anschließend jedoch stetig zu. Die in Tabelle 11 aufgezeigten Daten zeigen diesen Trend noch deutlicher. Die Tabelle zeigt besonders für den Zuul-Service eine erhöhte CPU-Last in der direkten Gegenüberstellung. Da die Instrumentierung konstante Last für jeden zu erzeugenden Span produziert, ist diese bei dem ohnehin weniger Last produzierenden Zuul-Service schwerer gewichtet.

	Minute 1	**Minute 2**	**Minute 3**	**Durchschnitt**
Nicht instrumentierte Services				
Business-Core-Service	7.06	6.50	5.44	6.30
Zuul-Service	4.45	3.66	3.80	3.97
Gesamt	11.51	10.16	9.24	10.30
Instrumentierter Services				
Business-Core-Service	5.87	6.79	7.12	6.59
Zuul-Service	4.75	4.50	6.05	5.10
Gesamt	10.62	11.29	13.17	11.69

Tabelle 11: CPU-Auslastung (in %) durch Services während der Testzeiträume

Der sich durchschnittlich ergebende CPU-Wert von 10,30 % bzw. 11,69 % im Fall der Instrumentierung zeigt einen relativ gestiegenen Verbrauch von 13,5 % an. Dies ent-

spricht einem Anstieg von 3,4 % je zu erzeugendem Span innerhalb einer Anfrageverarbeitung. Diese geringen Mehrkosten sind unter Berücksichtigung der Ergebnisse der Analyse von Antwortzeiten in einem erwarteten Bereich. Absolut betrachtet bewirkt die Instrumentierung lediglich 1 % zusätzliche CPU-Last.

Ein weiterer wichtiger Faktor einer verteilten Architektur ist die Netzwerklast, welche durch einen Service erzeugt wird. Die Bandbreite der vorhandenen Infrastruktur ist limitiert. Somit bedeutet eine höhere Netzwerklast Investitionen in die Infrastruktur. Daher wird die zusätzliche Last durch die Erzeugung von Spans analysiert.

Die durch das Dynatrace-Monitoring erhobenen Messungen ergeben, dass durch den Zuul-Service für Empfang und Beantwortung der eintreffenden Anfragen im nicht instrumentierten Zustand etwa 160 kB/s an Bandbreite benötigt werden. Wie aus Abbildung 52 ersichtlich, bleibt dieser Wert während des Testzeitraums (12:19–12:22) nahezu konstant, was der Konstanz in Menge und Art der Anfragen entspricht. Die durch den Business-Core-Service erzeugte und empfangene Netzwerklast umfasst lediglich den Datenaustausch zwischen den Services, welcher unter 1 kB/s beträgt und somit mit der verwendeten Skala nicht sichtbar wird. Ein anderes Bild ergibt sich für den instrumentierten Graphen. Hier ist der Business-Core-Service trotz der Verwendung einer reduzierten Skala deutlich sichtbar.

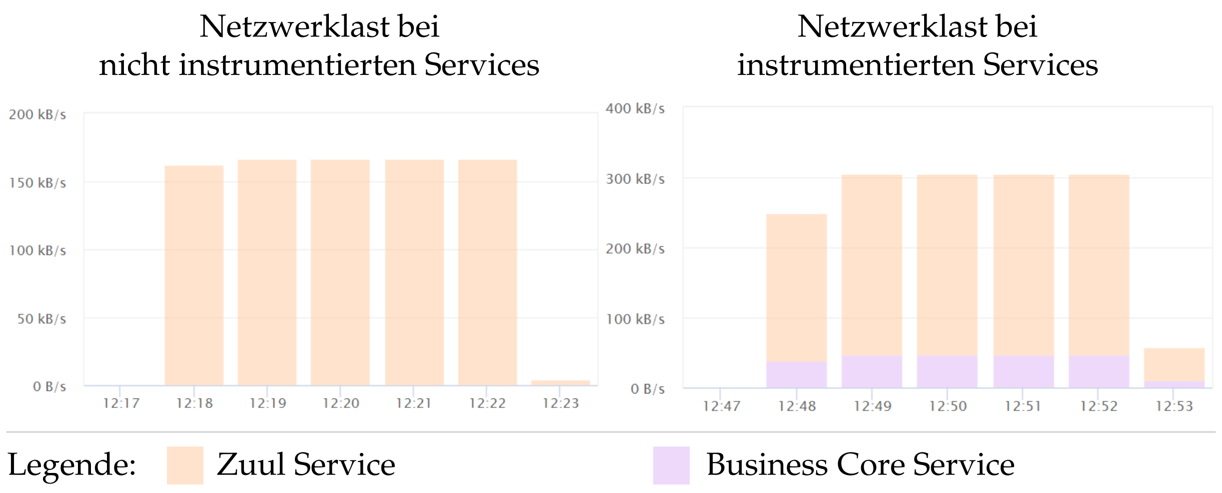


Abbildung 52: Netzwerklast (in kB/s) durch Services während der Testzeiträume

Eine detaillierte Analyse der in Tabelle 12 geführten Werte zeigt auf, dass sich die Netzwerklast im Vergleich der nicht instrumentierten und der instrumentierten Umgebung mit einem Zuwachs von 138,9 kB/s nahezu verdoppelt.

	Minute 1	**Minute 2**	**Minute 3**	**Durchschnitt**
Nicht instrumentierte Services				
Zuul-Service	166.00	166.00	166.00	166.00
Business-Core-Service	0.60	0.60	0.60	0.60
Gesamt	166.60	166.60	166.60	166.60
Instrumentierter Services				
Zuul-Service	259.00	259.00	258.00	258.67
Business-Core-Service	46.90	46.90	46.80	46.87
Gesamt	305.90	305.90	304.80	305.53

Tabelle 12: Netzwerklast (in kB/s) durch Services während der Testzeiträume

Vergleicht man die durchschnittliche Netzwerklast des Business-Core-Service mit und ohne Instrumentierung, so kann man feststellen, dass diese von 0,60 kB/s um mehr als 780 % auf 46,87 kB/s anstiegen. Der Anstieg des Zuul-Service umfasst lediglich 55 %. Der Gesamtanstieg liegt bei 83 %. Diese unterschiedlichen Anstiege sind darauf zurückzuführen, dass die Größe eines Spans nahezu konstant ist, im relativen Vergleich jedoch umso schwerer ins Gewicht fällt, umso geringer die Datenmenge der zugrundeliegenden Anfrage ist. Da durch den Business-Core-Service lediglich um ein vielfaches kleinere Datenmengen ausgetauscht werden, als ein Span umfasst, besteht in diesem Vergleich eine besonders schlechte Bilanz.

Mit einer um 88 % erhöhten Netzwerklast ist die Instrumentierung als schwerwiegender Faktor dieser Metrik zu berücksichtigen. Durch die Konfiguration einer Sampling-Rate kann die Instrumentierung jedoch angepasst werden, um eine reduzierte Menge an Spans zu erzeugen.

Die durchgeführten Performance- und Ressourcenanalysen zeigen, dass die für den Architekturerkennungsservice notwendige instrumentierte Microservice-Architektur ohne Performanceeinbußen und mit geringer zusätzlicher CPU-Last betrieben wer-

den kann. Außerdem wird sichtbar, dass die durch den zusätzlichen Traffic erzeugte Netzwerklast nicht unerheblich ist und somit bei der Konfiguration der Instrumentierung ein Augenmerk auf die Menge zu sammelnder Daten sowie die Rate der zu erzeugenden Spans gelegt werden sollte, um diese Last zu reduzieren.

6.3 Limitierungen

Die prototypische Implementierung ist aufgrund konzeptueller Entscheidungen sowie technischer Abhängigkeiten in ihrem Einsatzumfeld und den erzielenden Ergebnissen limitiert. Dieses Kapitel dient der Benennung und Erläuterung jener Limitierungen.

Technische Abhängigkeit zu Architekturkomponenten

Zur korrekten Funktionsweise des Architekturerkennungsdienstes sind mehrere Komponenten zwingend erforderlich. Dies umfasst eine für das Distributed Tracing notwendige Instrumentierung aller Microservices. Diese muss Spans erzeugen, welche durch einen Zipkin-Server verarbeitet werden können. Des Weiteren benötigt der Erkennungsservice das Service Repository Eureka zur Erkennung entfallener Architekturkomponenten. Alle Microservices müssen mit einem Eureka-Client ausgestattet sein und ihren Status an einen Eureka-Server kommunizieren. Weitere Informationen zu den technischen Abhängigkeiten sind in Kapitel 5.1 nachzulesen.

Metainformationen zu Relationen, Revisionen und Komponenten

Durch den Prototyp werden Metainformationen in Form von Annotationen für alle erkannten Komponenten einer Architektur gesammelt. Durch den Prototyp ist jedoch nicht gesichert, dass eine dieser Annotationen verfügbar ist. Die Verfügbarkeit ist dadurch bedingt, ob sie durch die Instrumentierung der Microservice-Architektur erhoben wird. Da durch den Prototyp jegliche von der Instrumentierung gelieferte Zusatzinformation abgelegt wird, bietet dies jedoch auch die Möglichkeit, die Menge an Metainformationen durch eine Rekonfiguration der Instrumentierung zu verändern, ohne Anpassungen an dem Prototyp vornehmen zu müssen.

Asynchrone Verbindungen und Relationen

Eine im Rahmen dieser Arbeit adressierte Forschungsfrage betrifft die Erkennung synchroner und asynchroner Abhängigkeiten zwischen Microservices. Der Microser-

vice ist in der Lage, asynchrone Verbindungen auszuweisen, insofern diese durch die Instrumentierung der Clients aufgezeichnet werden. Dies erfordert, dass die Instrumentierung die Erzeugung neuer Threads durch die Erzeugung eines separaten Spans dokumentiert, welcher als Elternelement eines nachfolgenden Requests dient. Erfolgt die Dokumentation einer asynchronen Kommunikation auf diese Weise, wird die Asynchronität durch den Prototyp als Annotation der Relation dokumentiert. Dies ist z. B. durch die in der Evaluation verwendete Instrumentierung Spring Sleuth gegeben. Erfolgt keine Dokumentation der Asynchronität oder erfolgt diese auf eine andere als die zuvor beschriebene Weise, so werden asynchrone Verbindungen nicht entsprechend erkannt und gekennzeichnet.

Erkennung von Architekturkomponenten wie Datenbanken

Der Prototyp ist nicht in der Lage, Architekturkomponenten zu erfassen, welche nicht durch das Distributed Tracing oder das Vorhandensein in Eureka dokumentiert werden. Dies umfasst im Besonderen jegliche Komponenten, welche über andere Schnittstellen als eine REST-API kommunizieren, z. B. Datenbanken, Streams und Dateiserver sowie Abhängigkeiten zu diesen. Durch eine Erweiterung von Instrumentierungen und des Prototyps könnten auch diese Komponenten künftig erkannt werden.

Revisionserkennung bei neuen Softwareversionen

Die prototypische Implementierung basiert auf dem Konzept, dass eine Softwareänderung eines Microservice zu Beziehungsänderungen führen kann. Daher wird mit jeder Softwareänderung eine neue Revision jenes Microservice im Architekturmodell erzeugt. Jede Revision kann ihre eigenen Beziehungen pflegen, wodurch Revisionen zeitlich unterschiedliche Zustände eines Service darstellen können. In seiner derzeitigen Implementierung fehlt dem Prototyp eine Erkennung der Softwareversion. Mit jedem Neustart eines Service wird eine neue Revision angelegt. Dies unterliegt der Annahme, dass eine neue Softwareversion einen Serviceneustart bedingt. Dies führt jedoch dazu, dass häufig neue Revisionen erzeugt werden, obwohl ein Service lediglich neu gestartet, jedoch seine Software nicht verändert wird. Die derzeitige Revisionierung ist somit kein zuverlässiger Indikator für Softwareänderungen der zur Revision gehörenden Komponente.

Relationen sind an Revisionen gebunden

Mit der Erzeugung einer neuen Revision werden alle bisherigen Relationen als unsicher eingestuft und aus dem Architekturmodell entfernt. Dieses Konzept verfolgt das Ziel, dass in vorherigen Softwareversionen eines Microservice bestehende Relationen möglicherweise nicht mehr existieren und somit aus dem Architekturmodell entfernt werden müssen. Da jedoch der Entfall einer Relation nicht erkannt werden kann, werden jegliche Relationen der betroffenen Revision bis zur erneuten Beobachtung entfernt. Dies bedeutet, dass nach dem Neustart eines Service (neue Revision) jegliche Verbindungen neu erkannt werden müssen und somit das Architekturmodell nur mit mittelfristiger Laufzeit und ohne hochfrequentierte Neustarts eines Microservice vollständig ist. Besonders in automatisierten Umgebungen, in welchen Hosts je nach Last gestartet und beendet werden, führt dies zu einer unvollständigen Abhängigkeitserkennung.

Erkennung virtueller Hosts sowie Beziehungen auf Hardware-Ebene

Der Prototyp erkennt Hardware-Komponenten in seiner derzeitigen Form lediglich anhand ihrer IP-Adresse. Es ist jedoch in der Praxis häufig anzutreffen, dass ein Host über mehrere IP-Adressen verfügt. Der Architekturerkennungsservice sollte erkennen, dass jene IP-Adressen zu einer gemeinsamen Hardware gehören und Abhängigkeiten korrekt ermitteln. Im aktuellen Zustand hingegen würden durch den Prototyp zwei Hosts ermittelt werden, was nicht der Realität entspricht. Des Weiteren können Hosts, welche in einer virtualisierten Umgebung erzeugt und betrieben werden, nicht als solche erkannt werden. Dies führt dazu, dass auch ihre Abhängigkeit vom Hostsystem, auf welchem die virtuellen Maschinen (VM) betrieben werden, nicht in dem Architekturmodell abgebildet ist.

Modellierung des Business-Layers

Der Architekturerkennungsdienst bietet API-Endpunkte, um die Elemente des Business-Layers mit ihren Relationen zum Architekturmodell hinzuzufügen. Diese Endpunkte werden durch die Weboberfläche der prototypischen Implementierung im Rahmen der Prozessmodellierung verwendet. Das somit erweiterte Architekturmodell stellt also eine Vermischung von Soll-Daten (Business-Layer) und aus Analysen gewonnenen Ist-Daten (Application- und Technology-Layer) dar. Die Modellierung sollte durch einen technischen Vorgang wie das Process Mining ersetzt werden.

7. Fazit und Ausblick

Mit der prototypischen Implementierung kann ein System realisiert werden, welches ein umfassendes Architekturmodell der technischen und der Businessebene zusammenführt und Beziehungen zwischen den Ebenen aufzeigt. Das in Echtzeit bereitgestellte Modell erlaubt Modellanalysen, welche durch die erzeugten Architekturmodelle bisheriger Lösungen nicht möglich waren und die Aufschluss erlauben über Einflüsse technischer Komponenten auf Geschäftsaktivitäten entsprechend dem Ist-Zustand einer Architektur. Durch den Prototyp kann außerdem aufgezeigt werden, dass die Ermittlung technischer Entitäten ausschließlich auf Basis des Distributed Tracing und in Echtzeit möglich ist. Entitäten des Application- sowie des Technology-Layers können ermittelt und Intralayer sowie Interlayer-Beziehungen erkannt werden. Durch eine Weboberfläche kann das Architekturmodell um Businessentitäten erweitert und teilautomatisiert ein Regelwerk zur Definition von Abhängigkeiten zwischen technischen und Geschäftsentitäten erzeugt werden. Durch Streaming-Technologien können Events der Architekturerkennung nicht nur in Echtzeit ermittelt, sondern ebenfalls in Echtzeit an konsumierende Applikationen weitergegeben werden. Dass durch das Distributed Tracing der Entfall einer Architekturentität nicht erkennbar ist, kann durch die Datenzusammenführung mit Informationen des Service Repository Eureka wettgemacht werden, wodurch nicht nur eine Architektur erkannt werden kann, sondern auch Veränderungen einzelner Entitäten entdeckt werden können. Durch einen neuen Ansatz der Visualisierung in Form einer modifizierten Adjacency-Matrix ist es gelungen, eine Repräsentationsmöglichkeit zu finden, welche die Visualisierung sowohl großer als auch kleiner Architekturen ermöglicht.

Es kann im Rahmen der Evaluation aufgezeigt werden, dass das eingesetzte Monitoring-Werkzeug der Host-Instrumentierung für Distributed Tracing lediglich geringfügige Mehrkosten erzeugt und somit einen praktischen Einsatz ermöglicht. Lediglich die Datenmenge, welche innerhalb der Infrastruktur transportiert wird, steigt signifikant und erfordert weitere Optimierungen. Besonders bei hochfrequentierten Services, welche geringe Datenmengen austauschen, ist der relative Overhead deutlich.

Der Prototyp unterliegt einer Reihe von Limitierungen, welche zu einem großen Teil nicht konzeptueller Natur, sondern dem verfügbaren Zeitkontingent für Implemen-

tierungen im Rahmen dieser Arbeit geschuldet sind und durch eine Erweiterung aufgelöst werden können. Die Ermittlung des Entfalls von Beziehungen während der Versionsänderung eines Service kann jedoch nicht befriedigend gelöst werden, da das Konzept den Verlust aller in der Vergangenheit erkannten Beziehungen bis zu einer Neuentdeckung vorsieht. Das Konzept basiert auf der Annahme, dass Services zumindest mittelfristig betrieben werden, um ein vollständiges Architekturmodell zu erzeugen. Besonders in hochfrequent veränderlichen Architekturen, wie jenen mit lastabhängigem automatisiertem Deployment, führt dies zu einem oft unvollständigen Modell. Durch die Verbesserung des Revisionserkennungsalgorithmus können die Auswirkungen dieser Limitierung jedoch stark reduziert werden.

Auf jede der in Kapitel 1 beschriebenen Forschungsfragen können Antworten gefunden werden, welche nachfolgend kurz zusammengefasst werden.

Wie kann eine Microservice-Architektur durch Anwendung von Distributed Tracing ermittelt werden?

Durch die Auswertung adressierter Endpunkte einzelner Spans im Kontext eines Traces können Architekturentitäten sowie deren Abhängigkeiten erkannt werden. Entsprechend des in Kapitel 5.4.3 vorgestellten Algorithmus können jene Informationen zu einem Architekturmodell zusammengeführt werden.

Welche Arten von Beziehungen bestehen zwischen Architekturkomponenten und wie können diese automatisch erkannt werden?

Es bestehen Beziehungen zwischen Komponenten des gleichen EA-Layers, welche als Intralayer-Beziehungen bezeichnet werden. Diese können aufgrund des Verlaufs eines Trace auf Applikationsebene automatisiert erkannt werden (siehe Kapitel 5.2.1). Auf Ebene des Business-Layers erfolgt eine Modellierung dieser Beziehungen. Durch eine Erweiterung mit der Technologie des Business Process Mining ist eine automatisierte Lösung denkbar.

Des Weiteren bestehen ebenenübergreifende Beziehungen von einer übergeordneten zu einer untergeordneten Ebene, welche als Interlayer-Beziehungen bezeichnet werden. Zwischen Business- und Application-Layer erfolgt die automatische Erkennung auf Basis eines Regelwerks (siehe Kapitel 5.2.2), zwischen Application- und Techno-

logy-Layer ist eine Erkennung durch Identifikatoren wie IP-Adresse und Port eines Service möglich.

Außerdem können Relationen in direkter und indirekter Abhängigkeit unterschieden werden (siehe Kapitel 5.3.3).

Wie können Nebenläufigkeit und Synchronität erkannt werden?

Durch die Instrumentierung des Distributed Tracing können nebenläufige Methodenaufrufe dokumentiert werden. Das Architekturerkennungssystem wertet diese aus und kann auf dieser Basis die Nebenläufigkeit oder Synchronität einer Kommunikation bestimmen (siehe Kapitel 3.3.3).

Wie können Veränderungen der Microservice-Architektur entdeckt werden?

Das Hinzukommen von Architekturentitäten kann sowohl durch die Analyse des Distributed Tracing sowie durch Analyse des Service Repository erkannt werden. Intralayer-Beziehungen werden aus dem Trace-Verlauf des Distributed Tracing ermittelt. Entfallene Entitäten werden durch einen Modellabgleich mit den Informationen des Service Repository sichtbar (siehe Kapitel 5.2.1, 5.4.3, 5.4.4).

Wie kann ein smartes User Interface bereitgestellt werden, um Business-Services mit Business-Semantik zu erweitern?

Die in Kapitel 5.2.2 vorgestellte Webanwendung zeigt unter Verwendung der in Kapitel 5.2.1 vorgestellten API auf, wie durch die Kombination eines Modellierungstools für Geschäftsprozesse sowie eine Oberfläche zur Regelerzeugung und Verwaltung Business-Semantik zum Architekturmodell hinzugefügt werden können.

Die prototypische Implementierung schafft neue Transparenz für den Anwender. Ausblickend verfügt sie jedoch darüber hinaus über weiteres Potential als Grundlage für darauf aufbauende Systeme und Analysen.

Das Architekturmodell ist derart konzipiert, dass es Zusatzinformationen an Komponenten, Revisionen und Beziehungen zulässt. Somit können Performanceanalysen bezüglich dieser Entitäten durchgeführt und direkt im Architekturmodell auf Basis von Annotationen manifestiert werden.

Das Architekturmodell ermöglicht ebenfalls eine Ursachenanalyse beginnend bei der Feststellung einer Störung einer Geschäftsaktivität bis hin zum ausführenden Service und dessen Host.

Neben der Ursachenanalyse können mit dem umfassenden Architekturmodell außerdem Vorhersagen von Einflüssen und Risiken durch technische Komponenten auf Entitäten der Geschäftsebene getroffen werden, was Kostenanalysen ermöglicht und im Entscheidungsprozess für Skalierungsstrategien als zusätzlicher Input dienen kann.

Durch seine weitreichende API und die Verwendung der Streaming-Technologien zur Kommunikationsänderung ergeben sich diverse neue Möglichkeiten des Reportings und der Datenanalyse, basierend auf den durch den Architekturerkennungsservice ermittelten und bereitgestellten Daten.

Literaturverzeichnis

Alonso, I. A., Verdún, J. C. & Caro, E. T. (2010, 13-15 Dec. 2010). *The IT implicated within the enterprise architecture model: Analysis of architecture models and focus IT architecture domain.* Paper presented at the 2010 IEEE International Conference on Service-Oriented Computing and Applications (SOCA).

Anne Lapkin, P. A., Brian Burke, Betsy Burton, R. Scott Bittler, Robert A. Handler, Greta A. James, Bruce Robertson, David Newman, Deborah Weiss, Richard Buchanan, Nicholas Gall (2008). Gartner Clarifies the Definition of the Term 'Enterprise Architecture'.

Apache Foundation. (2017a). Abgerufen am 05.09.2017 18:37 von https://kafka.apache.org/

Apache Foundation. (2017b). Abgerufen am 05.09.2017 19:07 von https://cwiki.apache.org/confluence/display/KAFKA/Clients

Apache Software Foundation. (2017). Apache JMeter. Abgerufen am 21.10.2017 18:23 von http://jmeter.apache.org/

Bakshi, K. (2017, 4-11 March 2017). *Microservices-based software architecture and approaches.* Paper presented at the 2017 IEEE Aerospace Conference.

BlazeMeter. (2017). BlazeMeter Sense. Abgerufen am 22.10.2017 13:04 von https://sense.blazemeter.com/

Bootstrap. (2017). Bootstrap - The most popular HTML, CSS, and JS library in the world. Abgerufen am 19.09.2017 18:34 von http://getbootstrap.com/

Burrows, B. H. S. L. A. B. M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S. & Shanbhag, C. (2010). Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. *Google Technical Report dapper-2010-1, April 2010.*

Celonis SE. (2017). Abgerufen am 27.10.2017 13:56 von https://www.celonis.com

client IO s.r.o. (2017). Rappid Diagramming Framework. Abgerufen am 19.09.2017 von https://www.jointjs.com/

Dynatrace LLC. (2017a). Digital Performance Management und Application Performance Monitoring (APM). Abgerufen am 21.10.2017 18:15 von https://www.dynatrace.de/

Dynatrace LLC. (2017b). Erkennung und Visualisierung von Anwendungstopologien. Abgerufen am 27.10.2017 16:39 von https://www.dynatrace.de/kompetenzen/application-topology-discovery/

Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software*: Addison-Wesley.

GitHub Inc. (2017). GitHub. Abgerufen am 26.10.2017 14:28 von https://github.com/

Goolge Inc. (2017). Angular. Abgerufen am 19.09.2017 18:26 von https://angular.io/

Granchelli, G., Cardarelli, M., Francesco, P. D., Malavolta, I., Iovino, L. & Salle, A. D. (2017a, 5-7 April 2017). *MicroART: A Software Architecture Recovery Tool for Maintaining Microservice-Based Systems.* Paper presented at the 2017 IEEE International Conference on Software Architecture Workshops (ICSAW).

Granchelli, G., Cardarelli, M., Francesco, P. D., Malavolta, I., Iovino, L. & Salle, A. D. (2017b, 5-7 April 2017). *Towards Recovering the Software Architecture of Microservice-Based Systems.* Paper presented at the 2017 IEEE International Conference on Software Architecture Workshops (ICSAW).

Haki, M. K. & Forte, M. W. (2010, 5-10 July 2010). *Service Oriented Enterprise Architecture Framework.* Paper presented at the 2010 6th World Congress on Services.

Hall, S. (2016). OpenTracing Aims for a Clearer View of Processes in Distributed Systems. Abgerufen am 06.10.2017 20:40 von

HashiCorp. (2017). Consul by HashiCorp. Abgerufen am 29.10.2017 16:00 von https://www.consul.io/

Hasselbring, W. (2017). Architecture Discovery | Kieker. Abgerufen am 27.10.2017 16:20 von http://kieker-monitoring.net/architecture-discovery/

IEEE. (2011). ISO/IEC/IEEE Systems and software engineering -- Architecture description. *ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000)*, 1-46. doi:10.1109/IEEESTD.2011.6129467

Jonkers, H., Burren, R. v., Arbab, F., Boer, F. d., Bonsangue, M., Bosma, H., . . . Zanten, G. V. v. (2003, 16-19 Sept. 2003). *Towards a language for coherent enterprise architecture descriptions.* Paper presented at the Seventh IEEE International Enterprise Distributed Object Computing Conference, 2003. Proceedings.

Kong inc. (2017). King - Open-Source API Management and Microservice Management. Abgerufen am 29.10.2017 17:51 von https://getkong.org/

Martincevic, N. (2016). DDD: Context is King – Kein Context, keine Microservices. Abgerufen am 12.10.2017 12:07 von https://www.informatik-aktuell.de/entwicklung/methoden/ddd-context-is-king-kein-context-keine-microservices.html

Microsoft. (2017). Abgerufen am 05.09.2017 20:14 von https://github.com/Microsoft/kafka-proxy-ws

Netflix inc. (2017). Zuul Gateway Service. Abgerufen am 29.10.2017 17:47 von https://github.com/Netflix/zuul

Netflix Inc. (2017a). Eureka. Abgerufen am 05.10.2017 19:25 von https://github.com/Netflix/eureka

Netflix Inc. (2017b). Eureka at a glance. Abgerufen am 30.08.2017 17:35 von https://github.com/Netflix/eureka/wiki/Eureka-at-a-glance

Netflix Inc. (2017c). Eureka REST operations. Abgerufen am 30.08.2017 20:07 von https://github.com/Netflix/eureka/wiki/Eureka-REST-operations

Newman, S. (2015). *Building Microservices*: O'REILLY.

OpenTracing. (2017). Introduction into OpenTracing. Abgerufen am 06.10.2017 21:20 von http://opentracing.io/documentation/

Pivotal Software Inc. (2017a). Spring Cloud. Abgerufen am 23.10.2017 von https://cloud.spring.io

Pivotal Software Inc. (2017b). Spring Cloud Sleuth. Abgerufen am 29.10.2017 21:02 von https://github.com/spring-cloud/spring-cloud-sleuth

Probst, K. & Becker, J. (2016). Engineering Trade-Offs and The Netflix API Re-Architecture. Abgerufen am 05.10.2017 18:35 von https://medium.com/netflix-techblog/engineering-trade-offs-and-the-netflix-api-re-architecture-64f122b277dd

Process Mining Group. (2017). ProM Tools. Abgerufen am 27.10.2018 13:53 von http://www.promtools.org

Reinhold, E. (2016). Lessons Learned on Uber's Journey into Microservices. Abgerufen am 05.10.2017 von https://www.infoq.com/presentations/uber-darwin

Rotter, C., Illés, J., Nyíri, G., Farkas, L., Csatári, G. & Huszty, G. (2017, 7-9 March 2017). *Telecom strategies for service discovery in microservice environments.* Paper presented at the 2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN).

Saat, J., Aier, S. & Gleichauf, B. (2009). *Assessing the Complexity of Dynamics in Enterprise Architecture Planning – Lessons from Chaos Theory*. Paper presented at the AMCIS 2009 Proceedings. 808.

Schaefer, R. (2016). From Monolith to Microservices at Zalando. Abgerufen am 05.10.2017 18:25 von https://jobs.zalando.com/tech/blog/goto-2016-from-monolith-to-microservices

Shah, H. & Kourdi, M. E. (2007). Frameworks for Enterprise Architecture. *IT Professional, 9*(5), 36-41. doi:10.1109/MITP.2007.86

SmartBear Software. (2017a). *World's Most Popular API Framework | Swagger.* Abgerufen am 14.09.2017 19:40 von https://swagger.io/

SmartBear Software. (2017b). Swagger API Registry. Abgerufen am 29.10.2017 18:08 von https://smartbear.com/product/swaggerhub/features/browse/

The Apache Software Foundation. (2017). Apache HTrace. Abgerufen am 26.10.2017 19:34 von http://htrace.incubator.apache.org/

The OpenZipkin Authors. (2017a). Architecture of OpenZipkin. Abgerufen am 06.10.2017 22:00 von http://zipkin.io/pages/architecture.html

The OpenZipkin Authors. (2017b). Existing Instrumentations of OpenZipkin. Abgerufen am 06.10.2017 22:15 von http://zipkin.io/pages/existing_instrumentations

The OpenZipkin Authors. (2017c). OpenZipkin - A distributed tracing System. Abgerufen am 07.10.2017 03:40 von http://zipkin.io/

The OpenZipkin Authors. (2017d). "zipkin-server" Repository at Github.com. Abgerufen am 06.09.2017 15:50 von https://github.com/openzipkin/zipkin/tree/master/zipkin-server

The OpenZipkin Authors. (2017e). Zipkin Data Model. Abgerufen am 01.09.2017 19:13 von http://zipkin.io/pages/data_model.html

van der Aalst, W. (2011). Process Discovery: An Introduction *Process Mining: Discovery, Conformance and Enhancement of Business Processes* (pp. 125-156). Berlin, Heidelberg: Springer Berlin Heidelberg.

van der Aalst, W. (2015). Extracting Event Data from Databases to Unleash Process Mining. In J. vom Brocke & T. Schmiedel (Eds.), *BPM - Driving Innovation in a Digital World* (pp. 105-128). Cham: Springer International Publishing.

van der Aalst, W., Adriansyah, A., de Medeiros, A. K. A., Arcieri, F., Baier, T., Blickle, T., . . . Wynn, M. (2012). Process Mining Manifesto. In F. Daniel, K. Barkaoui, & S. Dustdar (Eds.), *Business Process Management Workshops: BPM 2011 International Workshops, Clermont-Ferrand, France, August 29, 2011, Revised Selected Papers, Part I* (pp. 169-194). Berlin, Heidelberg: Springer Berlin Heidelberg.

van Hoorn, A., Rohr, M., Hasselbring, W., Waller, J., Ehlers, J., Frey, S. & Kieselhorst, D. (2009). Continuous Monitoring of Software Services: Design and Application of the Kieker Framework.

van Hoorn, A., Waller, J. & Hasselbring, W. (2012). *Kieker: a framework for application performance monitoring and dynamic software analysis*. Paper presented at the Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering, Boston, Massachusetts, USA.

Wagner, T. (2015). Microservices without the Servers. Abgerufen am 05.10.2017 18:40 von https://aws.amazon.com/de/blogs/compute/microservices-without-the-servers/

Winter, R. & Fischer, R. (2006, 16-20 Oct. 2006). *Essential Layers, Artifacts, and Dependencies of Enterprise Architecture.* Paper presented at the 2006 10th IEEE International Enterprise Distributed Object Computing Conference Workshops (EDOCW'06).

Wolff, E. (2016). *Microservices | Grundlagen flexibler Softwarearchitekturen*: dpunkt.verlag.

X-Trace. (2017). Abgerufen am 26.10.2017 19:37 von http://www.x-trace.net

Yale, Y., Silveira, H. & Sundaram, M. (2016, 3-5 Oct. 2016). *A microservice based reference architecture model in the context of enterprise architecture.* Paper presented at the 2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC).

Anhang A: XML-Response von /eureka/v2/apps

Folgende XML-Response stellt beispielhaft die zur Verfügung gestellten Daten durch den Endpunkt „GET /eureka/v2/apps“ dar. Sind mehrere Services (hier „application“ vorhanden, wiederholt sich der <application>-Tag.


```xml
<applications>
   <versions__delta>1</versions__delta>
   <apps__hashcode>UP_8_</apps__hashcode>
   <application>
      <name>DEUTSCHEBAHN-MOBILITY-SERVICE</name>
      <instance>
         <instanceId>
            PC192-168-2-50:deutschebahn-mobility-service:6003
         </instanceId>
         <hostName>PC192-168-2-50</hostName>
         <app>DEUTSCHEBAHN-MOBILITY-SERVICE</app>
         <ipAddr>192.168.2.50</ipAddr>
         <status>UP</status>
         <overriddenstatus>UNKNOWN</overriddenstatus>
         <port enabled="true">6003</port>
         <securePort enabled="false">443</securePort>
         <countryId>1</countryId>
         <dataCenterInfo>
            <name>MyOwn</name>
         </dataCenterInfo>
         <leaseInfo>
            <renewalIntervalInSecs>30</renewalIntervalInSecs>
            <durationInSecs>90</durationInSecs>
            <registrationTimestamp>1504108525643</registrationTimestamp>
            <lastRenewalTimestamp>1504113896069</lastRenewalTimestamp>
            <evictionTimestamp>0</evictionTimestamp>
            <serviceUpTimestamp>1504108525133</serviceUpTimestamp>
         </leaseInfo>
         <metadata>
            <appType>mobility-service</appType>
         </metadata>
         <homePageUrl>http://PC192-168-2-50:6003/</homePageUrl>
         <statusPageUrl>http://PC192-168-2-50:6003/info</statusPageUrl>
         <healthCheckUrl>http://PC192-168-2-50:6003/health</healthCheckUrl>
         <vipAddress>deutschebahn-mobility-service</vipAddress>
         <secureVipAddress>deutschebahn-mobility-service</secureVipAddress>
         <isCoordinatingDiscoveryServer>false</isCoordinatingDiscoveryServer>
         <lastUpdatedTimestamp>1504108525643</lastUpdatedTimestamp>
         <lastDirtyTimestamp>1504108524972</lastDirtyTimestamp>
         <actionType>ADDED</actionType>
      </instance>
   </application>
</applications>
```

Anhang B: Json-Daten eines Zipkin Spans

Folgendes Json-Modell stellt beispielhaft die Datenstruktur eines Zipkin Spans dar. Quelle: (The OpenZipkin Authors, 2017e),


```
{
    "traceId": "bd7a977555f6b982",
    "name": "get",
    "id": "bd7a977555f6b982",
    "timestamp": 1458702548467000,
    "duration": 386000,
    "annotations": [
        {
            "endpoint": {
                "serviceName": "zipkin-query",
                "ipv4": "192.168.1.2",
                "port": 9411
            },
            "timestamp": 1458702548467000,
            "value": "sr"
        },
        {
            "endpoint": {
                "serviceName": "zipkin-query",
                "ipv4": "192.168.1.2",
                "port": 9411
            },
            "timestamp": 1458702548853000,
            "value": "ss"
        }
    ],
    "binaryAnnotations": []
}
```

Anhang C: Exemplarische Antwort des Architektur-Endpunkts

Folgendes Json-Modell ist exemplarisch für die Antwort einer Anfrage an den Endpunkt: „GET api/v1/ad/architecture“


```json
{
    "snapshot": 1504978986792,
    "components": {
        "849": {
            "component": {
                "id": 637,
                "type": "Service",
                "name": "MAPS-HELPER-SERVICE"
            },
            "child-relations": [
                {
                    "callee": 905,
                    "annotations": {
                        "ad.discovered_at": "1504975120954"
                    }
                }
            ],
            "parent-relations": [
                {
                    "caller": 856,
                    "annotations": {
                        "http.path": "/distance",
                        "http.url": "http://localhost:7000/distance",
                        "ad.discovered_at": "1502749164309",
                        "http.method": "GET",
                        "http.host": "localhost",
                    }
                }
            ]
        },
    ...
    },
    "root-components": [
        869,
        855
    ]
}
```

Anhang D: E-R Diagramm des Architekturerkennsungsservice