Service-Registry

Eine zentrale Komponente in einer Microservice-Architektur

Pascal Giessler, Danny Forschner, Sebastian Abeck

Eine Microservice-Architektur umfasst viele kleine Services, eine Service-Registry ermöglicht die Service-Discovery, deren Aufgabe es ist, Services innerhalb einer Servicelandschaft auffindbar zu machen und somit die Wiederverwendbarkeit der Services zu fördern. In diesem Artikel sollen die Grundkonzepte einer Service-Registry, die daraus resultierenden Vorteile sowie ein Lösungsansatz im Rahmen eines universitären Kontextes vorgestellt werden.

Verzeichnisdienst

Im Rahmen der serviceorientierten Architekturen (SOA) wurde bereits vor einigen Jahren der Verzeichnisdienst Universal Description, Discovery and Integration (UDDI) vorgestellt, der heutzutage allerdings kaum Anwendung findet. Neben UDDI sind mit Eureka, Consul, etcd und Apache ZooKeeper neue Technologien in den Fokus gerückt, die es erlauben, Webservices (nachfolgend kurz Services) in einem Verzeichnis oder in einer sogenannten Service-Registry zu veröffentlichen und für Service-Nutzer verfügbar zu machen.

Die Schnittstellen moderner Services bestehen meist aus Ansätzen wie REpresentational State Transfer (REST), die entweder selbstbeschreibend im Sinne von Hypermedia sind oder mit Hilfe einer Spezifikationssprache unabhängig von der Implementierung definiert und anschließend in einem solchen Verzeichnis bereitgestellt werden können.

SOA und Microservices

Bei einer serviceorientierten Architektur handelt es sich nicht um eine konkrete Technik, sondern um eine Abstraktion ([Mel10], s. Abb. 1). Neben den Merkmalen und Prinzipien einer solchen Architektur, wie beispielsweise der losen Kopplung, der Wiederverwendbarkeit oder auch dem Einsatz von Standards, spielt das klassische Service-Modell aus Abbildung 1 eine wichtige Rolle.

Heute wird neben der klassischen SOA dagegen häufig der moderne Microservice-Ansatz betrachtet, der aus der Praxis entstanden und auf anwendungsnahen Konzepten aufgebaut ist [New15]. Der Microservice-Ansatz kann daher als eine konkrete Ausprägung einer SOA angesehen werden und basiert auf Konzepten wie dem Domain-Driven Design, bei dem die abzubildende Geschäftsdomäne in Form eines Domänenmodells abgebildet wird [New15, Eva03].

Das Domänenmodell bildet wiederum die Grundlage für den Service-Schnitt beziehungsweise die Aufteilung in verschiedene Services [Eva03]. Für die Wiederverwendbarkeit der Service-Funktionalitäten werden sogenannte Schnittstellen (Application Programming Interfaces, APIs) veräußert ([Erl08], s. Abb. 2). Im Falle von Webservices wird hier oftmals von einem Web-API gesprochen, da die Schnittstelle über das Web zugänglich gemacht und dabei zumeist das Anwendungsschichtprotokoll HyperText Transfer Protocol (HTTP) verwendet wird.

API-Spezifikation und API-First zur Beschreibung eines Service

Unter einem API verstehen wir eine Schnittstelle zur Anwendungsprogrammierung, die die Kommunikation mit einem Service ermöglicht. An dieser Stelle möchten wir darauf hinweisen, dass wir unter APIs in diesem Artikel grundsätzlich Web-APIs verstehen. APIs ermöglichen die Nutzung von bereitgestellten Funktionalitäten eines Service und bilden einen Vertrag zwischen Service und Service-Nutzer. Dieser beschreibt, wie der Service-Nutzer mit dem Service kommunizieren kann [Erl08].

Heute besitzen immer mehr Services ressourcenorientierte oder auf Hypermedia basierende APIs, was sich aus dem Architekturstil REST abgeleitet hat. REST ist eine Technologie, die es ermöglicht, leichtgewichtigere APIs zu entwickeln, als es

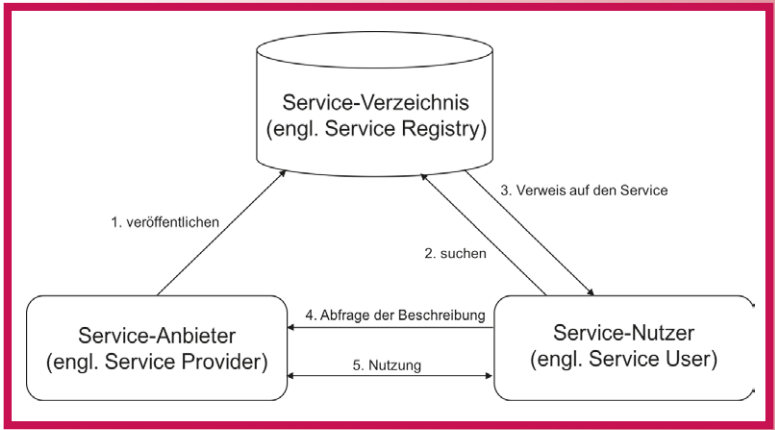

Abb. 1: Das klassische Service-Modell einer SOA [Mel10]

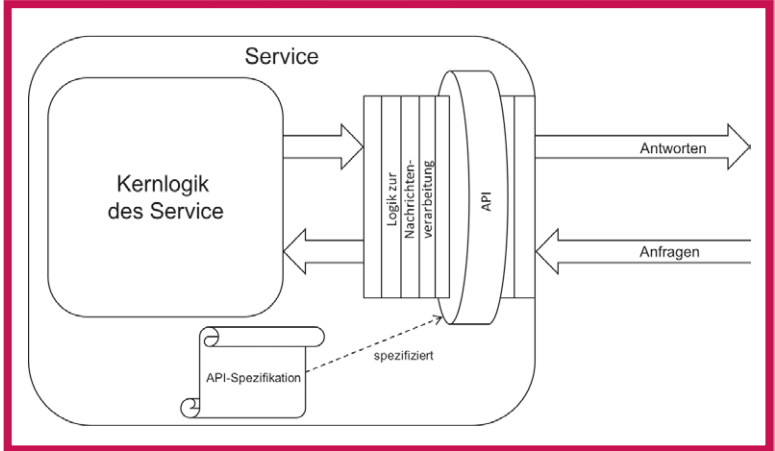

Abb. 2: Aufbau eines modernen Webservice, modifiziert nach [Erl08]

beispielsweise früher mit der Web Service Description Language (WSDL) möglich war. Große Firmen wie Zalando [FrSch15] verwenden heute den Entwicklungsansatz API-First [APIF], welcher das API vor der eigentlichen Implementierung spezifiziert. Die APIs werden anschließend detailliert analysiert und bewertet. Als Ergebnis liefert dieser Ansatz klar definierte und qualitätsbewusste APIs. API-First stellt daher zunächst das API selbst in den Vordergrund und nicht die Implementierung der eigentlichen Services [APIF].

Als Werkzeug zur Spezifikation von APIs im Kontext von REST existieren Ansätze wie OpenAPI [OAI] (früher unter dem Namen Swagger bekannt) oder RAML [RAML].

Service-Discovery

In einer SOA bilden Services das zentrale Architekturelement. Heute entwickeln Service-Anbieter zahlreiche Services und bieten deren Funktionalitäten, meist gegen eine Nutzungsgebühr, zur weiteren Verwendung an. Zum Beispiel ermöglicht das FlightAware API [FlightXML] die Abfrage von weltweiten Flugdaten. Die Veräußerung von Funktionalitäten über sogenannte offene APIs hat sich mittlerweile zu einem eigenen Geschäftsfeld entwickelt [HBR15].

Unter offenen APIs verstehen sich APIs, welche auch von externen Konsumenten genutzt werden können und damit nicht nur für die interne Wiederverwendung gedacht sind. Diese Art von APIs unterliegt dann meist auch weiteren Verträgen, welche die Art der Nutzung des ausgewählten Service im Hinblick auf die Kommerzialisierung genauer definieren. Hierfür liefern Jacobson et al. ein entsprechendes Buch, welches APIs aus Geschäftssicht näher beleuchtet [JBW11]. Die grundsätzliche Herausforderung stellt, neben der Bereitstellung eines Service sowie der Veräußerung von Funktionalität über ein API, die Service-Discovery dar.

Unter Service-Discovery versteht man allgemein die Anforderungen an die Suche und das Auffinden eines Service innerhalb eines Netzwerkes [Erl08]. Service-Discovery kann mit Hilfe einer Service-Registry umgesetzt und ermöglicht werden. Im Deutschen kann Service-Discovery auch mit dem Wort „Auffindbarkeit“ übersetzt werden. Die Serviceorientierung hat vor allem der Service-Discovery ihren Erfolg zu verdanken, da sie einem Entwickler hilft festzustellen, ob eine Funktionalität bereits durch einen anderen Service im Zuge des Single Responsibility Principle (SRP) erbracht wird. Demnach unterstützt die Service-Discovery das zentrale Prinzip der Wiederverwendbarkeit eines Service innerhalb einer SOA [Erl08].

Um die Suche nach einem Service über das Internet oder auch innerhalb einer Organisation zu ermöglichen, benötigt man einen Verzeichnisdienst beziehungsweise eine Service-Registry, welche Kenntnis über alle existierenden Services einer SOA hat. Als Grundlage für die Suche nach konkreten Services kann die API-Spezifikation dienen, da diese die Funktionalitäten eines Service von der eigentlichen Implementierung abstrahiert und damit die zugrunde liegende Domäne abbildet. Allerdings reicht diese Art der Beschreibung eines API häufig nicht aus, um den Service vollständig zu beschreiben [LGG10]. Aus diesem Grund werden API-Spezifikationen häufig um weitere semantische Informationen wie natürliche Sprache ergänzt.

Für die Suche können nun Auswahlkriterien auf die semantischen Informationen oder Teile der API-Spezifikation angewendet werden. Der schlimmste Fall ist, wenn ein passender Service über richtige Kriterien gesucht, dieser aber aufgrund einer falschen oder schlechten Beschreibung nicht gefunden werden kann. Aufgrund dieser Tatsache ist eine Service-Registry auch ein zentraler und sehr wichtiger Grundpfeiler innerhalb einer SOA im Zuge der Wiederverwendbarkeit. Die klare Definition von APIs und semantischen Informationen ist erfolgsentscheidend für eine SOA, da sonst bestehende Services nicht wiederverwendet oder wiedergefunden werden können. Dies führt dazu, dass sich die Funktionalitäten von Services innerhalb der SOA überlappen und somit Redundanz erzeugt wird.

Obwohl die Wiederverwendbarkeit ein lobenswertes Ziel ist, können sich dadurch auch Nachteile ergeben, die im Vorfeld abgewogen werden müssen, wie eine Reduktion der Service-Autonomie. Daneben kann die Service-Registry auch dazu dienen, eine Übersicht über sämtliche Services einer Serviceland-schaft zu erhalten.

Manuelle- und autonome Discovery

Die Service-Discovery kann in zwei grundlegende Kategorien aufgeteilt werden, wobei der Schwerpunkt in der Art und Weise liegt, wie diese letztlich umgesetzt werden beziehungsweise was deren primäres Ziel ist [Erl08, John08]:

- Die *manuelle* Discovery (engl. manual discovery) findet zur Entwicklungszeit (eng. design time) statt. Dabei wird von einer Person manuell ein Verzeichnis anhand von Suchkriterien durchsucht, um eine Beschreibung eines Service zu erhalten. Es wird also ein bereits existierender Service durch einen Entwickler gesucht.
- Bei der *autonomen* Discovery (engl. autonomous discovery) wird diese Aufgabe hingegen von einem Anfrageagenten, also einer Maschine, durchgeführt. Sie kann entweder zur Entwicklungs- oder auch zur Laufzeit (engl. run time) stattfinden.

Der Unterschied zwischen beiden Konzepten liegt unter anderem in der API-Spezifikation und den semantischen Informationen, die bei der manuellen Discovery für einen Menschen und der autonomen Discovery für eine Maschine lesbar und verständlich sein müssen. Außerdem weisen API-Spezifikationen und semantische Informationen bei der autonomen Discovery deutlich höhere Anforderungen an die Beschreibung auf. Dies liegt unter anderem daran, dass sich Maschinen schwertun, menschliche Sprache eindeutig zu interpretieren [LGG10].

Ein weiterer Punkt, der hierbei nicht vernachlässigt werden darf, ist das Vertrauen in eine Maschine. Es ist nicht klar, ob Menschen auf die Auswahl und die Entscheidungsfindung einer Maschine jederzeit vertrauen können. Bei der autonomen Discovery werden semantische Informationen benötigt, die zwingend von Maschinen eindeutig interpretiert werden können. Aus diesen Gründen sind Discovery-Prozesse heute fast ausschließlich für Menschen optimiert. Zumindest dessen Beteiligung ist heutzutage meist notwendig [Erl08, John08].

Auffindbarkeit und Interpretierbarkeit

Ein weiterer wichtiger Teil ist – neben der Auffindbarkeit – die Interpretierbarkeit. Beide Teile sind sehr stark verwandte Serviceentwurfsmerkmale und werden deshalb unter dem Begriff der Service-Discovery zusammengefasst [Erl08]. Unter Auffindbarkeit versteht man streng genommen die Suche und das Finden eines Service, wie es bisher beschrieben wurde. Entscheidende Informationen liefert die API-Spezifikation sowie die zusätzlichen semantischen Informationen.

Wurde ein Service erfolgreich gefunden, muss durch die Interpretierbarkeit der gewonnenen Informationen dafür gesorgt werden, dass die Informationen klar und eindeutig von einem Nutzer oder von einer Maschine verstanden werden. Die Interpretierbarkeit kann als eine Art Maß der Qualität der Kom-

munikation zwischen Service und Service-Nutzer verstanden werden [Erl08].

Umsetzung einer Service-Registry

Um die vorgestellten Konzepte der Service-Discovery in einem konkreten Projekt vorzustellen, möchten wir an dieser Stelle unsere SmartCampus Service-Registry vorstellen, die Service-Discovery umsetzt. SmartCampus ist eine serviceorientierte Webanwendung, die von der Forschungsgruppe Cooperation & Management (C&M) des Karlsruher Institut für Technologie (KIT) in Kooperation mit dem Fraunhofer Institut für Optronik, Systemtechnik und Bildauswertung (IOSB) in Karlsruhe entwickelt wird.

Die durch die einzelnen Services der SmartCampus-Architektur bereitgestellte Funktionalität soll Studierenden, Mitarbeitern und Gästen das Lernen, Lehren und Forschen am KIT erleichtern. Dazu bietet SmartCampus beispielsweise einen Campus-Plan mit Gebäudeinformationen, eine Möglichkeit, freie Arbeitsplätze aufzufinden sowie zu reservieren, und ermöglicht für Studierende mit Behinderung eine barrierefreie Navigation auf dem Campus [KITSC]. SmartCampus als mobile Webanwendung kann überall auf dem Campus genutzt werden.

Im Rahmen der SOA von SmartCampus liefern Services die funktionalen Bausteine des Gesamtsystems, die mit Hilfe einer Service-Registry verwaltet werden sollen. Diese Registry enthält dabei Referenzen zu vorhandenen und sich im Betrieb befindenden Services. Ein Hauptziel der SmartCampus Service-Registry ist es, eine ganzheitliche und aktuelle Sicht der Servicelandschaft aufzuzeigen. Informationen über Services, die in einer Service-Registry abgelegt sind, können jederzeit von Entwicklern abgerufen werden. Mit der SmartCampus Service-Registry sollen unter anderem Services innerhalb der Architektur wiederverwendbar gemacht werden, um neue Entwicklungen und Adaptionen zu beschleunigen und eine strukturierte und übersichtliche Architektur von SmartCampus zu schaffen.

Wahl der Technologie

Um für SmartCampus eine Service-Registry zu entwickeln, haben wir zunächst bestehende Open-Source-Lösungen analysiert, die bereits die technische Umsetzung von Service-Discovery unterstützen. Neben Apache ZooKeeper [ZK], Consul [Consul] und etcd [etcd] ist uns bei unseren Recherchen vor allem Eureka [Eureka] aufgefallen, welches von Netflix benutzt und öffentlich zur Verfügung gestellt wird. Aufgrund der Architektur, die sich sehr gut mit unseren Konzepten und Anforderungen vereinen lässt, des existierenden Proof-of-Concept am Beispiel von Netflix, der einfachen Integration in unser Gesamtsystem sowie des geringen operativen Aufwands, konnte sich Eureka in unserem Auswahlprozess gegenüber den Konkurrenztechnologien durchsetzen.

Eureka wurde von Netflix zur intelligenten Lastverteilung (engl. load balancing) und Ausfallsicherung (engl. failover) im Zuge von volatilen Umgebungen entwickelt [AWSEx, Jan15]. Der sogenannte Eureka-Server ist ein auf Java-basierender Service mit einer ressourcenorientierten Schnittstelle, um registrierte Services auffindbar zu machen. Für die Kommunikation mit dem Eureka-Server dient der Eureka-Client. Eureka erlaubt es, einen Service innerhalb einer Microservice-Architektur mit *Register* unter einem logischen Namen zu registrieren, über den der Service anschließend angesprochen werden kann.

Eine typische Architektur von Eureka ist in Abbildung 3 dargestellt, die neben dem Eureka-Server aus dem Applikationsclient und dem Applikationsservice besteht. Der Applikationsclient stellt Anfragen an den Applikationsservice, der wiederum die Anfragen beantwortet. Im Modell der SOA entsprechen diese Rollen dem Service und dem Service-Nutzer. Bei Eureka betrachtet man sogenannte Eureka-Cluster, die den Bereich einer Region abdecken und durch einen oder mehrere Server verwaltet werden.

Um einen Service bei Eureka zu registrieren, sendet der Eureka Client eine *Register*-Nachricht (HTTP-Request) an den Eureka-Server (Listing 1 repräsentiert eine beispielhafte Konfiguration im Kontext von Spring Boot). Dabei werden der logische Name des Service und die Lokation der Instanz angegeben. Alle 30 Sekunden muss der Service ein Lebenszeichen (engl. Heartbeat) an den Eureka-Server senden, um seine Erreichbarkeit mitzuteilen. Dazu wird eine *Renew*-Nachricht gesendet. Empfängt der Server ca. 90 Sekunden lang kein Lebenszeichen vom Eureka-Client, wird dieser aus der Service-Registry entfernt.

Die Informationen über die Registrierung und die Auffrischung der Registrierung wird an alle Eureka-Server innerhalb eines Clusters repliziert. Ein Eureka-Client fragt regelmäßig beim Eureka-Server nach den gespeicherten Informationen in der Service-Registry. Der Client verwendet dazu den Aufruf der *Get Registry*-Methode aus der Eureka-Client-Bibliothek. Sobald die Informationen beim Eureka-Client verfügbar sind, werden diese vom Client lokal in einem Cache gespeichert, woraufhin der Eureka-Client diese Informationen nutzen kann, um andere Services zu finden. Um einen Service abzumelden, kann der Eureka-Client eine *Cancel*-Nachricht an den Eureka-Server senden.

Ein weiterer wichtiger Punkt ist das Hinzufügen eigener Metadaten wie einer API-Spezifikation. Bei der Registrierung eines Service werden dessen Metadaten in der Service-Registry gespeichert und stehen anschließend für die Suche der Services bereit. Metadaten können bei Eureka durch die Definition von Key/Value-Paaren hinzugefügt werden. Mit *eureka.metadata.mykey=myvalue*

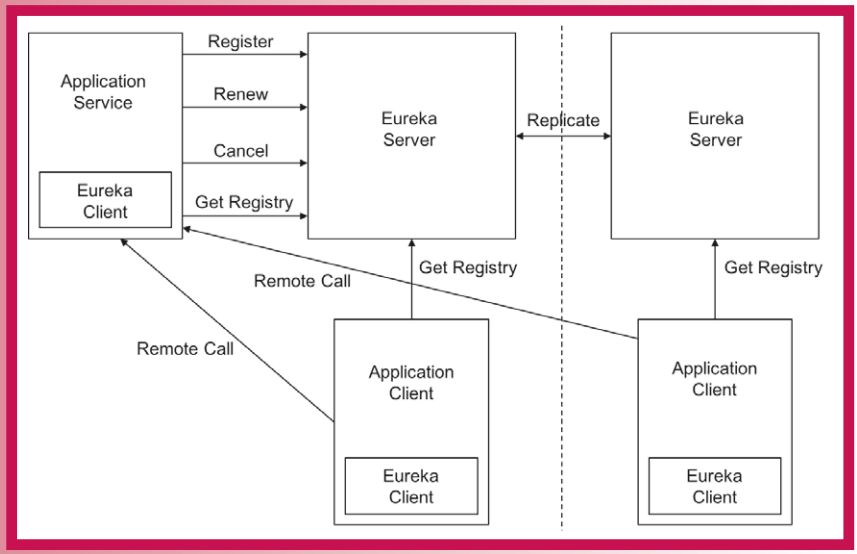

Abb. 3: Eureka-Architektur [Eureka]

kann das statische Key/Value-Paar *mykey:myvalue* angelegt werden. Eureka unterstützt jedoch auch das dynamische Anlegen von Key/Value-Paaren [Eureka].

```
server:
  port: 443
spring:
  application:
    name: WorkspaceService
eureka:
  client:
    serviceUrl:
      defaultZone: https://registry.smartcampus.kit.edu
  instance:
  preferIPAddress: true
```

Listing 1: Konfiguration für die Registrierung eines neuen Service bei Spring Boot

Umsetzung

Für die SmartCampus-Service-Registry haben wir die in Abbildung 4 dargestellte Architektur entwickelt, die an unsere Anforderungen angepasst ist. Die SmartCampus Service-Registry soll dabei manuelle Service-Discovery zur Entwicklungszeit unterstützen. In der Abbildung wird ein Service von SmartCampus dargestellt, der in einer Laufzeitumgebung hochgefahren wird und sich unmittelbar danach bei der Service-Registry anmeldet, um seine Erreichbarkeit mitzuteilen.

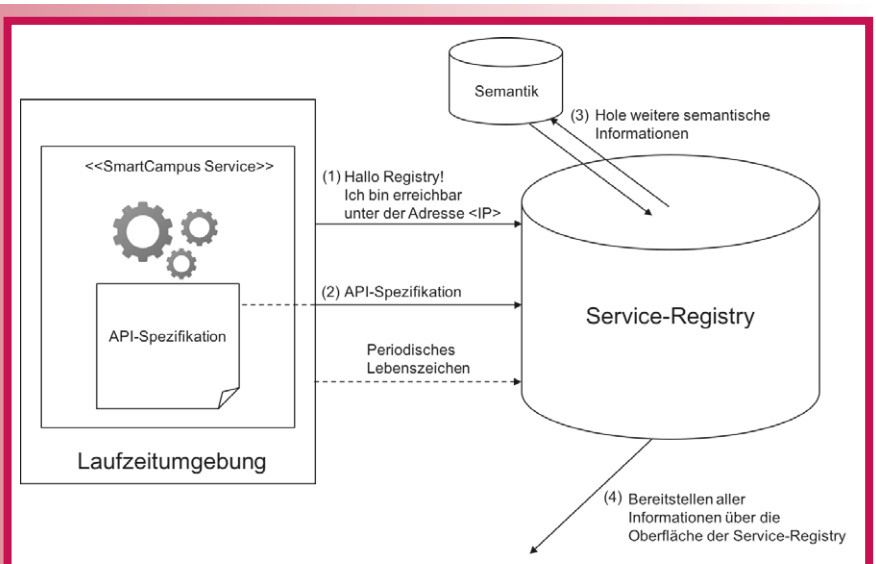

Abb. 4: Die SmartCampus-Service-Registry

Da jeder Service seine API-Spezifikation selber bereitstellen muss, kann der Service diese im zweiten Schritt automatisch an die Service-Registry senden. Der Eureka-Server nimmt diese Metadaten entgegen und speichert sie mit Hilfe des bereitgestellten Key/Value-Speichers. Die API-Spezifikation ist dabei bereits bei der Entwicklung des Service mit OpenAPI erstellt worden. Die darin enthaltenen Informationen können jedoch bei Bedarf um weitere semantische Informationen ergänzt werden, die beispielsweise auf ontologischen Ansätzen beruhen.

Basierend auf Eureka haben wir für unsere Service-Registry ein Frontend entwickelt, um die Servicelandschaft von SmartCampus darzustellen und die Interaktion zwischen der Service-Registry und einem Entwickler von SmartCampus zu ermöglichen. Außerdem haben wir eine Rechteverwaltung definiert, um Studierenden und Mitarbeitern unterschiedliche Rollen zuzuweisen.

Um die Service-Discovery zu ermöglichen, liefern wir über die Weboberfläche Informationen zu jedem Service und stellen Entwicklern eine durchsuchbare Liste aller registrierten Services bereit. In einer Kurzübersicht werden so der Name, die Kurzbeschreibung, die Verfügbarkeit und die Qualität der Schnittstelle entsprechend festgelegter Richtlinien des Service dargestellt. Die Analyse der Schnittstelle entsprechend selbst definierter API-Richtlinien wird durch einen weiteren Service erbracht und an den Eureka-Server übermittelt.

Zeigt ein Entwickler Interesse an einem Service, kann er dessen API-Spezifikation mithilfe von Swagger UI oder weitere semantische Informationen in Form von natürlicher Sprache, einem Domänenmodell oder Ontologien über die Weboberfläche der Service-Registry erhalten.

Fazit

Eine Service-Registry, wie sie im SmartCampus-Projekt verwendet wird, kann die Entwicklung in SOAs oder im modernen Microservice-Ansatz erheblich beschleunigen und vereinfachen. Service-Discovery ist aus unserer Sicht notwendig, um eine Servicelandschaft zu pflegen und die von Services bereitgestellte Funktionalität klar voneinander abzugrenzen, um Redundanz zu vermeiden. Außerdem ist eine Service-Registry der entscheidende Grundpfeiler der Wiederverwendbarkeit. Entwickler besitzen zu jeder Zeit eine aktuelle Sicht auf die Servicelandschaft, deren Services gefunden und wiederverwendet werden können.

Eine Service-Registry kann weiter die Rolle einer zentralen Prüfstelle der APIs der Services einnehmen, da API-Spezifikationen anhand von Richtlinien geprüft werden können. Mit Hilfe der grafischen Oberfläche kann die Servicelandschaft von jedem Entwickler eingesehen werden, wobei Verfügbarkeit und Qualität in den Vordergrund gerückt werden.

Literatur und Links

[APIF] API-First, `http://www.api-first.com/`

[AWSEx] Amazon Web Services, AWS-Fallbeispiel: Netflix, `https://aws.amazon.com/de/solutions/case-studies/netflix/?nc1=h_ls`

[Consul] `https://www.consul.io/`

[Erl08] Th. Erl, SOA – Principles of Service Design, Prentice Hall, 2008

[etcd] `https://coreos.com/etcd/`

[Eureka] `https://github.com/Netflix/eureka/wiki`

[Eva03] E. J. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley, 1. Auflage 2003

[FlightXML] FlightAware API, `https://de.flightaware.com/commercial/flightxml/`

[FrSch15] Th. Frauenstein, H. Schmeisky, On APIS and the Zalando Guild, Zalando, 2015, `https://tech.zalando.de/blog/on-apis-and-the-zalando-api-guild/`

[HBR15] B. Iyer, M. Subramaniam, The Strategic Value of APIs, Harvard Business Review, 7.1.2015, `https://hbr.org/2015/01/the-strategic-value-of-apis`

[Jan15] St. Janser, Eureka – Microservice-Registry mit Spring Cloud, heise Developer, 20.10.2015, http://www.heise.de/developer/artikel/Eureka-Microservice-Registry-mit-Spring-Cloud-2848238.html

[JBW11] D. Jacobson, G. Brail, D. Woods, APIs: A Strategy Guide – Creating Channels with Application Programming Interfaces, O'Reilly Media, 2011

[John08] F. T. Johnsen et al., Web Services and Service Discovery, Forsvarets forskningsinstitutt/Norwegian Defence Research Establishment (FFI), 9.5.2008, http://www.ffi.no/no/Rapporter/08-01064.pdf

[KITSC] SmartCampus, Karlsruher Institut für Technologie, http://cm.tm.kit.edu/smartcampus.php

[LGG10] M. Lanthaler, M. Granitzer, Ch. Gütl, Semantic Web Services: state of the art, in: Proc. of the ITS 2010

[Mel10] I. Melzer et al., Service-orientierte Architekturen mit Web Services, Spektrum Akademischer Verlag, 4. Auflage 2010

[New15] S. Newman, Building Microservices, O´Reilly, 2015

[OAI] OpenAPI, https://openapis.org/

[RAML] RESTful API Modeling Language, http://raml.org/

[ZK] Apache ZooKeeper, https://zookeeper.apache.org/

Pascal Giessler ist Head of Development bei der ZWEI14 Digital GmbH in Offenburg-Appenweier. Neben dieser Tätigkeit strebt er eine externe Promotion im Bereich der qualitätsorientierten Entwicklung moderner Services im Kontext einer SOA am Karlsruher Institut für Technologie (KIT) innerhalb der Forschungsgruppe Cooperation & Management (C&M) an.
E-Mail: pascal.giessler@kit.edu

Danny Forschner absolviert ein Masterstudium im Fach Informatik am Karlsruher Institut für Technologie (KIT). Er arbeitet derzeit an seiner Masterarbeit in der Forschungsgruppe Cooperation & Management (C&M) mit dem Thema: Analyse und Entwicklung einer Service-Registry für SmartCampus.

Sebastian Abeck leitet an der Fakultät für Informatik des KIT die Forschungsgruppe Cooperation & Management (C&M). Seine Forschungsinteressen betreffen die serviceorientierten Architekturen, das Internet der Dinge und das Identitäts- und Zugriffsmanagement.