Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D - 70569 Stuttgart

Bachelorarbeit Nr. 42

Konzept und Implementierung einer generischen Service Invocation Schnittstelle für Cloud Application Management basierend auf TOSCA

Michael Zimmermann

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. Frank Leymann
Betreuer:	Dipl.-Inf. Uwe Breitenbücher
Beginn am:	02.01.2013
Beendet am:	04.07.2013
CR-Nummer:	C.2.4, D.2.11, D.2.12, J.0

Kurzfassung

Cloud Computing ist ein viel diskutiertes Thema in der Informations- und Kommunikationstechnologie. Es ermöglicht Unternehmen sich auf ihr Kerngebiet zu konzentrieren ohne dabei auf professionelle IT-Infrastruktur verzichten zu müssen. Allerdings besteht demgegenüber die Gefahr eines Vendor-Lock-in, also die Abhängigkeit eines bestimmten Cloud Anbieters. Diesem Problem nimmt sich TOSCA [29] an. TOSCA ist ein Standard zur Beschreibung von interoperablen Cloud Anwendungen. TOSCA ermöglicht unter anderem die Beschreibung des Aufbaus, Deployments und Managements. Die Universität Stuttgart entwickelt eine Laufzeitumgebung namens OpenTOSCA [23] für diesen Standard. Eine erste prototypische aber dennoch funktionelle Implementierung ist bereits fertiggestellt. In dieser werden Aufrufe von Services allerdings außerhalb des Sichtbarkeitsbereichs des OpenTOSCA Containers ausgeführt.

In der vorliegenden Bachelorarbeit wird deshalb ein Konzept zum generischen Aufruf von Services durch eine zentrale Komponente des OpenTOSCA Containers vorgestellt. Dabei wird auf gestellte Anforderungen und getroffene Entwurfsentscheidungen ebenso eingegangen wie auf die Architektur und Möglichkeiten des erarbeiteten Lösungskonzepts. Weiterhin ist eine prototypische Implementierung des Konzepts Teil dieser Arbeit. Es werden daher wichtige sowie interessante Punkte der Implementierung dargestellt und erläutert.

Inhaltsverzeichnis

1 Einleitung

Cloud Computing ist einer Umfrage des Bundesverband Informationswirtschaft, Telekommunikation und neue Medien e.V. (BITKOM) zufolge das wichtigste Hightech-Thema der ITK-Branche 2013 [15]. Es ermöglicht unter anderem eine hohe Verfügbar- sowie Anpassungsfähigkeit, Kostenreduzierungen und die Nutzung leistungsfähiger IT-Ressourcen ohne Wissen über Administration und ähnlichem. Ein Problem des Cloud Computing ist allerdings der Lock-in-Effekt, also die Abhängigkeit des Kunden von einem bestimmten Cloud-Anbieter.

Demgegenüber wurde mit Topology and Orchestration Specification for Cloud Applications (TOSCA) [29] ein Standard zur Beschreibung von interoperablen und portablen Cloud Anwendungen geschaffen. Mit OpenTOSCA [23] wurde an der Universität Stuttgart weiterhin eine erste Open Source Implementierung einer Laufzeitumgebung für TOSCA entwickelt, welche aktuell kontinuierlich verbessert und erweitert wird.

In dem momentanen Entwicklungsstand von OpenTOSCA werden Services zum Management der Cloud Anwendung, wie `Pläne` oder `Implementation Artifacts` direkt und damit außerhalb des Sichtbarkeitsbereichs des Containers aufgerufen, was beispielsweise ein Logging oder Monitoring der ausgeführten Aufrufe unmöglich macht. Weiterhin steht dem Aufrufer kein einheitliches Interface zur Verfügung. Dadurch ist der Aufrufer gezwungen, sich mit den Besonderheiten verschiedenster Techniken wie zum Beispiel SOAP oder REST auseinander setzen zu müssen. Darüber hinaus wird zudem ein extra Programm zum Aufruf von `Plänen` benötigt.

Um diese Einschränkungen zu beseitigen, soll im Rahmen dieser Bachelorarbeit ein Konzept für eine zentrale Komponente - Service Invocation Schnittstelle genannt - für OpenTOSCA zum Aufruf von in TOSCA spezifizierten Services erar-

beitet werden. Weiterhin soll im praktischen Teil dieser Arbeit diese Funktionalität implementiert sowie in OpenTOSCA integriert werden.

Die Ausarbeitung ist folgendermaßen strukturiert: In Kapitel 2 werden dem Leser zuerst einige Grundlagen zum Verständnis dieser Arbeit nähergebracht. Dabei wird auf Cloud Computing, TOSCA, OSGi, OpenTOSCA sowie eine Erweiterung dazu, die Plan Invocation Engine [18] eingegangen. Außerdem werden mit Abschnitten über ESBs sowie Camel Möglichkeiten zur Integration verschiedener Komponenten und Services betrachtet. Anschließend werden in Kapitel 3 die an die Service Invocation Schnittstelle gestellten Anforderungen erarbeitet und dokumentiert. Daraufhin wird in Kapitel 4 unter Berücksichtigung der getroffenen Entwurfsentscheidungen das Konzept und die Architektur der Service Invocation Schnittstelle vorgestellt sowie die Möglichkeiten des gewählten Lösungskonzepts aufgezeigt. Im Anschluss daran wird in Kapitel 5 die Implementierung vorgestellt. In Kapitel 6 folgt eine Übersicht an getroffenen Annahmen zur Umsetzung der Service Invocation Schnittstelle. Anschließend wird in Kapitel 7 das Konzept beziehungsweise die Implementierung der Service Invocation Schnittstelle anhand der gestellten Anforderungen evaluiert, bevor in Kapitel 8 die Arbeit nochmals zusammengefasst und ein Ausblick auf weitere Optimierungsmöglichkeiten gegeben wird.

2 Thematische Grundlagen und verwandte Arbeiten

In diesem Kapitel werden zum Verständnis der Arbeit benötigte Grundlagen erläutert. Zuerst wird auf Cloud Computing und dessen Möglichkeiten eingegangen. Anschließend werden Grundlagen zu TOSCA erläutert und das Framework OSGi vorgestellt. Daraufhin wird der OpenTOSCA Container und eine Erweiterung dazu, die Plan Invocation Engine, betrachtet. Zum Ende des Kapitels wird auf Integrationsmöglichkeiten wie Enterprise Service Buse sowie Integrationsframeworks eingegangen.

2.1 Cloud Computing

Cloud Computing ist eine an die jeweils aktuellen Benutzeranforderungen elastisch anpassbare und über ein Netzwerk durch standardisierte Techniken erreichbare IT-Infrastruktur. Sie stellt sowohl Infrastruktur (Hardware wie z.B. Server), Plattformen als auch Software als Service bereit. [25]

Es wird zwischen vier wesentlichen Bereitstellungsmodellen (Deployment Models) unterschieden. Diese sind Public-, Private-, Community-, sowie Hybrid Cloud und werden folgend vorgestellt.

Public Cloud. Die Public Cloud ist eine von meist einem IT-Unternehmen angebotene und für beliebige Personen und Organisationen zugängliche Cloud. Der Kunde der Public Cloud kann dabei die für seine momentanen Bedürfnisse geeigneten Angebote wählen ohne teure Anschaffungen tätigen zu müssen. [25]

Private Cloud. Die Private Cloud ist eine für eine Organisation speziell eingerichtete Cloud. Benutzung sowie Zugang sind auf die Organisation und gegebenenfalls zusätzlich einen autorisierten Personenkreis wie zum Beispiel Kunden beschränkt. Die Hardware der Private Cloud befindet sich oftmals innerhalb der Organisation selbst und die Verwaltung erfolgt in der Regel intern. [25]

Community Cloud. Die Community Cloud ist der Private Cloud ähnlich, mit dem Unterschied, dass sich hier mehrere Organisationen eine Cloud teilen. Gründe dafür können die Reduzierung von Kosten oder gemeinsame Interessen sein. [24]

Hybrid Cloud. Die Hybrid Cloud ist eine beliebige Kombination aus herkömmlichen IT-Umgebungen sowie den vorherig vorgestellten Cloud-Arten. [25]

Weiterhin wird bei Cloud Computing zwischen drei typischen Service Arten unterschieden: IaaS, PaaS und SaaS.

IaaS (Infrastructure-as-a-Service). Bei IaaS wird dem Kunden vom IT-Dienstleister eine Infrastruktur (z.B. Server) bereitgestellt. Der Kunde muss zur Arbeit benötigte Software selbst installieren und verwalten. Ein Beispiel für IaaS ist Amazons EC2 [14]. [25]

PaaS (Platform-as-a-Service). Bei PaaS stellt der IT-Dienstleister dem Kunden eine komplette und betriebsbereite Plattform, bestehend aus Hard-, Middle- und gegebenenfalls Software zur Verfügung und kümmert sich auch um deren Administration. Auf dieser bereitgestellten Plattform kann der Kunde selbst Software betreiben und verwalten. Ein Beispiel für PaaS ist Microsofts Windows Azure [26]. [25]

SaaS (Software-as-a-Service). Bei SaaS stellt der IT-Dienstleister dem Kunden eine bestehende Software über das Internet oder Intranet zur Verfügung und kümmert sich darüber hinaus auch um deren Administration. Der Kunde kann beispielsweise per Web-Browser darauf zugreifen und die Software damit nutzen. Ein Beispiel für eine SaaS Anwendung ist Google Mail [20]. [25]

Gemeinsam haben alle drei Service-Arten, dass dem Kunden der Kauf von Hardware und gegebenenfalls auch Software erspart bleibt. Weiterhin bieten diese Konzepte eine hohe Skalierbarkeit und nehmen dem Kunden die Komplexität von Installationen sowie Administration ab.

Cloud Anwendungen bestehen typischerweise jedoch aus mehreren Cloud Services. Die Provisionierung der heterogenen Komponenten ist weiterhin aufgrund verschiedener Technologien (z.B. REST [19], SOAP [40], Chef [32], Puppet [38], AWS API, Azure API, usw.) nicht einfach automatisierbar. Hierfür müssen die verwendeten Technologien sowie verschiedene Arten von Provisionierungstools integriert werden. Zu diesem Zweck sind in TOSCA (siehe nächstes Kapitel) sogenannte `Pläne` vorgesehen. `Pläne` beschreiben Workflows und nutzen Managementoperationen verschiedener Services zum Management von Cloud Anwendungen und ermöglichen dadurch eine Automatisierung. Allerdings ist das Schreiben dieser `Pläne` aufgrund den verschiedenen genutzten Technologien sowie unterschiedlichen Interfaces sehr schwierig. Aus diesem Grund wird eine einheitliche Schnittstelle zum Aufruf der Managementoperationen benötigt, welche die verschiedenen Technologien dem `Plan` gegenüber verbirgt. Dadurch können `Pläne` einfacher erstellt werden und sind zudem weniger komplex.

2.2 TOSCA

In diesem Kapitel werden die für diese Arbeit wichtigsten Grundlagen von TOSCA [29] erläutert.

Die Topology and Orchestration Specification for Cloud Applications (TOSCA) ist eine Sprache zur Beschreibung von Cloud Anwendungen. Sie wurde von einer Initiative der OASIS (Organization for the Advancement of Structured Information Standards) [28] entwickelt und liegt aktuell in Version 1.0 [29] vor. Das Ziel von TOSCA ist es, einen Standard zur Interoperabilität und Portabilität von Cloud Umgebung zu definieren, der sowohl die automatische Provisionierung als auch Management ermöglicht. Dafür lassen sich mittels TOSCA die einzelnen Service-Komponenten einer Cloud Anwendung, deren Beziehungen zueinander sowie deren Managementoperationen unabhängig von einer konkre-

ten Cloud Umgebung beschreiben. Ein Dokument, das eine solche Beschreibung enthält wird TOSCA Definition genannt. [29]

Für diese Arbeit von besonderer Bedeutung sind die beiden TOSCA Elemente `Implementation Artifacts (IAs)` und `Pläne`, welche Services nach außen anbieten und folgend genauer betrachtet werden.

`Implementation Artifacts` bieten Managementoperationen einer Komponente einer Cloud Anwendung an. So ermöglicht ein `Implementation Artifact` zum Beispiel das Erstellen einer Datenbank auf einem Server oder das Installieren von dort benötigten Treibern. TOSCA erlaubt dabei die Implementierung verschiedenster `Implementation Artifact` Arten. So sind beispielsweise Web Archive (WAR), welche einen SOAP Web Service anbieten, ebenso möglich wie komplexe Skripte, welche auch über andere Technologien wie Chef [32] oder Puppet [38] verwaltet werden können und auf der Zielinfrastruktur ausgeführt werden.

`Pläne`, oder auch `Managementpläne` genannt, nutzen die von den `Implementation Artifacts` bereitgestellten Operationen zum Management der Service-Komponenten und damit der Cloud Anwendung. Äquivalent zu `Implementation Artifacts` können auch `Pläne` mittels verschiedenster Plansprachen, wie zum Beispiel BPMN (Business Process Model and Notation) [31] oder BPEL (Web Services Business Process Execution Language) [30] realisiert werden.

2.3 OSGi

Dieser Abschnitt stellt das OSGi Framework vor, welches zur Implementierung von OpenTOSCA (siehe nächstes Kapitel) genutzt wird.

Das OSGi Framework ist eine auf Java basierende Softwareplattform und ermöglicht eine Modularisierung von Anwendungen. Die OSGi Spezifikation [35] wird

von der OSGi Alliance [37] entwickelt. Es bestehen mehrere, sowohl kommerzielle als auch Open Source Implementierungen des OSGi Standards. Für die Implementierung von OpenTOSCA sowie der Service Invocation Schnittstelle wird Eclipse Equinox [17] genutzt.

Im Kontext von OSGi wird zwischen Softwarekomponenten – Bundles genannt – und Diensten – Services genannt – unterschieden. Bundles sind möglichst in sich abgeschlossene Einheiten, bestehend aus Klassen und Ressourcen, sind als JAR Archive gepackt und lassen sich im OSGi Framework deployen. Services implementieren Interfaces und bieten dessen Funktionalitäten anderen Komponenten als Service an. Ein Bundle kann mehrere Services anbieten. Weiterhin können mehrere Implementierungen (also Services) eines Interfaces sowie verschiedene Versionen eines Bundles innerhalb einer OSGi Umgebung vorhanden sein. Desweiteren wird Hot Deployment (das Hinzufügen und Starten zur Laufzeit) von OSGi ermöglicht.

Von besonderer Wichtigkeit zum Verständnis der Implementierungen von OpenTOSCA sowie der Service Invocation Schnittstelle sind Declarative Services. Declarative Services ermöglichen das konfigurieren von Services per XML-Konfigurationsdatei. Damit lässt sich für Komponenten definieren, welche Services angeboten und/oder benötigt werden. Dies ermöglicht unter anderem eine automatische Auflösung von Abhängigkeiten verschiedener Komponenten. Benötigte Services können auch als optional gekennzeichnet werden und müssen somit beim Start der Komponente nicht vorhanden sein. Weiterhin kann der Konsument eines Services durch bind- und unbind-Methoden dynamisch auf das Starten oder Stoppen eines Services reagieren und ermöglicht somit zum Beispiel die Umsetzung eines Plug-in-Systems wie es auch in OpenTOSCA, was im nächsten Kapitel vorgestellt wird, genutzt wird.

2.4 OpenTOSCA

In diesem Kapitel wird der aktuelle Implementierungsstand von OpenTOSCA [23], einem an der Universität Stuttgart entwickelten Open Source TOSCA Containers, beschrieben.

Mittels OpenTOSCA können Cloud Service Archives (`CSARs`) [29] installiert werden. `CSARs` beinhalten neben der TOSCA Definition alle benötigten Artefakte wie zum Beispiel `Implementation Artifacts` oder `Pläne`.

Abbildung 1 zeigt die Architektur von OpenTOSCA. Hauptkomponenten des Containers sind OpenTOSCAControl, IA-Engine und Plan-Engine.

Aufgabe der OpenTOSCAControl Komponente ist es, den Ablauf der Bearbeitung einer `CSAR`-Datei zu dirigieren. Sie bietet zudem Funktionen an, die durch die OpenTOSCA Container API mittels einer REST-Schnittstelle nach außen hin, unter anderem einer grafischen Benutzerschnittstelle, zur Verfügung gestellt werden.

Die IA-Engine ist für das Deployment von `Implementation Artifacts` zuständig, welche in der `CSAR` enthaltenen sind. Da `Implementation Artifacts`, wie in Kapitel 2.2 beschrieben, verschiedenster Art sein können, ist die IA-Engine mittels eines Plug-in-Systems realisiert. Dadurch können neue Plug-ins, welche das eigentliche Deployment der `Implementation Artifacts` ausführen, zur Laufzeit hinzugefügt werden. Die Endpunkte der erfolgreich deployten `Implementation Artifacts` werden in einer Endpunktdatenbank gespeichert. Aktuell wird das Deployment von Web Archives (WAR) auf Tomcat [6] sowie Axis Archives (AAR) auf Apache Axis [1] unterstützt.

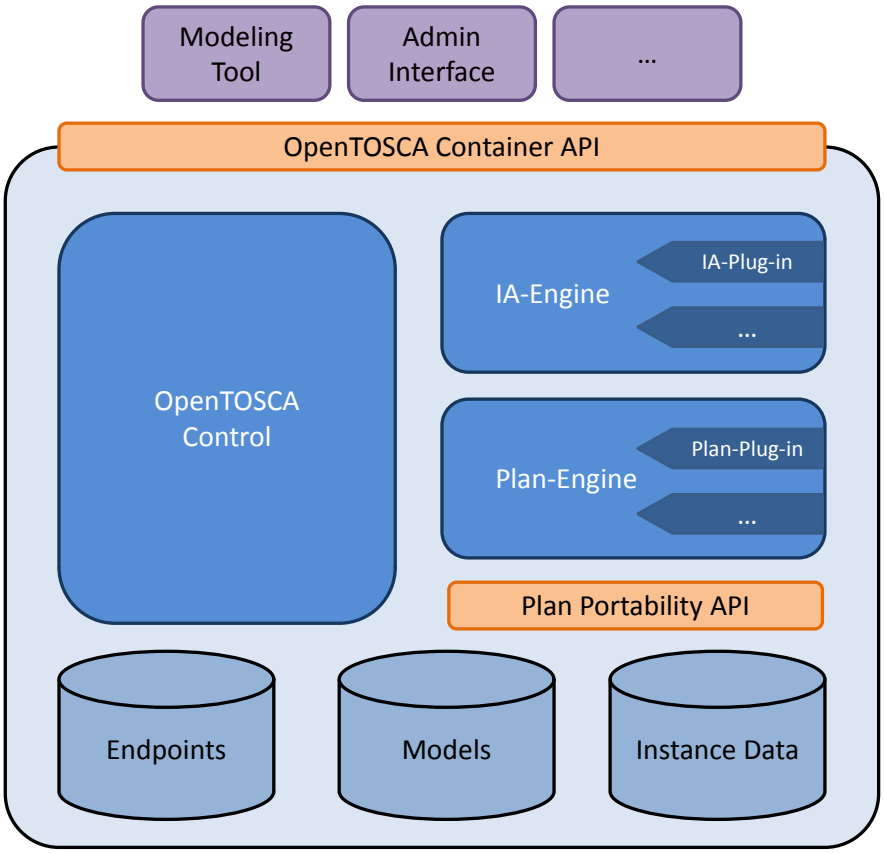

Abbildung 1: Architektur von OpenTOSCA (nach [23])

Äquivalent zur IA-Engine ermöglicht die Plan-Engine das Deployment von `Plänen`. Ebenfalls ist die Plan-Engine, um die Anzahl an unterstützten `Plan`-Arten einfach erweitern zu können, mit einem Plug-in-System ausgestattet. Ebenso speichert die Plan-Engine die Endpunkte der deployten `Pläne` in der Endpunktdatenbank. Der momentane Stand des OpenTOSCA Containers unterstützt lediglich das Deployment von BPEL `Plänen` auf dem WSO2 Business Process Server (BPS) [45]. Weiterhin werden in der aktuellen Implementierung vor dem Deployment die `Pläne` von der Plan-Engine an die von den `Implementation Artifacts` angebotenen Managementoperationen gebunden. Dafür wird der Plan zuerst analysiert und anschließend an die sich in der Endpunktdatenbank befindlichen Endpunkte gebunden.

Darüber hinaus gibt es noch weitere Komponenten, wie zum Beispiel einen InstanceDataService zur Haltung von Instanzdaten oder eine TOSCAEngine zur Verwaltung des TOSCA Modells.

Die Entwicklung von OpenTOSCA als erste Open Source Referenzimplementierung der TOSCA Spezifikation ist zum Zeitpunkt dieser Arbeit noch nicht abgeschlossen und wird unter anderem durch verschiedene Studienarbeiten vorangetrieben. Das nächste Kapitel stellt eine solche Erweiterung von OpenTOSCA vor.

2.5 Plan Invocation Engine

Dieser Abschnitt stellt die Plan Invocation Engine [18], eine Erweiterung des OpenTOSCA Containers vor.

Die Plan Invocation Engine wurde in einer parallel laufenden Bachelorarbeit entwickelt und ermöglicht dem Nutzer das Management von `Plänen` beziehungsweise Planinstanzen. Sie übernimmt unter anderem die Verwaltung von `CSAR`-sowie Prozessinstanzen und bietet auch eine Historie dieser an. Weiterhin erlaubt die Plan Invocation Engine das Erstellen der zum Aufruf eines `Plans` benötigten Nachricht mittels einer grafischen Benutzerschnittstelle (siehe Abbildung 2), zur Eingabe benötigter Parameter. Zum Aufrufen des `Plans` soll von der Plan Invocation Engine die in dieser Arbeit entwickelte Service Invocation Schnittstelle genutzt werden. [18]

Abbildung 2 zeigt einen Screenshot der grafischen Benutzeroberfläche der Plan Invocation Engine. Im Vordergrund sieht man unten rechts im Bild das Fenster zur Eingabe von Parametern. Im Hintergrund sieht man weitere Informationen über den `Plan` wie beispielsweise eine History über ausgeführte Aufrufe oder wann der `Plan` erstellt wurde.

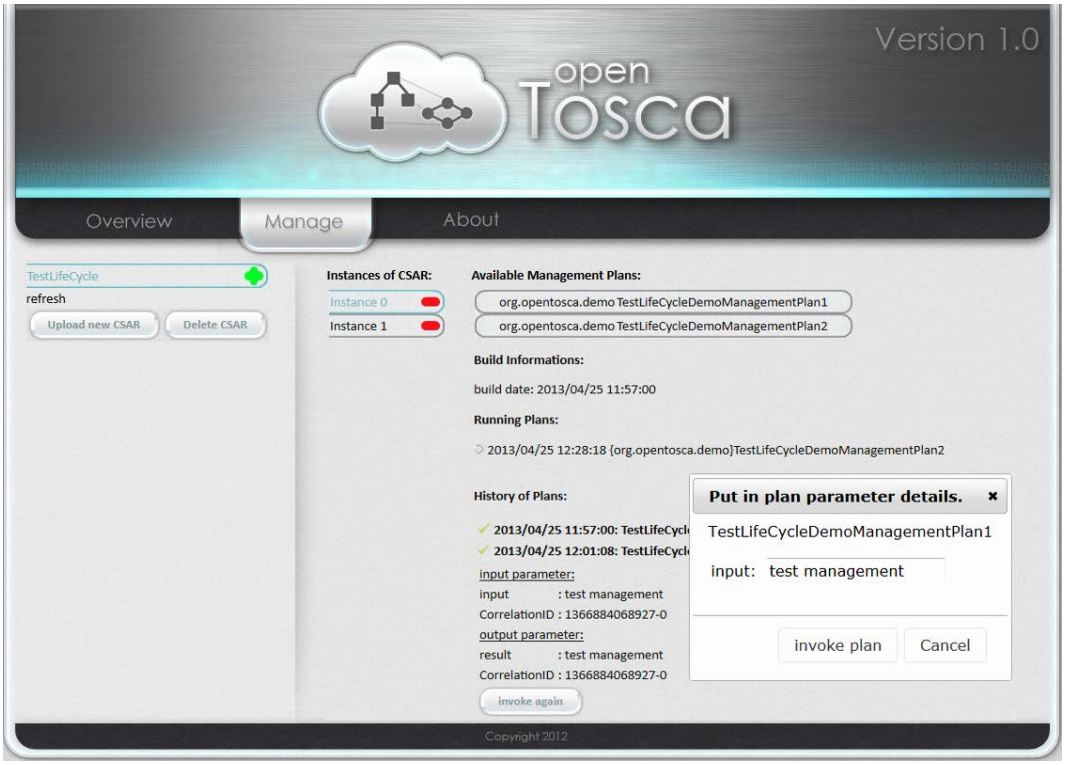

Abbildung 2: GUI der Plan Invocation Engine

2.6 ESB

In diesem Kapitel wird der sogenannte Enterprise Service Bus (ESB) behandelt. Auf eine detaillierte Erläuterung der Technologie wird an dieser Stelle verzichtet und ausschließlich die zum Verständnis grundlegenden Konzepte dargestellt. Für eine ausführliche Behandlung des Themas wird auf [16] verwiesen.

Für den Begriff ESB existiert eine Vielzahl an verschiedenen Definitionen. Exemplarisch wird an dieser Stelle die Definition von David A. Chappell aus seinem Buch *Enterprise Service Bus* [16, Seite 1] verwendet:

> An ESB is a standards-based integration platform that combines messaging, web services, data transformation, and intelligent routing to reliable connect and coordinate the interaction of significant numbers of diverse applications across extended enterprises with transactional integrity.

Nach Chappell ist ein Enterprise Service Bus also eine auf Standards beruhende Integrationsplattform, die sowohl Nachrichtenaustausch, Web Services, Datentransformation als auch intelligentes Routing miteinander verbindet. Ein ESB verbindet sowie koordiniert laut Chappell, Interaktionen verschiedenster Anwendungen eines Unternehmens zuverlässig (bezüglich der Transaktionssicherheit) miteinander.

Abbildung 3 illustriert eine beispielhafte Architektur, bestehend aus fünf verschiedenen Komponenten, einerseits ohne den Gebrauch eines ESBs (links) sowie andererseits mit einem ESB (rechts). Ohne die Nutzung eines ESBs kommunizieren die einzelnen Komponenten auf direktem Wege miteinander. Dies erfordert bei abweichenden Datenformaten oder Transportprotokollen, viele einzelne Schnittstellen der Komponenten untereinander. Mittels eines ESBs dagegen wird eine zentrale Kommunikationskomponente geschaffen, welche die verschiedenen Komponenten miteinander verbindet und die Kommunikation vereinfacht.

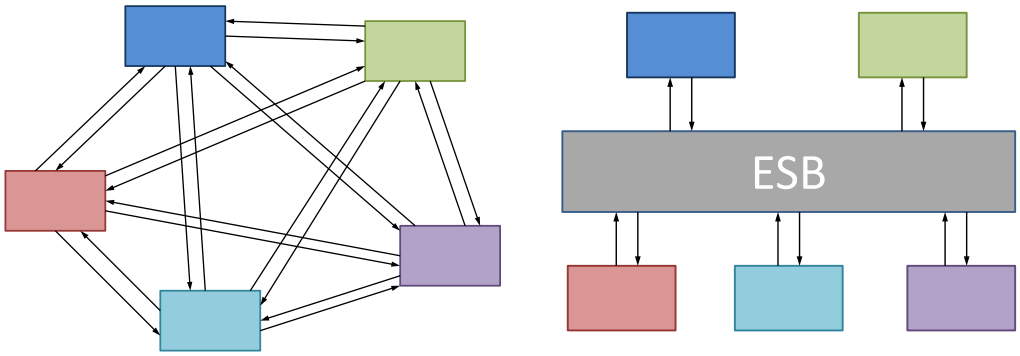

Abbildung 3: Beispiel-Architektur aus fünf Komponenten ohne (links) und mit (rechts) ESB

2.7 Camel

Dieser Abschnitt stellt das Open Source Integrations-Framework Apache Camel [2], welches zur Implementierung der Service Invocation Schnittstelle genutzt wird, vor.

Das Projekt Camel wurde 2007 gestartet und befindet sich aktuell in Version 2.11.0. Es ermöglicht zur Integration verschiedener Komponenten und Services sowohl das Routing von Nachrichten, das Konvertieren von Datenformaten als auch die Implementierung einer Vielzahl an Enterprise Integration Patterns (EIPs) [3]. Camel bietet dafür verschiedene domänenspezifische Sprachen (domain-specific languages, DSLs) wie zum Beispiel Java oder XML an. Aufgrund seines modularen Aufbaus und einer Vielzahl an vorhandenen Komponenten, ist Camel einerseits leichtgewichtig und flexibel sowie andererseits dennoch mächtig bezüglich des Funktionsumfangs. Weiterhin kann es sowohl als standalone Anwendung als auch eingebettet in beispielsweise einen OSGi Container genutzt werden. [2]

Zusammengefasst ermöglicht Camel also, analog zu einem ESB, die Integration verschiedener Komponenten zu einem Gesamtsystem. Bezüglich der Ähnlichkeit zu einem EBS, beschreiben die Verantwortlichen Camel auf der einen Seite selbst als:

> […] a rule based routing & mediation engine which can be used inside a full blown ESB, a message broker or a web services smart client. Though if you want to, you could consider that Camel is a small, lightweight embeddable ESB since it can provide many of the common ESB services like smart routing, transformation, mediation, monitoring, orchestration etc. [11]

Also ist gemäß der Entwickler Camel eine regelbasierte Routing- und Verbindungs-Engine, welche in einem „vollendeten“ ESB, einem Message-Broker oder einem Web Service Smart Client genutzt werden kann. Man könnte Camel jedoch auch aufgrund von vielen üblichen gebotenen ESB Funktionalitäten wie

intelligentes Routing, Transformation, Vermittlung, Monitoring, Orchestrierung usw. als einen kompakten, leichtgewichtigen und einbettungsfähigen ESB betrachten.

Allerdings stellen sie auf der anderen Seite auch fest, dass ein ESB ihrer Ansicht nach eher ein aus Integrationskomponenten bestehender Container sei. So sehen sie etwa Apache ServiceMix [4], welcher auf OSGi (und optional JBI[1]) basiert und damit eine standardisierte Integrationsplattform bietet, als einen richtigen ESB an:

> However our view is that an ESB is more of a container of integration components, so we view Apache ServiceMix to be a true ESB based around OSGi (and optionally JBI) to provide a standards based integration platform of components. [11]

Zusammenfassend ist es ihrer Meinung nach also durchaus zulässig Camel als ESB zu bezeichnen, sie selbst würden Camel allerdings nicht als kompletten ESB betiteln.

Im Folgenden werden einige zum Verständnis dieser Arbeit wichtige Konzepte von Camel vorgestellt.

Endpoint. Ein Endpoint wird durch eine URI identifiziert und beschreibt in Camel einen Endpunkt eines Kommunikations-Kanals. Beispielsweise beschreibt *jetty://http://localhost:9080/myservice* einen durch die Camel-Jetty-Komponente [12] realisierten Endpoint. [10]

Message und Exchange. Eine Message repräsentiert in Camel die Daten, die zwischen den verschiedenen Endpoints übergeben werden. Eine Message kann dabei aus Headern, einem Body sowie Attachments bestehen. Die Header sind in Camel per HashMap, der Body vom Typ Object implementiert. Exchange ist der Container einer Message und beinhaltet neben ihr unter anderem während der Bearbeitung aufgetretene Fehler oder Routing-Informationen. [10]

[1] Java Business Integration: Standard zur Beschreibung von Integrationssystemen.

Processor. Mittels eines Prozessors kann eine Message bearbeitet werden. Beispielsweise können weitere Daten hinzugefügt oder anhand der bestehenden Daten die Route geändert werden. Es können sowohl vorhandene Processors genutzt als auch eigene implementiert werden. [10]

Route. Eine Route besteht aus hintereinander hängenden Processors und Endpoints und repräsentiert den Bearbeitungsablauf einer Message [10]. Dieses Architekturmuster zur Beschreibung von Datenströmen wird als Pipes und Filter [21] bezeichnet. Abbildung 4 zeigt eine schematische Darstellung des Pipes und Filter Architekturmusters. Filter sind die Verarbeitungseinheiten. In ihnen werden die eingehenden Daten be- bzw. verarbeitet. Pipes sind Verbindungen zwischen den einzelnen Filtern.

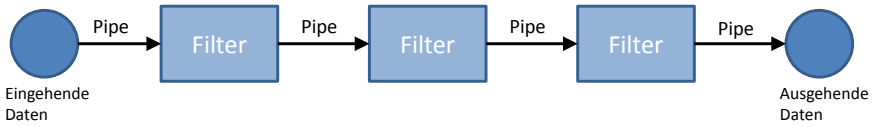

Abbildung 4: Pipes und Filter Architekturmuster

3 Anforderungen

In diesem Kapitel werden die an die Service Invocation Schnittstelle gestellten Anforderungen beschrieben. Zunächst werden in 3.1 funktionale Anforderungen benannt. Anschließend folgen in 3.2 an die Service Invocation Schnittstelle gestellte nichtfunktionale Anforderungen.

3.1 Funktionale Anforderungen

Dieser Abschnitt stellt die an die Service Invocation Schnittstelle gestellten funktionalen Anforderungen vor. Die Service Invocation Schnittstelle wird mit Hinblick auf diese Anforderungen konzipiert und entwickelt.

Möglichkeit zum Aufrufen von IAs sowie Plänen

Sowohl `Implementation Artifacts` als auch `Pläne` stellen ausführbare Services im Kontext von TOSCA dar. Aufgrund des Ziels der Service Invocation Schnittstelle, alle in TOSCA vorkommenden Services generisch aufrufbar zu machen, muss die Service Invocation Schnittstelle sowohl das Aufrufen von beliebigen `Implementation Artifacts` als auch von `Plänen` unterstützen.

Hauptsächlich werden mittels der Service Invocation Schnittstelle `Implementation Artifacts` durch `Pläne` aufgerufen werden. Allerdings sind zum Beispiel auch Aufrufe von `Implementation Artifacts` durch andere `Implementation Artifacts`, Aufrufe von `Plänen` durch `Implementation Artifacts` oder andere Varianten denkbar und müssen von der Service Invocation Schnittstelle unterstützt werden.

Des Weiteren muss insbesondere der Aufruf von `Plänen` durch die Plan Invocation Engine (siehe Kapitel 2.5) unterstützt werden.

Möglichkeit Asynchroner Aufrufe

Dem Aufrufer (Client) der Service Invocation Schnittstelle muss es möglich sein, diese asynchron aufrufen zu können. Dadurch muss der Aufrufer nicht warten, bis die Service Invocation Schnittstelle antwortet, und wird somit nicht an der Weiterarbeit blockiert.

Vor allem `Pläne`, welche die Mehrheit der Clients der Service Invocation Schnittstelle ausmachen werden, können stark von asynchronen Aufrufen bezüglich ihrer Bearbeitungsdauer profitieren. Sie können dadurch langläufige Prozesse frühzeitig initiieren und parallel andere notwendige Arbeitsschritte ausführen. Einem `Plan` ist es somit beispielsweise möglich die Aufträge zur Erstellung aller benötigten Datenbanken direkt zu Anfang des `Plans` zu erteilen und anschließend, während parallel die Datenbanken erstellt werden, auf einer, von den Datenbanken unabhängigen, virtuellen Maschine benötigte Treiber zu installieren.

Natürlich können neben `Plänen` auch alle anderen Aufrufer mit langläufigen und parallelen Prozessen von der Asynchronität der Kommunikation profitieren.

Dynamische Bestimmung der zum Aufruf von IAs/Plänen benötigten Informationen

Die Service Invocation Schnittstelle muss die Informationen, welche zum Aufrufen von `Implementation Artifacts` und `Pläne` benötigt werden, dynamisch beschaffen können. Dies verringert zum Einen die Anzahl an benötigten Übergabeparametern des Aufrufers an die Service Invocation Schnittstelle und ermöglicht zum Anderen das Erstellen deutlich generischer, beziehungsweise dynamischerer `Pläne` oder auch `Implementation Artifacts`.

Benötigte Informationen sind unter Anderem in der Endpunkt Datenbank gespeicherte Endpunkte[2] der jeweiligen `Implementation Artifacts` oder `Pläne` sowie die jeweilige Invocation-Art[3] des `Implementation Artifacts` oder `Plans`. Diese Daten können entweder aus der TOSCA Definition stammen oder zur Laufzeit vom OpenTOSCA Container zur Verfügung gestellt werden.

Darüber hinaus muss die Service Invocation Schnittstelle die Funktion bieten, weitergehende Informationen aus anderen Quellen beschaffen zu können. Ein Beispiel hierfür ist Informationen über eine bestimmte Operation eines `Implementation Artifacts`, aus der WSDL eines SOAP-Web-Services zu beschaffen.

Unterstützung verschiedener Invocation-Arten

Die Service Invocation Schnittstelle muss es ermöglichen, die sich in einer `CSAR` (siehe Grundlagen TOSCA, 2.2) befindlichen und in der zugehörigen TOSCA definierten `Implementation Artifacts` oder `Pläne` generisch aufrufbar zu machen.

Dies Umzusetzen setzt eine große Vielfalt an unterstützten Protokollen, Standards, Datenformaten usw. voraus. Aufgrund dessen muss die Service Invocation Schnittstelle so konzipiert sein, dass sie sich bezüglich ihrer unterstützten Invocation-Arten erweitern lässt.

Übergabe der Input-Parameter als Key/Value-Paare und XML-Document

Die für den letztendlichen Aufruf der `Implementation Artifacts` benötigten Parameter müssen der Service Invocation Schnittstelle als Key/Value-

[2] Endpunkte sind Adressen, unter der die, durch die IA-Engine deployten, Implementation Artifacts oder die, durch die Plan-Engine deployten, Pläne erreichbar sind.
[3] Die Invocation-Art beschreibt, welches Protokoll, Datenformat usw. zum Aufruf des Implementation Artifacts oder Plans nötig ist.

Paare (Schlüssel/Wert Paare) sowie als XML-Document übergeben werden können. Dies stellt zum Einen eine festgelegte und somit bekannte Übergabeform für die Parameter dar und ermöglicht zum Anderen eine einfache Bearbeitung dieser.

Verwendung des Instance Data Services

Die Service Invocation Schnittstelle muss mit dem Instance Data Service kommunizieren können.

Die Instanzdatenhaltung verfügt unter Anderem über eine Web Service Schnittstelle, die es `Plänen` und `Implementation Artifacts` ermöglicht, dort Instanz-Daten ablegen zu können. Dadurch können für einen späteren Gebrauch benötigte Daten, für andere `Pläne`, `Implementation Artifacts` und der Service Invocation Schnittstelle, zugänglich gemacht werden. Dies ermöglicht der Service Invocation Schnittstelle das Aktualisieren von Input-Parametern durch aktuellere, in dem Instance-Data-Service abgelegte, Werte und dadurch zum Beispiel den `Plänen` wiederrum eine deutlich flexiblere und effektivere Verwendung.

Anbindung an Container

Die Service Invocation Schnittstelle muss Teil des OpenTOSCA Containers (siehe 2.4) werden. Sie muss dementsprechend so implementiert sein, dass sie auf bestehende Komponenten des Containers zugreifen kann und beim Starten des Containers automatisch mit startet.

Dies bedeutet konkret, dass die Service Invocation Schnittstelle mit Hilfe von OSGi (siehe 2.3) oder einer dazu kompatiblen Technik implementiert werden muss, da sie ansonsten keinen Zugriff auf die vorhandenen und als OSGi Services implementierten Komponenten hätte.

3.2 Nichtfunktionale Anforderungen

In diesem Kapitel werden die nichtfunktionalen Anforderungen an die Service Invocation Schnittstelle vorgestellt. Diese Anforderungen stellen wichtige Eigenschaften der Service Invocation Schnittstelle dar und müssen bei der Entwicklung berücksichtigt werden.

Einfache Erweiterbarkeit

Die Service Invocation Schnittstelle muss so konstruiert sein, dass sie sich ohne größere Umbauarbeiten oder Änderungen der Architektur in ihrem Funktionsumfang erweitern lässt. Diese Erweiterungen beziehen sich dabei zum Einen auf die Möglichkeiten zum Aufrufen verschiedenster `Implementation Artifacts` und `Pläne` und zum Anderen auf die Möglichkeiten zum Aufrufen der Service Invocation Schnittstelle selber.

Darüber hinaus müssen aber auch weitere funktionserweiternde Komponenten, wie beispielsweise eine Logging-Komponente, einfach in die Service Invocation Schnittstelle integriert werden können.

Leistungsfähigkeit

Die Service Invocation Schnittstelle muss fähig sein, mehrere Aufrufe, auch verschiedenster Aufrufer, gleichzeitig bearbeiten zu können. Diese parallele Bearbeitung darf dabei jedoch, seitens der Service Invocation Schnittstelle, keine deutlich merkbaren Leistungseinbußen hinsichtlich der Bearbeitungsdauer verursachen.

Flexibilität

Aufgrund der Möglichkeit der Service Invocation Schnittstelle des Aufrufens verschiedenster `Implementation Artifacts`, ist eine Vorhersage über die verschiedenen von den aufgerufenen `Implementation Artifacts` zurückkommenden Datenformate nicht machbar. Die Service Invocation Schnittstelle muss deshalb flexibel im Umgang mit verschiedenen Datenformaten sein und eine große Anzahl an Datenformaten unterstützen und damit umgehen können.

4 Konzept & Architektur

In diesem Kapitel wird das entwickelte Konzept, sowie die Architektur der Service Invocation Schnittstelle veranschaulicht und erläutert. Zuerst werden, aufgrund der im vorherigen Kapitel formulierten Anforderungen, getroffene Entscheidungen dargelegt und begründet. Darauf folgend wird die Architektur der Service Invocation Schnittstelle beschrieben und anschließend Möglichkeiten des Lösungskonzepts aufgezeigt.

4.1 Entwurfsentscheidungen

Dieser Abschnitt legt wichtige Entscheidungen betreffend der Konzeption und Implementierung der Service Invocation Schnittstelle dar. Diese sind zum Einen Entscheidungen bezüglich genutzten Technologien und zum Anderen Entscheidungen bezüglich Interfaces von Komponenten, also der konkreten Implementierung.

Eine der zentralen Entscheidungen bei der Konzeption der Service Invocation Schnittstelle ist die Frage, mit welcher Art von Integrationstechnik sie realisiert werden sollte. Mit Hilfe eines fertigen ESB-Systems, einem Integrations-Frameworks oder einer kompletten Eigenentwicklung.

Abbildung 5 zeigt eine Übersicht der drei Möglichkeiten sowie eine Einschätzung der Verwendbarkeit dieser bezüglich der Integration der Service Invocation Schnittstelle in den OpenTOSCA Container und der damit verbundenen Komplexität. Im Folgenden wird dies detailliert erklärt.

Für die Wahl eines ESBs spricht, dass es sich dabei um ein mächtiges und je nach Wahl des ESBs, für seine Anwendungszwecke auch um ein bewehrtes und ausgereiftes Softwareprodukt handelt, dass es nur noch zu konfigurieren gilt. Einige der ESB Produkte sind allerdings auf ein bestimmtes Technologiegebiet

spezialisiert und eignen sich daher nicht für die Umsetzung der Service Invocation Schnittstelle. Der auf Apache Axis2 [13] basierte Apache Synapse ESB [5] zum Beispiel ist vorwiegend auf eine Web Service Umgebung spezialisiert [44]. Ohne Anpassungen bestehender (OSGi) Komponenten des OpenTOSCA Containers würde dies jedoch möglicherweise zu Problemen führen. Da die Komponenten per OSGi implementiert sind und keine Web Service Schnittstelle, wie zum Beispiel für SOAP bieten, können sie von Synapse ESB nicht direkt aufgerufen und damit nicht benutzt werden. Eine Möglichkeit zur Lösung des Problems ist das Umwandeln der Komponenten in Web Services oder das Erstellen einer Web Service Schnittstelle zum Aufruf der Komponenten. Dies würde allerdings einen erheblichen Umbauaufwand verursachen. Des Weiteren sind ESBs im Vergleich zu den beiden anderen Möglichkeiten zur Realisierung der Service Invocation Schnittstelle die deutlich schwergewichtigste und komplexeste Alternative und bringen teilweise nicht benötigte und damit überflüssige Funktionen mit sich [43].

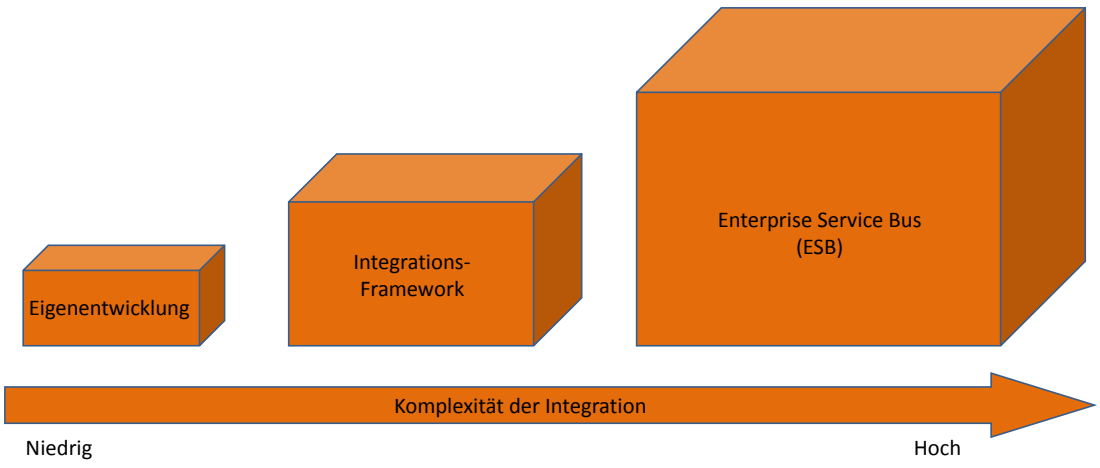

Abbildung 5: Möglichkeiten der Integrationstechnologien (nach [43] S.3)

Diesen Nachteil der überflüssigen Funktionen, hat die Entwicklung eines kompletten eigenen Systems, ohne Integrationswerkzeuge, nicht zur Folge. Es kann hier eine ganz flexible und komplett nach den konkreten Bedürfnissen angepasste Lösung realisiert werden. Allerdings bringt diese Alternative entweder, bei einer kompletten Eigenentwicklung mit Standard Java API, einen großen und noch wichtiger, vor allem größtenteils unnötigen Implementierungsaufwand mit sich. Für viele Kommunikationsstandards und Protokolle gibt es bereits fertige Bibliotheken die verwendet werden können und auch sollten. Denn eine selbst entwickelte Lösung wird mit einer relativen hohen Wahrscheinlichkeit nicht so ausgereift, wie eine bereits bewehrte Fremdbibliothek sein. Darauf zu verzichten resultiert also in überflüssiger Arbeit. Oder aber es besteht die Gefahr, bei der Verwendung einer Vielzahl solcher Bibliotheken, den Überblick über die Abhängigkeiten und den genauen Funktionalitäten der einzelnen Bibliotheken zu verlieren oder von Kompatibilitätsproblemen bei vielen Fremdbibliotheken verschiedenster Hersteller.

Ein mögliches Problem durch verschiedene Quellen der Bibliotheken hat die dritte Variante zur Realisierung der Service Invocation Schnittstelle nicht. Die Verwendung eines Integrations-Frameworks vereint die bisher bei den anderen beiden Realisierungsmethoden erwähnten Vorteile, ohne jedoch die vorherigen Nachteile mit sich zu bringen. So bieten Integrations-Frameworks zwar, ähnlich wie ESBs, Unterstützung für eine große Anzahl von Standards und Protokollen, lassen sich jedoch aufgrund dessen, dass es sich dabei de facto um kein eigenständiges Produkt, sondern Java Bibliotheken handelt, in bestehende Projekte leicht integrieren und in der Funktionalität einfach ergänzen. Zudem lassen Integrations-Frameworks, aufgrund ihres architektonischen Aufbaus aus einzelnen Funktionskomponenten, eine gleichermaßen leichtgewichtige wie flexible und individuell anpassbare Lösung des Problems zu. Des Weiteren besteht der große Vorteil eines Produktes aus einer Hand, was unter anderem aufeinander abgestimmte und damit hochgradig kompatible Komponenten, sowie eine einheitliche Art der Verwendung dieser, mit sich bringt.

Aufgrund der oben dargelegten Argumentation, wird die Möglichkeit des Integrationsframeworks in dieser Arbeit in Form von Apache Camel [2] gewählt.

Grund dafür ist unter Anderem eine große Anzahl an Komponenten (über 125, siehe [7]), welche viele verschiedene Techniken und Standards unterstützen. Beispielweise gibt es eine FTP-Komponente [9] zum versenden und empfangen von Dateien per FTP oder eine CXF-Komponente [8] zur Integration von SOAP Web Services. Weiterhin bietet Camel die Möglichkeit eigene Komponenten zur Funktionserweiterung unkompliziert entwickeln und benutzen zu können. Ein weiterer Grund für die Wahl von Camel ist die Möglichkeit, anders wie zum Beispiel bei den beiden alternativen Integrations-Frameworks Spring Integration [39] und Mule ESB [27], neben einer XML DSL auch eine Java DSL nutzen zu können [43]. Dies bringt die üblichen Vorteile einer IDE[4], wie zum Beispiel die automatische Vervollständigung von Code und damit ein effizienteres Arbeiten mit sich. Des Weiteren unterstützt Camel, im Gegensatz zu Mule ESB, OSGi und ermöglicht somit eine einfache Integration der Service Invocation Schnittstelle in den bestehenden OpenTOSCA Container [43]. Darüber hinaus ist Camel das in Apache ServiceMix [4], einem auf OSGi basierenden ESB Container, genutzte Integrations-Framework. Dies ermöglicht es, falls später ESB Funktionalitäten benötigt oder gewünscht werden, OpenTOSCA in Apache ServiceMix zu betreiben.

Um eine einfache Ergänzung des Funktionsumfangs der Service Invocation Schnittstelle hinsichtlich der unterstützten Invocation-Arten zu ermöglichen, wird ein Erweiterungssystem benötigt. Dies wird in Form eines Plug-In-Systems realisiert.

In der IA-Engine und Plan-Engine (siehe Grundlagen OpenTOSCA, 2.4) hat sich die Technik der OSGi Declarative Services (siehe Grundlagen OSGi, 2.3) zur Plug-in Verwaltung bewährt. Aufgrund dessen, wird die Service Invocation Schnittstelle in ihrem Kern ebenfalls als Engine, zur Verwaltung der durch

[4] Integrierte Entwicklungsumgebung (*integrated development environment).*

Declarative Services realisierten Plug-ins, konzipiert. Diese Beibehaltung einer einheitlichen Mechanik führt außerdem zu einer einheitlichen Architektur des Containers und damit zu einer höheren Wartbarkeit.

Eine weitere zu entscheidende Frage ist, ob für jede Invocation-Art eines `Implementation Artifacts` eine eigene Endpoint-Datenbank erstellt wird. Also beispielsweise eine eigene Datenbank für SOAP Web Service `Implementation Artifacts` und eine eigene Datenbank für REST Web Service `Implementation Artifacts`. Anhand der Datenbank kann dann von der Service Invocation Schnittstelle erkannt werden, wie das jeweils darin befindliche `Implementation Artifact` aufgerufen werden muss. Allerdings wird das Deployment der `Implementation Artifacts` und damit auch das Abspeichern der Endpoints durch die IA-Engine und deren Plug-ins erledigt. Die IA-Engine unterscheidet die `Implementation Artifacts` jedoch nicht anhand deren Invocation-Art (siehe Kapitel 3.1), wie die SI-Engine, sondern anhand ihrer `Implementation Artifact-`Art. Beispielsweise werden ein SOAP Web Service `Implementation Artifact` und ein REST Web Service `Implementation Artifact`, sofern beide als Web Application Archive (WAR) gepackt sind, zur selben `Implementation Artifact`-Art zugeordnet und damit auch durch das gleiche IA-Plug-in deployt[5]. Zum Aufrufen dieser beiden `Implementation Artifacts` sind jedoch zwei verschiedene SI-Schnittstellen Plug-ins notwendig: ein Rest- und ein SOAP-fähiges Plug-in. Zusammengefasst unterscheiden IA-Engine und SI-Engine `Implementation Artifacts` unterschiedlich (vgl. Invocation-Art und `Implementation Artifact-`Art). Daher bringt die Trennung der Endpoint-Datenbank keine Vorteile. Es können aber äquivalent zu Endpunkt-Informationen für die IA-Engine, auch die Invocation-Art eines `Implementation Artifacts` innerhalb der TOSCA Definition vermerkt

[5] Vereinfachte Darstellung. Die Wahl des IA-Plug-ins wird nicht ausschließlich durch die IA-Art bestimmt. IAs können Anforderungen an ein Plug-in stellen, die von diesem erfüllt werden müssen. Für weitergehende Informationen wird auf die TOSCA Spezifikation ([25] Kapitel 3.4 „Requirements & Capabilities“) verwiesen.

werden. Diese wird von der SI-Engine ausgelesen und damit das passende Plug-in bestimmt. Aufgrund dessen werden die Endpoints in einer gemeinsamen Datenbank gespeichert.

Weiterhin ist das Design des Interfaces der SI-Plug-ins, eine wichtige zu treffende Entscheidung. Die Plug-ins benötigen zum Aufrufen von `Implementation Artifacts` oder `Plänen` deren Endpoint, die aufzurufende Operation und die zu übergebenden Daten. Außerdem werden im Falle von asynchronen Aufrufen, wie sie zum Beispiel bei SOAP Web Services vorkommen, eine Correlation ID zur Bestimmung der zum Aufruf gehörenden Antwort benötigt. Das Design des Interfaces lässt sich auf verschiedene Arten umsetzen. Listing 1 und Listing 2 zeigen die zwei zur Wahl stehenden Alternativen.

```
public Object invoke(String endpoint, String operationName,
     HashMap<String, String> params, String correlationID);
```

Listing 1: Plug-in Interface Alternative 1

In der ersten Alternative (Listing 1) erfolgt die Übergabe der oben genannten Parameter mittels Standard Java Objekten wie Strings oder einer Map. Dadurch wird ein klar definiertes Interface festgelegt. Als Rückgabewert ist nur Object möglich, weil dies die Rückgabe verschiedenster Datentypen ermöglicht und dadurch die Flexibilität erhöht. Allerdings muss die Correlation ID ebenfalls mit zurück gegeben werden. Jedoch sind zwei einzelne Rückgabewerte in Java nicht möglich. Es muss dafür extra ein eigenes Objekt erstellt werden, welches den eigentlichen Rückgabewert plus die Correlation ID beinhaltet. Ein weiterer Kritikpunkt dieser Alternative ist die starre Form des Interfaces. Eine einfache Erweiterung dieses ist nicht möglich. Muss das Interface beispielsweise aufgrund geänderter Anforderungen angepasst werden, werden alle bis dahin bestehenden Plug-ins fehlerhaft und müssen angepasst werden. Eine solche Änderung

kann zum Beispiel ein weiterer benötigter (oder sogar optionaler) Parameter sein. Auch ist mit dieser Alternative die Übergabe der an das `Implementation Artifact` oder den `Plan` gerichteten Daten nur in Form einer HashMap [34] möglich. Dies ist nicht optimal, da zum Beispiel die Plan Invocation Engine ihre zum Aufruf eines `Plans` erzeugte Nachricht in Form eines org.w3c.dom.Document [33] Objekts übergibt. Dieses muss, damit es dem Plug-in übergeben werden kann, zuerst in eine HashMap umgewandelt werden. Allerdings muss zum Aufruf eines SOAP Web Service diese HashMap anschließend wieder zurück in eine SOAP Nachricht (also XML) transferiert werden.

```
public Exchange invoke(Exchange exchange);
```

Listing 2: Plug-in Interface Alternative 2

Das zweite zur Wahl stehende Interface (Listing 2) nutzt zur Übergabe der Parameter das zu Camel gehörende Exchange Objekt (siehe Grundlagen Camel, 2.7). Das Exchange Objekt wird als Container für Message Objekte mit beliebigem Inhalt genutzt. Im Gegensatz zur ersten Alternative, stellt Alternative 2 somit kein klar definiertes Interface bezüglich der zur Übergabe der Parameter genutzten Objekte dar. Das Exchange Objekt ermöglicht jedoch eine flexible Entwicklung sowie Nutzung der Plug-ins, also genau das, was für die Service Invocation Schnittstelle notwendig ist. In die Header der Message des Exchange Objekts können die Parameter wie Endpoint, Name der Operation usw. abgelegt werden, wohingegen die zu übergebenden Daten, in welcher Form auch immer, in den Body gelegt werden können (detailliertere Erläuterungen hierzu folgen in Kapitel 4.3). Dies ermöglicht zugleich auch eine einfache Erweiterung von Plug-ins ohne das Interface dafür anpassen zu müssen. Benötigt ein Plug-in beispielsweise weitere Daten, können diese einfach als neuer Header innerhalb des Message Objekts definiert werden. Plug-ins, die von den neuen Headern wissen und darauf vorbereitet sind, können diese nutzen. Alte Plug-ins jedoch, ohne

Wissen von den neuen Headern, funktionieren auch weiterhin problemlos. Ein weiterer Grund der für die Verwendung des Exchange Objekts und damit für Alternative 2 spricht ist die Tatsache, dass dieses ohnehin von Camel selbst als Container zum Transport von Messages genutzt wird und damit wiederum eine einheitliche Mechanik gewährleistet wird.

Aufgrund der dargelegten Argumentation, wird Alternative 2 (Listing 2) gewählt. Detailliertere Erläuterungen zur Verwendung des Exchange Objekts, wie zum Beispiel die Nutzung der Header, folgen in Kapitel 4.3.

4.2 Architektur

In diesem Kapitel wird die Architektur der Service Invocation Schnittstelle veranschaulicht und erläutert. Da sie Teil des OpenTOSCA Containers ist, wird sie entsprechend im Rahmen dessen dargestellt.

Abbildung 6 zeigt die Architektur der Service Invocation Schnittstelle. Eine der Hauptkomponenten der Service Invocation Schnittstelle ist, neben den bereits in Kapitel 2.4 vorgestellten Komponenten TOSCA Engine, Endpoint Service und Instance Data Service, die SI-Engine (siehe Kapitel 5.4). Sie bildet, durch ihre Verbindung zu anderen wichtigen Komponenten innerhalb des OpenTOSCA Containers, die zentrale Einheit der Service Invocation Schnittstelle. Weiterhin sind auch die Service Invocation APIs (siehe Kapitel 5.2 sowie 5.3), auf die später noch genauer eingegangen wird, ein wichtiger Bestandteil der Service Invocation Schnittstelle.

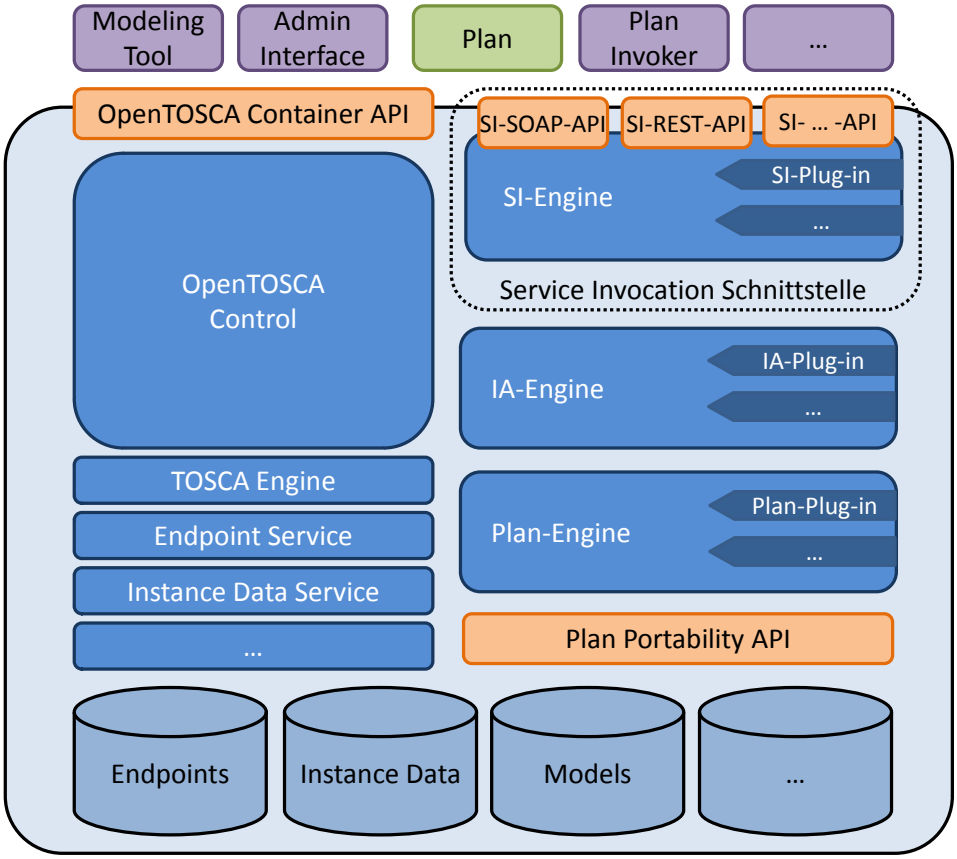

Abbildung 6: Architektur des OpenTOSCA Containers mit Service Invocation Schnittstelle

Die SI-Engine hat unter Anderem die Aufgabe, das zum Invoke-Request passende `Implementation Artifact`, sowie weitere benötigte Informationen, wie die Invocation-Art des `Implementation Artifacts`, mittels der TOSCA Engine zu bestimmen (äquivalent für `Pläne`). Weiterhin nutzt die SI-Engine den Endpoint Service zur Beschaffung der Endpunkte der `Implementation Artifacts` beziehungsweise der `Pläne`. Auch bietet eine Schnittstelle zum Instance Data Service der SI-Engine die Möglichkeit, dort von beispielsweise bereits ausgeführten `Plänen` gespeicherte Instanzdaten abzurufen und zu verwenden.

Die Aufrufe der `Implementation Artifacts` oder `Pläne` geschehen dann durch die so genannten SI-Plug-ins (siehe Kapitel 5.5). Diese Plug-ins bieten da-

bei jeweils Unterstützung für verschiedene Protokolle und Standards (Invocation-Arten), wie zum Beispiel das Versenden einer Message mittels SOAP über HTTP oder der Aufruf eines OSGi `Implementation Artifacts` und können bei Bedarf auch während der Laufzeit hinzugefügt und gestartet werden. Die Verwaltung der SI-Plug-ins übernimmt dabei, durch ein extra dafür konzipiertes OSGi basiertes Plug-in-System, ebenfalls die SI-Engine.

Die Funktionalität der Service Invocation Schnittstelle wird durch verschiedene Service Invocation APIs, wie z.B. eine SI-SOAP-API oder eine SI-REST-API zur Verfügung gestellt. Durch sie können andere Komponenten, Anwendungen, `Implementation Artifacts` und vor allem auch `Pläne` die Service Invocation Schnittstelle nutzen. Beispielsweise möchte ein `Plan` eine in TOSCA deklarierte Management Operation aufrufen, welche durch ein SOAP/HTTP `Implementation Artifact` implementiert ist. Dafür schickt der `Plan` der Service Invocation SOAP API eine Nachricht, welche alle benötigten Informationen enthält. Der Inhalt dieser Nachricht wird der SI-Engine weitergegeben. Dort werden weitere Daten wie z.B. der Endpunkt besorgt und anschließend alle Informationen an ein passendes Plug-in weitergegeben, wo der Aufruf schließlich ausgeführt wird. Der genaue Bearbeitungsablauf wird im folgenden Kapitel erläutert.

4.3 Beschreibung des gewählten Lösungskonzeptes

In diesem Kapitel wird detailliert auf das Konzept der Service Invocation Schnittstelle eingegangen und Möglichkeiten des Konzepts aufgezeigt.

Abbildung 7 zeigt den Aufbau der Komponenten der Service Invocation Schnittstelle samt deren Abhängigkeiten und Verbindungen sowie die Schnittstellen zu Services wie `Plänen` und `Implementation Artifacts`. Außerdem wird die Nutzung des Camel Exchange Objekts innerhalb der Service Invocation Schnittstelle dargestellt. Die Abbildung zeigt beispielhaft die Service Invocation SOAP API zum Aufruf der Service Invocation Schnittstelle mittels SOAP Nachrichten, wie sie zum Beispiel durch BPEL `Pläne` getätigt werden.

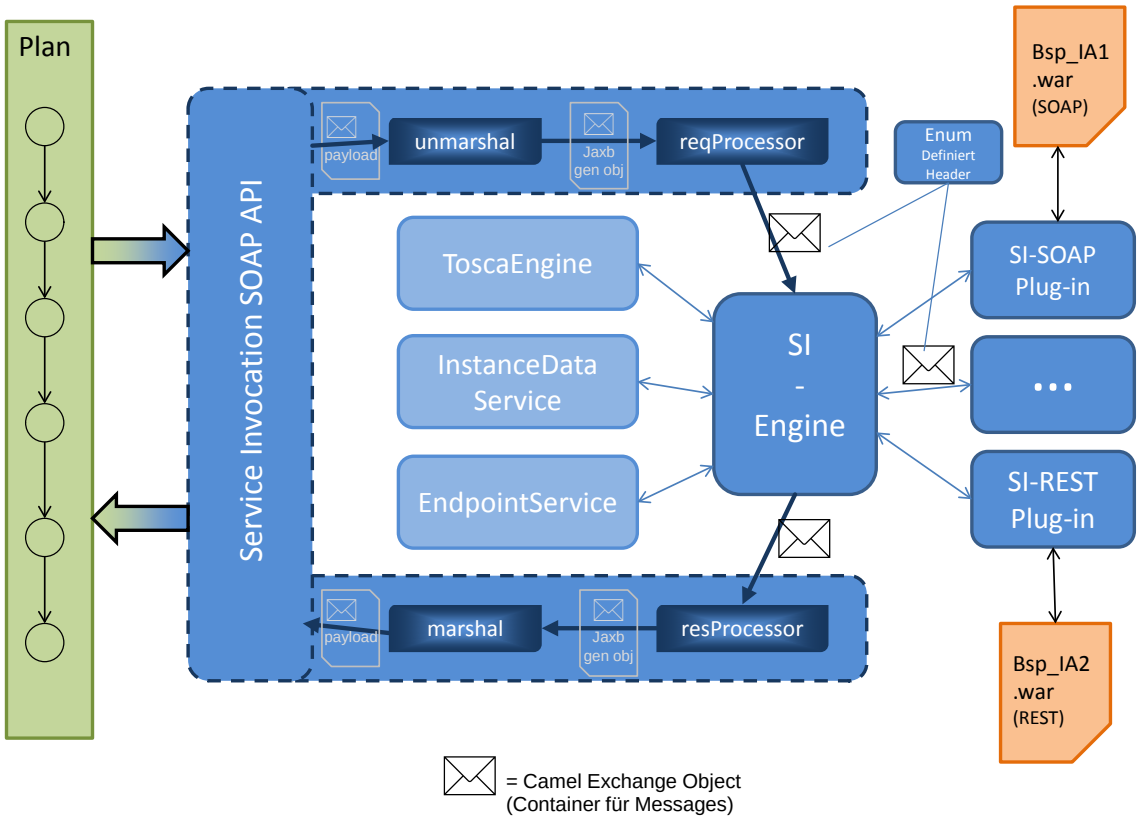

Abbildung 7: Konzeptioneller Aufbau der Komponenten

Abbildung 7 zeigt, dass die SI-Engine die zentrale Komponente innerhalb der Service Invocation Schnittstelle darstellt. Sie ermöglicht zum Einen die Kommunikation mit bestehenden und benötigten Komponenten des OpenTOSCA Containers (TOSCA Engine, Endpoint Service, Instance Data Service) und stellt zum Anderen die Verbindung zwischen den Service Invocation APIs und den verschiedenen SI-Plug-ins her (siehe auch Abbildung 8), übernimmt somit das Routing der Nachrichten innerhalb der Service Invocation Schnittstelle. Abbildung 8 verdeutlicht dies durch die Darstellung der Komponenten der Service Invocation Schnittstelle mittels eines Schichtendiagramms.

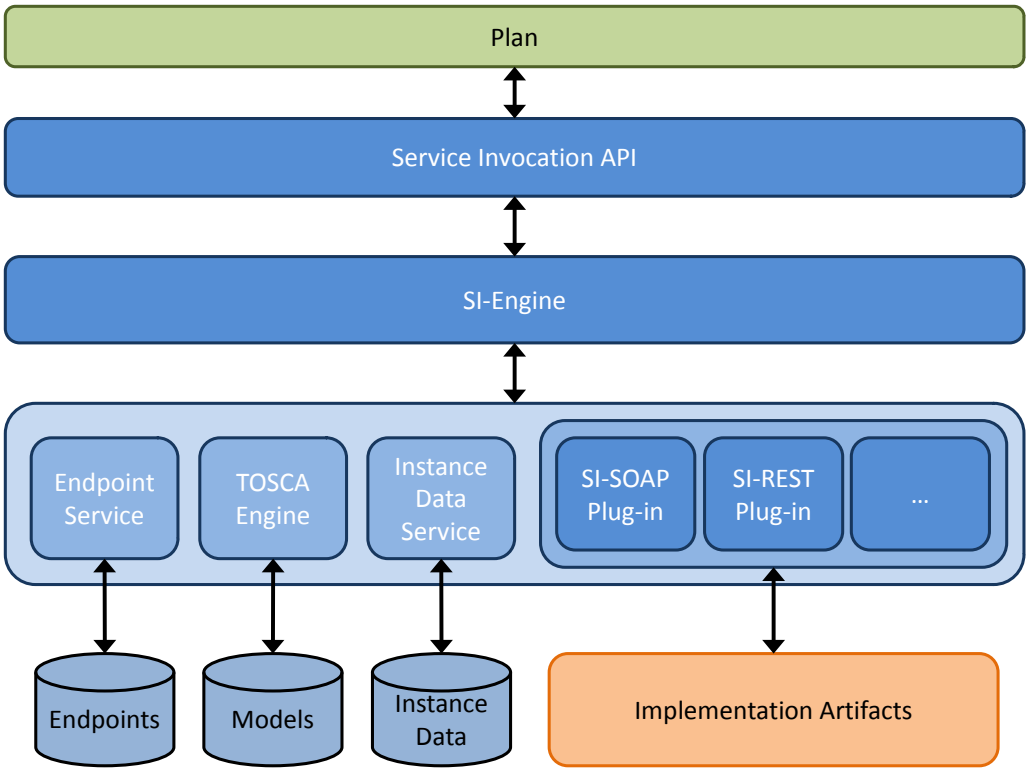

Abbildung 8: Service Invocation Schnittstelle als Schichtendiagramm

Abbildung 7 zeigt des Weiteren die Verwendung des Exchange Objekts. Innerhalb der Service Invocation Schnittstelle und deren Komponenten wird zur Übergabe der Daten und Nachrichten das von Camel bereitgestellte Exchange

Objekt genutzt. Die Kommunikation zwischen SI-Engine (und damit der Service Invocation Schnittstelle) und den restlichen Komponenten des OpenTOSCA Containers dagegen erfolgt per Übergabe einzelner Parameter. Dies hat unter Anderem den Vorteil, dass innerhalb der Service Invocation Schnittstelle zwar die Vorzüge des Exchange Objekts genutzt werden können, die restlichen Komponenten des Containers jedoch von Camel und dadurch auch von der Service Invocation Schnittstelle unabhängig bleiben und zudem nicht angepasst werden müssen.

Weiterhin ist in Abbildung 7 ein Enum dargestellt. Dieses Enum (siehe Kapitel 5.1) ist fest definiert und spezifiziert die Keys der Header des Message Objekts. Damit wird sichergestellt, dass die von den Service Invocation APIs an die SI-Engine und die von der SI-Engine an die SI-Plug-ins übergebenen Messages einen einheitlichen Aufbau bezüglich den in den Headern befindlichen Parametern haben. Dadurch können benötigten Informationen, wie zum Beispiel `CsarID` oder `ServiceTemplateID`, komponentenübergreifend identisch und zuverlässig ausgelesen werden. Man könnte zu diesem Zweck auch ein Objekt mit Feldern für die benötigten Parameter (`CsarID`, `ServiceTemplateID`, `OperationName`, ...) anlegen. Die HashMap als Objekt für die Header „simuliert“ zusammen mit dem Enum so gesehen ein solches Objekt zur Übergabe der erforderlichen Parameter. Allerdings ist die Lösung mit dem Enum, welches die Keys der benötigten Parameter in der Header-HashMap festlegt, flexibler und zudem so von Camel vorgesehen.

Abbildung 9 zeigt die Reihenfolge eines beispielhaften Bearbeitungsablaufs innerhalb der Service Invocation Schnittstelle vom Aufruf durch einen `Plan`, über die Bearbeitung der Anfrage in der SI-Engine und den Aufruf des `Implementation Artifacts`, bis zur Antwort zurück an den `Plan`.

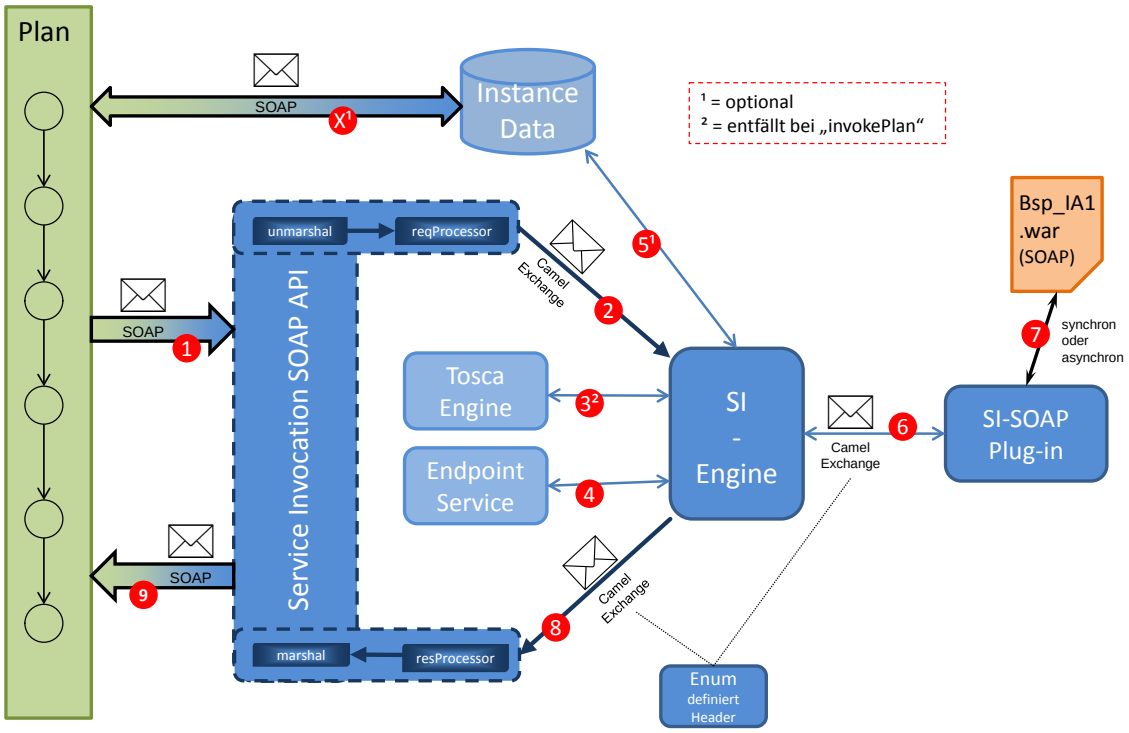

Abbildung 9: Bearbeitungsablauf zum Aufruf eines Services

Zuerst ruft der `Plan` die Service Invocation SOAP API mit den zum Aufruf des `Implementation Artifacts` benötigten Daten per SOAP Message auf (siehe 1). Die Service Invocation SOAP API unmarshallt diese SOAP Message, liest die übergebenen Daten aus, schreibt diese in ein Exchange Objekt und leitet dieses an die SI-Engine weiter (siehe 2). Durch das definierte Enum sind die im Header befindlichen Parameter zur Bestimmung des `Implementation Artifacts` in einem für die SI-Engine verständlichen Format.

Die SI-Engine holt sich daraufhin (siehe 3), mittels der TOSCA Engine und den bekommenen Daten, Informationen über das aufzurufende `Implementation Artifact`, wie Namen und Invocation-Art, ein. Außerdem bestimmt sie das message exchange pattern (MEP) [41] des `Implementation Artifacts`, anhand den in der TOSCA Definition für die aufzurufende Operation angegebenen Werten.

Im Rahmen der Service Invocation Schnittstelle werden nur die beiden an die WSDL 2.0[6] angelehnten In-Out sowie In-Only MEPs unterschieden. Das In-Out pattern gibt an, dass bei einer eingehenden Nachricht (Input) eine Antwort gegeben werden muss (Output). Listing 3 zeigt ein Beispiel des In-Out pattern in einer TOSCA Definition. Demgegenüber gibt das In-Only pattern an, dass auf eine Anfrage keine Antwort gesendet wird. Ein Beispiel hierfür wird in Listing 4 dargestellt.

```
01 <Operation name="createDB">
02     <InputParameters>
03           <InputParameter name="Size" type="xs:string"/>
04           <InputParameter name="Host" type="xs:string"/>
05           <InputParameter name="User" type="xs:string"/>
06           <InputParameter name="Password" type="xs:string"/>
07     </InputParameters>
08     <OutputParameters>
09           <OutputParameter name="URL" type="xs:string"/>
10     </OutputParameters>
11 </Operation>
```

Listing 3: Beispiel In-Out Pattern

```
01 <Operation name="deleteDB">
02     <InputParameters>
03           <InputParameter name="URL" type="xs:string"/>
04           <InputParameter name="User" type="xs:string"/>
05           <InputParameter name="Password" type="xs:string"/>
06     </InputParameters>
07 </Operation>
```

Listing 4: Beispiel In-Only Pattern

Anschließend ermittelt die SI-Engine anhand der zusätzlich gewonnenen Informationen den Endpunkt des `Implementation Artifacts` mit Hilfe des Endpoint Services (siehe 4).

[6] Web Services Description Language Version 2.0 (http://www.w3.org/TR/wsdl20/)

Optional, falls beim Aufruf der Service Invocation SOAP API eine ID einer CSAR-Instanz mitgegeben wurde, prüft die SI-Engine, ob für diese CSAR-Instanz gespeicherte Instanz Daten vorhanden sind und verwendet (eine genauere Erklärung folgt später) sie in diesem Falle (siehe 5). Die Instanz Daten können dabei von einem beliebigen Plan jederzeit vorher abgespeichert worden sein (siehe X in Abbildung 9).

Anschließend bestimmt die SI-Engine anhand der Invocation-Art des `Implementation Artifacts` das dazu passende SI-Plug-in und sendet diesem alle gesammelten Daten wiederum per Exchange Objekt zu (siehe 6).

Das Plug-in erstellt dann seinerseits aus den bekommenen Informationen eine Request-Message und sendet diese an das `Implementation Artifact` (siehe 7). Der Aufruf kann dabei, je nach Implementierung des `Implementation Artifacts`, synchron oder auch asynchron erfolgen und liegt in der Verantwortung des jeweiligen Plug-ins.

Die Antwort des `Implementation Artifacts` wird dann über das SI-Plug-in an die SI-Engine und von dort weiter an die Service Invocation SOAP API geleitet (siehe 8). Dort wird sie unmarshallt und in eine SOAP Message umgewandelt. Zum Schluss sendet die Service Invocation API schließlich diese SOAP Message zurück an den `Plan` (siehe 9), der nun mit den, von dem `Implementation Artifact`, erhaltenen Daten weiterarbeiten kann.

In diesem Beispiel erfolgte der Aufruf eines als SOAP Web Service implementierten `Implementation Artifacts` durch einen `Plan`. Natürlich sind aber auch andere Aufrufszenarien möglich. Zum Beispiel der Aufruf eines als RESTful Web Service oder OSGi-Service implementierten `Implementation Artifacts`. Weiterhin besteht auch die Möglichkeit des Aufrufens eines `Plans`, initiiert durch die Plan Invocation Engine (siehe Kapitel 2.5) oder anderen `Plan`.

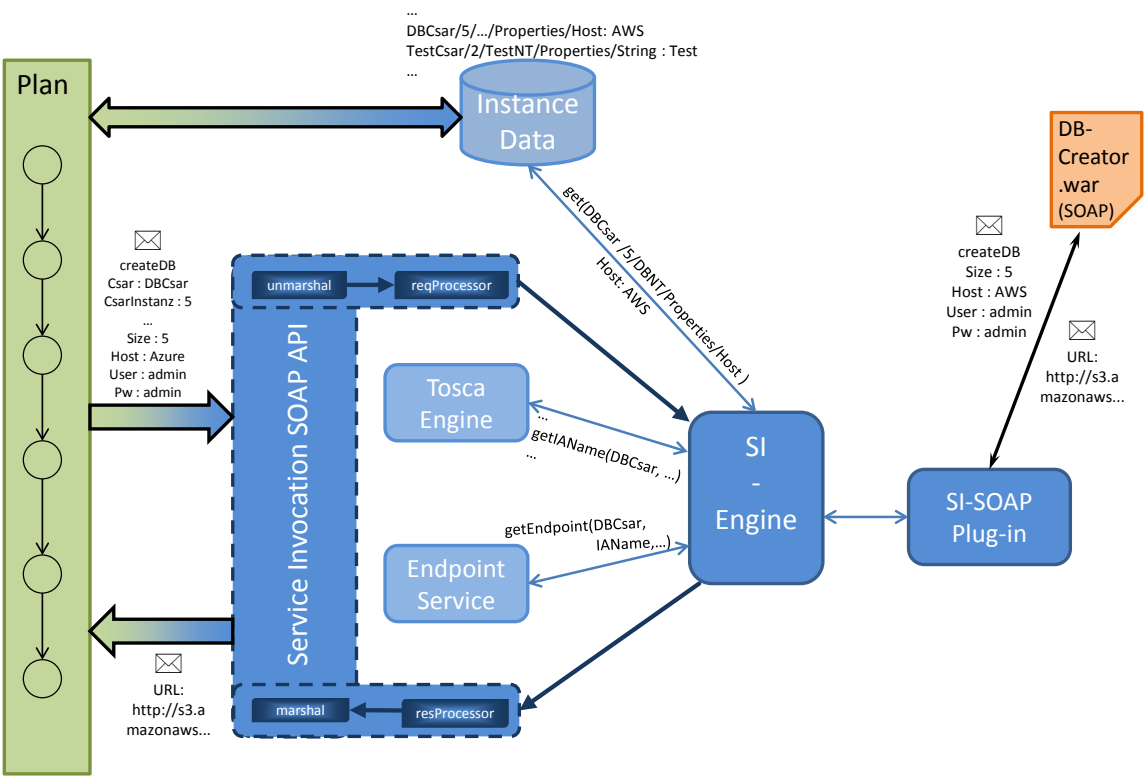

Abbildung 10: Beispielhafte Nachrichten / Aufrufe

Abbildung 10 zeigt das beschriebene Szenario - der Aufruf eines SOAP Web Service `Implementation Artifacts` durch einen `Plan` - an einem Beispiel mit konkreten Nachrichten und Werten. Die Nachrichten und Aufrufe sind innerhalb der Abbildung verkürzt dargestellt. Die kompletten Nachrichten sowie Aufrufe werden im Folgenden erläutert.

Listing 5 zeigt den ausführlichen SOAP Aufruf des `Plans` an die Service Invocation SOAP API. Ziel des Aufrufs ist es, das durch die Zeilen acht bis 18 spezifizierte `Implementation Artifact`, mit der Operation createDB (siehe Zeile 18) und den von Zeile 22 bis 40 angegebenen Parametern aufzurufen. Die Angabe von ReplyTo in Zeile 19 teilt der Service Invocation Schnittstelle mit, an welche Adresse sie den Callback mit der Antwort an den `Plan` zurück schicken

soll. Die MessageID (Zeile 20) wird von der Service Invocation Schnittstelle an den `Plan` zurück gesendet, da diese von der Workflow Engine benötigt wird, um die Antwort der Anfrage sowie der richtigen Planinstanz zuordnen zu können.

```
01 <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.
02                                 org/soap/envelope/" xmlns:
03                                 sch="http://siserver.
04                                 org/schema">
05    <soapenv:Header/>
06    <soapenv:Body>
07       <sch:invokeOperation>
08          <CsarID>DBCsar</CsarID>
09          <!--Optional:-->
10          <ServiceInstanceID>Instance5</ServiceInstanceID>
11          <ServiceTemplateIDNamespaceURI>
12             http://CsarDBCreator.org/DB/
13          </ServiceTemplateIDNamespaceURI>
14          <ServiceTemplateIDLocalPart>
15             DBCreator_ServiceTemplate
16          </ServiceTemplateIDLocalPart>
17          <NodeTemplateID>DB_NodeTemplate</NodeTemplateID>
18          <OperationName>createDB</OperationName>
19          <ReplyTo>http://localhost:1337/callback</ReplyTo>
20          <MessageID>A7ZD70AH</MessageID>
21          <!--Optional:-->
22          <Params>
23             <!--1 or more repetitions:-->
24             <Param>
25                <key>Size</key>
26                <value>5</value>
27             </Param>
28             <Param>
29                <key>Host</key>
30                <value>Azure</value>
31             </Param>
32             <Param>
33                <key>User</key>
34                <value>admin</value>
35             </Param>
36             <Param>
37                <key>Password</key>
38                <value>p8ilR6N9</value>
39             </Param>
40          </Params>
41       </sch:invokeOperation>
42    </soapenv:Body>
43 </soapenv:Envelope>
```

Listing 5: SOAP Nachricht eines Plans an die Service Invocation Schnittstelle zur Erstellung einer Datenbank

Die SOAP Nachricht wird von der Service Invocation SOAP API, wie in Abbildung 11 dargestellt, umgewandelt und an die SI-Engine weitergeleitet. Hierfür wird das in Kapitel 2.7 erläuterte Exchange Objekt genutzt, welches von Camel angeboten wird. Die für das `Implementation Artifact` bestimmten Parameter (Size, Host, User, Password) werden im Body der Message abgelegt. Die zur Bestimmung des `Implementation Artifacts` sowie zum Aufruf dessen benötigte Daten dagegen, werden im Header (mit den durch das Enum definierten Strings als Keys) des Message Objekts abgelegt. Weitere übergebene Werte wie ReplyTo oder MessageID werden ebenfalls im Header abgelegt. Diese Werte sind allerdings nicht durch das SI-Enum definiert, da sie nur von der SOAP-API selbst und nicht den anderen SI-Komponenten benötigt werden.

Header	CSARID : DBCsar SERVICEINSTANCEID_STRING : Instance5 SERVICETEMPLATEID_QNAME : {http://CsarDBCreator.org/DB/}DBCreator_ServiceTemplate NODETEMPLATEID_STRING : DB_NodeTemplate OPERATIONNAME_STRING : createDB ReplyTo : http://localhost:1337/callback MessageID : A7ZD70AH
Body	Size : 5 Host : Azure User : admin Password : p8ilR6N9

Abbildung 11: Von der Service Invocation SOAP API an die SI-Engine gesendete Message

Anschließend holt sich die SI-Engine benötigte Daten aus dem Header, bestimmt mit diesen und der Zuhilfenahme der TOSCA Engine das `Implementation Artifact` und besorgt sich mittels Endpoint Service den dazu gespeicherten Endpunkt. Dieser wird anschließend ebenfalls unter dem im SI-Enum definiertem Key (ENDPOINT_URI, siehe Kapitel 5.1) als Header des Message Objekts abgelegt.

Weiterhin überprüft die SI-Engine per Instance Data Service ob Instanz Daten für diese Anfrage gespeichert sind und holt sich diese gegebenenfalls von dort. Dies ist allerdings optional und erfolgt nur bei durch den Aufrufer angegebener ServiceInstanceID. Anschließend werden die sich im Body befindlichen Input-Parameter durch die jeweiligen Werte aus der Instanz Datenbank ersetzt und damit aktualisiert. So können beispielsweise IP Adressen einer zuvor angelegten Virtuellen Maschine genutzt werden.

Schließlich wird anhand der durch die TOSCA Engine ermittelten Invocation-Art des `Implementation Artifacts`, das dazu passende Plug-in gewählt und die inzwischen, zum Aufruf des `Implementation Artifacts`, mit allen benötigten Daten versehene Exchange Message dorthin weitergereicht.

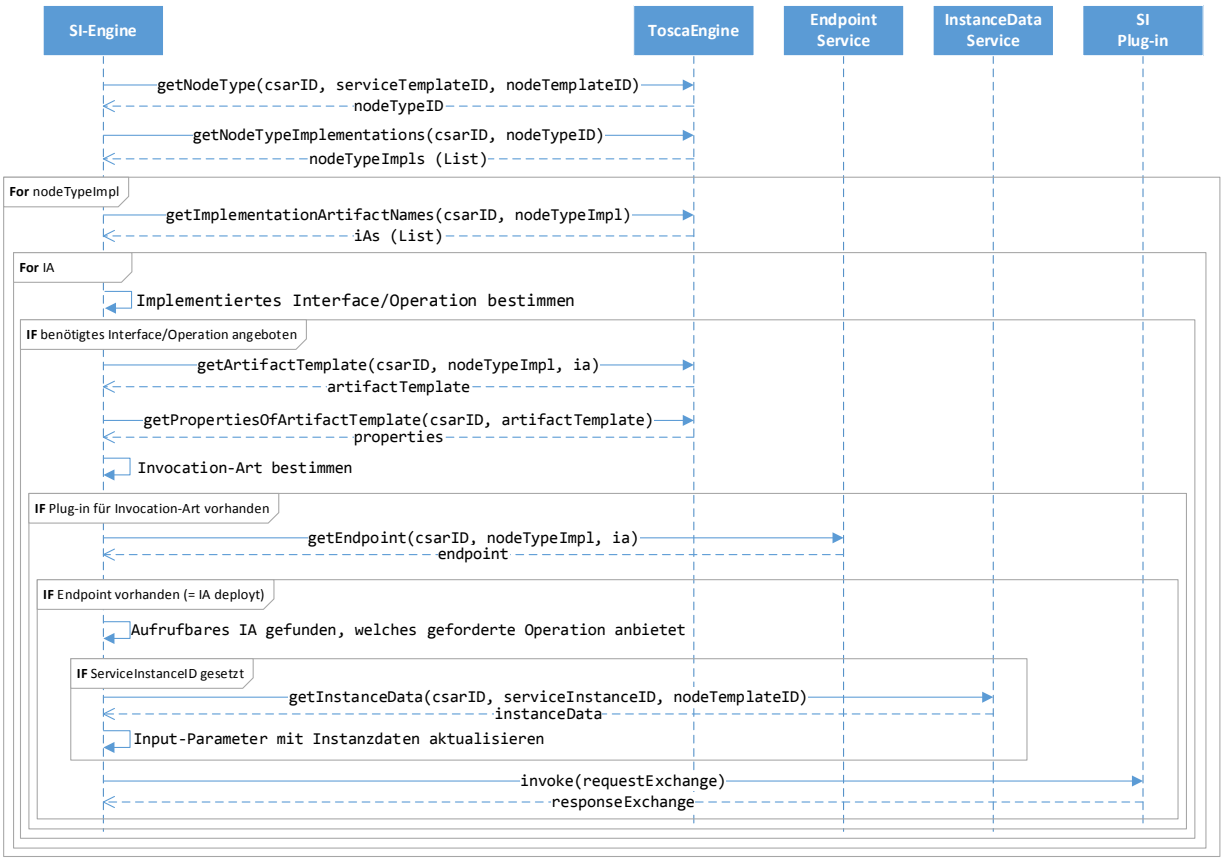

Abbildung 12: Sequenzdiagramm SI-Engine

Abbildung 12 stellt die Kommunikation der SI-Engine mit TOSCA Engine, Endpoint Service, Instance Data Service sowie eines SI-Plug-ins nochmals, mittels eines Sequenzdiagramms dar. Es zeigt unter anderem wie die SI-Engine mittels den übergebenen Parametern und der Hilfe der TOSCA Engine ein aufrufbares `Implementation Artifact` bestimmt. Dafür wird zuerst der zu den übergebenen Parametern `CsarID`, `ServiceTemplateID` und `NodeTemplateID` gehörende `NodeType` bestimmt (`NodeTypes` typisieren `NodeTemplates`). Anschließend werden für diesen `NodeType` alle entsprechenden `NodeTypeImplementations` abgefragt. Über die zurückbekommene Liste der `NodeTypeImplementations` wird anschließend iteriert und alle jeweiligen `Implementation Artifacts` ermittelt. Über diese Liste der `Implementation Artifacts` wird wiederrum iteriert und für jedes `Implementation Artifact` überprüft, ob es die geforderte Operation anbietet. Falls der Name der Operation innerhalb eines `NodeTypes` dabei eindeutig ist, langt die Angabe des Namens der Operation. Falls allerdings eine Operation nicht nur in einem Interface eines `NodeTypes` angeboten wird, muss zur eindeutigen Identifizierung zudem der Name des Interfaces angegeben werden. Wird die geforderte Operation angeboten, wird weiterhin überprüft, ob in der TOSCA Definition eine Invocation-Art spezifiziert wurde und ein dafür entsprechendes Plug-in verfügbar ist. Ist dies der Fall, wird per Endpoint Service überprüft, ob ein Endpunkt für dieses `Implementation Artifact` abgespeichert und es somit vom Container deployt wurde. Trifft dies ebenso zu, ist ein aufrufbares `Implementation Artifact` gefunden. Das Sequenzdiagramm zeigt weiterhin die Kommunikation mit dem Instanz Data Service (falls eine ServiceInstanceID angegeben wurde) sowie den Aufruf eines SI-Plug-ins.

In Abbildung 13 sieht man die durch die SI-Engine angereicherte Nachricht (Änderungen **fett** markiert) im Vergleich zu Abbildung 11. Zum Einen wurde der Endpunkt des `Implementation Artifacts` bestimmt und als Header hinzugefügt, zum Anderen wurde im Body ein Parameter durch einen Wert aus der Instanz Datenbank ersetzt. In diesem Beispiel wird dem `Implementation`

Artifact mitgeteilt, anstelle bei Microsoft Azure, bei Amazon Web Services (AWS) eine Datenbank zu erstellen. Ein zuvor ausgeführter Plan kann dies zum Beispiel abgespeichert haben, nachdem er überprüft hat, dass die Preise für Datenbanken bei AWS besser als bei Azure sind.

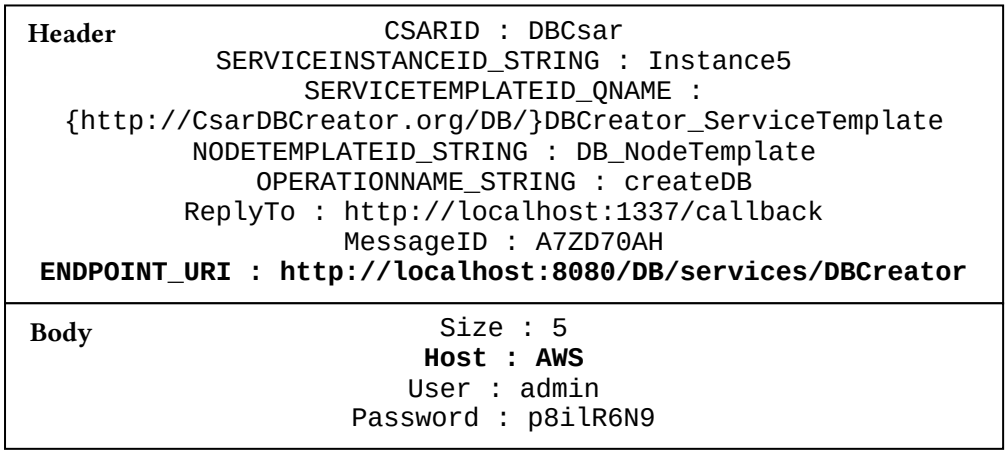

Abbildung 13: Durch die SI-Engine angereicherte und an ein SI-Plug-in gerichtete Message

Nachdem die Message an das zur Invocation-Art passende SI-Plug-in übergeben wurde, liest dieses die für den Aufruf des Implementation Artifact benötigten Information (OperationName, Endpoint) aus dem Header aus und holt die Übergabe-Parameter aus dem Body. Was weiterhin mit diesen Daten passiert und wie der eigentliche Aufruf des Implementation Artifacts (oder auch eines Plans) vonstattengeht ist Plug-in-spezifisch und kann sich somit von Plug-in zu Plug-in unterscheiden. Im Falle des SOAP/HTTP Plug-ins (Details in Kapitel 4.4.5) beispielsweise, werden weitere Informationen aus der zum Implementation Artifact gehörenden WSDL gelesen, eine der WSDL entsprechende SOAP Message erstellt (siehe Listing 6) und diese dann an das Implementation Artifact verschickt. Aus der Antwort, was im Falle einer SOAP Message der Body dieser darstellt, werden anschließend die Output-

Parameter entnommen und diese wiederum in den Body des Message Objekts gelegt und an die SI-Engine zurückgegeben.

```
01 <soap:Envelope
02 xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
03 soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
04    <soap:Body xmlns:m="http://www.DBIA.org/DBCreator">
05      <m:CreateDBRequest>
06       <m:Size>5</m:Size>
07       <m:Host>AWS</m:Host>
08       <m:User>admin</m:User>
09       <m:Password>p8ilR6N9</m:Password>
10      </m:CreateDBRequest>
11    </soap:Body>
12 </soap:Envelope>
```

Listing 6: Durch das SOAP/HTTP Plug-in erstellte SOAP Message

Listing 7 zeigt die Antwort des `Implementation Artifacts`. In diesem Beispiel wurde als Antwort, auf die Anfrage zur Erstellung einer Datenbank, die Adresse dieser Datenbank zurückgegeben.

```
01 <soap:Envelope
02 xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
03 soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
04    <soap:Body>
05      <CreateDBResponse>
06        <URL>http://s3.amazonaws.com/my-5GB-DB</URL>
07      </CreateDBResponse>
08    </soap:Body>
09 </soap:Envelope>
```

Listing 7: Antwort des Implementation Artifacts

Abbildung 14 stellt die an die SI-Engine und von dort an die Service Invocation SOAP API weitergeleitete Exchange Message dar. Im Body befinden sich die von dem `Implementation Artifact` erhaltenen Informationen, welche von dem SOAP-Plug-in als HashMap kodiert werden. Dafür wird bei eingehenden

SOAP Nachrichten der Name eines Elementes als Key und der Inhalt des Elementes als Value genommen. Es wird dabei angenommen, dass die Namen der von den `Implementation Artifacts` zurückgegebenen Elementen, mit den in der TOSCA Definition angegebenen Rückgabewerten überein stimmen.

Header	CSARID : DBCsar SERVICEINSTANCEID_STRING : Instance5 SERVICETEMPLATEID_QNAME : {http://CsarDBCreator.org/DB/}DBCreator_ServiceTemplate NODETEMPLATEID_STRING : DB_NodeTemplate OPERATIONNAME_STRING : createDB ReplyTo : http://localhost:1337/callback MessageID : A7ZD70AH ENDPOINT_URI : http://localhost:8080/DB/services/DBCreator
Body	**URL : http://s3.amazonaws.com/my-5GB-DB**

Abbildung 14: Rückgabe des SI-Plug-ins mit enthaltenen Informationen des Implementation Artifacts

Die Service Invocation SOAP API (siehe Kapitel 5.2) übernimmt anschließend den Body der Exchange Message und die MessageID aus dem Header in die an den `Plan` gerichtete SOAP Message (siehe Listing 8). Die im Body befindliche HashMap mit den zurückgegebenen Daten des aufgerufenen `Implementation Artifacts` wird dabei wie folgt konvertiert: Die Key-Value Paare der HashMap werden paarweise als Key- sowie Value-Elemente der SOAP Message überführt (siehe Listing 8). Der `Plan` bekommt dadurch eine einheitliche Antwort zurück, welche mit den in TOSCA spezifizierten Parametern (siehe Listing 3) übereinstimmt und unabhängig von der Technologie des aufgerufenen `Implementation Artifacts` ist. Da bereits der Aufruf der Service Invocation Schnittstelle (siehe Listing 5) durch die in der TOSCA spezifizierten Parameter (Listing 3) erfolgt, ergibt sich insgesamt eine für den Aufrufer generische und einheitliche Schnittstelle zum Aufruf von Services. Ein `Plan` benötigt somit zum Aufruf eines `Implementation Artifacts` lediglich

die in der TOSCA angegebenen Informationen und muss nichts über die konkrete Implementierung dessen wissen.

Zum Schluss wird die erstellte SOAP Message (Listing 8) an die zu anfangs vom `Plan` an die Service Invocation Schnittstelle übergebene und sich im Header befindliche Adresse (ReplyTo) gesendet.

```
01 <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.
02                                 org/soap/envelope/" xmlns:
03                                 sch="http://siserver.
04                                 org/schema">
05    <soapenv:Header/>
06    <soapenv:Body>
07       <sch:invokeResponse>
08          <MessageID>A7ZD70AH</MessageID>
09          <Params>
10             <Key>URL</Key>
11             <Value>http://s3.amazonaws.com/my-5GB-DB</Value>
12          </Params>
13       </sch:invokeResponse>
14    </soapenv:Body>
15 </soapenv:Envelope>
```

Listing 8: Nachricht der SOAP API zurück an den Aufrufer

Bisher wurde das Konzept der Service Invocation Schnittstelle ausschließlich mittels Service Invocation SOAP API und SOAP/HTTP SI-Plug-in erläutert. Das das Konzept der Service Invocation Schnittstelle jedoch weit mehr Möglichkeiten bietet, wird im folgenden Abschnitt verdeutlicht.

Abbildung 15 veranschaulicht die Möglichkeit zur Realisierung eines OSGI SI-Plug-ins. Dieses Plug-in ermöglicht den Aufruf von als OSGi Services implementierten `Implementation Artifacts`. Es sei angemerkt, dass der momentane Stand des OpenTOSCA Containers, aufgrund des Fehlens eines passenden IA-Plug-ins, noch keine OSGi `Implementation Artifacts` deployen kann, sie somit also vom Container noch nicht unterstützt werden. Die Service Invocation

Schnittstelle wurde jedoch auch mit Blick auf zukünftige `Implementation Artifact` Arten konzipiert und hält Konzepte dafür bereit.

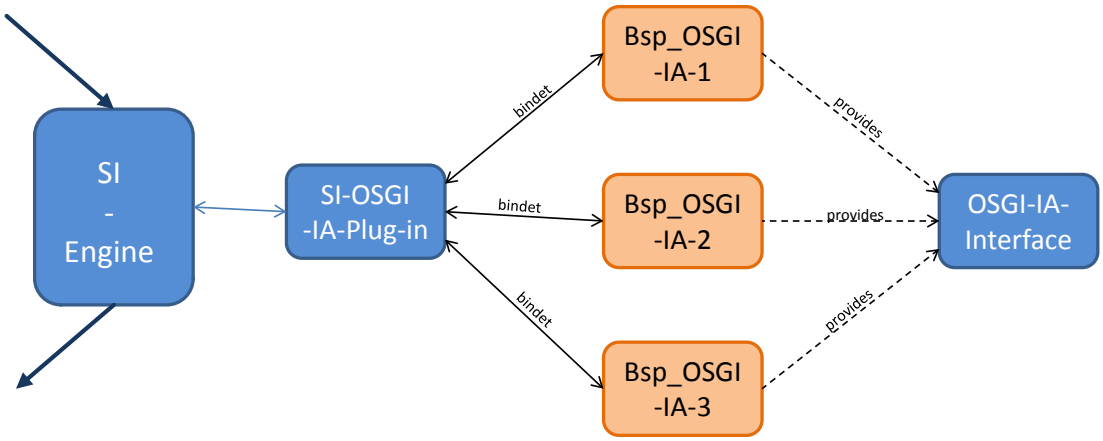

Abbildung 15: Beispiel für die Umsetzungsmöglichkeit eines weiteren Plug-in Types

Das OSGi-SI-Plug-in macht sich, äquivalent zur SI-Engine und dessen Plug-ins, die Möglichkeit von Declarative Services von OSGi zu Nutze. Dies bedeutet, dass OSGi `Implementation Artifacts` ein vorgegebenes Interface implementieren und dieses als Service anbieten müssen. Dies ist für die Umsetzung von Declarative Services nötig, ermöglicht aber das Hinzufügen und Starten der `Implementation Artifacts` zur Laufzeit (siehe Grundlagen OSGi, 2.3). Anhand des implementierten Interfaces und des dadurch spezifizierten Services, kann das OSGi-SI-Plug-in das OSGi `Implementation Artifact` an sich binden und dadurch nutzen. Als Endpoint des `Implementation Artifacts` muss in diesem Fall, die ID des jeweiligen OSGi Services gegeben sein. Listing 9 zeigt die vorgegebenen Methoden des Interfaces.

```
01 public void invoke(String operationName, HashMap<String,
02                    String> params);
03
04 public Object invoke(String operationName, HashMap<String,
05                      String> params);
06
07 public getID();
```

Listing 9: Methoden des OSGi Implementation Artifact Interface

Die beiden invoke Methoden werden benötigt, um die zwei MEPs request-response und one-way zu untersetzen. Per Methode getID wird beim binden des OSGi `Implementation Artifacts` dessen ID abgerufen und anschließend abgespeichert. Das SI-Plug-in kann anhand des übergebenen Endpunkts dann den richtigen OSGi Service wählen und aufrufen.

Wie bereits erwähnt wurde ist es auch möglich die Service Invocation Schnittstelle mit weiteren Service Invocation APIs zu erweitern. Dadurch kann die Funktionalität der Service Invocation Schnittstelle für weitere Techniken nutzbar gemacht werden. Das Hinzuschalten der Service Invocation APIs kann, äquivalent zu den Plug-ins, ebenfalls zur Laufzeit getätigt werden.

Abbildung 16 zeigt die Service Invocation Schnittstelle beispielhaft mit parallel betriebener Service Invocation SOAP sowie Service Invocation REST API. Dadurch wird, wie die Namen der APIs bereits verraten, der Aufruf der Service Invocation Schnittstelle und damit gleichzeitig auch der Aufruf von `Implementation Artifacts` und `Plänen` per SOAP sowie per REST ermöglicht.

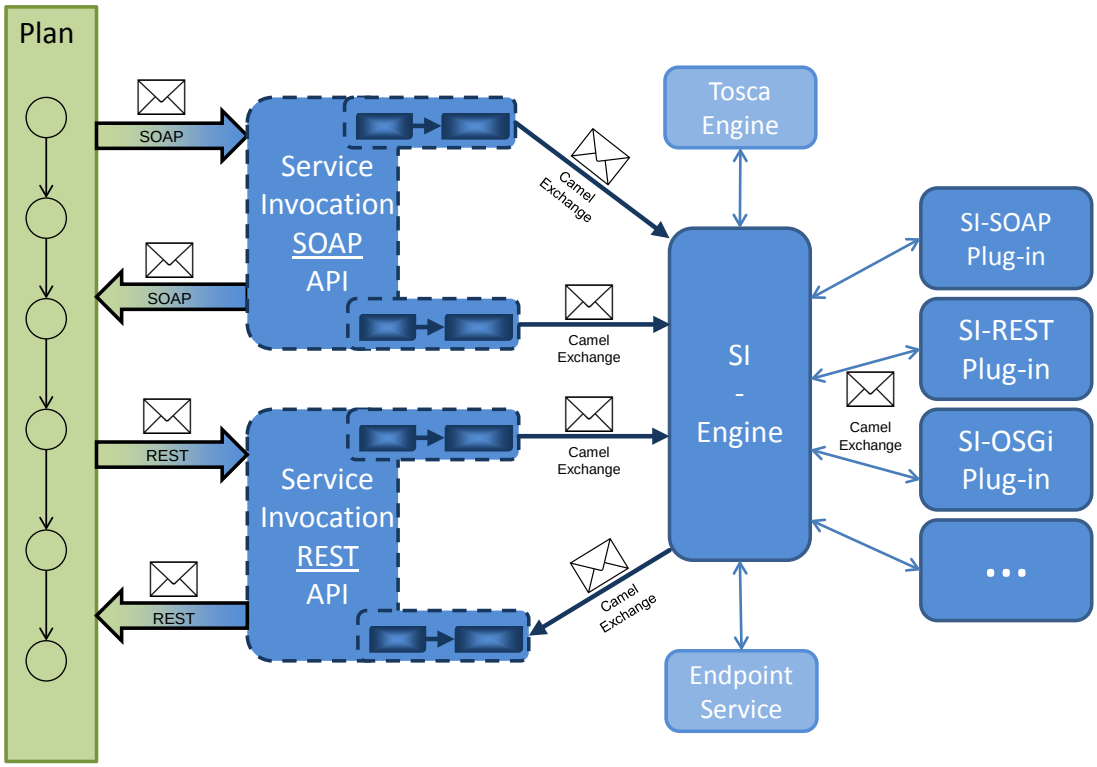

Abbildung 16: Beispiel für eine weitere Service Invocation API

Die im Rahmen dieser Bachelorarbeit implementierten Komponenten der Service Invocation Schnittstelle werden im folgenden Kapitel dargestellt.

5 Implementierung

In diesem Kapitel wird auf die Implementierung der Service Invocation Schnittstelle und deren Komponenten eingegangen. Zuerst wird das Enum zur Spezifizierung der Header der Exchange Message dargestellt. Danach folgen Darstellungen der Service Invocation API für SOAP sowie OSGi Events. Daraufhin wird die SI-Engine genauer erläutert und anschließend auf die SI-Plug-ins und deren Interface detaillierter eingegangen.

5.1 Service Invocation Enum

Dieser Abschnitt veranschaulicht das SI-Enum. Es wird benutzt, um die zum Aufruf eines `Implementation Artifacts` oder `Plans` benötigten Parameter in einer festgelegten Art und Weise übergeben zu können.

Listing 10 zeigt das festgelegte SI-Enum und die damit definierten Werte. Die Werte dienen als Key der Key/Value Paare der Header. Dadurch ist es den Komponenten der Service Invocation Schnittstelle möglich, benötigte Parameter auszulesen. CSARID dient der Identifikation der richtigen `CSAR`-Datei. SERVICEINSTANCEID_STRING wird benötigt, falls Daten per Instance Data Service abgerufen werden sollen und bestimmt dabei die Instanz. Per NODEINSTANCEID_STRING wird, falls angegeben, ebenfalls eine ID einer bestimmten Instanz übermittelt. Allerdings dient dieser Wert nicht der SI-Engine zur Abfrage von Instanz Daten mittels Instance Data Service, sondern wird einem `Implementation Artifact` als Parameter übergeben. Das `Implementation Artifact` besorgt sich damit benötigte Daten selbst per Instance Data Service. SERVICETEMPLATEID_QNAME dient der Bestimmung des `Service Templates` einer TOSCA Definition. Äquivalent bestimmt NODETEMPLATEID_STRING das `Node Template.`

OPERATIONNAME_STRING gibt die aufzurufende Operation des `Implementation Artifacts` oder `Plans` an. Falls der Name der Operation innerhalb eines `NodeTypes` nicht eindeutig ist, muss zudem der Name des zugehörigen Interfaces angegeben werden. Dies wird mittels INTERFACENAME_STRING gemacht. PLANID_QNAME wird benötigt, falls ein `Plan` aufgerufen werden soll und dient dabei der Identifikation des `Plans`.

```
public enum SIEnum {
	CSARID, SERVICEINSTANCEID_STRING, NODEINSTANCEID_STRING,
	SERVICETEMPLATEID_QNAME, NODETEMPLATEID_STRING,
	INTERFACENAME_STRING, OPERATIONNAME_STRING, PLANID_QNAME,
	ENDPOINT_URI, SPECIFICCONTENT_DOCUMENT
}
```

Listing 10: SI-Enum

Alle bisher genannten Werte werden der SI-Engine bereits von der entsprechenden Service Invocation API bei Bedarf übergeben. Ausschließlich die Werte von ENDPOINT_URI sowie gegebenenfalls SPECIFICCONTENT_DOCUMENT werden in der SI-Engine gesetzt. ENDPOINT_URI wird per Endpoint Service ermittelt und spezifiziert den Endpunkt des aufzurufenden `Implementation Artifacts` oder `Plans`. SPECIFICCONTENT_DOCUMENT wird über die TOSCA Engine ermittelt (falls in TOSCA Definition spezifiziert) und kann wichtige Informationen für das Plug-in, wie zum Beispiel über das Mapping der Parameter enthalten (siehe Kapitel 5.7).

5.2 Service Invocation SOAP API

In diesem Kapitel wird die Service Invocation SOAP API zum Aufruf der Service Invocation Schnittstelle per SOAP Messages erläutert. Weiterhin wird dabei auf die Nutzung von Camel eigegangen.

Listing 11 zeigt konzeptionell die implementierte Route und ihre Endpunkte (siehe Camel Grundlagen, Kapitel 2.7) der Service Invocation SOAP API. Die in den Zeilen eins bis 16 definierten Strings INVOKE, CALLBACK, ENGINE_IA und ENGINE_PLAN stellen die Endpunkte der Route dar. Die Route selbst, wird von Zeile 17 bis 22 definiert.

Der von Zeile eins bis sechs definierte String INVOKE dient als Endpunkt zum Aufruf der Service Invocation SOAP API. Durch ihn wird ein SOAP Web Service per CXF Komponente [8] auf *http://localhost:8081/invoker* gestartet. Als WSDL des Web Services dient die *invoker.wsdl* (siehe Anhang A1). Außerdem werden Service sowie Port aus der WSDL definiert.

```
01 String INVOKE = "cxf:http://localhost:
02                 8081/invoker?wsdlURL=META-INF/invoker.
03                 wsdl&serviceName={http://siserver.org/wsdl}
04                 SIServerInvokeService&portName={http:
05                 //siserver.org/wsdl}
06                 SIServerInvokePort";

07 String CALLBACK = "cxf:${header[ReplyTo]}?wsdlURL=META-
08                   INF/invoker.wsdl&serviceName={http:
09                   //siserver.org/wsdl}
10                   SIServerCallback&portName={http://siserver.
11                   org/wsdl}CallbackPort";

12 String ENGINE_IA = "bean:siengineinterface.
13                    SIEngineInterface?method=invokeOperation";

14 String ENGINE_PLAN = "bean:siengineinterface.
15                      SIEngineInterface?method=invokePlan";

16 from(INVOKE).unmarshal(requestJaxb).process(requestProcessor)
17      .choice().when(this.header(CxfConstants.OPERATION_NAME).
18      isEqualTo("invokeOperation")).to(ENGINE_IA).when(this.header(
19      CxfConstants.OPERATION_NAME).isEqualTo("invokePlan")).
20      to(ENGINE_PLAN).end().process(responseProcessor).
21      marshal(responseJaxb).recipientList(this.simple(CALLBACK));
```

Listing 11: Route der Service Invocation SOAP API

Der String CALLBACK (Zeile sieben bis elf) definiert den Endpunkt, der bei asynchronen Aufrufen zum Versenden der Antwortnachricht an den Aufrufer benötigt wird. Es wird dafür ebenfalls die CXF Komponente sowie die *invoker.wsdl* genutzt. Weiterhin ist der entsprechende Service sowie Port angegeben. Allerdings wird die Adresse, wohin die SOAP Message geschickt werden soll, jeweils dynamisch bestimmt. *${header[ReplyTo]}* bedeutet, dass aus der Exchange Message der Wert des Headers *ReplyTo* ausgelesen und als Adresse der Antwortnachricht benutzt werden soll. Dieser Wert muss beim Aufruf der Service Invocation SOAP API vom Aufrufer mitgeteilt werden. Die Service Invocation SOAP API unterstützt dabei die Übergabe, wie in dem im Kapitel 4.3 Listing 5 gezeigten Beispiel per Parameter im Body als auch per WS-Addressing Header [42]. Entsprechend kann auch die MessageID mit diesen beiden Möglichkeiten übergeben werden.

In den Zeilen zwölf und 13 sowie 14 und 15 werden die beiden Methoden der SI-Engine (siehe 5.4) als Endpunkte festgelegt. Ein Endpunkt zum Aufruf von `Implementation Artifacts` (ENGINE_IA) und ein Endpunkt zum Aufruf von `Plänen` (ENGINE_PLAN).

Zeile 16 bis 21 zeigt die Route, die nach einem Aufruf abgearbeitet wird. `From(INVOKE)` definiert den oben bereits erklärten INVOKE Endpunkt als Einstiegspunkt der Route. Die SOAP Message eines Aufrufs wird unmarshallt (`unmarshal(requestJaxb)`) und anschließend im `requestProcessor` prozessiert. Beim Prozessieren werden die durch das SI-Enum definierten Parameter sowie alle in der eingegangen SOAP Message definierten Header als Header der Exchange Message angelegt. Daraufhin wird der Header OPERATION_NAME ausgelesen und anhand dessen bestimmt, ob ein `Implementation Artifact` oder ein `Plan` aufgerufen werden soll und die dementsprechende Methode der SI-Engine aufgerufen (ENGINE_IA oder ENGINE_PLAN). Nachdem die SI-Engine beziehungsweise ein passendes Plug-in den Aufruf ausgeführt und die Exchange Message samt Antwort des `Implementation Artifacts` oder `Plans` an die Service Invocation SOAP

API zurückgegeben haben, wird diese erneut prozessiert (`process(responseProcessor)`). Dabei wird die Antwort aus dem Body der Exchange Message ausgelesen und – falls möglich – in ein marshall-fähiges Objekt umgewandelt. Anschließend wird das Antwort-Objekt marshallt (`marshal(responseJaxb)`) und als SOAP Message an den CALLBACK-Endpunkt geschickt.

5.3 Service Invocation OSGi-Event API

Dieses Kapitel stellt die Implementierung der Service Invocation OSGi-Event API vor. Insbesondere wird dabei die allgemeine Funktionsweise von OSGi Events sowie speziell die Zusammenarbeit mit der Plan Invocation Engine (siehe Kapitel 2.5) betrachtet.

Die Service Invocation OSGi-Event API ermöglicht die Nutzung der Service Invocation Schnittstelle per OSGi Event Admin Service [31, Kapitel 113]. Mittels des OSGi Event Admin Services können Events nach dem *publish and subscribe* Pattern [22] versendet und empfangen werden. Diese Technik ermöglicht eine asynchrone Kommunikation zwischen der Service Invocation Schnittstelle und beispielsweise der Plan Invocation Engine.

Abbildung 17 zeigt die Service Invocation Schnittstelle mit Service Invocation OSGi-Event API und Plan Invocation Engine. Desweiteren zeigt die Abbildung die Funktion des *publish and subscribe*-Patterns. Die Plan Invocation Engine sendet seine mit den benötigten Informationen angereicherte Nachricht an eine Liste (Request Topic). Die Service Invocation OSGi-Event API empfängt - da sie diese List abonniert (subscribed) hat - die Nachricht und kann sie weiterverarbeiten. Übergebene Informationen wie zum Beispiel `CSAR`-ID oder `Plan`-ID werden, wie bereits von der Service Invocation SOAP API bekannt, als Header und der übergebene Payload als Body eines Exchange Message Objekts gesetzt. Der Payload kann dabei beispielsweise eine HashMap mit Parametern als

Key/Value Paaren oder wie im Falle der Plan Invocation Engine vom Typ org.w3c.dom.Document [33] sein. Anschließend wird die Exchange Message der SI-Engine zur weiteren Bearbeitung weitergereicht. Nachdem die Antwort von der SI-Engine zurück gekommen ist, sendet die Service Invocation OSGi-Event API die Antwortnachricht an eine weitere Liste (Response Topic). Diese wiederum ist von der Plan Invocation Engine abonniert, welche dadurch die Antwortnachrichten empfangen und ihrerseits weiter bearbeiten kann.

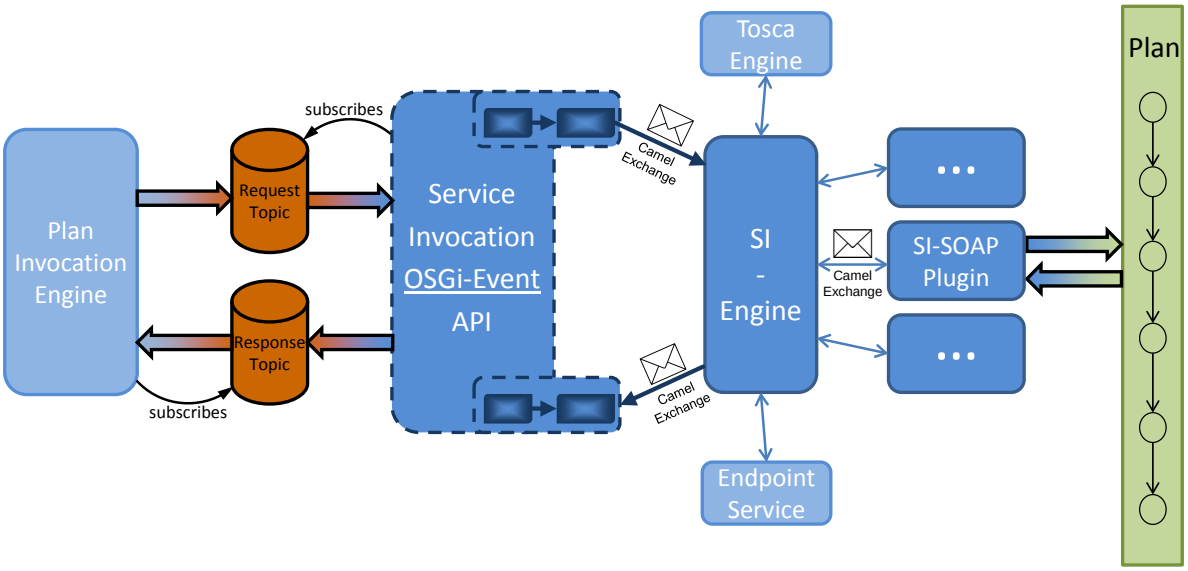

Abbildung 17: Aufruf eines Plans initiiert durch Plan Invocation Engine

Listing 12 zeigt anhand der Service Invocation OSGi-Event API, wie OSGi Event Services angewandt werden. Zuerst werden wie in jeder OSGi XML-Konfigurationsdatei (Component Description [36, Kapitel 112.2]) der Name (Zeile 02 bis 04) sowie die implementierende Klasse (Zeile 05 bis 06) der jeweiligen Komponente festgelegt. Damit die Komponente Events an Listen senden kann, muss der EventAdmin-Service gebunden werden (Zeile 07 bis 11). *bindEventAdmin* (Zeile 07) gibt die Methode zum Binden des Services an und muss in *org.opentosca.siengine.api. osgievent.SIEventHandler*, der in Zeile 05 bis 06 spezifizierten Klasse vorhanden und entsprechend implementiert sein. Zeile 12

bis 15 gibt an, dass die oben genannte Klasse den Service EventHandler anbietet. Dafür muss sie das Interface EventHandler und die dazugehörige Methode zum Empfangen von Events *handleEvent(Event event)* implementieren. Außerdem müssen die Namen der von der Komponente abonnierten Listen (*event.topics*, Zeile 16 bis 17) spezifiziert werden. In diesem Beispiel handelt es sich dabei um die Liste *org_opentosca_plans/requests*, welche zum Beispiel von der Plan Invocation Engine zum Verschicken von Anfragen zum Aufruf von `Plänen` genutzt wird.

```
01 <scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.
02                          0" immediate="false" name="org.
03                          opentosca.siengine.api.osgievent.
04                          SIEventHandler">
05    <implementation class="org.opentosca.siengine.api.
06                          osgievent.SIEventHandler"/>
07    <reference bind="bindEventAdmin" cardinality="1..1"
08                    interface="org.osgi.service.event.
09                    EventAdmin" name="EventAdmin"
10                    policy="static"
11                    unbind="unbindEventAdmin"/>
12    <service>
13       <provide interface="org.osgi.service.event.
14                          EventHandler"/>
15    </service>
16    <property name="event.topics" type="String"
17                   value="org_opentosca_plans/requests"/>
18 </scr:component>
```

Listing 12: Anwendung des OSGi Event Services

Äquivalent dazu abonniert die Plan Invocation Engine die Liste *org_opentosca_plans/responses*, über welche die Antwortnachrichten durch die Service Invocation OSGi-Event API verschickt werden. Das Senden eines Events erfolgt dabei über die von dem EventAdmin- Service angebotene Methode *postEvent(Event event)*.

5.4 Service Invocation Engine

In diesem Abschnitt wird die Implementierung der SI-Engine vorgestellt. Dabei wird verstärkt auf die von der SI-Engine angebotenen Methoden sowie das System zur Verwaltung der SI-Plug-ins eingegangen.

Die SI-Engine bietet zwei Methoden an. Zum Einen *InvokeOperation(Exchange exchange)* zum Aufruf von `Implementation Artifacts` und zum Anderen *invokePlan(Exchange exchange)* zum Aufruf von `Plänen`. Obwohl `Implementation Artifacts` als auch `Pläne` als Services gesehen werden können, müssen die Anfragen zum Aufruf dieser beiden Fälle unterschiedlich behandelt werden. In Abbildung 18 werden die Abläufe der beiden Methoden dargestellt.

Identisch haben beide angebotenen Methoden, dass sie zuerst die für sich relevanten durch das SI-Enum (siehe 5.1) definierten Header aus der Exchange Message auslesen. Im Falle von *invokeOperation* wären dies die Werte für CSARID, SERVICEINSTANCEID_STRING, NODEINSTANCEID_STRING, SERVICETEMPLATEID_QNAME, NODETEMPLATEID_STRING, INTERFACENAME_STRING und OPERATIONNAME_STRING. Im Falle von invokePlan CSARID, SERVICEINSTANCEID_STRING, NODETEMPLATEID_STRING und PLANID_QNAME. Im Anschluss daran wird – falls *invokeOperation* aufgerufen wurde – mithilfe der TOSCA Engine ein passendes `Implementation Artifact`, dessen Message Exchange Pattern sowie die dazugehörigen Properties bestimmt. In Kapitel 4.3 wurde bereits das entsprechende Sequenzdiagramm (Abbildung 12) vorgestellt. Das Abfragen von Instanz Daten bei vorhandener ServiceInstanceID und einer HashMap als Payload geschieht in beiden Fällen. Bei *invokeOperation* wird weiterhin überprüft, ob eine NodeInstanceID spezifiziert wurde. Wenn dies der falls ist und zudem der Payload in Form einer HashMap gegeben ist, wird die NodeInstanceID als Parameter in die HashMap übernommen. Damit können `Implementation Artifacts` aktuelle Werte aus

den Instanzdaten selbst bestimmen. Unabhängig der Methode wird anschließend der jeweilige Endpunkt des `Implementation Artifacts` oder `Plans` per Endpoint Service bestimmt. Für den Aufruf von `Implementation Artifacts` wird weiterhin die Invocation-Art zur Bestimmung des passenden Plug-ins anhand den vorherig abgefragten Properties bestimmt.

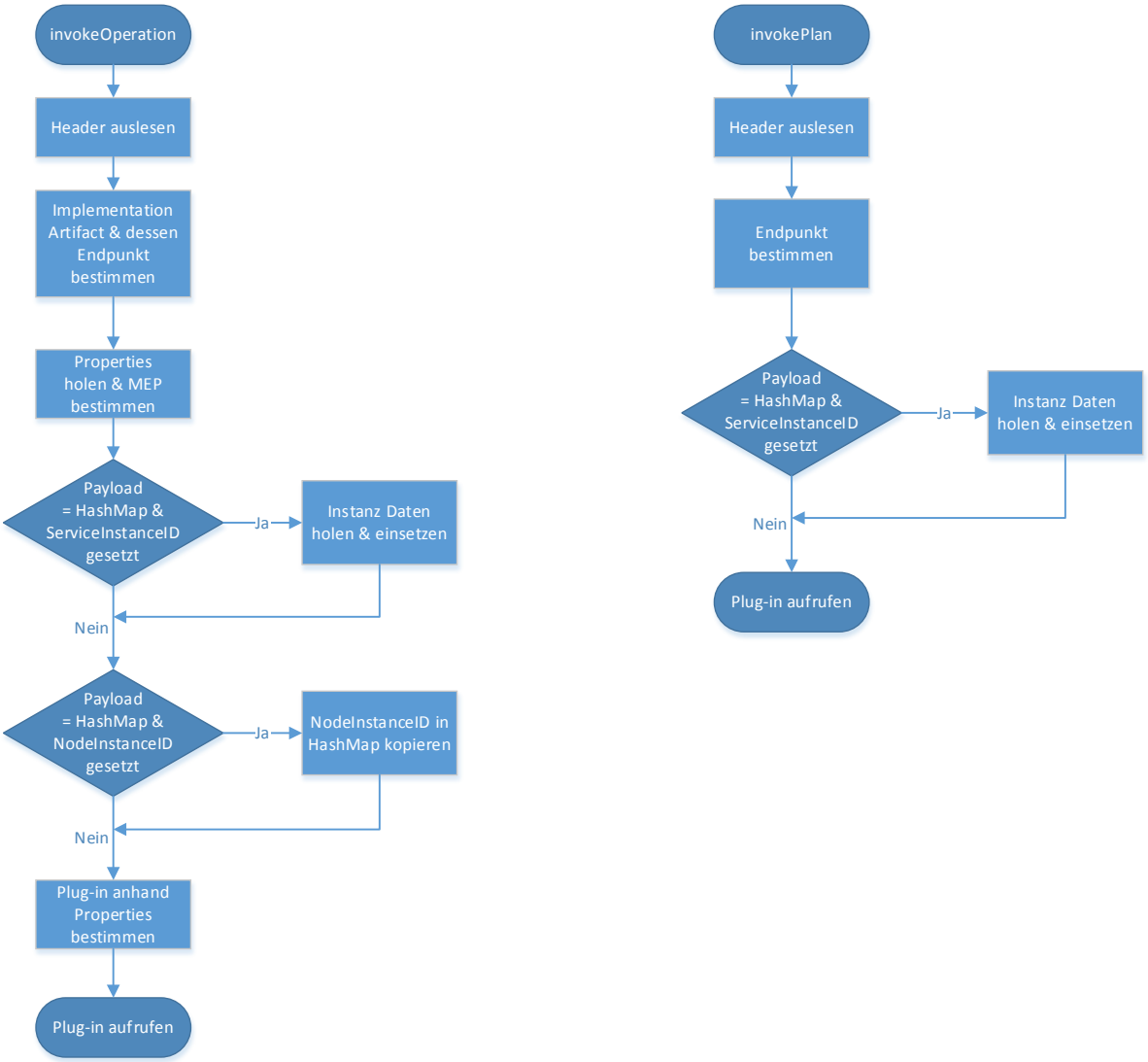

Abbildung 18: Flussdiagramme der beiden SI-Engine Methoden invokeOperation und invokePlan

Da der OpenTOSCA Container bisher ausschließlich BPEL `Pläne` unterstützt, wird beim Aufruf eines `Plans` die Exchange Message direkt an das SOAP Plug-in (siehe Kapitel 5.6) übergeben. Sollten zukünftig weitere `Plan`-Arten unterstützt werden, muss die Invocation-Art und damit die Wahl des Plug-ins anhand der `Managementplan`-Sprache bestimmt werden.

Listing 13 zeigt wie innerhalb der TOSCA Definition die Invocation-Art eines `Implementation Artifacts` angegeben wird. Dafür wird die von `Artifact Templates` gebotene Möglichkeit zur Definition von Properties (Zeile 03 bis 12) genutzt. Für die Service Invocation Schnittstelle sind in diesem Beispiel die Zeilen 08 bis 10 relevant. Per *InvocationType*-Element wird die Invocation-Art des `Implementation Artifacts` festgelegt. In diesem Fall ist dies *SOAP/HTTP*. Die SI-Engine würde somit die Anfrage zum Aufruf dieses `Implementation Artifacts` an ein SOAP/HTTP-fähiges Plug-in weiterleiten.

```
01    <p:ArtifactTemplate id="EC2VMService" type="toscatypes:
02                          WAR">
03          <p:Properties>
04                <opentosca:WSProperties>
05                      <opentosca:ServiceEndpoint>
06                            /services/EC2VMIAService
07                      </opentosca:ServiceEndpoint>
08                      <opentosca:InvocationType>
09                            SOAP/HTTP
10                      </opentosca:InvocationType>
11                </opentosca:WSProperties>
12          </p:Properties>
13          <p:ArtifactReferences>
14                <p:ArtifactReference reference=
15                      "IAs/EC2VMService/EC2-VM-Service.war" />
16          </p:ArtifactReferences>
17    </p:ArtifactTemplate>
```

Listing 13: Beispiel ArtifactTemplate mit Invocation-Art Angabe

Desweiteren ist in Listing 13 eine weitere Property (*ServiceEndpoint*) definiert. Diese wird von der IA-Engine zur Bestimmung des korrekten Endpunktes des `Implementation Artifacts` benötigt. Dadurch wird sichergestellt, dass die IA-Engine einen für die Service Invocation Schnittstelle nutzbaren Endpunkt per Endpoint Service abspeichert.

Eine wichtige Anforderung an die Service Invocation Schnittstelle ist die Möglichkeit, das Spektrum unterstützter Invocation-Arten erweitern zu können. Dies wird mittels eines Plug-in Systems erreicht, das nun genauer vorgestellt wird.

Das Plug-in-System wird mittels Declarative Services (siehe Kapitel 2.3) realisiert. Die SI-Plug-ins implementieren dafür das vorgegeben Interface *SIPluginInterface* (siehe Kapitel 5.5) und bieten jeweils den Service *ISIEnginePluginService* an. Dies ist in Zeile 04 bis 05 in Listing 14 zu sehen.

```
01 <scr:component ...>
02    <implementation class= .../>
03    <service>
04       <provide interface="org.opentosca.siengine.plugins.
05                           service.ISIEnginePluginService"/>
06    </service>
07 </scr:component>
```

Listing 14: Anbieten eines Services per OSGi XML-Konfigurationsdatei

```
01 <scr:component ...>
02    <implementation class= .../>
03    <reference bind="bindPluginService" cardinality="0..n"
04                    interface="org.opentosca.siengine.plugins.
05                    service.ISIEnginePluginService"
06                    name="SIPluginInterface" policy="dynamic"
07                    unbind="unbindPluginService"/>
08 </scr:component>
```

Listing 15: Binden eines Services per OSGi XML-Konfigurationsdatei

Da die SI-Engine alle diese Services bindet (siehe Listing 15, Zeile 03 bis 07), wird beim Start eines SI-Plug-ins in der SI-Engine automatisch eine Methode zum binden dieses Plug-ins aufgerufen. In diesem Beispiel lautet die Methode *bindPluginService* (siehe Zeile 03 in Listing 15).

Des Weiteren haben SI-Plug-ins per Interface eine Methode *getType()*, welche die Invocation-Art als String zurück gibt, vorgegeben. Diese Methode wird von der SI-Engine aufgerufen wenn ein Plug-in gebunden wird und der Rückgabewert (die Invocation-Art) zusammen mit dem jeweiligen Plug-in in einer Map abgelegt (siehe Listing 16, Zeile 03 bis 06). Zeile 01 und 02 zeigen die Map zur Verwaltung der SI-Plug-ins. Als Key dient die jeweilige Invocation-Art und entsprechend als Value das dazugehörige Plug-in. Beim Stoppen eines SI-Plug-ins, wird die unbind-Methode ausgeführt und dort das Plug-in aus der Map entfernt (Zeile 08 bis 11). Durch dieses System besitzt die SI-Engine jederzeit eine aktuelle Liste der verfügbaren SI-Plug-ins und kann damit anhand der Invocation-Art eines `Implementation Artifacts`, das dazu passende Plug-in wählen.

```
01    Map<String, ISIEnginePluginService> pluginServicesMap =
Collections.
02    synchronizedMap(new HashMap<String, ISIEnginePluginService >());

. . .

03   public void bindPluginService(ISIEnginePluginService plugin) {
04
05          pluginServicesMap.put(plugin.getType(), plugin);
06   }
07
08   public void unbindPluginService(ISIEnginePluginService plugin) {
09
10          pluginServicesMap.remove(plugin.getType());
11   }
```

Listing 16: Implementierung des Plug-in-Systems

5.5 Service Invocation Plug-in Interface

Dieses Kapitel stellt das Interface für SI-Plug-ins vor. Wie im vorherigen Kapitel bereits erläutert, müssen alle für die Service Invocation Schnittstelle nutzbaren Plug-ins dieses Interface implementieren und den Service *ISIEnginePluginService* anbieten.

```
01 public interface ISIEnginePluginService {
02
03     public Exchange invoke(Exchange exchange);
04
05     public String getType();
06
07 }
```

Listing 17: Interface der SI-Plug-ins

Listing 17 zeigt das für die SI-Plug-ins vorgegebene Interface. Das Interface definiert zwei Methoden. Die im vorangegangen Kapitel erläuterte Methode *getType()* (Zeile 05), welche die unterstützte Invocation-Art des Plug-ins zurück gibt und eine Methode *invoke(Exchange exchange)* (Zeile 03), zur Übergabe der Exchange Message an das SI-Plug-in. Weiterhin gibt die invoke-Methode die Exchange Message, mit der sie aufgerufen wurde, mit der Antwortnachricht des aufgerufenen `Implementation Artifacts` oder `Plans` im Body als Rückgabewert zurück.

5.6 Service Invocation SOAP/HTTP-Plug-in

In diesem Kapitel wird eine Implementierung des im Kapitel zuvor vorgestellten SI-Plug-in Interfaces vorgestellt. Konkret handelt es dabei um das SOAP/HTTP-Plug-in, also einem SI-Plug-in, welches `Implementation Artifacts` und `Pläne` per SOAP Message über HTTP aufrufen kann.

Nach dem Aufruf der invoke-Methode und der Übergabe der Exchange Message durch die SI-Engine liest das SOAP/HTTP-Plug-in den im Header angegebenen Endpunkt des `Implementation Artifacts` oder `Plans` aus. Es wurde im Verlauf dieser Arbeit bereits erklärt, dass sich die Endpunkte je nach benötigtem Plug-in unterscheiden können. Im Falle des SOAP/HTTP-Plug-ins muss der Endpunkt entweder direkt auf die WSDL Definition des `Implementation Artifacts` beziehungsweise `Plans` verweisen oder aber dies durch das Anhängen von „?wsdl“ tun. Falls zum Beispiel die WSDL-Definition eines aufzurufenden `Implementation Artifacts` unter der Adresse *http://localhost:8080/EC2IA/services/EC2Service?wsdl* erreichbar ist, muss entweder diese Adresse oder *http://localhost:8080/EC2IA/services/EC2Service* als Endpunkt gegeben sein. Dies ist notwendig, da die WSDL-Definition die Schnittstelle zum Aufruf des Webservices beschreibt und die darin enthaltenen Informationen zum Erstellen sowie zum Verschicken der Aufrufnachricht benötigt und deshalb durch das Plug-in ausgelesen werden. So wird beispielsweise das MEP bestimmt oder, falls eine HashMap mit den Input Parametern als Body der Exchange Message übergeben wird, anhand der WSDL-Definition eine korrekte SOAP Message daraus generiert.

Das SOAP/HTTP-Plug-in unterstützt in der aktuellen Implementierung drei Austauscharten von Nachrichten. Diese werden nun folgend erklärt sowie in Abbildung 19 veranschaulicht.

One-Way (In-Only): Der Service wird per Soap Message aufgerufen, sendet aber keine Antwortnachricht zurück. Listing 18 zeigt beispielhaft eine One-Way Operation in einer WSDL-Definition.

```
01          <wsdl:operation name="one-wayOperation">
02                <wsdl:input message="tns:inputMessage">
03                </wsdl:input>
04          </wsdl:operation>
```

Listing 18: One-Way Operation

Request-Response (In-Out): Der Client (also das Plug-in) sendet eine SOAP Message an den Service und wartet bis die Antwortnachricht eintrifft. Ein Beispiel einer Request-Response Operation ist in Listing 19 zu sehen.

```
01          <wsdl:operation name="request-responseOperation">
02                <wsdl:input message="tns:requestMessage">
03                </wsdl:input>
04                <wsdl:output message="tns:responseMessage">
05                </wsdl:output>
06          </wsdl:operation>
```

Listing 19: Request-Response Operation

Request-Callback (asynchrones Request-Response): Der Client sendet eine SOAP Message an den Service, erwartet aber keine direkte Antwort. Stattdessen wird er informiert wenn die Antwort des Services eintrifft. Hierfür muss dem Service eine Adresse für den Callback sowie eine MessageID mitgegeben werden. Die MessageID muss in der Antwortnachricht des Services wieder enthalten sein, um eine Korrelation der Aufrufnachricht mit der Antwortnachricht herstellen zu können. Das SOAP/HTTP-Plug-in ermöglicht dabei die Übergabe von Callback-Adresse und MessageID per Parameter im SOAP-Body der Message sowie per WS-Addressing Header. Allerdings wird Request-Callback aus WSDL-Sicht mittels zwei One-Way Operationen realisiert und ist alleine aus der WSDL-Definition nicht spezifizierbar. Daher müssen zur endgültigen Bestim-

mung zu den aus der WSDL erworben Informationen noch zusätzlich die in Kapitel 4.3 erklärten MEPs (In-Only & In-Out) aus der TOSCA Definition hinzugenommen werden. Dementsprechend ergibt eine In-Only Operation (aus der TOSCA Definition) und eine In-Only Operation (aus der WSDL-Definition), eine One-Way SOAP Message (also ohne Antwortnachricht) an den Service. Demgegenüber ergibt sich aus einer In-Out Operation (aus der TOSCA Definition) und einer In-Only Operation (aus der WSDL-Definition) ein Nachrichtenaustausch per Request-Callback.

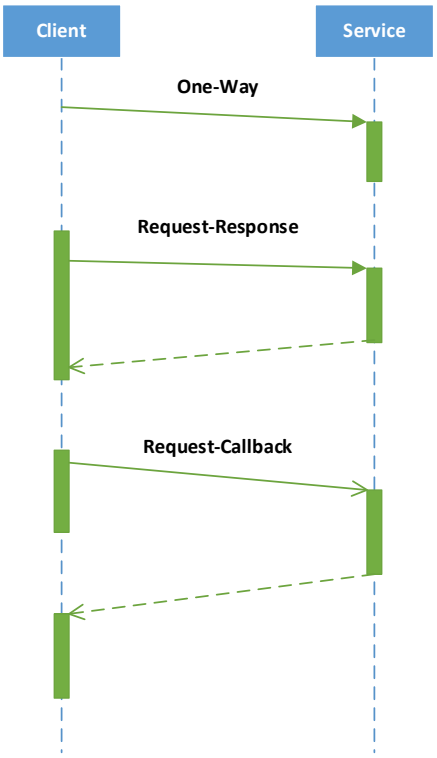

Abbildung 19: Darstellung der drei von dem SOAP/HTTP-Plug-in unterstützten MEPs

5.7 Service Invocation REST/HTTP-Plug-in

Dieser Abschnitt erläutert die Umsetzung des REST/HTTP-Plug-ins. Insbesondere wird dabei darauf eingegangen, wie Mapping-Informationen der zu übergebenden Parameter in der TOSCA Definition angegeben werden können.

HTTP erlaubt die Übergabe von Parametern sowohl innerhalb der URL (query string) als auch im Body der Nachricht. Um dem Plug-in zu ermöglichen, die Nachricht entsprechend der Implementierung des `Implementation Artifacts` aufzubauen, können Informationen über das Mapping der Parameter in der TOSCA Definition angegeben werden. Diese Informationen werden in der SI-Engine mittels der TOSCA Engine bestimmt und dem Plug-in übergeben. Listing 20 zeigt das Schema zur Beschreibung der Mapping-Informationen, welche innerhalb des `Implementation Artifact`-Elements (als *artifact specific content*[7]) definiert werden müssen.

```
01 <DataAssign>
02      <Operations>
03              <Operation name="String" ?
04                      interfaceName="String" ?
05                      endpoint="no | yes" ?
06                      params="queryString | payload" ?
07                      contentType="urlencoded | xml" ?
08                      method="POST | GET"  ? >
09              </Operation> +
10      </Operations>
11 </DataAssign>
```

Listing 20: Schema zur Beschreibung des Parameter-Mappings

`Implementation Artifacts` implementieren die von einem `NodeType` angebotenen Interfaces und Operationen. Da ein `Implementation Artifact` neben einer einzelnen Operation eines Interfaces auch alle Operationen eines Interfaces oder alle Interfaces samt Operationen implementieren kann, werden die beiden Attribute *name* (Zeile 03) sowie *interfaceName* (Zeile

[7] Siehe [29] Kapitel 7.

04) benötigt. Sie spezifizieren, für welche Operation die jeweils angegebenen Mapping-Informationen gelten. Implementiert das `Implementation Artifact` jedoch ohnehin nur eine Operation, ist die Angabe von *name* sowie *interfaceName* nicht nötig. Dementsprechend ist auch die Angabe von *interfaceName* nur dann nötig, falls das `Implementation Artifact` mehrere Interfaces implementiert. Per *endpoint*-Attribut (Zeile 05) kann angegeben werden, dass der Name der Operation sowie des Interfaces an den von dem Endpoint Service erhaltenen Endpunkt angehängt (mit „/“ als Trennzeichen) werden soll. Das *params*-Attribut (Zeile 06) spezifiziert, ob die zu übergebenden Parameter als Teil der URL (*queryString*) oder im Body (*payload*) der Nachricht angegeben werden sollen. Falls sie im Body übergeben werden, kann zudem der gewünschte *Content-Type* (Zeile 07) spezifiziert werden. Aktuell werden die beiden Content-Types application/x-www-form-urlencoded (*urlencoded*) sowie application/xml (*xml*) unterstützt. Weiterhin kann mittels *method-Attribut* (Zeile 08) die geforderte HTTP-Methode (*POST* oder *GET*) angegeben werden.

Zusätzlich zu den Mapping-Informationen hat das Plug-in eigene Annahmen bzw. Anforderungen bezüglich des Aufbaus einer Nachricht. So werden beispielsweise falls die Übergabe der Parameter per Query String erfolgt, die einzelnen Key-Value-Paare mittels „&“ und die Key-Value-Werte per „=“ voneinander getrennt. Weiterhin steht vor dem ersten Key-Wert ein „?“ zur Abgrenzung des Endpunktes von den Parametern. Listing 21 zeigt ein Beispiel[8] für die Angabe einer solchen Mapping-Definition und die daraus resultierende Anfrage. Identisch (mit „&“ und „=“ Zeichen zur Trennung) werden auch die Parameter im Body kodiert, falls sie *urlencoded* übergeben werden. Für die Übergabe per XML werden die Parameter wie folgt umgewandelt: Die Namen der einzelnen Parameter werden zu Elementen mit dem Wert des jeweiligen Parameters als Content. Weiterhin wird der Name der Operation als Root Element verwendet. Ein Beispiel hierfür wird in Listing 22 gezeigt.

[8] Als Grundlage für die in Listing 21 und 22 gezeigten Beispiele, wird die in Abbildung 13 dargestellte Exchange Message angenommen. Die eigentlich darin mit zu übergebenden Mapping-Informationen werden im jeweiligen Beispiel angegeben.

```
<DataAssign>
   <Operations>
      <Operation name="createDB"
                 endpoint="yes"
                 params="queryString"
                 method="GET" >
      </Operation>
   </Operations>
</DataAssign>
```

```
GET /DB/services/DBCreator/createDB?Size=5&Host=AWS&User=
      admin&Password=p8ilR6N9 HTTP/1.1
Host: localhost:8080
…
```

Listing 21: Beispiel für Parameterübergabe per Query String

```
<DataAssign>
   <Operations>
      <Operation name="createDB"
                 params="payload"
                 contentType="xml" >
      </Operation>
   </Operations>
</DataAssign>
```

```
POST /DB/services/DBCreator HTTP/1.1
Host: localhost:8080
…

      <createDB>
        <Size>5</Size >
        <Host>AWS</Host >
        <User>admin</User >
        <Password>p8ilR6N9</Password >
      </createDB>
```

Listing 22: Beispiel für Parameterübergabe mit Content-Type *xml*

Zukünftig könnte man auch die Web Application Description Language (WADL), äquivalent zur WSDL im SOAP-Plug-in, zur Beschaffung von benötigten Informationen nutzen. Da diese aber selten genutzt wird und zudem ungleich komplexer ist, wurde sich in dieser ersten prototypischen Implementierung des Plug-ins für ein eigenes, für diese spezielle Aufgabe ausgelegtes, Schema zur Angabe der Informationen entschieden.

6 Annahmen

Damit das in den vorherigen Abschnitten erläuterte Konzept und die darauf beruhende Implementierung korrekt funktionieren kann, müssen einige Annahmen getroffen und Anforderungen an den `Plan`- beziehungsweise TOSCA Definition-Ersteller sowie Plug-in Entwickler (der Service Invocation Schnittstelle) gestellt werden. Diese werden in diesem Kapitel dargelegt und erläutert.

Richtigkeit der übergebenen Parameter

Die an die Service Invocation Schnittstelle übergebenen Parameter lassen sich in zwei verschiedene Gruppen einteilen. Einerseits die zur Bestimmung des gewünschten `Implementation Artifacts` beziehungsweise `Plans` benötigten Daten und andererseits die für den Aufruf des `Implementation Artifacts` oder `Plans` geforderten Daten.

Die zur Bestimmung des passenden `Implementation Artifacts` beziehungsweise des passenden `Plans` an die Service Invocation Schnittstelle übergebenen Parameter, müssen korrekt und innerhalb des OpenTOSCA Containers bekannt sein. Innerhalb des OpenTOSCA Containers bekannt bedeutet dabei, dass diese Daten, wie zum Beispiel `CSARID` oder `ServiceTemplateID`, über die TOSCA Engine abrufbar und damit als Java Objekte abgebildet sind. Ist dies nicht der Fall, können zwingend erforderliche Informationen wie zum Beispiel der Name des `Implementation Artifacts` und damit dessen Endpunkt nicht bestimmt und damit der gewünschte Aufruf nicht getätigt werden.

Weiterhin müssen die, für den Aufruf des `Implementation Artifacts` beziehungsweise des `Plans`, benötigten Input-Parameter den durch das `Implementation Artifact` oder den `Plan` geforderten Parametern entsprechen. Dies bedeutet, dass zum Einen die Namen der übergebenen Parameter sowie zum Anderen die Anzahl der Parameter (nachdem zusätzliche Parameter

wie z.B. Instanzdaten hinzugefügt wurden) identisch sein müssen. Außerdem muss die angegebene auszuführende Operation ebenfalls zu `Implementation Artifact` beziehungsweise `Plan` sowie deren Input-Parametern passen. Des Weiteren muss, falls eine Operation innerhalb eines `NodeTypes` in verschiedenen Interfaces vorhanden ist, der Name des Interfaces angegeben werden. Wird dieser nicht angegeben, wird davon ausgegangen, dass die Operation innerhalb des `NodeTypes` eindeutig ist und das `Implementation Artifact` nur anhand dessen bestimmt.

Vorhandensein eines Endpunktes

Ohne Informationen unter welchem Namen oder welcher Adresse ein bestimmtes `Implementation Artifact` erreichbar ist kann dieses nicht aufgerufen werden. Daher müssen durch die IA-Engine beziehungsweise deren Plug-ins deployte `Implementation Artifacts` sowie äquivalent deployte `Pläne` einen korrekten Endpunkt mittels des Endpoint Services gespeichert haben.

Diese Endpoints können dabei je nach Art des `Implementation Artifacts` unterschiedlich aussehen. Der Endpoint eines als SOAP Web Service implementierten `Implementation Artifacts` würde beispielsweise als eine URL, welche die Adresse zur WSDL-Datei des SOAP Web Services angibt, gespeichert sein. Ein Endpunkt eines als OSGi-Service implementiertes `Implementation Artifact` würde dagegen, wie bereits in 4.3 dargestellt, durch die ID dieses OSGi-Services dargestellt werden. Die verschiedenen Plug-ins der Service Invocation Schnittstelle wissen dann, wie diese verschiedenen Endpoints interpretiert werden müssen.

Des Weiteren müssen die deployten `Implementation Artifacts` und `Pläne` (zum Aufrufen) aus der OpenTOSCA Container Umgebung erreichbar sein.

Nutzung vorgegebener Interfaces

Neue, zu entwickelnde SI-Plug-ins müssen entsprechend den vorhandenen Plug-ins das dafür vorgegebene Interface (siehe Kapitel 5.5) implementieren. Dieses ist für die korrekte Funktionsweise des durch OSGi realisierten Plug-in Systems notwendig (siehe OSGi Grundlagen 2.3).

Außerdem müssen sich die angegebenen Invocation-Arten, welche gleichzeitig zur Identifizierung der Plug-ins genutzt werden, voneinander unterscheiden.

Weiterhin müssen, für das im vorigen Abschnitt dargestellte Konzept (siehe Kapitel 4.3) eines für OSGi-`Implementation Artifacts` entwickelten Plug-ins, diese `Implementation Artifacts` ebenfalls das dafür vorgesehene Interface implementieren.

Benötigte Informationen innerhalb der TOSCA Definition

Benötigte Informationen, wie zum Beispiel die Invocation-Art eines `Implementation Artifacts`, müssen in der TOSCA Definition angegeben sein. Diese Informationen werden von der Service Invocation Schnittstelle unter anderem zur Bestimmung des passenden SI-Plug-ins benutzt. Ohne diese Daten kann kein passendes Plug-in gefunden werden und das Aufrufen des gewünschten `Implementation Artifacts` schlägt fehl.

Weiterhin sollten auch die optionalen Input und Output Werte der Operationen der `NodeTypes` zur Bestimmung des message exchange patterns (MEP) angegeben sein. Falls nicht angegeben, wird ansonsten standardmäßig von request-response ausgegangen.

Gebrauch des Enums

Das in Kapitel 4.4.1 vorgestellte SI-Enum ist zum Austausch der benötigten Informationen zwingend notwendig. Es definiert die Header der Exchange Message und stellt somit ein standardisiertes Format sicher.

Neue Service Invocation APIs müssen deshalb, äquivalent der bestehenden Service Invocation SOAP API, dieses Enum nutzen. Andernfalls kann die SI-Engine benötigte Informationen nicht auslesen und der gewünschte Aufruf des `Implementation` `Artifacts` oder `Plans` kann nicht ausgeführt werden.

7 Überprüfung des Konzepts und der Implementierung

In diesem Kapitel wird beschrieben, ob und wie die in Kapitel 3 gestellten Anforderungen in Konzept sowie Implementierung der Service Invocation Schnittstelle umgesetzt wurden.

Die erste aufgestellte Anforderung war, dass per Service Invocation Schnittstelle sowohl `Implementation Artifacts` als auch `Pläne` aufgerufen werden können. Dass diese Anforderung umgesetzt wurde, zeigt beispielsweise Kapitel 5.4. Dort wird die Implementierung der SI-Engine vorgestellt und gleichzeitig gezeigt wie Aufrufe von `Implementation Artifacts` als auch `Plänen` bearbeitet werden. Weiterhin sollte die Service Invocation Schnittstelle der, in einer anderen Bachelorarbeit entwickelten, Plan Invocation Engine (siehe Kapitel 2.5) nutzbar gemacht werden. Dies wurde durch die OSGi-Event-API umgesetzt, welche in Kapitel 5.3 vorgestellt wurde.

Eine weitere gestellte Anforderung war, asynchrone Aufrufe zu ermöglichen. Eine beispielhafte Bearbeitung eines asynchronen Aufrufes der SOAP-API samt funktionsweise von MessageID sowie ReplyTo (Callback-Adresse) wurde in Kapitel 4.3 gezeigt. Darüber hinaus wurde in Kapitel 5.3 dargestellt, wie die Plan Invocation Engine mittels Publish-Subscribe Pattern asynchron mit der Service Invocation Schnittstelle kommunizieren kann.

Weiterhin sollten zum Aufruf eines Services benötigte Informationen dynamisch beschafft werden können. Diese Anforderung wurde durch die Anbindung von TOSCA Engine sowie Endpoint Service umgesetzt. Das in Kapitel 4.3 abgebildete Sequenzdiagramm zeigt die Anbindung der genannten Komponenten an die Service Invocation Schnittstelle. Außerdem wird in Kapitel 5.6 dargelegt, wie das SOAP/HTTP-Plug-in anhand der WSDL eines Web Services beispielsweise das MEP bestimmt.

Eine große Anzahl an unterstützten Invocation-Arten, beziehungsweise die Möglichkeit diese einfach erweitern zu können, war eine weitere gestellte Anforderung. Um dies zu ermöglichen, wurde die SI-Engine mit einem Plug-in System (siehe Kapitel 5.4) ausgestattet, welches das Hinzufügen und Starten von Plug-ins zur Laufzeit ermöglicht. Weiterhin wurde Camel, welches eine Vielzahl an Komponenten für verschiedene Standards und Protokollen bietet (siehe Kapitel 2.7), zur Implementierung der Service Invocation Schnittstelle genutzt.

Eine andere Anforderung war, die Übergabe der Input-Daten sowohl per HashMap [34] als auch (XML-)Document [33] zu ermöglichen. In unter anderem Kapitel 4.1 sowie Kapitel 5.3 wird auf die Möglichkeit zur Übergabe von verschiedenen Datentypen eingegangen sowie in Kapitel 4.3 ein Beispielaufruf für die Übergabe einer HashMap dargestellt.

Darüber hinaus sollte die Service Invocation Schnittstelle, um zuvor abgelegte Instanzdaten abfragen und damit die übergebenen Parameter aktualisieren oder ergänzen zu können, mit dem Instanz Data Service kommunizieren können. Die Anbindung des Instanz Data Service wird in Kapitel 4.3 gezeigt sowie anhand eines Beispiels beschrieben.

Des Weiteren sollte die Service Invocation Schnittstelle in den bestehenden OpenTOSCA Container (siehe Kapitel 2.4) integriert werden. Um dies zu ermöglichen, wurde sie per OSGi (siehe Kapitel 2.3) realisiert (siehe beispielsweise Kapitel 5.4).

Eine weitere Anforderung war, dass sich die Service Invocation Schnittstelle einfach erweitern lassen soll. Dies wird unter anderem durch die Verwendung von Camel und des damit verbundenen Pipes-Filter Pattern (siehe Kapitel 2.7), wodurch sich weitere Verarbeitungsschritte einfach in eine Route integrieren lassen, sowie des Plug-in Systems (siehe Kapitel 5.4) ermöglicht. Zudem lassen sich neue APIs einfach integrieren, da alle Daten zwischen den einzelnen SI-Komponenten durch die Nutzung des Exchange Objekts und SI-Enums einheitlich übergeben werden (siehe Kapitel 4.3 sowie Kapitel 5.1).

Weiterhin sollte die Service Invocation Schnittstelle Aufrufe verschiedener Aufrufer parallel und ohne merkbare Leistungseinbußen bewerkstelligen können. Dies wird zum Einen durch die Möglichkeit von asynchronen Aufrufen (siehe Kapitel 4.3) sowie durch die Implementierung der einzelnen SI-Komponenten als OSGi-Services (siehe Kapitel 4.2) und zum Anderen durch die Nutzung von Camel als Integrationsframework (siehe Kapitel 4.1) realisiert.

Die letzte aufgestellte Anforderung war, die Service Invocation Schnittstelle möglichst flexibel bezüglich Datenformate umzusetzen. Dies wird ebenfalls durch die Verwendung von Camel erreicht. Camel bietet durch die Vielzahl an Komponenten (siehe Kapitel 2.7) bereits nativ eine große Anzahl an unterstützten Datenformaten sowie Typ-Konverter an, welche sich zudem noch durch eigene erweitern lässt. Zudem lassen sich im Body des verwendeten Message Objekts beliebige Datentypen ablegen. Kapitel 4.3 zeigt dies anhand eines Beispiels.

8 Zusammenfassung und Ausblick

Im Folgenden werden die wichtigen Erkenntnisse dieser Arbeit nochmals zusammengefasst sowie einen Ausblick auf Aspekte zur weiteren Bearbeitung gegebenen.

Ziel der vorliegenden Bachelorarbeit war es, ein Konzept für eine Erweiterung des OpenTOSCA Containers zum generischen Aufruf von in TOSCA referenzierten Services zu erarbeiten und dieses in die Praxis umzusetzen. Zu diesem Zweck wurde nach der Erarbeitung der zum Verständnis benötigten Grundlagen (Kapitel 2) ein Anforderungskatalog (Kapitel 3) erstellt. Unter Berücksichtigung dessen wurde ein Überblick über mögliche Techniken und konkrete Implementierungen zur Integration verschiedenster Komponenten geschaffen sowie darauf beruhende Entwurfsentscheidungen getroffen (Kapitel 4.1). Dabei hat sich ergeben, dass aufgrund der vorgestellten Vorteile, Apache Camel die beste Alternative zur Umsetzung darstellt.

Darauf aufbauend wurde die Architektur der zu entwickelnden Komponente entworfen (Kapitel 4.2) und das dazugehörige Konzept erarbeitet (Kapitel 4.3). Von besonderer Bedeutung hierfür war die Integration der benötigten bestehenden Komponenten in die zu entwickelnde Erweiterung. Anhand der im weiteren Verlauf dieser Arbeit aufgezeigten Möglichkeiten konnte dargelegt werden, dass das zuvor erarbeitete Lösungskonzept den geforderten Anforderungen entspricht.

Im anschließenden Implementierungsteil (Kapitel 5) wurden die bis dahin abstrakten Beschreibungen der einzelnen Komponenten und deren Funktionalität durch ausgewählte Code- beziehungsweise XML-Auszüge oder Abbildungen konkretisiert und erläutert. Dabei konnte ebenfalls beispielhaft gezeigt werden, wie einfach andere Erweiterungen von OpenTOSCA die in dieser Arbeit entwickelte Komponente nutzen können (Kapitel 5.3).

Jedoch mussten auch, um die korrekte Funktionsweise der entworfenen Komponente zu sichern, einige Annahmen bezüglich TOSCA Definitionen, `Plänen` oder Komponenten-Erweiterungen getroffen werden (Kapitel 6).

Insgesamt kann festgestellt werden, dass mit dieser Arbeit ein Konzept sowie eine erste prototypische Implementierung davon zum Aufruf von Services im Kontext von TOSCA geschaffen wurde. Aktuell wurden sowohl eine SOAP-HTTP-API (Kapitel 5.2) als auch eine OSGi-Event-API (Kapitel 5.3) zum Aufruf der Service Invocation Schnittstelle implementiert. Weiterhin wurde ein SOAP/HTTP-Plug-in (Kapitel 5.6) zum Aufruf von SOAP Web Services sowie ein REST/HTTP-Plug-in (Kapitel 5.7) umgesetzt. Um eine größer Anzahl an Standards und Protokollen zu unterstützen, müssen weitere Plug-ins und APIs implementiert werden. Aufgrund der darauf ausgerichteten Konzeption, stellt das Einbinden von neuen Komponenten allerdings keine große Schwierigkeit dar.

Abbildungsverzeichnis

Listingsverzeichnis

Abkürzungsverzeichnis

AAR	Axis Archive
API	Application Programming Interface
AWS	Amazon Web Services
BPEL	WS-Business Process Execution Language
BPMN	Business Process Model and Notation
BPS	Business Process Server
CSAR	Cloud Service Archive
DSL	Domain Specific Language
EIP	Enterprise Integration Pattern
ESB	Enterprise Service Bus
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IA	Implementation Artifact
IaaS	Infrastructure as a Service
ID	Identifikationsbezeichnung
IDE	Integrated Development Environment
IT	Informationstechnik
ITK	Informations- und Kommunikationstechnik
JAR	Java Archive
MEP	Message Exchange Pattern
OASIS	Organization for the Advancement of Structured Information Standards
PaaS	Platform as a Service
REST	Representational State Transfer
SaaS	Software as a Service
SIS	Service Invocation Schnittstelle
TOSCA	Topology and Orchestration Specification for Cloud Applications
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WADL	Web Application Description Language
WAR	Web Application Archive
WSDL	Web Services Description Language
XML	Extensible Markup Language
XSD	XML Schema Definition

Literaturverzeichnis

[1] Apache Software Foundation: Apache Axis, URL: http://axis.apache.org/

[2] Apache Software Foundation: Apache Camel , URL: http://camel.apache.org/

[3] Apache Software Foundation: Apache Camel: Enterprise Integration Patterns, URL: http://camel.apache.org/enterprise-integration-patterns.html

[4] Apache Software Foundation: Apache ServiceMix, URL: http://servicemix.apache.org/

[5] Apache Software Foundation: Apache Synapse Enterprise Service Bus (ESB), URL: http://synapse.apache.org/

[6] Apache Software Foundation: Apache Tomcat, URL: http://tomcat.apache.org/

[7] Apache Software Foundation: Components, URL: http://camel.apache.org/component.html

[8] Apache Software Foundation: CXF Component, URL: http://camel.apache.org/cxf

[9] Apache Software Foundation: FTP/SFTP/FTPS Component, URL: http://camel.apache.org/ftp2.html

[10] Apache Software Foundation: Getting Started with Apache Camel, URL: http://camel.apache.org/book-getting-started.html

[11] Apache Software Foundation: Is Camel an ESB?, URL: http://camel.apache.org/is-camel-an-esb.html

[12] Apache Software Foundation: Jetty Component, URL: http://camel.apache.org/jetty

[13] Apache Software Foundation: Welcome to Apache Axis2/Java, URL: http://axis.apache.org/axis2/java/core/

[14] AWS: Amazon EC2, URL: http://aws.amazon.com/de/ec2/

[15] BITKOM: Die wichtigsten Hightech-Themen 2013, URL: http://www.bitkom.org/de/presse/30739_74757.aspx

[16] Chappell, David A. : Enterprise Service Bus, O'Reilly Media 2004

[17] Eclipse: Equinox, URL: http://www.eclipse.org/equinox/

[18] Endres, Christian: Management von Cloud Applikationen in OpenTOSCA. Cloud Application Management in OpenTOSCA, Bachelorarbeit, 03.05.2013

[19] Fielding, Roy Thomas: Architectural Styles and the Design of Network-based Software Architectures URL: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[20] Google: Gmail, URL: https://mail.google.com/

[21] Hohpe , Gregor/ Woolf , Bobby: Pipes and Filters, URL: http://www.eaipatterns.com/PipesAndFilters.html

[22] Hohpe , Gregor/ Woolf , Bobby: Publish-Subscribe Channel, URL: http://www.eaipatterns.com/PublishSubscribeChannel.html

[23] IAAS Universität Stuttgart: OpenTOSCA - Open Source Laufzeitumgebung für TOSCA, URL: http://www.iaas.uni-stuttgart.de/OpenTOSCA/

[24] IT Wissen: Community Cloud, URL: http://www.itwissen.info/definition/lexikon/Community-Cloud-community-cloud.html

[25] Manhart, Klaus: Cloud Dienste - Das müssen Sie wissen! Cloud Computing - SaaS, PaaS, IaaS, Public und Private (02.08.2011), URL: http://www.tecchannel.de/server/cloud_computing/2030180/cloud_computing_das_muessen_sie_wissen_saas_paas_iaas/

[26] Microsoft: Windows Azure, URL: http://www.windowsazure.com/de-de/

[27] Mule Community: Mule ESB - The easiest way to integrate anything, anywhere, URL: http://www.mulesoft.org/

[28] OASIS: Advancing open standards fort he information society, URL: https://www.oasis-open.org

[29] OASIS: Topology and Orchestration Specification for Cloud Applications Version 1.0, URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.html

[30] OASIS: Web Services Business Process Execution Language, URL: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

[31] OMG: Business Process Model and Notation , URL: http://www.bpmn.org/

[32] Opscode: Chef, URL: http://www.opscode.com/chef/

[33] Oracle: Document (Java Platform SE 6), URL: http://docs.oracle.com/javase/6/docs/api/org/w3c/dom/Document.html

[34] Oracle: HashMapURL (Java Platform SE 6), URL: http://docs.oracle.com/javase/6/docs/api/java/util/HashMap.html

[35] OSGi Alliance: OSGi Service Platform Core Specification (Release 4, Version 4.2, June 2009), URL: http://www.osgi.org/download/r4v42/r4.core.pdf

[36] OSGi Alliance: OSGi Service Platform Service Compendium (Release 4, Version 4.2, August 2009), URL: http://www.osgi.org/download/r4v42/r4.cmpn.pdf

[37] OSGi Alliance: OSGi - The Dynamic Module System for Java, URL: http://www.osgi.org/Main/HomePage

[38] Puppet Labs: IT Automation Software for System Administrators, URL: https://puppetlabs.com/

[39] Spring: Spring Integration, URL: http://www.springsource.org/spring-integration

[40] W3C: SOAP Specifications, URL: http://www.w3.org/TR/soap/

[41] W3C: Web Services Description Language (WSDL) Version 2.0 Part 2: Predefined Extensions, URL: http://www.w3.org/TR/2004/WD-wsdl20-extensions-20040803/#patterns

[42] W3C: Web Services Addressing (WS-Addressing), URL: http://www.w3.org/Submission/ws-addressing/

[43] Wähner, Kai: Mittler zwischen den Welten. Freie Integrations-Frameworks auf der Java-Plattform (13.08.2012), URL: http://www.heise.de/developer/artikel/Freie-Integrations-Frameworks-auf-der-Java-Plattform-1666403.html

[44] Wissmeier, Jörg / Kraus, Adrian: Top Ten ESB: Viele ESBs, viele Möglichkeiten, URL: http://jaxenter.de/artikel/Top-Ten-ESB-Viele-ESBs-viele-Moeglichkeiten

[45] WSO2: WSO2 Business Process Server , URL: http://wso2.com/products/business-process-server/

Alle aufgeführten Weblinks wurden das letzte Mal am 1.07.2013 geprüft.

Anhang A. WSDL der SOAP-API

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <wsdl:definitions xmlns:xsd="http://www.w3.
03                             org/2001/XMLSchema"
04     xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:
05            tns="http://siserver.org/wsdl"
06     xmlns:ns="http://siserver.org/schema" xmlns:soap="http:
07           //schemas.xmlsoap.org/wsdl/soap/"
08     name="SIServerImplService" targetNamespace="http://siserver.
09       org/wsdl">
10
11     <wsdl:types>
12            <xsd:schema xmlns="http://www.w3.org/2001/XMLSchema">
13                   <xsd:import namespace="http://siserver.org/schema"
14                          schemaLocation="invoker.xsd" />
15            </xsd:schema>
16     </wsdl:types>
17     <wsdl:message name="invokeOperationMessage">
18            <wsdl:part element="ns:invokeOperation"
19                     name="invokeOperation">
20            </wsdl:part>
21     </wsdl:message>
22     <wsdl:message name="invokeOperationAsyncMessage">
23            <wsdl:part element="ns:invokeOperationAsync"
24                     name="invokeOperationAsync">
25            </wsdl:part>
26     </wsdl:message>
27     <wsdl:message name="invokeOperationSyncMessage">
28            <wsdl:part element="ns:invokeOperationSync"
29                     name="invokeOperationSync">
30            </wsdl:part>
31     </wsdl:message>
32     <wsdl:message name="invokePlanMessage">
33            <wsdl:part element="ns:invokePlan" name="invokePlan">
34            </wsdl:part>
35     </wsdl:message>
36     <wsdl:message name="invokeResponse">
37            <wsdl:part element="ns:invokeResponse"
38                     name="invokeResponse">
39            </wsdl:part>
40     </wsdl:message>
41     <wsdl:portType name="InvokePortType">
42            <wsdl:operation name="invokeOperation">
43                   <wsdl:input message="tns:invokeOperationMessage">
44                   </wsdl:input>
45            </wsdl:operation>
46            <wsdl:operation name="invokeOperationAsync">
47                   <wsdl:input message="tns:invokeOperationAsyncMessage">
48                   </wsdl:input>
49            </wsdl:operation>
50            <wsdl:operation name="invokeOperationSync">
51                   <wsdl:input message="tns:invokeOperationSyncMessage">
52                   </wsdl:input>
53                   <wsdl:output message="tns:invokeResponse">
54                   </wsdl:output>
```

```
55            </wsdl:operation>
56            <wsdl:operation name="invokePlan">
57                  <wsdl:input message="tns:invokePlanMessage">
58                  </wsdl:input>
59            </wsdl:operation>
60      </wsdl:portType>
61      <wsdl:portType name="CallbackPortType">
62            <wsdl:operation name="callback">
63                  <wsdl:input message="tns:invokeResponse">
64                  </wsdl:input>
65            </wsdl:operation>
66      </wsdl:portType>
67      <wsdl:binding name="InvokeBinding" type="tns:
68                    InvokePortType">
69            <soap:binding style="document"
70                  transport="http://schemas.xmlsoap.org/soap/http" />
71            <wsdl:operation name="invokeOperation">
72                  <soap:operation soapAction="http://siserver.
73                              org/invokeOperation"
74                        style="document" />
75                  <wsdl:input>
76                        <soap:body use="literal" />
77                  </wsdl:input>
78            </wsdl:operation>
79            <wsdl:operation name="invokeOperationAsync">
80                  <soap:operation soapAction="http://siserver.
81                              org/invokeOperationAsync"
82                        style="document" />
83                  <wsdl:input>
84                        <soap:body use="literal" />
85                  </wsdl:input>
86            </wsdl:operation>
87            <wsdl:operation name="invokeOperationSync">
88                  <soap:operation soapAction="http://siserver.
89                              org/invokeOperationSync"
90                        style="document" />
91                  <wsdl:input>
92                        <soap:body use="literal" />
93                  </wsdl:input>
94                  <wsdl:output>
95                        <soap:body use="literal" />
96                  </wsdl:output>
97            </wsdl:operation>
98            <wsdl:operation name="invokePlan">
99                  <soap:operation soapAction="http://siserver.
100                             org/invokePlan"
101                       style="document" />
102                 <wsdl:input>
103                       <soap:body use="literal" />
104                 </wsdl:input>
105           </wsdl:operation>
106     </wsdl:binding>
107     <wsdl:binding name="CallbackBinding" type="tns:
108                   CallbackPortType">
109           <soap:binding style="document"
110                 transport="http://schemas.xmlsoap.org/soap/http" />
111           <wsdl:operation name="callback">
112                 <wsdl:input>
113                       <soap:body use="literal" />
114                 </wsdl:input>
```

```
115           </wsdl:operation>
116     </wsdl:binding>
117     <wsdl:service name="InvokerService">
118           <wsdl:port binding="tns:InvokeBinding" name="InvokePort">
119                 <soap:address location="http://localhost:8081/invoker" />
120           </wsdl:port>
121           <wsdl:port binding="tns:CallbackBinding"
122                    name="CallbackPort">
123                 <soap:address location="http://localhost:8088/callback" />
124           </wsdl:port>
125     </wsdl:service>
126 </wsdl:definitions>
```

Anhang B. XSD der SOAP-API

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
03     xmlns:ns="http://siserver.org/schema"
04          attributeFormDefault="unqualified"
05     elementFormDefault="unqualified" targetNamespace="http:
06                    //siserver.org/schema">
07     <xs:complexType name="ParamsMapItemType">
08           <xs:sequence>
09                 <xs:element name="key" type="xs:string" />
10                 <xs:element name="value" type="xs:string" />
11           </xs:sequence>
12     </xs:complexType>
13     <xs:complexType name="ParamsMap">
14           <xs:sequence>
15                 <xs:element maxOccurs="unbounded" name="Param"
16                        type="ns:ParamsMapItemType" />
17           </xs:sequence>
18     </xs:complexType>
19           <xs:complexType name="Doc">
20           <xs:sequence>
21                 <xs:any minOccurs="0" maxOccurs="1"
22                   processContents="skip"/>
23           </xs:sequence>
24     </xs:complexType>
25     <xs:element name="invokeOperation" type="ns:
26                  invokeOperationAsync" />
27     <xs:element name="invokeOperationAsync" type="ns:
28                  invokeOperationAsync" />
29     <xs:complexType name="invokeOperationAsync">
30           <xs:sequence>
31                 <xs:element minOccurs="1" maxOccurs="1" name="CsarID"
32                        type="xs:string" />
33                 <xs:element minOccurs="0" maxOccurs="1"
34                       name="ServiceInstanceID"
35                        type="xs:string" />
36                 <xs:element minOccurs="0" maxOccurs="1"
37                       name="NodeInstanceID"
38                        type="xs:string" />
39                 <xs:element minOccurs="1" maxOccurs="1"
```

```
40                      name="ServiceTemplateIDNamespaceURI" type="xs:string" />
41                <xs:element minOccurs="1" maxOccurs="1"
42                      name="ServiceTemplateIDLocalPart" type="xs:string" />
43                <xs:element minOccurs="1" maxOccurs="1"
44                     name="NodeTemplateID"
45                      type="xs:string" />
46                <xs:element minOccurs="0" maxOccurs="1"
47                     name="InterfaceName"
48                      type="xs:string" />
49                <xs:element minOccurs="1" maxOccurs="1"
50                     name="OperationName"
51                      type="xs:string" />
52                <xs:element minOccurs="1" maxOccurs="1" name="ReplyTo"
53                      type="xs:string" />
54                <xs:element minOccurs="1" maxOccurs="1" name="MessageID"
55                      type="xs:string" />
56                <xs:choice>
57                      <xs:element minOccurs="0" name="Params" type="ns:
58                      ParamsMap" />
59                      <xs:element minOccurs="0" name="Doc" type="ns:Doc" />
60                </xs:choice>
61          </xs:sequence>
62    </xs:complexType>
63    <xs:element name="invokeOperationSync" type="ns:
64                invokeOperationSync" />
65    <xs:complexType name="invokeOperationSync">
66          <xs:sequence>
67                <xs:element minOccurs="1" maxOccurs="1" name="CsarID"
68                      type="xs:string" />
69                <xs:element minOccurs="0" maxOccurs="1"
70                     name="ServiceInstanceID"
71                      type="xs:string" />
72                <xs:element minOccurs="0" maxOccurs="1"
73                     name="NodeInstanceID"
74                      type="xs:string" />
75                <xs:element minOccurs="1" maxOccurs="1"
76                      name="ServiceTemplateIDNamespaceURI" type="xs:string" />
77                <xs:element minOccurs="1" maxOccurs="1"
78                      name="ServiceTemplateIDLocalPart" type="xs:string" />
79                <xs:element minOccurs="1" maxOccurs="1"
80                     name="NodeTemplateID"
81                      type="xs:string" />
82                <xs:element minOccurs="0" maxOccurs="1"
83                     name="InterfaceName"
84                      type="xs:string" />
85                <xs:element minOccurs="1" maxOccurs="1"
86                     name="OperationName"
87                      type="xs:string" />
88                <xs:choice>
89                      <xs:element minOccurs="0" name="Params" type="ns:
90                      ParamsMap" />
91                      <xs:element minOccurs="0" name="Doc" type="ns:Doc" />
92                </xs:choice>
93          </xs:sequence>
94    </xs:complexType>
95    <xs:element name="invokePlan" type="ns:invokePlan" />
96    <xs:complexType name="invokePlan">
97          <xs:sequence>
98                <xs:element minOccurs="1" maxOccurs="1" name="CsarID"
99                      type="xs:string" />
```

```
100                 <xs:element minOccurs="0" maxOccurs="1"
101                         name="ServiceInstanceID"
102                         type="xs:string" />
103                 <xs:element minOccurs="1" maxOccurs="1"
104                         name="PlanIDNamespaceURI"
105                         type="xs:string" />
106                 <xs:element minOccurs="1" maxOccurs="1"
107                         name="PlanIDLocalPart"
108                         type="xs:string" />
109                 <xs:element minOccurs="1" maxOccurs="1"
110                         name="OperationName"
111                         type="xs:string" />
112                 <xs:element minOccurs="1" maxOccurs="1" name="ReplyTo"
113                         type="xs:string" />
114                 <xs:element minOccurs="1" maxOccurs="1" name="MessageID"
115                         type="xs:string" />
116                 <xs:choice>
117                         <xs:element minOccurs="0" name="Params" type="ns:
118                          ParamsMap" />
119                         <xs:element minOccurs="0" name="Doc" type="ns:Doc" />
120                 </xs:choice>
121         </xs:sequence>
122     </xs:complexType>
123     <xs:element name="invokeResponse" type="ns:invokeResponse"
124                 />
125     <xs:complexType name="invokeResponse">
126         <xs:sequence>
127                 <xs:element minOccurs="0" maxOccurs="1" name="MessageID"
128                         type="xs:string" />
129                 <xs:choice>
130                         <xs:element minOccurs="0" name="Params" type="ns:
131                          ParamsMap" />
132                         <xs:element minOccurs="0" name="Doc" type="ns:Doc" />
133                 </xs:choice>
134         </xs:sequence>
135     </xs:complexType>
136 </xs:schema>
```

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

_____

Ort, Datum, Unterschrift