Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 53

Verwaltung von Instanzdaten eines TOSCA Cloud Services

Marcus Eisele

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Frank Leymann
Betreuer/in:	Dipl.-Inf. Florian Haupt
Beginn am:	30. April 2013
Beendet am:	30. Oktober 2013
CR-Nummer:	C.2.4, H.0

Kurzfassung

Die Topology and Orchestration Specification for Cloud Applications (TOSCA) ermöglicht die portable und interoperable Beschreibung von Cloud Anwendungen, deren Deployment und deren Verwaltung. Die Abfolge der ausgeführten Operationen innerhalb der Anwendungsstruktur kann hierbei durch Management-Pläne modelliert werden.

Die Beschreibungen der Anwendung alleine sind jedoch nicht ausreichend um Instanzen einer solchen sinnvoll zu verwalten. Während der Ausführung der Management-Pläne fallen komponentenspezifische Daten an, deren persistente Speicherung und Bereitstellung gewährleistet sein muss. Die Pläne benötigen während ihrer Ausführung, neben den Instanzdaten, den Zugriff auf anwendungsspezifische Dateien. Zur Sicherstellung der Portabilität der Pläne, muss dieser Zugriff möglichst unabhängig von der eingesetzten TOSCA-Laufzeitumgebung sein. Die Verwaltung von Instanzdaten und Sicherstellung der Portabilität sind beides Aufgaben einer TOSCA-Laufzeitumgebung.

Diese Arbeit identifiziert Anforderungen an einen Dienst, der diese Aufgaben realisiert, und zeigt den Entwurf und die Implementierung eines solchen. Dies wird exemplarisch am Beispiel des OpenTOSCA-Containers, der eine an der Universität Stuttgart entwickelte TOSCA-Laufzeitumgebung ist, durchgeführt. Dieser wird im Zuge dieser Arbeit um eine Instanzdatenverwaltungs- und Portabilitäts-Schnittstelle erweitert.

Inhaltsverzeichnis

Abkürzungsverzeichnis

API	application programming interface
BPEL	WS-Business Process Execution Language
BPMN	Business Process Model and Notation
CMS	Content-Management-System
CSAR	Cloud Service Archive
DA	DeploymentArtifact
FMC	Fundamental Modeling Concepts
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IA	ImplementationArtifact
IAAS	Institut für Architektur von Anwendungssystemen
IaaS	Infrastructure-as-a-Service
ID	Identifier
IDC	International Data Corporation
IPVS	Institut für Parallele und Verteilte Systeme
IT	Informationstechnik
JAXB	Java Architecture for XML Binding
JPA	Java Persistence API
NIST	National Institute of Standards and Technology
ODE	Orchestration Director Engine
PaaS	Platform-as-a-Service
PHP	PHP: Hypertext Preprocessor
QName	qualified Name
REST	Representational State Transfer
SaaS	Software-as-a-Service
SOA	service-oriented architecture
SQL	Structured Query Language
TOSCA	Topology and Orchestration Specification for Cloud Applications
UML	Unified Modeling Language
URI	Uniform Resource Identifier

URL	Uniform Resource Locator
VM	virtuelle Maschine
W3C	World Wide Web Consortium
WAR	Web application ARchive
XLink	XML Linking Language
XML	Extensible Markup Language

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

1 Einleitung

Die Entwicklung und Prognosen der letzten Jahre zeigen, dass *Cloud-Computing* einer der wichtigsten Zweige der Informationstechnik (IT) ist und auch zukünftig bleiben wird. Mitte des Jahres 2012 prognostizierte die International Data Corporation (IDC) bis ins Jahre 2016 eine Annäherung der Ausgaben für öffentliche Cloud-Dienste an die 100 Milliarden Dollar Marke, was einem jährlichem Wachstum von 26,4% in diesem Zeitraum entsprechen würde.[IDC] Einer der Gründe für dieses starke Wachstum ist, dass immer mehr Unternehmen Bereiche ihrer IT-Infrastruktur auslagern. Dieser Vorgang wird als *Outsourcing* bezeichnet und ist besonders beliebt, wenn die ausgelagerten Bereiche nicht Teil des Kerngeschäfts sind.

Hat sich ein Unternehmen erstmals für einen Cloud-Anbieter entschieden, so ist es zunächst an diesen gebunden. Ein Wechsel des Anbieters ist in vielen Fällen für das Unternehmen sehr schwer und nur mit hohen Kosten verbunden möglich. Dieser Sachverhalt wird in der Literatur als *vendor-lockin* bezeichnet.

Gründe für diesen *vendor-lockin* sind vielfältig. Es existieren wenige oder kaum Standards zur Definition der Dienste von Cloud-Anbietern, deshalb bestehen viele Lösungen der Cloud-Anbieter aus proprietären Komponenten. Aus diesem Grund müssen Cloud-Anwendungen häufig, speziell für einen gewissen Anbieter, maßgeschneidert werden. In Folge dessen können bestehende Daten oder gesamte Cloud-Anwendungen häufig nicht, oder nur in Verbindung mit hohen Kosten, migriert werden.

Die Gründe für eine solche Migration sind allerdings vielfältig. Einer dieser Gründe ist die dynamische Preisgestaltung vieler Anbieter, welche zur Folge hat, dass ein Unternehmen, das auf den Anbieter angewiesen ist, keine mittel- oder langfristige Kontrolle über seine eigenen IT-Kosten hat. An dieser Stelle erkennen wir die Notwendigkeit für ein Unternehmen, die *Cloud* permanent zu überwachen um ggf. sehr zeitnah den Anbieter wechseln zu können.[SHI+13, S. 69ff] Topology and Orchestration Specification for Cloud Applications (TOSCA) soll unter anderem genau dieses Problem angehen und lösen.

Motivation von TOSCA ist es Cloud-Computing wertvoller zu machen, indem es gelingt die halb-automatische Erstellung und Verwaltung von Anwendungsschicht (*application layer*) Diensten zwischen verschiedenen Cloud-Implementierungsumgebungen zu portieren, ohne die Zusammenarbeitsfähigkeit (*interoperability*) einzuschränken. TOSCA stellt eine Sprache zur Verfügung, welche ermöglicht Cloud-Dienste und Verwaltungsabläufe zum Erstellen oder Modifizieren dieser Dienste einheitlich zu beschreiben. Ein in TOSCA beschriebener Dienst ist so unter Umständen zu einer Vielzahl von Anbietern kompatibel. Diese Tatsache stellt einen großen Mehrwert für die bisherige Situation des Cloud-Computing dar, denn bisher war es selten oder nur schwer möglich von einem Cloud-Anbieter zu einem anderen zu wechseln oder sogar Dienste dieser beiden zu kombinieren. [OAS13, S.7]

Motivation und Ziel

Bei der Durchführung des Studienprojektes LeGO4TOSCA, das sich mit der Implementierung von TOSCA-konformen Bausteinen, sogenannte *NodeTypes*, beschäftigte, haben wir einige wichtige Erkenntnisse über die Automatisierung des Lebenszyklus von Cloud-Anwendungen erlangen können. Einige der dort erkannten Anforderungen sind nun in diese Bachelorarbeit eingeflossen.

Während des Betriebs von Cloud-Anwendungen existieren und entstehen wichtige komponentenspezifische Daten. Beispielsweise besitzt eine virtuelle Maschine (VM) eine zugewiesene IP-Adresse und ein gestartetes Image hat zugehörige Anmeldedaten (*Credentials*). Wir nennen diese Daten, da sie einer eindeutigen Instanz zuzuordnen sind, Instanzdaten. Außerdem muss regelmäßig bei der automatisierten Installation und Verwaltung einer Cloud-Anwendungen auf anwendungsspezifische Dateien jeglicher Art zugegriffen werden.

Bei der Erstellung der TOSCA wurden diese Anforderungen bereits identifiziert und berücksichtigt. Es gibt sogenannte *Properties* um Instanzdaten zu spezifizieren. Die Funktionalität um auf anwendungsspezifische Daten zuzugreifen wird in TOSCA durch das Prinzip der ImplementationArtifacts (IAs) und DeploymentArtifacts (DAs) abgedeckt.

Um TOSCA-Anwendungen zu interpretieren wird eine TOSCA-Laufzeitumgebung benötigt, die diese Anforderungen erfüllt und diese Prinzipien umsetzt. Eine Implementierung einer solchen Laufzeitumgebung ist der, an der Universität Stuttgart entwickelte, OpenTOSCA-Container. Dieser hatte zu Beginn dieser Bachelorarbeit

jedoch nicht die Möglichkeit diese beiden Anforderungen ausreichend abzudecken und soll durch die Ergebnisse der Arbeit um diese Anforderungen erweitert werden.

Im Rahmen dieser Bachelorarbeit werden deshalb theoretische Anforderungen an einen solchen Dienst zur Instanzdatenverwaltung und Bereitstellung von IAs und DAs identifiziert. Die identifizierten Anforderungen werden als Grundlage für den Entwurf einer internen Schnittstelle für den OpenTOSCA-Container dienen. Diese Schnittstelle wird im Laufe der Arbeit in Form eines OSGi-Bundles realisiert werden. Die Funktionalität der internen Schnittstelle soll auch außerhalb des Containers zur Verfügung stehen, weshalb die bereits bestehende Representational State Transfer (REST)-Schnittstelle des OpenTOSCA-Containers erweitert werden wird.

Gliederung

Im Folgenden werden die Kapitel der Arbeit in chronologischer Reihenfolge dargestellt:

Kapitel 2 – Grundlagen: Hier werden die grundlegenden Begrifflichkeiten, die für das Verständnis der Arbeit erforderlich sind, erläutert. Besonders wird hierbei auf die Zusammenhänge der TOSCA-Spezifikation sowie auf die Architektur des OpenTOSCA-Containers eingegangen.

Kapitel 3 – Anforderungen an einen Dienst zur Instanzdatenverwaltung: Dieses Kapitel beschäftigt sich mit der Identifikation der Anforderungen an die Schnittstellen. Diese Schnittstellen werden besonders hinsichtlich ihres späteren Einsatzes betrachtet.

Kapitel 4 – Entwurf: Entwurf der internen und REST-Schnittstelle, sowie Konzept der Integration in den OpenTOSCA-Container unter Berücksichtigung der, im vorherigem Kapitel identifizierten, Anforderungen.

Kapitel 5 – Implementierung: Dieses Kapitel beschreibt, die für die Implementierung der Schnittstellen notwendigen Details und Besonderheiten. Es stellt eine Ergänzung der vorhergehenden Kapitel dar und baut auf diesen auf.

Kapitel 6 – Validierung des Konzepts und der Implementierung: Prüfung des entworfenen Konzepts und der Implementierung hinsichtlich der Erfüllung der Anforderungen.

Kapitel 7 – Zusammenfassung und Ausblick: In diesem Kapitel werden die Ergebnisse der Arbeit zusammengefasst und Ausblicke auf die weitere Entwicklung aufgezeigt.

Verwendung der englischen Sprache

An vielen Stellen dieser Arbeit werden englische Fachbegriffe benutzt werden. Diese Begriffe werden, sofern es sich nicht um allgemein bekannte Wörter handelt, im Text erklärt oder übersetzt werden. Eine Ausnahme hiervon bilden die Begriffe der englischsprachigen TOSCA-Spezifikation. Um Mehrdeutigkeiten zu vermeiden wurde von einer Übersetzung dieser Begriffe in die deutsche Sprache abgesehen.

2 Grundlagen

Dieses Kapitel soll die für das Verständnis der Arbeit relevanten Grundlagen vermitteln. Hierzu werden sowohl allgemeine Begrifflichkeiten als auch, mit der Thematik verwandte, Technologien erklärt.

2.1 Cloud

Der Begriff des *Cloud-Computing* ist ein weitreichender und es wurde vielmals versucht den Begriff klar und präzise zu definieren. Eine der häufig verwendeten und anerkannten Definitionen ist die Definition nach dem National Institute of Standards and Technology (NIST), die wie folgt von mir in die deutsche Sprache übersetzt wurde:

> Cloud-Computing ist ein Modell um zu jeder Zeit bei Bedarf Netzwerkzugriff zu einem geteilten Pool von Rechenressourcen (z.B. Netzwerk, Server, Speicher, Anwendungen und Dienste) zu erhalten, die mit minimalem Verwaltungsaufwand oder Eingriff seitens des Dienstleisters schnell bereitgestellt oder freigegeben werden können. Dieses Cloud-Modell fördert die Verfügbarkeit und besteht aus fünf essentiellen Charakteristika, drei Dienstmodellen (*service models*) und vier Betriebsmodellen (*deployment models*).[PM11]

Im Wesentlichen unterscheidet sich diese Definition von anderen Definitionen indem hier einerseits auf die drei gängigen Abstraktionsebenen (Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS) und Software-as-a-Service (SaaS)) und andererseits auf die möglichen Liefermodelle (*Private*, *Public*, *Hybrid* und *Community Cloud*) eingegangen wird. [MRV11]

Generell stellt ein Cloud-Anbieter, welcher auch die interne IT-Abteilung sein kann, Dienste zur Verfügung. Abhängig von der Art des vorliegenden Dienstmodells handelt es sich dabei um unterschiedliche Ressourcen. Die verschiedenen Dienstmodelle sollen anhand des Beispiels einer weitverbreiteten Art von Webanwendung,

eines Content-Management-System (CMS)-Systems, erklärt werden. In diesem Beispiel wird davon ausgegangen, dass es sich dabei um eine auf PHP: Hypertext Preprocessor (PHP)-basierende Webanwendung handelt, welche die Datenhaltung mittels eines angebundenen Datenbanksystems erledigt.

Bei IaaS wird Rechnerinfrastruktur, wie VMs, Speicher und Netzwerke, zur Verfügung gestellt, die der Anwender abhängig von seinen Nutzerrechten selbstständig verwalten kann. In den meisten Fällen erhält der Anwender Administratorrechte und ist dann selbst für die Installation, Verwaltung und Pflege der von ihm benötigten Software verantwortlich. In unserem Beispiel bedeutet dies, dass der Anwender nach der Provisionierung einer VM selbstständig einen Webserver und eine Datenbank auf dem System installieren muss. Im Anschluss muss auch die Webanwendung, die unter Umständen noch konfiguriert werden muss, installiert werden.

PaaS hingegen wählt den Ansatz, anstatt der Infrastruktur, die Plattform zur Verfügung zu stellen. Plattformen sind oft Laufzeitumgebungen, Datenbanken oder Serveranwendungen unterschiedlicher Art. Bei dieser Art des Dienstangebots profitiert der Anbieter davon, dass sich mehrere Kunden eine dieser Installationen der Plattformen teilen können. Im Beispiel der Webanwendung sucht sich der Anwender einen Anbieter, der ihm einen Webserver-Dienst mit Datenbank zur Verfügung stellt und installiert auf dieser Plattform dann die Webanwendung, die er im Anschluss nur noch konfigurieren muss.

Bei SaaS gibt es einen Anbieter, der diese Webanwendung als Dienst zur Verfügung stellt. Nach Erhalt der Zugangsdaten kann der Kunde diesen Dienst, im Anschluss an eine ggf. kurze Konfiguration, direkt benutzen.

In der Regel empfiehlt es sich eine umso höhere Abstraktionsebene zu wählen (von IaaS zu SaaS) desto anspruchsloser die Anwendung an ihre Umgebung ist. Wenn eine Anwendung nur in einer sehr speziellen Umgebung läuft, ist es oft unvermeidlich diese Umgebung mittels eines IaaS-Dienstes selbstständig herzustellen.IaaS bietet im Gegensatz zu SaaS zwar viel mehr Möglichkeiten der Konfiguration an, aber mit großer Macht folgt große Verantwortung. Als Kunde ist man bei IaaS für die Konfiguration des Betriebssystems, der Firewall und vieler anderer Dinge selbst verantwortlich. Bei SaaS ist dies Aufgabe des Anbieters.

Die vier Auslieferungsmodelle unterscheiden sich weitgehend darin wer auf die geteilten Ressourcen Zugriff hat.
Die *Private Cloud* stellt den typischen Fall da, wenn ein Unternehmen alleinig auf die in der Cloud existierenden Ressourcen zugreifen kann. Die *Public Cloud* hingegen

ist für die Allgemeinheit geöffnet und prinzipiell jeder kann Ressourcen der Cloud benutzen. Die *Community Cloud* ist typisch für Anwendungen bei den Zusammenarbeit zwischen Unternehmen oder Personen notwendig ist und Datenaustausch zwischen diesen stattfinden muss. Zwischen diesen drei Formen gibt es noch vielerlei Mischformen, die unter dem Überbegriff *Hybrid Cloud* zusammengefasst werden.

2.2 REST

Roy Fielding hat in seiner Dissertation "Architectural Styles and the Design of Network-based Software Architectures" REST als einen Architekturstil für verteilte Hypermedia Systeme vorgestellt. REST definiert eine Menge von architektonischen Einschränkungen (*constraints*) die, wenn sie als Ganzes eingehalten werden, positive Eigenschaften der entwickelten Anwendung betonen. [Fie00]

Diese positiven Eigenschaften kann man auch als Grundkonzepte von REST ansehen. Diese Grundkonzepte umfassen:

Anwendungsübergreifend standardisierte Identifikation Verwenden von menschenlesbaren Uniform Resource Identifiers (URIs) um Instanzen von relevanten Ressourcen der Anwendung zu identifizieren

Hypermedia Verwenden von Links um Ressourcen miteinander zu verbinden. Steuern des Funktionsflusses mittels Links.

Schnittstelle mit fest definierten Mengen von Operationen Jede Ressource unterstützt die gleichen Operationen (*GET*, *POST*, *PUT*, *DELETE*, *HEAD* und *OPTIONS*)

Unterschiedliche Ressourcenpräsentationen Unterschiedliche Repräsentationen für unterschiedliche Anforderungen. Beispielsweise kann für ein und dieselbe URI eine Hypertext Markup Language (HTML)-Datei für die Browser geliefert werden aber auch eine Extensible Markup Language (XML) für Anwendungen, die diese URI aufrufen.

Statuslose Kommunikation Die Kommunikation mit dem Client ist frei von einem Sitzungszustand (*stateless*). Als Konsequenz davon ist ein Client ungebunden von einem speziellen Server und jede zukünftige Anfrage kann auch von

einem anderem Server bearbeitet werden. Dies ist unter Anbetracht der Skalierbarkeit und der Ausfallsicherheit interessant, da jegliche Zugriffe nun frei verteilt werden können.

[Til11]

2.3 TOSCA

Topology and Orchestration Specification for Cloud Applications (TOSCA) ist ein auf XML basierendes Metamodel für die Beschreibung des strukturellen Aufbaus von Cloud-Anwendungen. Diese Struktur kann durch das Ausführen von Plänen in einer geeigneten Laufzeitumgebung (*runtime enviroment*) oder das Interpretieren der definierten Anwendungsstruktur ausgeführt und verwaltet werden. Zu diesen verwaltenden Aufgaben gehören generell Instanziierung, Management und die Terminierung der Instanzen. Je nach Anwendung handelt es sich dabei um unterschiedliche Managementoperationen.

Der folgende Abschnitt basiert auf der TOSCA [OAS13] und soll auf die Konzepte dieser eingehen. Er soll es dem Leser ermöglichen die weiteren Kapitel in einem klarem Kontext zu sehen. Dieses Kapitel erhebt keinerlei Anspruch auf Vollständigkeit, es sollen lediglich die für die Ausarbeitung relevanten Themen erläutert werden.

2.3.1 Verschiedene Ebenen des Modells

TOSCA modelliert auf zwei konzeptionellen Ebenen, auf Type- und Template-Ebene. Zur Laufzeit existiert noch eine dritte Abstraktionsebene, die Instanz-Ebene, welche nicht Teil des Modells der TOSCA ist. Der Zusammenhang zwischen den einzelnen im Folgenden genannten Entitäten ist in Abbildung 2.1 auf der nächsten Seite illustriert. Der folgende Abschnitt erläutert dies anhand der NodeTypes, das Prinzip ist in TOSCA generell anwendbar. Auf *RelationshipTypes*, *RelationshipTemplates* und *RelationshipInstances* soll an dieser Stelle nicht explizit eingegangen werden, sie werden im Abschnitt 2.3.2 erklärt, wenn es um die Beschreibung der Beziehungen geht.

NodeTypes sind wiederverwendbare Entitäten, sie beschreiben abstrakt Komponenten einer Cloud-Anwendung, wie beispielsweise Anwendungen, VMs oder Speicherkomponenten, und definieren so den Typ eines oder mehrerer NodeTemplates.

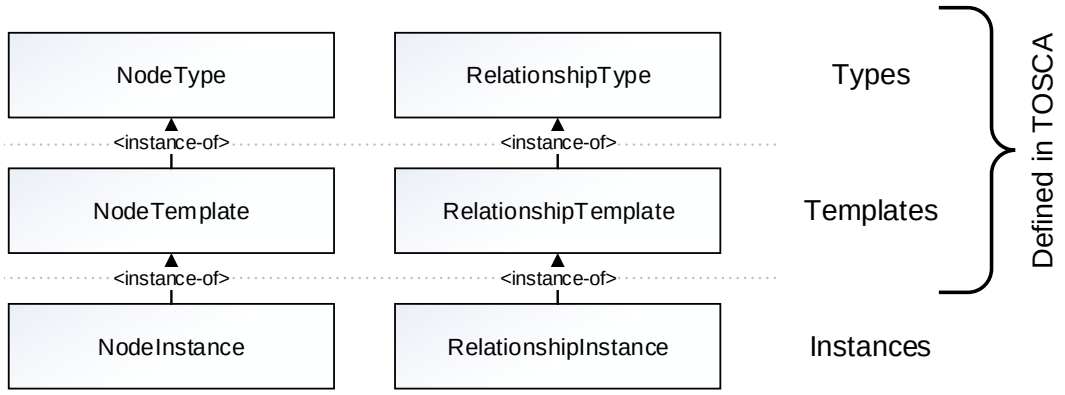

Abbildung 2.1: TOSCA Drei-Schichten-Modell nach [BBLS12]

Die Definition eines NodeTypes spezifiziert die beobachtbaren Eigenschaften. Diese Eigenschaften umfassen die Struktur der Properties, die vom NodeType benötigten Requirements, die zur Verfügung gestellten Capabilities sowie das Interface der unterstützen Management-Operationen des NodeTypes.[OAS13]

NodeTemplates sind Instanziierungen dieser NodeTypes, hierbei werden unter anderem konkrete Werte für die im NodeType definierten Properties gesetzt. NodeTemplates sind konkrete Vorlagen für erzeugbare Instanzen. Wenn ein NodeTemplate instanziiert wird sprechen wir von einer *NodeInstance*. Diese Instanzen repräsentieren reale, instanziierte Komponenten, beispielsweise einen konkreten Apache Webserver.

2.3.2 Templates und Types

Die in diesem Abschnitt beschriebenen Beziehungen zwischen Template und Type sowie NodeTemplate und RelationshipTemplate werden in Abbildung 2.2 auf der nächsten Seite anhand eines Beispiels erläutert.

Ein *TopologyTemplate* definiert die Struktur eines Dienstes. Dafür werden NodeTemplate und RelationshipTemplates benutzt, die zusammen das Topologie-Modell (*topology model*) als einen gerichteten Graphen beschreiben. NodeTemplates und RelationshipTemplates sind hierbei Instanzen von NodeTypes und RelationshipTypes. Ein NodeTemplate spezifiziert so das Vorkommen eines NodeTypes als

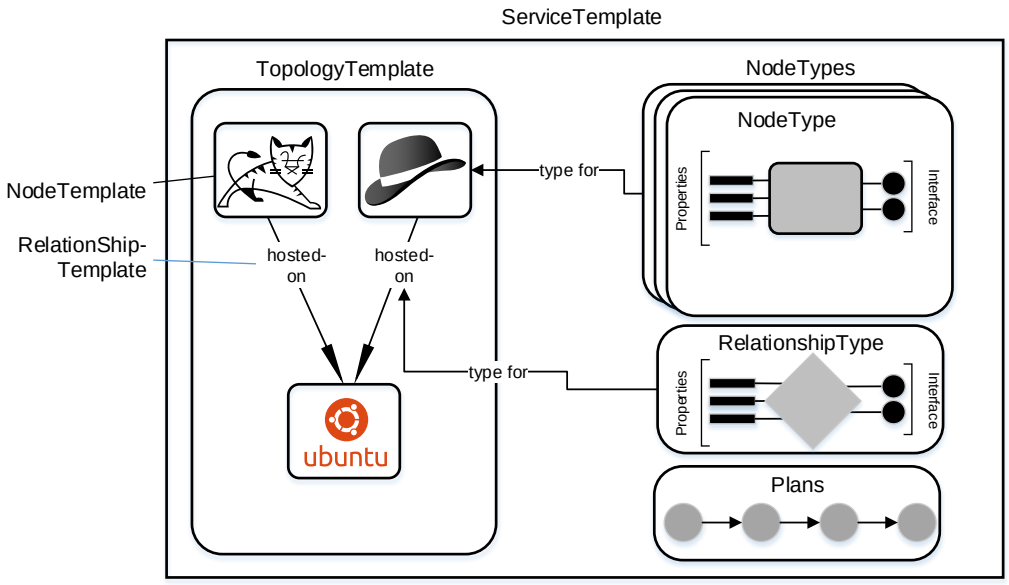

Abbildung 2.2: Zusammenhang zwischen verschiedenen Template-Typen nach [BBLS12]

Servicekomponente. RelationshipTemplates werden dann genutzt um die einzelnen NodeTemplates logisch miteinander zu verknüpfen. [OAS13]

NodeTypes sind deshalb stark auf Wiederverwendbarkeit ausgelegt und enthalten zusätzlich ImplementationArtifacts (IAs) und DeploymentArtifacts (DAs). IAs sind Artefakte, die Operationen des NodeTypes implementieren. [OAS13]

Ein Betriebssystem-NodeType könnte beispielsweise eine Operation zum Absetzen eines Kommandozeilenbefehls besitzen. Das IA dieser Operation könnte als eine REST-Operation implementiert sein und in Form einer Web application ARchive (WAR)-Datei zur Verfügung stehen.

DAs werden benötigt um ein NodeType bzw. ein konkretes NodeTemplate zu instanziieren. In dem vorherigem Beispiel könnte ein typisches DA ein Abbild des verwendeten Betriebssystems sein. Damit ein NodeType mit einem bestimmten TOSCA-Container eingesetzt werden kann muss dieser sowohl IAs als auch DAs unterstützen.

Das *ServiceTemplate* vereint alle diesen Templates und Typen, inklusive der darin enthaltenen Artefakte, zusammen mit den Plänen zu einer ganzheitlichen Servicebeschreibung. Die TOSCA beschreibt im ServiceTemplate lediglich die Struktur der Anwendung. Die Verwaltung, besonders die Erstellung und Terminierung, der Service- und NodeInstances wird von sogenannten Plänen gehandhabt. Diese Pläne sind durch ein Prozessmodell, beispielsweise durch einen ein- oder mehrstufigen Arbeitsablauf (*Workflow*), definiert. Die TOSCA definiert hierfür keine neue Modellierungssprache sondern benutzt bestehende Technologien, wie beispielsweise Business Process Model and Notation (BPMN)[OMG11] oder WS-Business Process Execution Language (BPEL)[OAS07]. [OAS13]

2.3.3 Instanzen, Properties und State

Instanzen sind in dem Kontext dieser Bachelorarbeit instanziierte NodeTemplates. Dies sind die für die Instanzdatenverwaltung relevanten Instanzen. Sie haben konkrete Properties, bei denen es sich um in TOSCA definierte XML-Dokumente handelt, auf die an vielen Stellen zugegriffen werden muss. Properties können prinzipiell als Variablen im Kontext einer Instanz benutzt und angesehen werden, sie können Daten wie IP-Adressen, Passwörter, Installationsordner und sonstige spezifische Werte repräsentieren. IAs benötigen Zugriff auf diese Daten, genauso ist denkbar, dass Pläne diese Daten gezielt lesen und auch manipulieren müssen.

Es ist nicht garantiert, dass immer dasselbe IA zur Ausführung einer Operation genutzt wird oder das jeweilige IA in der Zwischenzeit nicht neu deployed wurde. Deshalb wurde während der Durchführung des LeGO4TOSCA-Projekts die Erfahrung gemacht, dass es sich anbietet den Zustand eines IAs in den Properties der zugehörigen Instanz zu speichern.

Der *State* ist ein in TOSCA definierter Wert, der den Zustand einer Instanz eines NodeTemplates angibt, beispielsweise könnten für eine Webanwendung die Zustände "deployed","running","stopped" und "undeployed" definiert sein. An dieser Stelle sei aber daraufhingewiesen, dass der State aus Sicht des TOSCA-Containers lediglich eine Textrepräsentation ist. TOSCA-Container besitzen keine definierte Logik, wie beispielsweise Monitoring, um diesen Zustand automatisch zu ändern. Für das Modifizieren des States sind also Pläne und IAs verantwortlich.

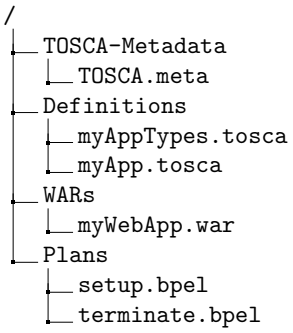

Abbildung 2.3: Beispielstruktur einer gültigen CSAR-Datei

2.3.4 CSAR

Ein Cloud Service Archive ist ein im Zuge der TOSCA spezifiziertes Zip-Archiv mit der Endung $*$.csar. Es paketiert Metadaten und benötigte Artefakte einer Cloud-Anwendung zusammen zu einer Datei. Ein Cloud Service Archive (CSAR) muss mindestens einen TOSCA-Metadata Ordner und einen Definitions Ordner enthalten, darüber hinaus gibt es keine weiteren Einschränkungen an den Aufbau des Archives. [OAS13] Ein Beispiel für eine gültige CSAR stellt die Abbildung 2.3 dar.

2.4 OpenTOSCA

OpenTOSCA[1] ist eine an der Universität Stuttgart entwickelte Open Source TOSCA-Laufzeitumgebung. Eine erste Version dieses TOSCA-Containers entstand 2012 während eines Studienprojekts der beiden Institute Institut für Architektur von Anwendungssystemen (IAAS)[2] und Institut für Parallele und Verteilte Systeme (IPVS)[3] an der Universität Stuttgart und wird seitdem weiterentwickelt.

[1]Webseite des OpenTOSCA-Containers: http://www.iaas.uni-stuttgart.de/OpenTOSCA/
[2]Webseite des IAAS: http://www.iaas.uni-stuttgart.de/
[3]Webseite des IPVS: http://www.ipvs.uni-stuttgart.de/

2.4.1 Architektur

Der folgende Abschnitt soll auf die Architektur und grundlegende Funktionsweise des OpenTOSCA-Containers eingehen, spezieller Fokus soll hierbei auf der Kommunikation zwischen den einzelnen Komponenten der *TOSCA-Engine* liegen. Dazu werden zuerst die einzelnen Komponenten und deren Funktionen und im Anschluss daran der Ablauf eines Deployment-Vorgangs einer CSAR beschrieben.

Grob lässt sich die Architektur des OpenTOSCA-Containers in 6 Komponenten unterteilen. Dies sind die *Container-API* (application programming interface), *TOSCA-Engine*, *IA-Engine*, *Plan-Engine*, sowie die *Control-* und *Core-Komponenten*. Einen groben Überblick vermittelt das dazugehörige Fundamental Modeling Concepts (FMC)-Aufbaudiagramm (s. Abbildung 2.4 auf Seite 25) [BBH+13].

Die *Core-Komponenten* bilden den Kern der Architektur und beschäftigen sich primär mit den grundlegenden Aufgaben wie Persistenz, Dateiverwaltung und der Speicherung von Informationen der anderen Komponenten.

Die *IA-Engine* und *Plan-Engine* haben recht ähnliche Funktionsweisen allerdings unterschiedliche Zuständigkeiten, die IA-Engine kümmert sich um das Deployment von IAs und die Plan-Engine um das Deployment von Plänen. Beide Engines bedienen sich hierbei an auf sie zugeschnitte Plugins, die für eine gewisse Art von Datei (z.B. eine WAR-Datei) Logik zur Verfügung stellen, durch dieses dynamische Plugin-System sind die beiden Engines beliebig erweiterbar.

Die *TOSCA-Engine* ist die Komponente, die sich mit der Verarbeitung der in der CSAR enthaltenden ServiceTemplates beschäftigt. Sie liest, validiert und löst Referenzen innerhalb des TOSCA-XMLs auf und stellt die daraus resultierenden Informationen anderen Komponenten zur Verfügung.

Für den Ablauf des Deployments einer CSAR-Datei ist die *Control-Komponente* verantwortlich, sie stößt Operationen anderer Komponenten an und überprüft zu jeder Zeit ob eine vom Benutzer gestartete Operation im aktuellem Deployment-Zustand auch ausgeführt werden darf.

Als letzte Komponente bildet die *Container-API* die Schnittstelle nach außen. Sie ist eine in Jersey[4] implementierte REST-Schnittstelle und stellt so die für das Verarbeiten einer CSAR notwendige Logik mittels eines Webservices zur Verfügung.

[4]Webseite der Jersey-API: https://jersey.java.net/

Auf diese Schnittstelle wird in Abschnitt 4.8, wenn es um die Erweiterung dieser Schnittstelle geht, noch genauer eingegangen.

Der Ablauf des Deployment einer CSAR sieht beim OpenTOSCA-Container wie folgt aus. Er lässt sich logisch in 4 Schritte unterteilen: Das Hochladen der CSAR-Datei, das anschließende Verarbeiten der hochgeladenen CSAR, das Deployment der IAs und das Deployment der Pläne.

Das Hochladen einer CSAR kann auf mehreren Wegen geschehen. Im Anschluss an das Hochladen veranlasst die Container-API das Entpacken und die nachfolgende Speicherung der in der CSAR enthaltenen Dateien. Das Speichern kann mittels sogenannter StorageProvider angepasst werden, je nach ausgewähltem StorageProvider können diese Dateien entweder auf dem lokalem Dateisystem des Containers oder in einem Cloud-Dateisystem gespeichert werden. Wenn weder die CSAR noch die in der CSAR enthalten Definitionen fehlerhaft sind, wird im Anschluss der Deployment-State in der Control-Komponente auf STORED gesetzt.

Der nächste Schritt im Ablauf ist das Anstoßen der Verarbeitung (processing) der in dem CSAR enthaltenen TOSCA-Definitionen. Dafür muss eine Anfrage an einen, für das CSAR-spezifischen, CSARControl-Pfad der Container-API gesendet werden. Daraufhin wird die Core-Komponente von der Container-API dazu veranlasst die TOSCA-Definitionen zu verarbeiten. Bei der Verarbeitung ist wie bereits vorher erwähnt die TOSCA-Engine maßgeblich beteiligt, sie liest die Definitions ein und löst dabei Referenzen auf und speichert in den Definitions enthaltene Dokumente und Pläne. Nach dem Einlesen ist die TOSCA-Engine bereit Operationen betreffend des CSAR entgegen zu nehmen.

Nun ist es möglich die in den Definitions enthaltenen IAs zu deployen, dies wird mittels REST-Aufruf an den bereits beim Verarbeiten des CSAR benutzten CSARControl-Pfads angestoßen. Dieser Aufruf wird von der Container-API über die Core-Komponente zur IA-Engine delegiert, diese wählt aufgrund der Art des IAs das dazu passende Plugin aus, welches dann das Deployment übernimmt. Dies kann beispielsweise die Installation einer WAR-Datei auf einem lokalen Tomcat-Server sein.

Als letzten Schritt müssen nun noch die Managementpläne auf eine passende Laufzeitumgebung ausgeliefert (deployed) werden. Diese Operation wird mittels eines Aufrufs auf den bereits bekannten CSARControl-Pfad ausgeführt. Bei OpenTOSCA

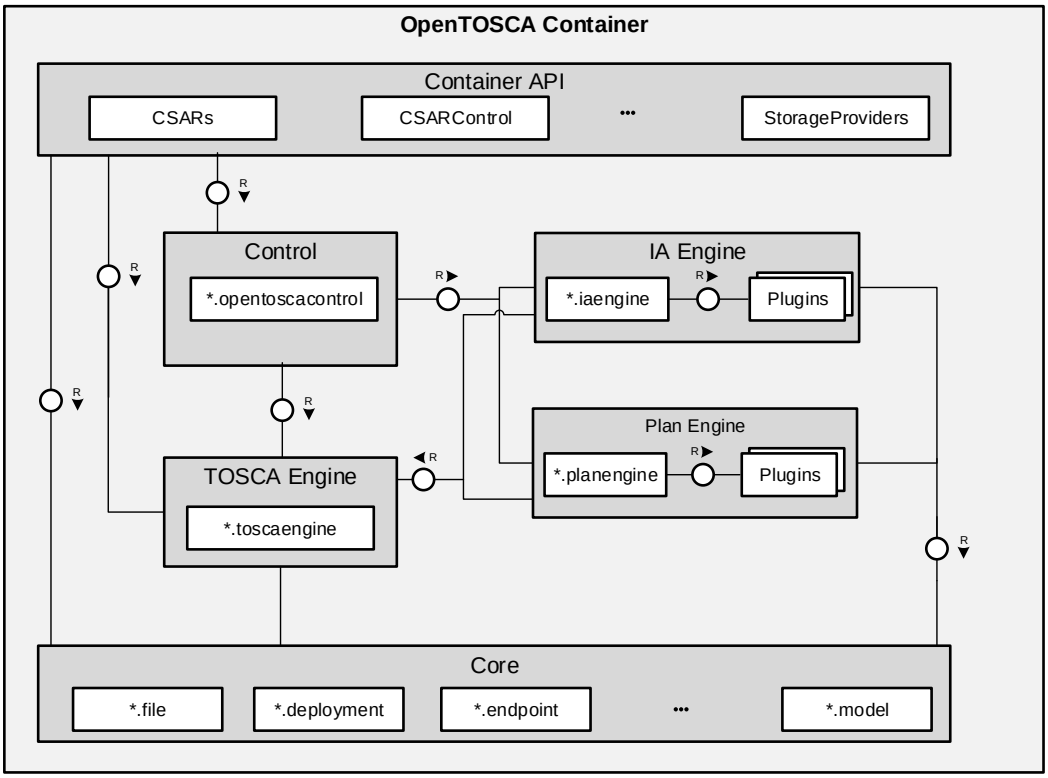

Abbildung 2.4: FMC-Aufbaudiagramm der OpenTOSCA-Struktur nach [BBH+13]

werden dabei als BPEL vorliegende Pläne auf eine lokale Apache Orchestration Director Engine (ODE)[5] deployed.

Nach Ausführung dieser Schritte sind alle IAs korrekt installiert und die durch die deployten Managementpläne zur Verfügung gestellten Funktionalitäten können korrekt verwendet werden.

[5] Apache ODE: http://ode.apache.org/

Listing 2.1 Mit JPA-Annotationen versehene Beispielklasse

```
1  @Entity
2  public class Mitarbeiter {
3
4    @Id
5    private int id;
6
7    @Column(name = "fullName", columnDefinition = "VARCHAR(128)")
8    private name;
9
10   private String getName(){...}
11   private void setName(String name){...}
12 }
```

3 Anforderungen an einen Dienst zur Instanzdatenverwaltung

Dieses Kapitel setzt sich mit den Anforderungen auseinander, die an einen Dienst zur Instanzdatenverwaltung und zur Bereitstellung von IAs und DAs gestellt werden. Diese Anforderungen werden weitgehend aus Sicht eines Nutzers eines TOSCA-Containers (speziell OpenTOSCA-Nutzer) betrachtet werden und beruhen stark auf Erfahrungen, die während des Studienprojektes LeGO4TOSCA bei der Entwicklung von NodeTypes gemacht wurden.

Ein OpenTOSCA-Nutzer kann jeder sein, der einen Webservice benutzen kann, weitgehend werden diese Schnittstellen aber von automatischen Build- bzw. Managementplänen und von IAs benutzt werden um zur Laufzeit Informationen bezüglich NodeInstances oder NodeType-spezifischen Dateien zu erlangen.

Die folgenden Anforderungen konnten im Laufe der Arbeit identifiziert werden und werden in diesem Kapitel an entsprechender Stelle weiter erläutert:

Die Anforderungen im Einzelnen sind unterteilt in Serviceinstanz-spezifische, Nodeinstanz-spezifische, Nodetemplate-spezifische und weitere funktionale Anforderungen. Bevor diese im Detail betrachtet werden, wird nun noch auf die theoretischen Annahmen, die für die folgenden Anforderungen notwendig sind, eingegangen.

3.1 Theoretische Annahmen

In Abbildung 2.2 auf Seite 20 wurde bereits auf die unterschiedlichen Abstraktionsebenen von TOSCA eingegangen. Da hier nun immer konkrete Instanzen dieser Abstraktionsebenen betrachtet werden, muss zuerst überlegt werden mittels welcher Werte eine solche Instanz genau identifiziert werden kann. In der Tabelle 3.1 auf Seite 29 wird auf das Verhältnis zwischen diesen Instanzen und deren Identifiern eingegangen.

Für die Betrachtung der Tabelle 3.1 auf der nächsten Seite ist vorausgesetzt, dass es sich bei *qualified Name (QName)* um einen fully qualified Name handelt, bei dem ein *Namespace*-Element und ein lokaler Teil (*localpart*) angegeben sein muss. An Stellen an denen ein QName nicht explizit einen Namespace benötigt, da dieser implizit aufgrund des Kontexts bereits bekannt ist, wird nur der notwendige *localpart* als Zeichenkette modelliert.
Dies ist bei jeder Operation, bei dem ein NodeTemplate innerhalb eines ServiceTemplates identifiziert wird, der Fall. Das NodeTemplate befindet sich immer im Namensraum (*namespace*) des umschließenden ServiceTemplates — eine Angabe des Namensraums des NodeTemplates ist so überflüssig.

Für den vorliegenden Anwendungsfall der Instanzdatenverwaltung lässt sich die Identifikation aber leicht abwandeln. NodeTemplates sind in ServiceTemplates geschachtelt, sie befinden sich immer im selben Namespace. Bei der Installation einer Cloud-Anwendung werden zuerst die ServiceInstances (die ihr zugehöriges Template kennen) und im Anschluss daran die einzelnen NodeInstances erstellt, durch diese temporale Abhängigkeit kann bei der Instanziierung eines NodeTemplates auf die Angabe des Namespaces verzichtet werden und stattdessen die zugehörige ServiceInstance angegeben werden, die bei der Erstellung sowieso benötigt wird. Im weiteren Verlauf soll also immer wenn es um Identifikation einer NodeInstance geht, der Namespace aus der zugehörigen ServiceInstance bezogen werden. In Tabelle 3.2 auf der nächsten Seite wird der bei der Erstellung einer NodeInstance angepasste und in der Arbeit als Grundlage verwendete Zustand der Beziehung zwischen den Templates und Instanzen aufgezeigt.

3.2 ServiceInstance-spezifische Anforderungen

3.2.1 Erstellen einer ServiceInstance

Eine ServiceInstance ist eine Instanz eines ServiceTemplates welches durch eine ServiceTemplateID identifizierbar ist. Die ToscaEngine des OpenTOSCA-Containers benötigt jedoch zusätzlich die ID der CSAR, die das ServiceTemplate enthält. Ein Nutzer der späteren Schnittstelle muss also die Möglichkeit haben mittels diesen Parametern eine Instanz eines ServiceTemplates zu erzeugen.

Beim Erstellen eines ServiceTemplates hat man die Möglichkeit Kardinalitäten zu spezifizieren, welche die minimale und maximale Anzahl der Vorkommen von NodeTemplates festlegen. Um das Aufsetzen von Topologien zu vereinfachen soll

Instanz	eindeutiger Identifier	Identifier, die bei Erstellung der Instanz angegeben werden müssen
ServiceTemplate	ServiceTemplateID (QName)	—
NodeTemplate	NodeTemplateID (QName)	—
ServiceInstance	ServiceInstanceID (generiert bei Erstellung)	ServiceTemplateID (QName)
NodeInstance	NodeInstanceID (generiert bei Erstellung)	ServiceInstanceID, NodeTemplateID (QName)

Tabelle 3.1: Identifikation von TOSCA-Elementen und Beziehung zwischen Instanzen und Templates

Instanz	eindeutiger Identifier	Identifier, die bei Erstellung der Instanz angegeben werden müssen
ServiceTemplate	ServiceTemplateID (QName)	—
NodeTemplate	NodeTemplateID (QName); alternativ: ServiceTemplateID (QName), NodeTemplateID (String)	—
ServiceInstance	ServiceInstanceID (generiert bei Erstellung)	ServiceTemplateID (QName)
NodeInstance	NodeInstanceID (generiert bei Erstellung)	ServiceInstanceID , NodeTemplateID (String)

Tabelle 3.2: Angepasste Identifikation von TOSCA-Elementen und Beziehung zwischen Instanzen und Templates

bei der Instanziierung eines ServiceTemplates zusätzlich für jedes vorhandene NodeTemplate eine Instanz, bzw. eine NodeInstance, erzeugt werden. Nach Erstellung der ServiceInstance müssen Informationen bezüglich dieser und aller während des Erstellungsprozesses erzeugten NodeInstances mittels eindeutiger IDs abrufbar sein.

3.2.2 Löschen einer ServiceInstance

Für die Löschung einer ServiceInstance muss lediglich die ServiceInstance-ID angegeben werden. Bei dem Löschvorgang müssen außer der Serviceinstance selbst auch noch alle zur Serviceinstance dazugehörigen NodeInstances gelöscht werden. Im Anschluss an die Löschung sollen keinerlei Informationen bezüglich dieser Instanzen mehr existieren. Diese Art von Löschung, bei der abhängige Elemente gelöscht werden, nennt man *cascading delete*.

3.2.3 Abfragen von ServiceInstance-Informationen

Es muss die Möglichkeit bestehen, Informationen bezüglich einer bestimmten ServiceInstance, von der die ID bekannt ist, zu erhalten. Diese Informationen müssen Aufschluss über folgende Details geben:

- Zeitpunkt der Erstellung der Instanz
- zugehörige CSAR-ID
- QName des ServiceTemplates, das als Vorlage für die ServiceInstance diente
- Name des ServiceTemplates, das als Vorlage diente
- Referenzen auf alle NodeInstances, die zu dieser ServiceInstance gehören

3.2.4 Finden von ServiceInstance-IDs anhand von Filtern

Neben der Informationsabfrage zu einer bestimmten ServiceInstance muss auch die Möglichkeit bestehen, mittels der Angabe von Filtern, spezielle ServiceInstances bzw. die IDs der ServiceInstances zu finden. Mögliche Filterkriterien sollten hierbei vor allem der Name des verwendeten ServiceTemplates sein, sowie dessen ID.

Ein gängiger Anwendungsfall für die Verwendung dieser Filter, ist die Frage nach allen Instanzen eines gewissen ServiceTemplates. Die Funktionalität des Findens von ServiceInstances anhand des verwendeten Templates wird dann eine Antwort auf genau diese Frage liefern.

3.2.5 Prüfung der Existenz einer ServiceInstance

Neben der Abfrage von Informationen bezüglich ServiceInstances ist es notwendig auch eine Möglichkeit zu haben, welche die Existenz einer ServiceInstance prüft.

3.3 NodeInstance-spezifische Anforderungen

3.3.1 Erstellen einer NodeInstance

Das Erstellen einer NodeInstance wird einerseits implizit beim Erstellen einer ServiceInstance aufgerufen und kann andererseits auch explizit nach Erstellung einer ServiceInstance veranlasst werden. Bei der Erstellung müssen Parameter angegeben werden, die einen eindeutigen Rückschluss auf die zugehörige ServiceInstance und auf das zu instanziierende NodeTemplate liefern. NodeInstances haben im Gegensatz zu ServiceInstances noch Instanzdaten, für diese Daten können in TOSCA Standardwerte im Template definiert werden, die beim Erstellen einer NodeInstance auch gesetzt werden müssen.

3.3.2 Löschen einer NodeInstance

Die Löschung einer NodeInstance erfordert, dass diese eindeutig identifiziert ist, dafür wird die NodeInstanceID als Parameter benötigt. Nach der Löschung der NodeInstance dürfen keinerlei Informationen dieser mehr verfügbar sein.

3.3.3 Abfragen von NodeInstance-Informationen

Bei NodeInstance Informationen handelt es sich einerseits, ähnlich wie bei der Abfrage von ServiceInstance-Daten, um Meta-Informationen zur Erstellung und andererseits um den sogenannte State und die Properties einer NodeInstance. Was

man unter dem State und den Properties einer NodeInstance versteht wurde bereits in Abschnitt 2.3.3 auf Seite 21 erläutert.

Diese Abfrage muss also folgende Informationen liefern:

- Zeitpunkt der Erstellung der Instanz
- QName des NodeTemplates, das als Vorlage für die NodeInstance diente
- Name des NodeTemplates, das als Vorlage diente
- Referenzen auf die zugehörige ServiceInstance
- State der NodeInstance
- Properties der NodeInstance

3.3.4 Ändern von NodeInstance-Informationen

Das Modifizieren von Instanzdaten ist Hauptaufgabe dieses Dienstes, es muss also gewährleistet sein, dass er diese Aufgabe erledigen kann. Um Instanzdaten ändern zu können muss einerseits die Instanz identifiziert werden und andererseits müssen neue Werte der Informationen in der Anfrage enthalten sein. Nach der Durchführung der Änderungen der Daten einer Instanz müssen die neuen Daten unter der bisher verwendeten Adresse zur Verfügung stehen.

3.3.5 Abfragen des NodeTypes einer NodeInstance

Bei der Entwicklung von IAs im LeGO4TOSCA Studienprojekt wurde an vielen Stellen eine Möglichkeit zur Bestimmung des NodeTypes einer bestimmte NodeInstance benötigt. Besonders wichtig war die Unterscheidung der NodeTypes von Microsoft Windows[1] und Ubuntu[2] um anderen IAs (vor allem IAs von Anwendungs-NodeTypes) eine Information für die interne Entscheidung der einzusetzenden Logik zu bieten. Dies ist ein sicherlich häufiger Anwendungsfall, da sich durch die Unterscheidung in den IAs selbst, Pläne generischer schreiben und vielseitiger einsetzen lassen.

[1]Microsoft Windows: http://windows.microsoft.com/en-us/windows/home
[2]Ubuntu: http://www.ubuntu.com/

3.3.6 Finden von NodeInstance-IDs anhand von Filtern

Neben der Abfrage von Informationen bezüglich NodeInstances muss auch die Möglichkeit bestehen eine NodeInstance, bzw. die ID der NodeInstance, aufgrund von Parametern zu finden. Denkbar ist, dass nach der Erstellung einer ServiceInstance und der impliziten Erstellung der dazugehörigen NodeInstances die ID einer bestimmten NodeInstance unbekannt ist. Nun kann mittels der bekannten ServiceInstance-ID und des NodeTemplates in Erfahrung gebracht werden wie die ID dieser bestimmten Instanz ist.

3.3.7 Prüfung der Existenz einer NodeInstance

Neben der Abfrage von Informationen bezüglich NodeInstances muss auch eine allgemeine Möglichkeit existieren, die überprüft ob eine NodeInstance überhaupt existiert. Dies kann zu Validierungszwecken und zur Fehlerbehandlung sehr wichtig sein.

3.4 NodeTemplate-spezifische Anforderungen

3.4.1 Link zu einem oder mehreren Artefakten eines NodeTemplates erhalten

Diese Anforderung bezieht sich nun auf die Komponente des Diensts, die sich mit dem Bereitstellen von Artefakten beschäftigt. Im LeGO4TOSCA Projekt wurden IAs entwickelt, die jeweils eine Installationsmethode bereit stellten. Bei der Implementierung wurde deutlich, dass diese Methoden die Möglichkeit benötigen auf Daten, die in dem zugehörigem CSAR enthalten sind, zuzugreifen. Problematisch war dies, weil in den *TOSCA-Definitionen* nur relative Pfade zu den DAs angegeben werden. Für die effektive Verwendung muss aber eine Möglichkeit bestehen absolute, d.h. direkt herunterladbare, Referenzen für bestimmte Artefakte zu ermitteln.

Denkbar wäre ein Szenario in dem ein Anwendungs-NodeType die entsprechende Anwendung mittels einer Installations-Methode installiert. Abstrakt betrachtet wird die Installation einen Download einer *.zip-Datei, sowie das Entpacken dieser Datei beinhalten. Für den Download benötigt

das IA nun eine Adresse an der diese Datei heruntergeladen werden kann. Um die häufigsten Anwendungsfälle abzudecken müssen folgende Filtermöglichkeiten bestehen:

- serviceTemplateID
- nodeTemplateID
- Art des gesuchten Artefaktes (DA oder IA)
- Name des Artefaktes

3.5 Weitere funktionale Anforderungen

3.5.1 Persistenz

Die Lebensdauer einer Cloud-Anwendung kann sehr lange sein, es ist also eine sehr realistische Annahme, dass sowohl die Maschine auf der der TOSCA-Container läuft als auch der TOSCA-Container selbst einmal abstürzt oder neugestartet werden muss. Die Instanzdaten müssen nach so einem Vorfall natürlich weiterhin verfügbar sein, da ohne diese Daten die Verwaltung des Dienstes mit hoher Wahrscheinlichkeit unmöglich wird. Der Dienst zur Instanzdatenverwaltung muss seine Daten also persistent speichern.

3.5.2 Integration in bestehende Dienste

Der zu entwickelnde Dienst muss sich in die bisherige Architektur des OpenTOSCA-Containers integrieren und sich so weit wie möglich ähnlicher Technologien wie bereits implementiere Komponenten bedienen. Es muss generell auf Konsistenz zur bestehenden Architektur geachtet werden, besonders wenn es um die Entwicklung der REST-Schnittstelle geht um den späteren Nutzern eine einheitliche Erfahrung mit der Container-API zu ermöglichen. Was dies im Einzelnen bedeutet soll im nachfolgendem Kapitel Entwurf aufgezeigt werden.

4 Entwurf

Dieses Kapitel beschäftigt sich systematisch und konzeptionell mit dem Entwurf der Schnittstelle des internen Dienstes und dem Entwurf der externen REST-Schnittstelle. Es wird ein Konzept ausgearbeitet werden, wie der zu entwickelnde Dienst in die Architektur des OpenTOSCA-Containers zu integrieren ist. Dabei wird zuerst auf den Ist- und Sollzustand eingegangen um im Anschluss sich mit der notwendigen Umsetzung der in Kapitel 3 identifizierten Anforderungen auseinanderzusetzen.

Beim Entwurf des Dienstes soll nach dem *Top-Down-Ansatz*, bei dem die Schnittstellendefinition vor der Implementierung durchgeführt wird, vorgegangen werden. Diese Vorgehensweise bietet sich an, da durch die in der Analyse identifizierten Anforderungen bereits definiert ist welche Operationen dieser Dienst zur Verfügung stellen muss, um den gestellten Anforderungen gerecht zu werden. Der Fokus liegt aufgrund des gewählten Ansatzes deshalb am Anfang dieses Abschnittes auf den Schnittstellen und der Interaktion der einzelnen Komponenten. Gegen Ende des Kapitels wird sich der Entwurf dann detaillierter mit den eingesetzten Technologien und verwendeten Datentypen auseinandersetzen.

4.1 Einschränkungen

Der OpenTOSCA-Container ist *CSAR-aware*. Das bedeutet, dass für die Herstellung eindeutiger Beziehungen zu ServiceTemplates und anderen Templates immer die jeweilige CSAR-ID des dazugehörigen CSAR angegeben werden muss. Diese Abhängigkeit befindet sich in der TOSCA-Engine verankert. Da die zu entwerfenden Dienste die TOSCA-Engine nutzen wird an einigen Stellen des Entwurfs die Angabe von CSAR-IDs notwendig, wobei an den entsprechenden Stellen dadurch eine CSAR-ID in den Parametern enthalten sein wird.

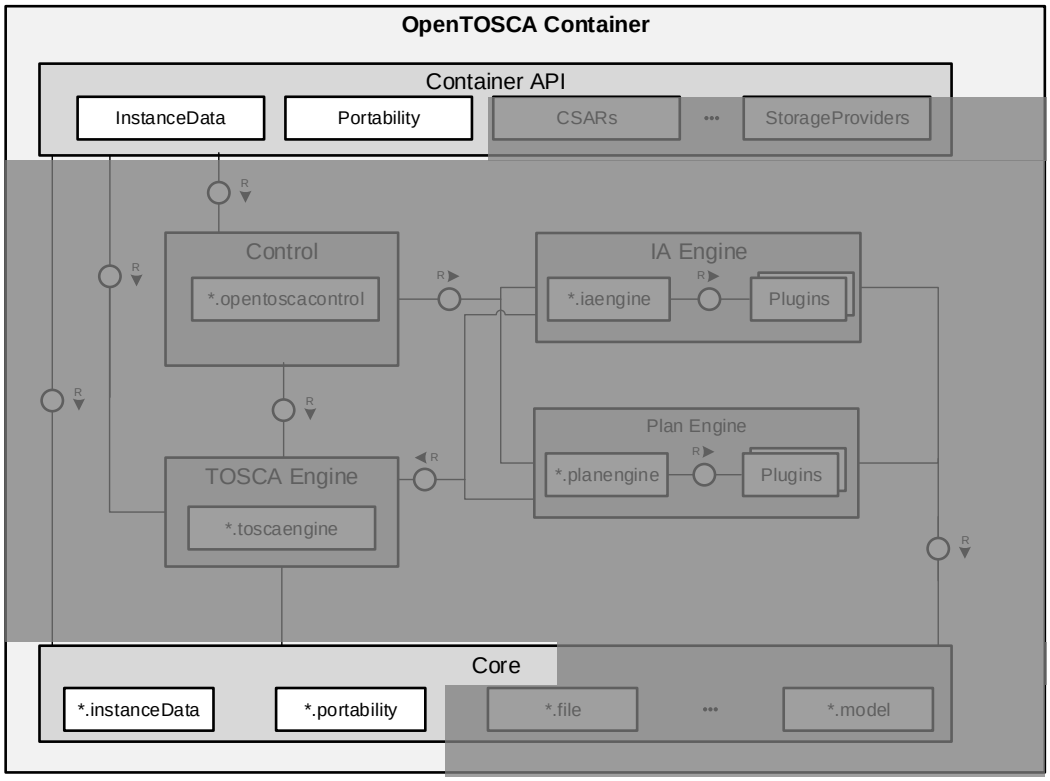

Abbildung 4.1: Erweitertes FMC-Aufbaudiagramm der OpenTOSCA-Struktur, bisher bestehende Komponenten sind ausgegraut - es wurden keine Komponenten entfernt

4.2 Ist- / Sollzustand

Auf den Ist-Zustand der Komponenten wurde bereits in Kapitel 3 Anforderungen an einen Dienst zur Instanzdatenverwaltung, besonders in Form des FMC-Aufbaudiagramms (s. Abbildung 2.4 auf Seite 25), ausreichend eingegangen. Durch die Entwicklung des Dienstes wird sich die Architektur des OpenTOSCA-Containers ändern, dies wird sich auch im Schaubild niederschlagen. Es steht einerseits die Erweiterung der REST-Schnittstelle um weitere Funktionen bevor und andererseits wird der OpenTOSCA-Container selbst intern neue Komponenten beinhalten. Der Ist-Zustand der REST-Schnittstelle des OpenTOSCA-Containers

wurde bisher nicht behandelt und soll in Abschnitt 4.8, wenn es um den Entwurf der veränderten REST-Schnittstelle geht, aufgezeigt werden.

Der Sollzustand lässt sich durch die Änderungen an Architektur und Schnittstelle gut beschreiben. Die Umsetzung der in Kapitel 3 identifizierten Anforderungen soll in mehreren OSGi-Modulen bzw. Komponenten umgesetzt werden. Die Entscheidung der Implementierung in mehreren Komponenten wurde getroffen, da nach einiger Überlegung klar wurde, dass die Anforderung der Bereitstellung von Links zu Artefakten eines NodeTemplates (vgl. Abschnitt 3.4.1 auf Seite 33) nicht in den Tätigkeitsbereich eines Instanzdatenverwaltungs-Dienstes passt. Deshalb wird zum Einen ein Dienst zur Instanzdatenverwaltung und zum Anderen ein Dienst, der sich generell mit den weiteren Anforderungen zur Portierbarkeit oder auch Portabilität von solchen Anwendungen beschäftigt, entwickelt. Der zusätzliche Dienst wird im Weiteren *Portability-API* genannt. Die bisher einzige geplante Operation ist die Umsetzung der in Abschnitt 3.4.1 geforderten Bereitstellung von Downloadlinks für Artefakte. Diese Schnittstelle soll, sobald weitere Anforderungen ersichtlich sind, erweitert werden. Diese neuen Komponenten werden auch weitere Anforderungen an die bereits vorhandene TOSCA-Engine stellen, die im Zuge dessen während der Entwicklung um diese weitere Funktionen erweitert werden muss. Eine detaillierte Analyse dieser neuen Anforderungen und Funktionen erfolgt in Abschnitt 4.6. Die geplanten Änderungen der Architektur sind in Abbildung 4.1 in einer modifizierten Version des bisherigen FMC-Aufbaufdiagramms visualisiert.

Als Ergebnis dieser Arbeit wird der OpenTOSCA-Container um einen Dienst zur Instanzdatenverwaltung, der seine Daten persistent speichert, erweitert sein. Ebenso wird er einen Dienst zur Sicherstellung der Portabilität von Management-Plänen erhalten, welcher Informationen zu Artefakten bereitstellt. Die Funktionalität dieser beiden Dienste wird mittels der bereits bestehenden und in dieser Arbeit zusätzlich erweiterten REST-API verfügbar gemacht, wobei diese Implementierung aus den von der Portability-API generierten Artefakt-Links, die im OpenTOSCA-Container gültig sind, allgemein gültige Referenzen zu Dateien konvertieren muss. Der Grund für diese strikte Trennung zwischen internen Schnittstellen und REST-API ist die Wiederverwendbarkeit und Trennung von Zuständigkeiten. Falls in einiger Zeit eine andere Art von Zugriff als die zu implementierende REST-Schnittstelle benötigt wird kann diese ebenso die interne Dienstschnittstelle nutzen um die Funktionalität verfügbar zu machen.

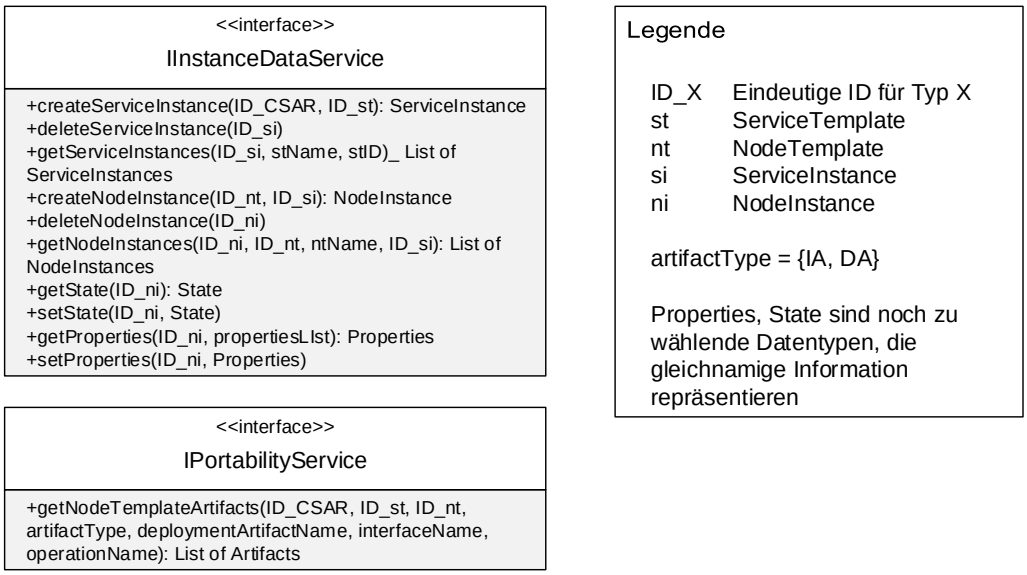

Abbildung 4.2: UML-Klassendiagramm der beiden Interfaces IInstanceDataService und IPortabilityService

4.3 Schnittstellen

Der Entwurf der beiden Schnittstellen orientiert sich sehr stark an Kapitel 3, das die Anforderungen an die Schnittstellen aufzeigt. Jede dieser Anforderungen wird durch eine Operation realisiert. Das Resultat dieses Abschnittes ist durch Abbildung 4.2 grafisch dargestellt. Die Entstehung dieser Grafik soll im folgenden Abschnitt erläutert werden.

Es werden zuerst die Operationen bezüglich Erstellung und Löschung von Instanzen, also `createServiceInstance`, `deleteServiceInstance`, `createNodeInstance` und `deleteNodeInstance`, behandelt. Beim Betrachten der Operationen fällt auf, dass bei jeder dieser Operationen eine Entität genau identifiziert werden muss. Zum Beispiel muss bei der Erstellung einer ServiceInstance ein bestimmtes ServiceTemplate identifiziert werden. Im ersten Schritt dieser Betrachtung wurden alle zu identifizierten Entitäten für alle Operationen definiert, danach wurden diese Entitäten wiederum durch ihre Identifier (siehe Tabelle 3.2 auf Seite 29) ersetzt, so dass man die Parameter dieser Operationen erhält.

Für die weiteren Operationen `getState`, `setState`, `getProperties` und `setProperties` war klar, da es sich um NodeInstance-spezifische Operationen handelt, dass hier jeweils genau eine NodeInstance identifiziert werden muss. Die beiden set-Operationen benötigen darüber hinaus auch noch einen Wert, der beim Ausführen der Operation gesetzt werden soll.

Die beiden Methoden `getServiceInstances` und `getNodeInstances` stellen einen Sonderfall dar, da sie nicht direkt eine spezielle Instanz identifizieren sondern viel mehr Filterkriterien vorgeben, die zur Selektion von Instanzen genutzt werden sollen. Nach einiger Überlegung wurde hier der Entschluss gefasst die beiden Anforderungen "Prüfung der Existenz einer ServiceInstance" und "Prüfung der Existenz einer NodeInstance" zusammen mit den beiden Filtermethoden umzusetzen. Das Resultat dieser Entscheidung ist die Aufnahme der NodeInstance-ID in die `getNodeInstances-Parameter` und die Aufnahme der ServiceInstanceID in die `getServiceInstances-Parameter`. Die weiteren Parameter wurden durch einen Blick in die gestellten Anforderungen an die Suchmöglichkeiten in Abschnitt 3.2.4 auf Seite 30 und 3.3.6 auf Seite 33 bestimmt.

Die Schnittstelle des PortabilityService hat nur eine Methode, die es ermöglicht Artefakte eines bestimmten ServiceTemplate eines CSARs abzufragen und nach gewissen Kriterien zu filtern. Auf die Möglichkeiten der Filterung wurde bereits ausgiebig in Abschnitt 3.4.1 auf Seite 33 eingegangen, diese sollen hier nicht erneut ausführlich beschrieben werden.

4.4 Interaktion

Die geplante Interaktion der neuen Schnittstellen mit den bestehenden Komponenten soll anhand eines recht simplen Beispiels zur Erstellung einer ServiceInstance aufgezeigt werden. Als Grundlage hierfür nehmen wir die, in Abbildung 2.2 auf Seite 20, beschriebene Topologie. Diese Betrachtung soll aber, aus Gründen des Umfangs, auf die Installation der Ubuntu-NodeInstance beschränkt werden. Der Aufruf der restlichen Installationen würde lediglich das Beispiel sowie die Grafik vergrößern und kaum zusätzliches Wissen vermitteln.

Das Beispiel betrachtet lediglich die notwendige Arbeit in Zusammenhang mit der Instanzdaten- und der Portability-API. Es wird davon ausgegangen, dass die entsprechende CSAR bereits mittels des OpenTOSCA-Containers vollständig verarbeitet wurde und die Pläne sich so aufrufen lassen. In dem Beispiel wird ein von einem asynchrones Ubuntu-IA ausgegangen, das eine Install-Methode anbietet, die

nach Fertigstellung eine Nachricht über die erfolgreiche Installation an den Aufrufenden schickt. Die Installation wird auf einer entfernten Virtualisierungsplattform durchgeführt, die dafür eine gültigen Hypertext Transfer Protocol (HTTP)-Link zu einem Datenträgerabbild des zu installierenden Betriebssystems benötigt. Der Ubuntu-NodeType hat ein solches Image als DA definiert.

Einige aufgezeigte Funktionen der TOSCA-Engine sind vor der Bachelorarbeit noch nicht verfügbar und müssen im Zuge dieser ebenso implementiert werden. Auf diese Erweiterungen soll in Abschnitt 4.6, der die Erweiterung der TOSCA-Engine behandelt, eingegangen werden. Bei diesem Beispiel wird außerdem davon ausgegangen, dass im ServiceTemplate der Beispieltopologie korrekt die minimale Anzahl der Instanzen der einzelnen NodeTemplates auf 1 gesetzt ist. Dies hat zur Folge, dass der Buildplan, der die Topologie aufsetzt, lediglich einen Request an die Instanzdaten-Engine senden muss, da diese dann bei der Erstellung der ServiceInstance ebenso die notwendigen NodeInstances erstellt.

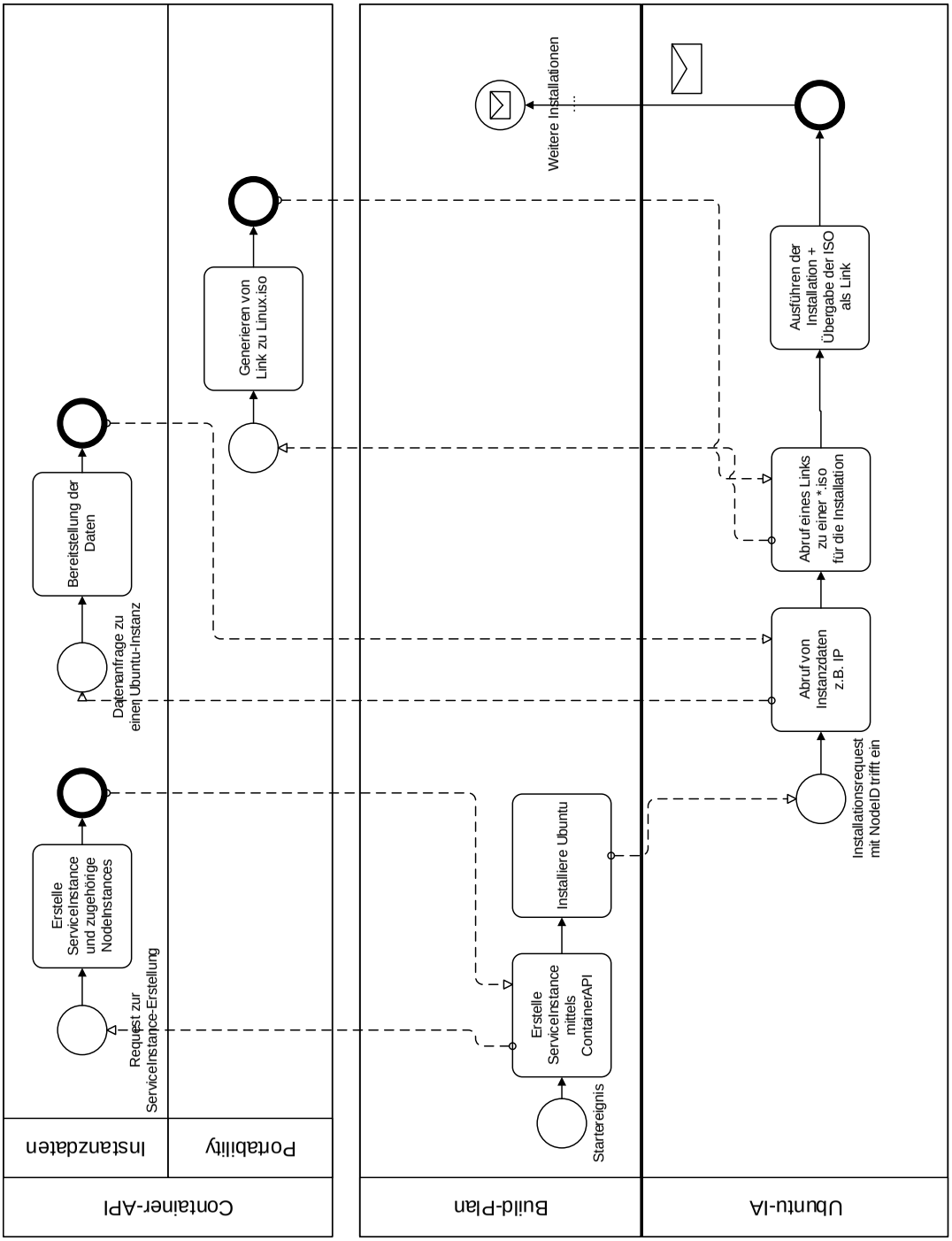

Abbildung 4.3: Beispielhafter Ablauf eines Build-Plans

Ablauf des Beispiels zwischen Plan, IA und Container-API

Der Buildplan hat einen relativ einfachen Ablauf, der in Abbildung 4.3 auf Seite 41 ausführlich aufgezeigt wird. Bei diesem Ablauf ruft der Plan erst die Instanzdatenkomponente der Container-API auf um eine ServiceInstance des ServiceTemplates zu erstellen. Die Container-API veranlasst, wie im oberen Verantwortlichkeitsbereich (*Swimlane*) ersichtlich, nun eine Instanziierung der ServiceInstance und aller zugehörigen NodeInstances. Daraufhin erhält der Plan die synchrone Antwort mit der darin enthaltenen ServiceInstanceID der ServiceInstance. Als nächster Schritt wird das Ubuntu-Betriebssystem installiert. Für den Plan stellt diese Operation einen Aufruf des Ubuntu-IA mit anschließendem Warten auf die asynchrone Fertigstellungsnachricht dar. Die asynchrone Nachricht wird in der Abbildung durch ein Briefsymbol repräsentiert.

Während der Installationsoperation, die in der unteren *Swimlane* abläuft, holt das Ubuntu-IA im ersten Schritt erforderliche Daten bezüglich der Instanz von der Instanzdatenkomponente um diese im Anschluss zur Installation des Betriebssystems zu nutzen. In diesem Beispiel handelt es sich bei diesen Daten exemplarisch um die IP-Adresse der unterliegenden Virtualisierungsplattform und Zugangsdaten zu dieser.

Nachdem das IA nun die IP-Adresse und die Zugangsdaten zu der Virtualisierungsplattform hat, benötigt es noch das Datenträgerabbild des Betriebssystems. Diese Operation wird in Abbildung 4.3 durch die Aktivität mit der Bezeichnung "Abruf eines Links zu einer *.iso für die Installation" repräsentiert. Um diese Operation durchzuführen geht es an die Portability-Komponente der Container-API heran und ruft die Methode zur Bereitstellung von Artefakten mit den entsprechenden Parametern auf. Die Antwort dieses Dienstes enthält nun einen gültigen Link zu dem Datenträgerabbild, das im Ubuntu-NodeType als DA definiert wurde. Jetzt kann das Ubuntu-IA die Installation mit allen notwendigen Daten aufrufen.

Nach der Fertigstellung dieser Operation sendet das IA Daten an die Instanzdaten-Schnittstelle. Bei der Installation entstehen oftmals wichtige Informationen, wie beispielsweise IP-Adresse oder das zufällig gewählte Passwort des Betriebssystems. Diese instanzspezifischen Informationen müssen für die spätere Verwendung dauerhaft gespeichert werden. Von diesem Schritt wird aus Gründen der Übersichtlichkeit der Grafik abgesehen, die Möglichkeit dazu soll dennoch erwähnt werden. Nach dieser Operation ist die Aufgabe des Ubuntu-IAs erledigt. Es sendet nun eine Nachricht an den Build-Plan, damit dieser mit eventuellen weiteren Schritten fortfahren kann. Im Falle eines Fehlers kann an dieser Stelle, die im Plan durch einen

Brief symbolisiert ist, eine andere Antwort erfolgen, die entweder zum Abbruch des Plans führen kann oder von diesem kompensiert werden muss.

Ablauf aus Sicht des Containers

Dieser Abschnitt soll nun ausführlich auf den Ablauf innerhalb des OpenTOSCA-Containers eingehen und betrachtet hierfür die folgenden Komponenten des OpenTOSCA-Containers: Container-API, TOSCA-Engine, Portability-Dienst und den Instanzdaten-Dienst.

Erstellung der ServiceInstance mittels Container-API

Der gesamte folgende Abschnitt bezieht sich auf den ersten Schritt des Plans (s. Abbildung 4.3 auf Seite 41: Erstellung einer ServiceInstance inklusive ihrer dazugehörigen NodeInstances). Der nun beschriebene Ablauf ist in Abbildung 4.4 auf Seite 45 grafisch dargestellt.

Der Plan ruft die Methode zur Erstellung einer ServiceInstance der REST-Schnittstelle des Containers auf. Die Container-API ruft nun unmittelbar die `createServiceInstance`-Methode des internen InstanceDataService auf und veranlasst so diesen eine ServiceInstance mit den übergebenen Parametern zu erstellen. Die Schritte, die nun zuerst ablaufen, sind weitgehend Validierungen der übermittelten Parameter. Dabei wird neben einer Prüfung der Werte zusätzlich eine Prüfung der Existenz des, in den Parametern spezifizierten, ServiceTemplates durchgeführt. Dies geschieht indem die TOSCA-Engine nach einer Liste aller ServiceTemplates in der angegebenen CSAR gefragt wird und diese Liste dann nach dem zu erstellenden ServiceTemplate durchsucht wird.

Im nächsten Schritt werden mittels eines Aufrufs der TOSCA-Engine die Angaben zur Kardinalität der einzelnen NodeTemplates des ServiceTemplates ermittelt. Der Rückgabe-Wert der `getInstanceCountsOfNodeTemplates ByServiceTemplateID`-Methode ist eine assoziative Speicherstruktur bzw. `Map`, die NodeTemplateIDs und eine Datenstruktur, welche die Minimal- und Maximalanzahl beinhaltet, miteinander verknüpft. Mit Hilfe dieser Map und der darin enthaltenen Datenstruktur kann eine Schleife initialisiert werden, die für alle in dem ServiceTemplate enthaltenen NodeTemplates genau so viele NodeInstances erstellt wie mittels der Minimalanzahl spezifiziert. Nachdem dieser Schritt abgeschlossen ist wurden die ServiceInstance und die dazugehörigen NodeInstances

erfolgreich erstellt. Der *InstanceDataService* gibt das erstellte ServiceInstance-Objekt an die Container-API zurück, welche dann als letzten Schritt einen Link zu dieser ServiceInstance generiert, der dann letztendlich dem Plan als Antwort zurückgesendet wird.

Dieses Beispiel lässt sich auch die Erstellung von NodeInstances übertragen, da diese beiden Operationen sehr ähnlich sind und sich weitgehend nur durch die Wahl der Parameter unterscheiden. Deshalb soll an dieser Stelle auch kein weiteres Beispiel zur Erstellung einer NodeInstance erfolgen.

Abruf von Instanzdaten mittels Container-API

Dieser Abschnitt bezieht sich auf die beispielhafte Operation zum Abruf von Instanzdaten in Abbildung 4.3 auf Seite 41. Wie diese Operation im OpenTOSCA-Container umgesetzt werden soll wird in Abbildung 4.5 auf Seite 46 verdeutlicht. Hierbei schickt der Plan eine `getProperties`-Anfrage an die Container-API. Diese Anfrage enthält einerseits die NodeInstanceID und andererseits eine Liste der Properties, die der Plan vom Container benötigt. In der Abbildung ist der Parameter `null` gewählt, dies bedeutet in der Praxis, dass der Service alle für diesen Knoten bekannten Properties zurückgeben wird. Falls hier Properties in dieser Liste spezifiziert werden, generiert der `InstanceDataService` ein eigenes Properties-Element, das er dann mit den in der Liste angegebenen Werten befüllt. Dieser Parameter bietet also die Möglichkeit sich nur die Werte, die wirklich benötigt werden, zurückgeben zu lassen. Die Container-API ruft den internen `InstanceDataService` auf, der im ersten Schritt die NodeInstance inklusive der dazugehörigen Instanzdaten lädt. Falls die im Aufruf übergebene Properties-Liste gefüllt ist, ist noch eine Filterung der Properties notwendig. Dies wird gelöst, indem ein neues XML-Dokument erzeugt wird und alle in der Liste enthaltenen Properties in dieses neue Dokument hineinkopiert werden. Am Ende wird das Properties-Dokument der NodeInstance bzw. das soeben neu erstellte, mit den gefilterten Properties gefüllte, Dokument zurückgeben. Die REST-API gibt das erhaltene Dokument an den aufrufenden Plan weiter.

Abruf von Links zu Artefakten mittels Container-API

Die Operation zum Abruf eines Links zu einer *.iso für die Installation (s. Abbildung 4.3 auf Seite 41) beschäftigt sich mit der Beschaffung eines Links zu einem

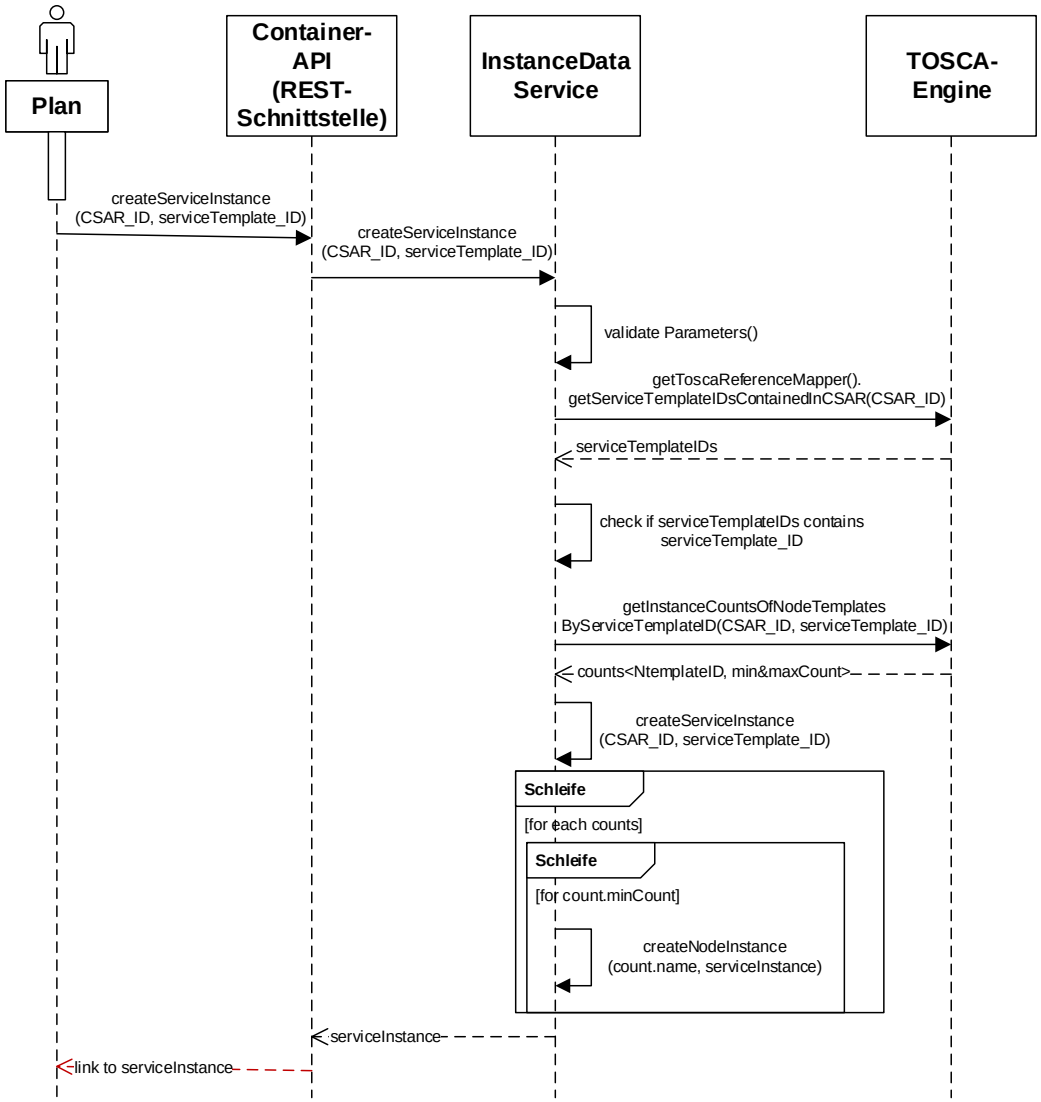

Abbildung 4.4: UML-Sequenzdiagramm: ServiceInstance-Erstellung innerhalb des OpenTOSCA-Containers

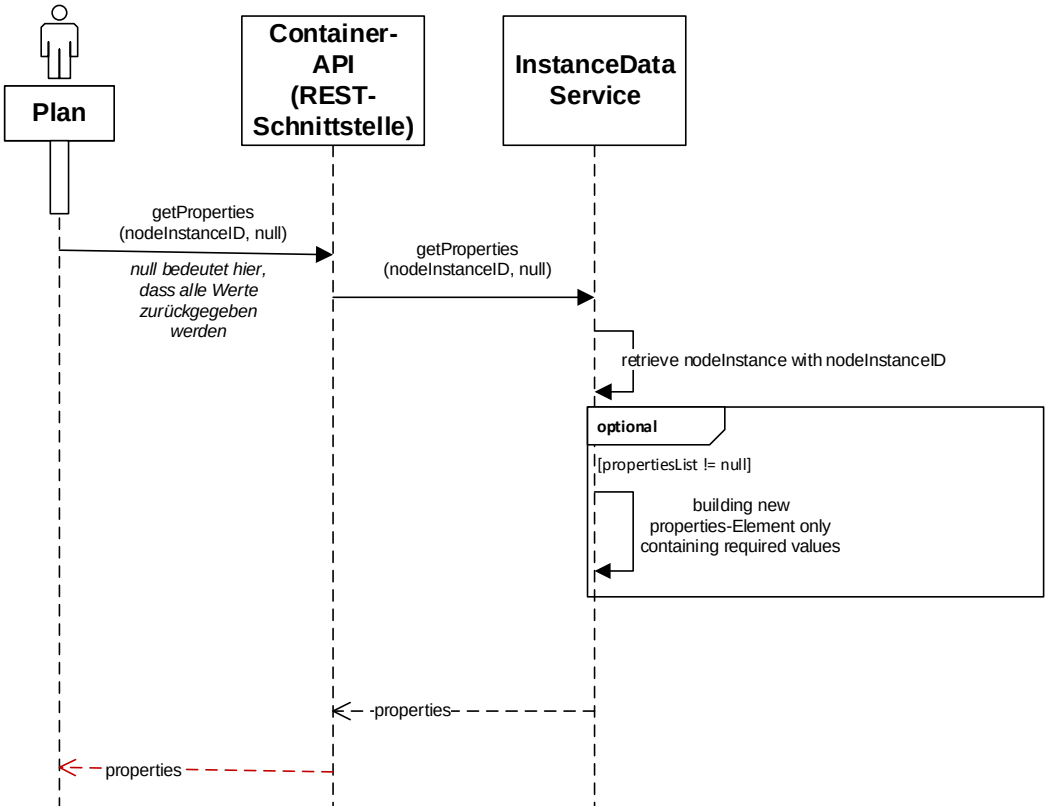

Abbildung 4.5: UML-Sequenzdiagramm: Abruf von Instanzdaten innerhalb des OpenTOSCA-Containers

DA. Die Portability-Komponente der Container-API stellt eine Methode mit dieser Funktionalität zur Verfügung. Die genaue Funktionsweise wird in Form des Ablauf eines Aufrufs aus Sicht des OpenTOSCA-Containers in Abbildung 4.6 auf der nächsten Seite dargestellt.

In der Abbildung erkennt man, dass das IA die Container-API mittels der `getNodeTemplateArtifacts`-Methode aufruft und dabei artefaktspezifische Parameter übergibt. Die Container-API ruft intern die `getNodeTemplateArtifacts`-Methode des zuständigen Portability-Dienstes auf, welcher sich dann um die Erstellung der Liste mit den entsprechenden Links kümmert. Im Detail geschieht dies

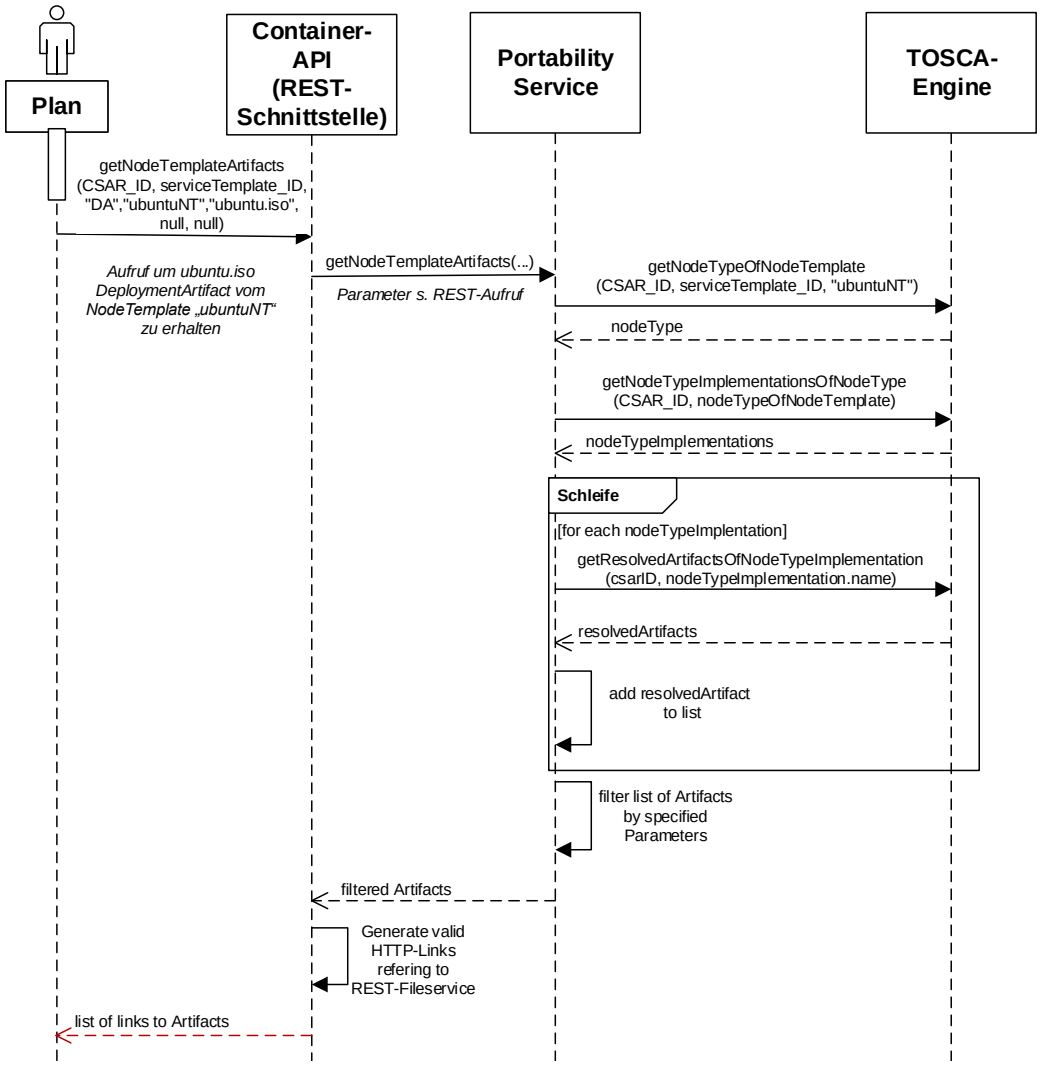

Abbildung 4.6: UML-Sequenzdiagramm: Abruf von Links zu Artefakten innerhalb des OpenTOSCA-Containers

durch enge Zusammenarbeit mit der TOSCA-Engine, da diese alle Informationen bezüglich des ServiceTemplates besitzt.

Im ersten Schritt besorgt sich die Portability-API den NodeType des in den Parametern spezifizierten Templates, um im Anschluss mittels dieser Information alle *NodeTypeImplementations* des NodeTypes zu erfragen. Für alle *NodeTypeImplementations* werden nun, mit Hilfe der `getResolvedArtifactsOfNodeTypeImplementation`-Methode der TOSCA-Engine, alle `resolvedArtifacts` in einer Liste gesammelt. `ResolvedArtifact` ist eine Datenstruktur, welche bereits aufgelöste Referenzen, also keinerlei Referenzen selbst mehr enthält. Bei dem Inhalt kann es sich aber um artefaktspezifischen Inhalt (*ArtifactSpecificContent*) oder um den relativen Pfad zur Datei innerhalb der CSAR handeln, mehr dazu aber im Abschnitt 4.5.

Die erstellte Liste wird im Anschluss entsprechend den spezifizierten Parametern gefiltert und zurück an die Container-API gegeben. Die zurückgegebene Liste enthält wie bereits erwähnt entweder ArtifactSpecificContent oder eine innerhalb der CSAR gültige Referenz. Der ArtifactSpecificContent kann direkt zurückgegeben werden, die relative Referenz muss allerdings noch umgewandelt werden, da einem IA diese Information nicht genügt. Das IA benötigt zur Verwendung der Schnittstelle einen gültigen Link zur Datei auf die referenziert wird.

Deshalb wird in der Container-API nun für jede Referenz ein entsprechender Link zur REST-Schnittstelle des FileServices generiert, an dem die Datei mittels HTTP-Request für das IA abrufbar ist. Diese modifizierte Liste gibt die Container-API nun an das aufrufende IA zurück, das mit den zur Verfügung gestellten Informationen weiterarbeiten kann.

4.5 Analyse der Beschaffenheit von Artefakten in TOSCA

Die Beschreibung der letzten Operation hat noch einige Fragen offen gelassen. Die `getResolvedArtifacts`-Methode erscheint noch recht abstrakt, welche Funktion diese Methode genau hat soll im Folgenden erläutert werden.

Um die Funktion genau zu definieren erscheint es sinnvoll zuerst einen Blick in die TOSCA [OAS13] zu werfen und die Möglichkeiten zur Definition von IAs und DAs genauer zu analysieren.

Die zu implementierende Funktion muss auf jeden Fall alle für das NodeTemplate relevanten Artefaktinformationen liefern. Diese Informationen setzen sich aus zwei

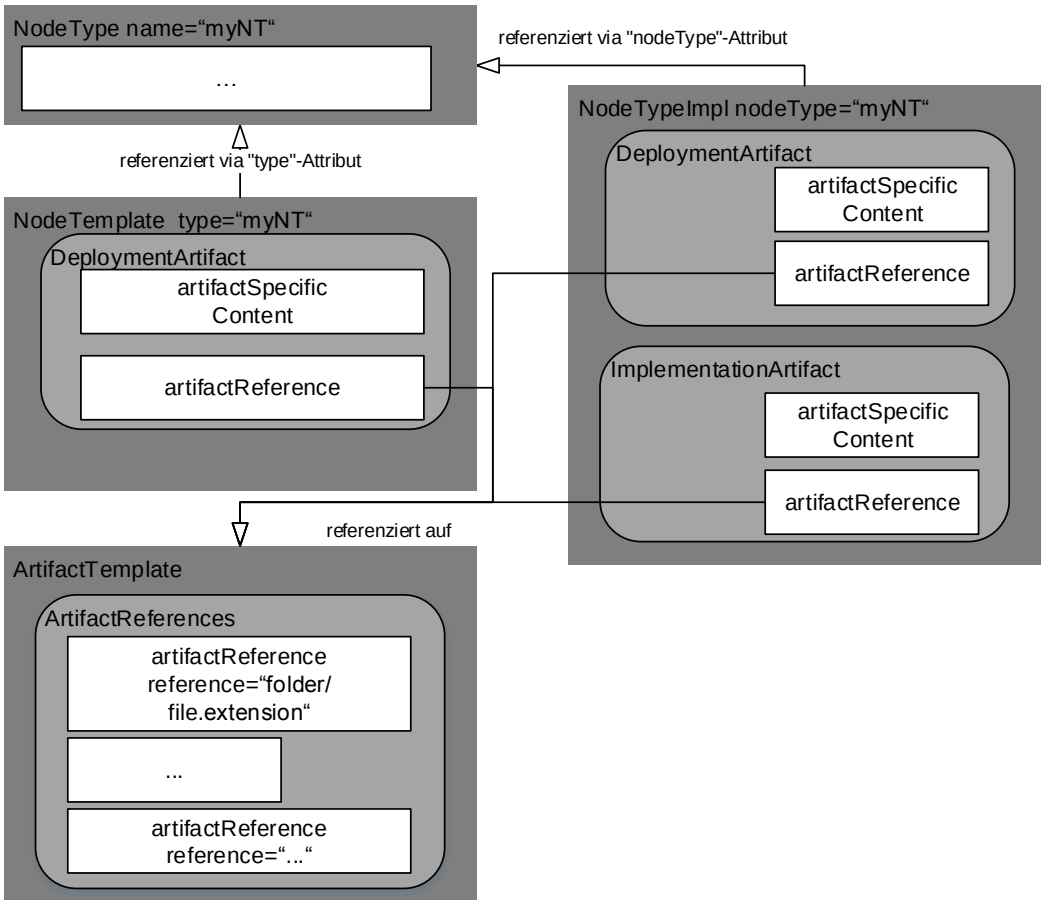

Abbildung 4.7: Referenzierung von DAs, IAs und deren ArtifactSpecificContent

strukturell unterschiedlichen Typen zusammen. Zum Einen kann ein Artefakt sogenannten ArtifactSpecificContent haben, in dem der Inhalt eines Artefaktes direkt definiert wird. Es handelt sich also um eine Möglichkeit ein Artefakt direkt im entsprechenden Template zu definieren. Diese Möglichkeit kann genutzt werden um zum Beispiel einen Link oder einen Namen eines entfernt vorhandenen Images anzugeben oder direkt im Template ein Skript zu verfassen, das als Artefakt dient. Zum Anderem besteht die Möglichkeit in einer Definition eines Artefaktes auf ein sogenanntes *ArtifactTemplate* zu referenzieren. In diesem bestehen viele weitere Möglichkeiten ein Artefakt genauer zu beschreiben, hier soll aber primär auf die

Listing 4.1 Beispielhafte Definition eines DA mit artifactSpecificContent, einer NodeTypeImplementation und eines ArtifactTemplates, das von dem DA und dem IA der NodeTypeImplementation referenziert wird

```
<NodeTemplate id="myApp" name="myApplication" type="types:ApplicationType">
      ...
      <DeploymentArtifacts>
            <DeploymentArtifact artifactRef="myNamespace:myScript"
                artifactType="types:script" name="myScript">
                  <ArtefaktSpezifischerTag>irgend ein
                      Text</ArtefaktSpezifischerTag>
            </DeploymentArtifact>
      </DeploymentArtifacts>
</NodeTemplate>
...
<NodeTypeImplementation name="myNodeTypeImpl" nodeType="types:NodeType">
      <ImplementationArtifacts>
            <ImplementationArtifact artifactRef="myNamespace:myScript"
                artifactType="types:script"/>
      </ImplementationArtifacts>
</NodeTypeImplementation>
...
<ArtifactTemplate id="myScript" type="type:Script">
      <ArtifactReferences>
            <ArtifactReference reference="IAs/Scripts/myScript.sh"/>
            <ArtifactReference reference="IAs/Scripts/myScript.bat"/>
      </ArtifactReferences>
</ArtifactTemplate>
...
```

Möglichkeit der Referenzierung innerhalb der CSAR eingegangen werden. Im ArtifactTemplate ist es möglich eine Liste von Referenzen anzugeben, die auf Dateien innerhalb der CSAR verweisen. Listing 4.1 zeigt einen Auszug eines ServiceTemplates, das neben einem DA mit ArtifactSpecificContent exemplarisch zeigt, wie ein ArtifactTemplate von einem IA referenziert wird.[OAS13]

4.6 Erweiterung der TOSCA-Engine

Dieser Abschnitt beschäftigt sich mit der Erweiterung der bereits bestehenden TOSCA-Engine. Diese muss, um die beiden neuen Dienste zur Bereitstellung der Artefakte und der Instanzdaten mit ausreichend Daten und Informationen zu versorgen, erweitert werden.

Die beiden Dienste werden Funktionen benötigen um sowohl Informationen bezüglich NodeTemplates und NodeTypeImplementations abzufragen, als auch die Möglichkeit bieten Referenzen auf ArtifactTemplates aufzulösen. Viele dieser Methoden sind bereits vorhanden und in der existierenden Implementierung der Schnittstelle bereits funktionsfähig. Nach ausgiebiger Analyse konnten die zusätzlich benötigten Methoden identifiziert werden. Abbildung 4.8 auf Seite 52 zeigt die bestehende Schnittstelle der OpenTOSCA-Engine inklusive der im Laufe der Arbeit aufgenommenen Methoden.

Es ist erkennbar, dass die TOSCA-Engine die benötigten Funktionen für den Dienst zur Verwaltung der Instanzdaten weitgehend bereits besitzt. Einzig die Funktionen `getNameOfReference`, die benötigt wird um das Name-Attribut der Service- oder NodeTemplates zu bestimmen, `getInstanceCounts OfNodeTemplatesByServiceTemplateID`, welche die Mindest- und Maximalanzahl für Instanzen von gewissen NodeTemplates eines ServiceTemplates liefert, und `doesNodeTemplateExist`, die lediglich überprüft ob ein NodeTemplate in einem ServiceTemplate existiert, müssen hinzugefügt werden.

Hauptteil der Erweiterung werden dementsprechend, die für die Portability-API benötigten Funktionen `getResolvedArtifactsOfNodeTemplate` und `getResolved ArtifactsOfNodeTypeImplementation` sein, welche sich um die Dereferenzierung der Artefaktreferenzen innerhalb von NodeTemplate und NodeType-Implementation kümmern werden (vgl. Abbildung 4.7 auf Seite 49). Der Rückgabewert dieser Methoden wird eine Datenstruktur sein, die sowohl DAs als auch IAs inklusive deren ArtifactSpecificContent und den Inhalt der referenzierten ArtefactTemplates enthalten wird.

4.7 Persistenz

Die Anforderungen an die Persistenz der beiden Dienste sind sehr unterschiedlich. Die Portability-API muss ihre Daten nicht persistent speichern, da sie implizit die Persistenz der TOSCA-Engine nutzt. Sie hat keine zusätzlichen Informationen und gibt lediglich aufbereitete Funktionen der TOSCA-Engine weiter und macht diese verfügbar. Ganz anders sieht dies beim Dienst zur Verwaltung der Instanzdaten aus, dessen Daten müssen persistent gehalten werden. Diese Daten müssen nicht nur gespeichert werden, sondern sie müssen bei den Anfragen auch nach gewissen Kriterien gefiltert werden können. Aus diesem Grund sollen die Daten in einer

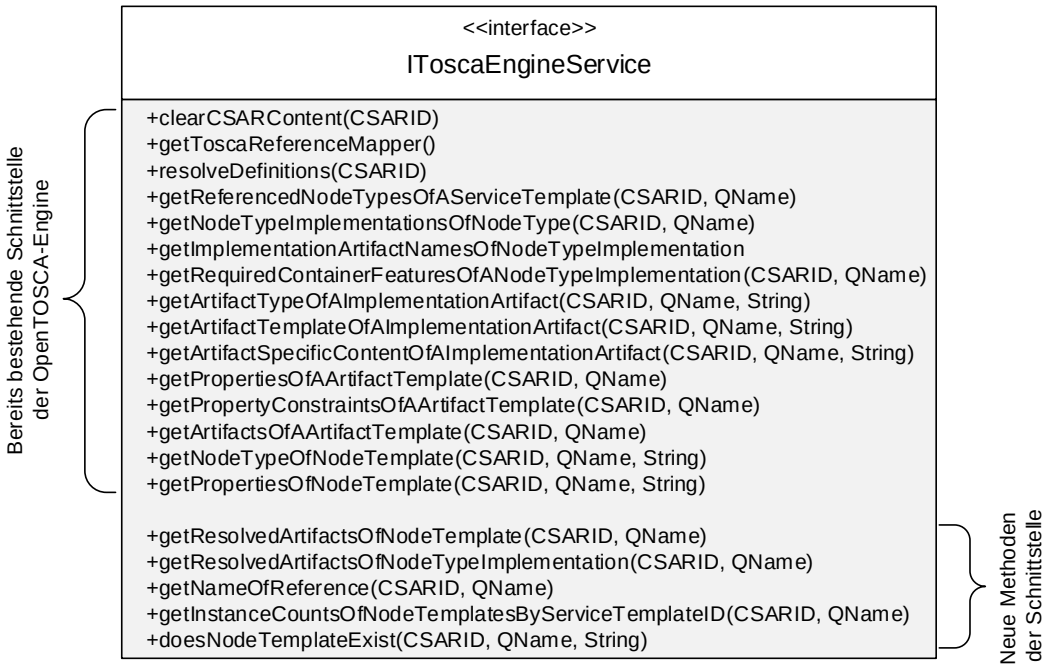

Abbildung 4.8: UML-Klassendiagramm der TOSCA-Engine Schnittstelle nach der Erweiterung im Zuge der Entwicklung

Datenbank gespeichert werden, mit deren Hilfe später auch der Filtermechanismus umgesetzt werden soll.

An vielen Stellen der Implementierung des OpenTOSCA-Containers werden bereits Daten persistent gespeichert, an diesen Stellen wird bereits auf eine Datenbank zur Speicherung zurückgegriffen. Die bestehende Implementierung nutzt Eclipse Link[1], die Referenzimplementierung der Java Persistence API (JPA) in Kombination mit einer lokalen Derby[2]-Datenbank.

Im Abschnitt 3.5.2 auf Seite 34, der die Integration in bestehende Dienste beschreibt, wurde bereits erwähnt, dass die Entwicklung der neuen Dienste sich homogen in die bestehende Architektur integrieren soll. Nach erfolgreicher Analyse der

[1]Eclipse Link: http://www.eclipse.org/eclipselink
[2]Apache Derby: http://db.apache.org/derby/

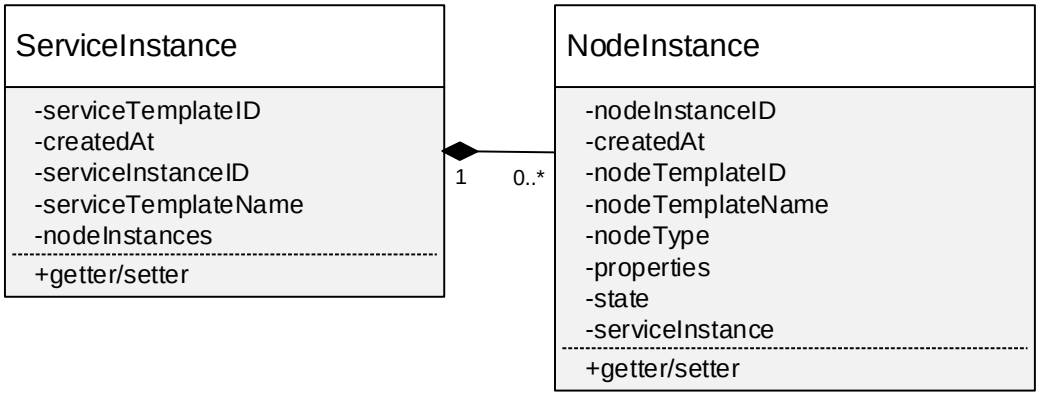

Abbildung 4.9: UML-Klassendiagramm der ServiceInstance- und NodeInstance-Klasse

Umsetzbarkeit der geforderten Funktionalität mittels EclipseLink, wurde sich an dieser Stelle für den Einsatz dieser Persistenzmöglichkeit entschieden.

Das Datenmodell des zu entwickelten Instanzdaten-Dienstes besteht nur aus Service- und NodeInstances. ServiceInstances besitzen folgende Attribute: Eindeutige ServiceInstanceID, Erstelldatum, ServiceTemplateID, ServiceTemplateName und eine Liste aller zugehörigen NodeInstances.

NodeInstances bestehen aus eindeutiger NodeInstanceID, NodeTemplateID und NodeTemplateName des zugehörigen Templates, zugehörigen NodeType, Erstelldatum, der zugehörigen ServiceInstance und außerdem State und Properties der Instanz. Auf die genaue Umsetzung dieses Businessmodells mittels JPA soll bei der Implementierung in Abschnitt 5.2 auf Seite 67 eingegangen werden. Die Struktur der beiden Klassen ist im Klassendiagramm in Abbildung 4.9 dargestellt.

4.8 REST

Dieser Abschnitt geht auf die Erweiterung der REST-Schnittstelle ein. Abbildung 4.10 auf der nächsten Seite zeigt die erweiterte REST-Schnittstelle des OpenTOSCA-Containers inklusive des schemenhaften Ist-Zustandes. Die API soll zwei weitere Pfade erhalten "/instanceData" und "/portability". Die Verwendung der Ressourcen ist, wie bei REST generell, selbsterklärend.

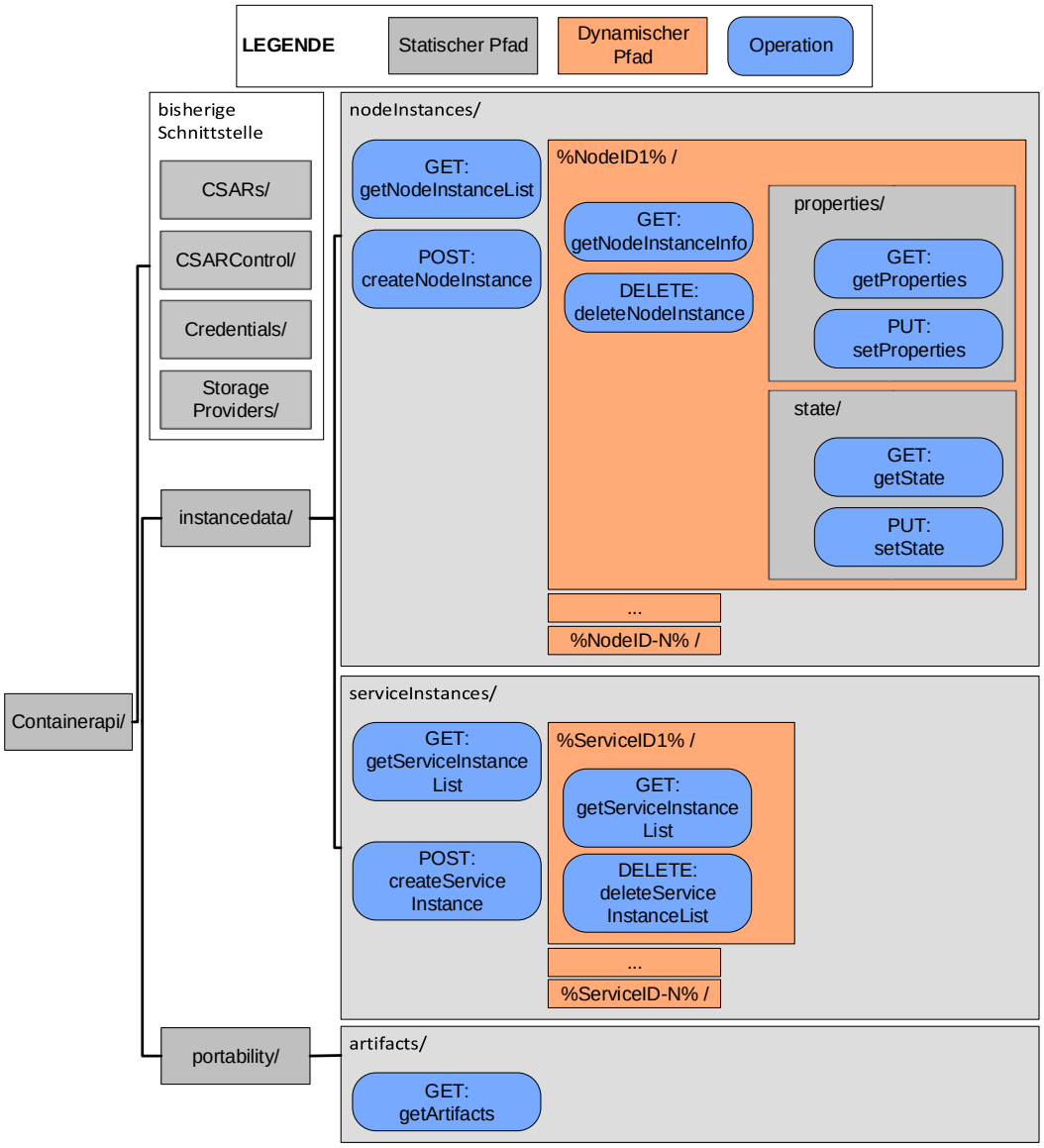

Abbildung 4.10: Erweiterung der REST-Schnittstelle des OpenTOSCA-Containers

Der InstanceData-Pfad soll alle Anfragen bezüglich der Instanzdaten-Engine verwalten, hierzu beinhaltet der Pfad auf oberster Ebene zwei weitere Pfade, die Zugriff auf NodeInstances und ServiceInstances liefern. Einer dieser Pfade ist der NodeInstances-Pfad. Er bietet die Operationen GET und POST an um einerseits mittels GET und der Angabe von Filterparametern eine gefilterte Liste von NodeInstances zu erhalten und andererseits die Erstellung von NodeInstances, also die Instanziierung von NodeTemplates, zu ermöglichen. Um Zugriff auf instanzspezifische Operationen zu erhalten wird an den NodeInstances-Pfad die ID der entsprechenden Instanz angehängt. An diesem dynamischen Pfad kann ein GET abgesetzt werden um Informationen zu der Instanz zu erhalten, ein DELETE löscht die der Identifier (ID) entsprechende NodeInstance. Zugriff auf die Properties und den State geschieht durch die zum instanzspezifischen Pfad relativen Pfade "/properties" und "/state". Auf diesen Ressourcen sind GET und PUT-Operationen möglich um die jeweiligen Informationen abzufragen oder zu überschreiben. Der NodeInstances-Pfad ist in Abbildung 4.10 in der oberen rechten Ecke inklusive aller untergeordneten Pfaden und Ressourcen dargestellt.

Ähnlich wie bei den NodeInstances sieht dies bei den ServiceInstances aus. Auf dem ServiceInstances-Pfad sind GET und POST-Operation möglich. Die GET-Operation liefert eine Liste, die den Anfrageparametern entsprechenden ServiceInstances beinhaltet. Mittels POST-Operation können an dieser Stelle ServiceInstances erstellt werden. Relativ zu diesem Pfad existiert ein dynamischer Pfad, der durch Angabe der ServiceInstanceID einen Link zu einer spezifischen ServiceInstance darstellt. Auf dieser Ressource sind wieder die beiden Operationen GET, um Informationen zu der spezifizierten ServiceInstance zu ermitteln, und DELETE, um die spezifizierte ServiceInstance und all ihre zugehörigen NodeInstances zu löschen, möglich. Der ServiceInstance-Pfad ist, ebenso wie der NodeInstances-Pfad, in Abbildung 4.10 dargestellt.

Die PortabilityAPI ist unter dem Pfad "/portability" erreichbar. Sie ist um unteren Bereich der Abbildung 4.10 dargestellt und bietet unter dem Pfad "/artifacts" eine GET-Operation an, die es ermöglicht Artefakte anhand von Parametern zu ermitteln.

Verlinkung mittels XLink

REST beabsichtigt darzustellen, wie sich gut konzipierte Web Anwendungen verhalten. Es wird ein Netzwerk von Hypermedia Dokumenten, also z.B. Webseiten, beschrieben, in dem der Benutzer voranschreitet indem er Links zwischen diesen

Listing 4.2 Beispielhafte Umsetzung der XLink-Spezifikation

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns:root xmlns:xlink="http://www.w3.org/1999/xlink"
    xmlns:ns="http://example.com">
      <ns:self xlink:href="http://mydomain.com" xlink:title="meine Webseite"
          xlink:type="simple"/>
</ns:root>
```

Webseiten auswählt [Fie00]. Eine REST-API soll dabei unabhängig von statischen Ressourcenamen sein, es muss jediglich der Einstiegspunkt der Anwendung bekannt sein, der es ermöglicht weitere Ressourcen aufgrund der gelieferten Links zu erschließen. [Fie08] Eine der Konsequenzen dieser Anforderung ist neben der Verlinkung einzelner Ressourcen, das Zurückgeben von Links als Ergebnis von Erstelloperationen.

Die Beschaffenheit dieser generierten Links muss um die maschinelle Verarbeitung sicherzustellen einem Standard entsprechen. Die World Wide Web Consortium (W3C) definiert für die Verlinkung innerhalb XML-Dokumenten einen Standard namens XML Linking Language (XLink)[3] zur Verfügung. XLink liegt mittlerweile seit Mai 2010 in der Version 1.1 vor und ermöglicht es zwischen Ressourcen zu verlinken und diese Links mittels geeigneten Abfragesprachen einfach zu extrahieren.[WDMO10]

Um für unseren Verwendungszweck ausreichende Links nach der XLink-Spezifikation zu erstellen genügt es es einen Link vom simple-Typ zu erstellen, der die Attribute `href` und `title` besitzt. Dabei gibt `href` den Ort und `title` den Namen der referenzierten Ressource an. [WDMO10] Ein Beispiel für die Art der während der Implementierung einzusetzenden Verlinkung stellt Listing 4.2 dar.

Rückgabetypen der REST-Schnittstelle

Die bisherigen Pfade der OpenTOSCA-REST-Schnittstelle geben ihre Ergebnisse als reinen Text, genauer gesagt als String, zurück oder nutzen zur Rückgabe strukturierte Informationen im XML-Format. Im Bezug auf die Konsistenz der Schnittstelle wäre ein anderes Verhalten an dieser Stelle sehr schlecht, deshalb werden die beiden Pfade dies auch auf diese Art handhaben.

[3] XLink Spezifikation: http://www.w3.org/TR/xlink11/

Listing 4.3 Beispielhafte Rückgabe der GET-Operation auf dem NodeInstances-Pfad

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:NodeInstanceList xmlns:ns1="http://www.w3.org/1999/xlink"
    xmlns:ns2="http://opentosca.org/api/pp">
      <ns2:self ns1:href="http://localhost:1337/containerapi/
          instancedata/nodeInstances" ns1:title="self" ns1:type="simple"/>
      <ns2:nodeinstances>
            <ns2:link
                ns1:href="http://localhost:1337/containerapi/instancedata/
                nodeInstances/1" ns1:title="1" ns1:type="simple"/>
            <ns2:link
                ns1:href="http://localhost:1337/containerapi/instancedata/
                nodeInstances/10" ns1:title="10" ns1:type="simple"/>
            <ns2:link
                ns1:href="http://localhost:1337/containerapi/instancedata/
                nodeInstances/11" ns1:title="11" ns1:type="simple"/>
      </ns2:nodeinstances>
</ns2:NodeInstanceList>
```

Aus Abschnitt 4.8 folgt die Anforderung, dass POST-Operationen einen Link auf die durch die Operation erstellten Objekte liefern. Delete-Operationen hingegen sollen, falls die Operation erfolgreich war, ein OK zurückgeben. Da diese simplen OK- und Zeichenketten-Rückgabewerte alle recht selbsterklärend sind und auf die Verlinkung bereits im vorhergehenden Abschnitt 4.8 ausreichend eingegangen wurde, soll im folgenden Abschnitt auf die speziellen Rückgabetypen der einzelnen GET-Operationen eingegangen werden. Dabei sollen die GET-Operationen der */nodeInstances-*, */serviceInstances-* und */portability/artifacts*-Pfade getrennt betrachtet werden.

GET-Operationen Rückgabewerte des NodeInstance-Pfads

Der NodeInstance-Pfad hat insgesamt vier GET-Operation. Die GET-Operation des NodeInstances-Pfad selbst, die GET-Operation auf dem dynamischen ID-Pfad und die beiden GET-Operation auf State- und auf dem Properties-Pfad. Die GET-Operation auf dem NodeInstances-Pfad wird Links zu den im Aufruf passenden NodeInstances liefern. Ein beispielhafter Rückgabewert dieser Operation ist in Listing 4.3 ersichtlich.

Die GET-Operation auf dem dynamischen ID-Pfad liefert Informationen bezüglich einer spezifischen NodeInstance mit der im Pfad spezifizierten ID. Die Operationen

Listing 4.4 Beispielhafte Rückgabe der GET-Operation auf dem dynamischen NodeInstances-ID-Pfad

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:NodeInstance xmlns:ns1="http://www.w3.org/1999/xlink"
    xmlns:ns2="http://opentosca.org/api/pp"
    created-at="2013-08-10T09:54:55.038+02:00"
    nodeInstanceID="http://opentosca.org/nodetemplates/instances/1"
    nodeTemplateID="{http://www.example.com/demo}MySql"
    nodeTemplateName="MySQL" serviceInstanceID=
    "http://opentosca.org/servicetemplates/instances/1">
      <ns2:Link ns1:href="http://localhost:1337/containerapi/instancedata/
          nodeInstances/1" ns1:title="self" ns1:type="simple"/>
      <ns2:Link ns1:href="http://localhost:1337/containerapi/instancedata/
          serviceInstances/1" ns1:title="ServiceInstance" ns1:type="simple"/>
      <ns2:Link ns1:href="http://localhost:1337/containerapi/instancedata/
          nodeInstances/1/properties" ns1:title="Properties"
          ns1:type="simple"/>
      <ns2:Link ns1:href="http://localhost:1337/containerapi/instancedata/
          nodeInstances/1/state" ns1:title="State" ns1:type="simple"/>
      <ns2:NodeType>{http://www.example.com/ToscaTypes}MySqlType</ns2:NodeType>
</ns2:NodeInstance>
```

liefern neben vielen Informationen, in Form von Attributen, noch einige Links zu zugehöriger ServiceInstance, Properties und State. Ein beispielhafter Rückgabewert ist in Listing 4.4 dargestellt.

Die GET-Operation auf dem State-Pfad liefert lediglich den State zurück in dem die NodeInstance sich gerade befindet. Dies ist im Generellen ein QName wird aber als String zurückgeliefert. REST sieht dafür den Datentyp `text/plain` vor, der an dieser Stelle dafür auch benutzt werden wird.

Letzte Operation dieses Abschnitts ist die GET-Operation des Properties-Pfads. Abhängig von den Parametern gibt er die Properties der, mittels des Pfads spezifizierten, NodeInstance zurück. Je nachdem ob die Rückgabe der Properties mittels der Parameter eingeschränkt ist, gibt die REST-Schnittstelle das gesamte `Properties-Document` zurück oder generiert aus den in den Parametern angegeben Properties-Namen ein neues Dokument, das die spezifizierten Properties enthält.

Die Struktur des Rückgabewert hängt einerseits von der Beschaffenheit der *Default-Properties* in der Definition im ServiceTemplate ab und andererseits davon ob einschränkende Parameter in der Anfrage definiert wurden. Wenn Parameter in der Abfrage spezifiziert wurden ist der Rückgabewert ein neues Dokument in einem

Listing 4.5 Beispielhafte Rückgabe der GET-Operation auf dem Properties-Pfad einer NodeInstance mit spezifiertem List-Parameter

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Properties>
  <demo:AdminUser
      xmlns:demo="http://www.example.com/demo">admin</demo:AdminUser>
  <demo:AdminPassword
      xmlns:demo="http://www.example.com/demo">admin</demo:AdminPassword>
</Properties>
```

Listing 4.6 Beispielhafte Rückgabe der GET-Operation auf dem ServiceInstances-Pfad

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:ServiceInstanceList xmlns:ns1="http://www.w3.org/1999/xlink"
    xmlns:ns2="http://opentosca.org/api/pp">
      <ns2:serviceinstances>
            <ns2:link ns1:href="http://localhost:1337/containerapi/
                instancedata/serviceInstances/1" ns1:title="1"
                ns1:type="simple"/>
      </ns2:serviceinstances>
</ns2:ServiceInstanceList>
```

eigenen zur Portability-API gehörendem Namespace. Ein Beispiel für diesen Fall stellt Listing 4.5 dar. Falls keine Parameter angegeben werden hängt der Rückgabewert zu stark von den definierten Properties ab, so dass hier an dieser Stelle kein weiteres repräsentatives Beispiel gegeben werden kann.

GET-Operationen Rückgabewerte des ServiceInstance-Pfads

Die GET-Operationen des ServiceInstance-Pfads verhalten sich recht ähnlich wie die des NodeInstance-Pfads. Ein GET auf der ServiceInstances-Ressource selbst gibt eine Liste von ServiceInstances, die mittels Filterparametern noch weiter eingeschränkt werden können. Eine beispielhafte Rückgabe dieser Operation ist in Listing 4.6 abgebildet. Ein GET auf dem dynamischen ID-Pfad gibt, ähnlich wie beim NodeInstance-Pfad, Informationen zu der mittels ID spezifizierten ServiceInstance zurück. Ein für diese Operation beispielhafter Rückgabewert ist in Listing 4.7 auf der nächsten Seite enthalten.

Listing 4.7 Beispielhafte Rückgabe der GET-Operation auf dem dynamischen ServiceInstance-Pfad

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:ServiceInstance xmlns:ns1="http://www.w3.org/1999/xlink"
    xmlns:ns2="http://opentosca.org/api/pp"
    created-at="2013-08-09T18:31:54.196+02:00" csarID="SugarCRM3.csar"
    serviceInstanceID="http://opentosca.org/servicetemplates/instances/1"
    serviceTemplateID=
    "http://www.example.com/demo}SugarCRM_CSPRD01_ServiceTemplate"
    serviceTemplateName="SugarCRM (CSPRD01) Service Template">
      <ns2:Link ns1:href="http://localhost:1337/containerapi/instancedata/
          serviceInstances/1" ns1:title="self" ns1:type="simple"/>
      <ns2:nodeInstances>
            <ns2:nodeInstance
                ns1:href="http://localhost:1337/containerapi/instancedata/
                nodeInstances/1"
                ns1:title="http://opentosca.org/nodetemplates/instances/1"
                ns1:type="simple"/>
            <ns2:nodeInstance
                ns1:href="http://localhost:1337/containerapi/instancedata/
                nodeInstances/2"
                ns1:title="http://opentosca.org/nodetemplates/instances/2"
                ns1:type="simple"/>
      </ns2:nodeInstances>
</ns2:ServiceInstance>
```

GET-Operationen Rückgabewerte des Portability-Pfads

Der Portability-Pfad hat, wie bereits erwähnt, die alleinige Funktion zur Abfrage von Artefakten, welche unter dem `Artifact`-Pfad mittels eines GET-Requests nutzbar ist. Die Abfrage kann mittels Parametern gezielt eingeschränkt werden. Ein Beispiel für diese Art von Rückgabewert ist in Listing 4.8 auf der nächsten Seite zu sehen. Auffällig hierbei ist der im `references`-Tag (Auszeichner) enthaltene Pfad, der einen Downloadlink des Artefakts darstellt. Dieser muss auf Grundlage des von der internen Portability-API erhaltenem Pfads berechnet werden. Auf diese notwendige Umwandlung soll im folgendem Abschnitt 4.8 eingegangen werden.

Listing 4.8 Beispielhafte Rückgabe der GET-Operation auf dem Artifact-Pfad

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Artifacts xmlns="http://opentosca.org/planportability/rest">
      <deploymentArtifacts/>
      <implementationtArtifacts>
            <implementationArtifact operationName="ConfigureScript"
                type="{http://www.example.com/ToscaTypes}Script">
                  <references>
                        <ref>http://localhost:1337/containerapi/CSARs/
                            SugarCRM3.csar/Content/IAs/Scripts/ApacheWebServer/
                            install.sh</ref>
                  </references>
            </implementationArtifact>
      </implementationtArtifacts>
</Artifacts>
```

Listing 4.9 Beispielhafte Konvertierung einer relativen Pfadreferenz in eine absolute Referenz

```
Relativer Pfad
  "IAs/Scripts/ApacheWebServer/install.sh"
=> Absolute Referenz:
  "http://<Host URL des Containers>:1337/containerapi/CSARs/
    SugarCRM3.csar/Content/IAs/Scripts/ApacheWebServer/install.sh"
```

Umwandlung der von der internen Portability-API generierten Links zu extern erreichbaren Links

Im vorherigen Abschnitt wurde bereits auf die Tatsache aufmerksam gemacht, dass man statt relativer Referenzen absolute Referenzen benötigt. Ein Nutzer der Container-API benötigt nicht den Pfad der Datei innerhalb der CSAR, er braucht eine Uniform Resource Locator (URL), an der das Artefakt selbst verfügbar ist. Die REST-Schnittstelle des OpenTOSCA-Container besitzt bereits einen Pfad, der es ermöglicht die Struktur einer CSAR zu traversieren und Dateien der CSAR direkt herunterzuladen.

Die REST-API wird also einen Mechanismus benötigen um den relative Pfad eines Artefakts innerhalb der CSAR in eine URL, bei der das Artefakt verfügbar ist, zu konvertieren. Ein Beispiel für die Konvertierung einer relativen Referenz in eine absolute stellt Listing 4.9 dar. In Listing 4.7 auf der vorherigen Seite wurde die Referenz schon korrekt aufgelöst und die im ref-Tag enthaltene Referenz ermöglicht es dem Anfragenden das entsprechende Artefakt herunterzuladen.

5 Implementierung

Dieses Kapitel soll auf die Details und Besonderheiten der Implementierung der in Kapitel 4 entworfenen Lösung eingehen. Dabei wird primär auf allgemeine Schritte der Implementierung, die bisher nicht ausreichend behandelt wurden, sowie auf die Umsetzung der Persistenz- und Filteranforderung eingegangen werden. Andere notwendige Implementierungsschritte sind durch den Entwurf weitgehend ausreichend beschrieben und mussten lediglich umgesetzt werden.

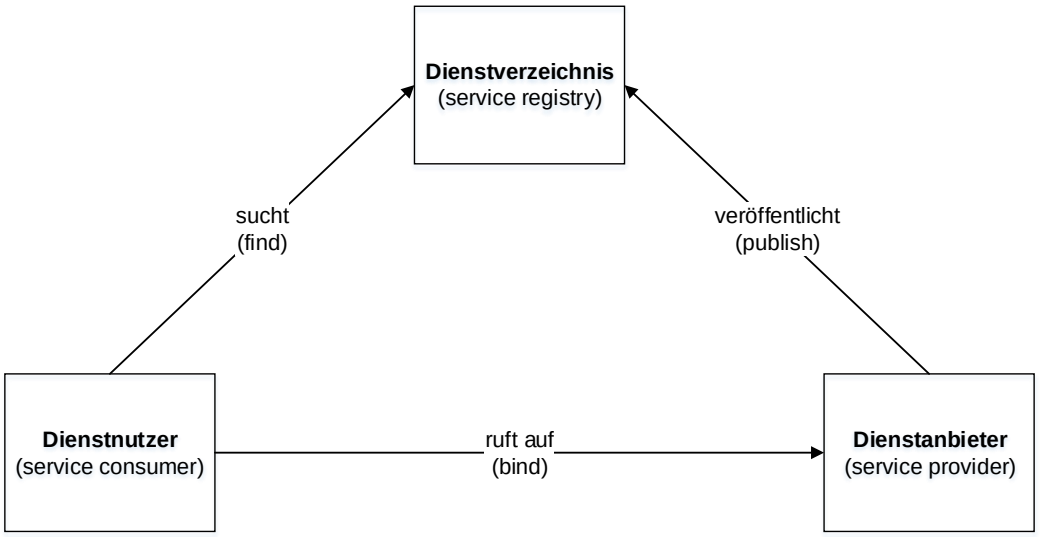

Abbildung 5.1: SOA-Dreieck, beschreibt die Beziehungen zwischen den Rollen einer service-oriented architecture nach [Mel10]

5.1 OSGi

OSGi spezifiziert ein dynamisches Modulsystem für Java, die sogenannte OSGi Service Platform. Das Framework implementiert das Konzept der service-oriented architecture (SOA) und bietet so die Grundlage für modularisierte Lösungen in Java. Abbildung 5.1 auf der vorherigen Seite stellt die Beziehung zwischen den einzelnen Beteiligten dieses Architekturmusters da.
Ein Service wird lediglich durch seine Schnittstelle spezifiziert und ist so unabhängig von einer Implementierung. Auf der einen Seite ist der Dienstanbieter (*Service Provider*), der diese Schnittstelle implementiert und sein Dienstangebot beim Dienstverzeichnis (*Service Registry*) veröffentlicht (*publish*). Auf der anderen Seite ist ein Dienstnutzer (*Service Consumer*), der nur die Schnittstelle des zu verwendeten Services kennt und im Dienstverzeichnis nach einer passenden Implementierung dieser Schnittstelle sucht (*find*). Enthält das Verzeichnis eine passende Implementierung kann er diese benutzen. Die Rolle des *Dienstverzeichnisses* wird in OSGi vom Framework umgesetzt. Den Kern des OSGi-Frameworks bilden Module, welche im OSGi-Kontext *Bundles* genannte werden. In diesen Bundles werden sowohl Schnittstellen definiert als auch implementiert. Die Bundles nehmen so Rolle der *Dienstanbieter* ein, können aber ebenso andere Dienste nutzen und so ebenso als *Dienstnutzer* auftreten.

Diese Architektur führt zu einer losen Kopplung zwischen Dienstnutzer und Dienstanbieter. Im OSGi-Framework können Bundles zur Laufzeit installiert, gestartet, gestoppt, aktualisiert und deinstalliert werden. Während der Laufzeit kann es von einem Bundle mehrere Versionen geben (Versionierung) und von einer Schnittstelle mehrere Implementierungen (Varianten).[Wüt08]

Notwendiges Projektsetup

Bei den einzelnen Komponenten des OpenTOSCA-Containers handelt es sich um OSGi-Bundles. Die typische OSGi-Projektstruktur innerhalb des OpenTOSCA-Containers wurde beibehalten. Deshalb wurden für beide Dienste jeweils drei Projekte erstellt: Ein Projekt für die Schnittstelle des zu implementierenden Dienstes, ein Projekt für die Implementierung selbst, sowie ein Projekt für die dazugehörigen Testfälle. Diese Struktur wird anhand der Struktur der InstanceData-Engine in Abbildung 5.2 auf der nächsten Seite gezeigt.

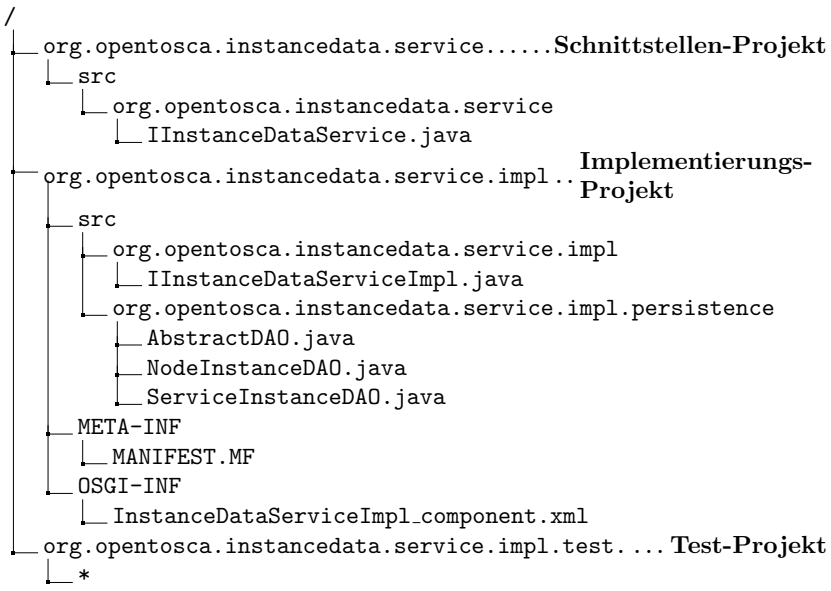

Abbildung 5.2: Struktur der zum Instanzdatenverwaltungsdienst gehörenden OSGi-Projekte

Das Projekt `org.opentosca.instancedata.service` besteht lediglich aus dem gleichnamigen Package, welches das Interface `IInstanceDataService` beinhaltet. Die Implementierung selbst befindet sich in dem Projekt `org.opentosca.instancedata.service.impl` zusammen mit den beiden Ordnern `META-INF` und `OSGI-INF`. Der `META-INF`-Ordner beinhaltet die `MANIFEST.MF`, die Name, Version, importierte Packages und weitere bundlespezifische Informationen enthält. Die in `OSGI-INF` enthalten Dateien geben Aufschluss darüber welche Schnittstellen das Bundle zur Laufzeit referenziert und zur Verfügung stellt.

Die Änderungen in der bestehenden REST-API bestanden darin zwei neue Packages für die beiden Dienste innerhalb des Projekts `org.opentosca.containerapi` zu erstellen. Die Wurzelressourcen der neuen Packages mussten noch in der Klasse `JerseyApplication` hinzugefügt werden.

5.2 Implementierung der Persistenz- und Filteranforderung

Dieser Abschnitt stellt die für die Persistenz- und Filterumsetzung notwendigen Erweiterungen des in Abbildung 4.9 auf Seite 53 dargestellten Datenmodells dar. Dabei wird im ersten Unterabschnitt mit den Grundlagen der eingesetzten Technologie JPA begonnen. Der Einsatz von JPA wurde bereits im Entwurf (vgl. Abschnitt 4.7 auf Seite 51) festgelegt. Im Anschluss an die Grundlagen von JPA werden in darauf folgendem Unterabschnitt die für die Implementierung notwendigen Schritte erläutert.

JPA

Bei der JPA[1] handelt es sich um eine Spezifikation, welche die permanenten Speicherung von Objekten in Java beschreibt. Der Einsatz einer Implementierung der JPA ermöglicht die Speicherung von Objekten in relationalen Datenbanken.

Um Objekte einer Klasse persistent zu speichern, muss in einer zugehörigen `persistence.xml` eine `persistence-unit` erstellt werden, welche die zu persistierende Klasse einschließt. Hier werden außerdem für die Speicherung besonders im Bezug auf die eingesetzte Datenbank wichtige Parameter gesetzt. [JPA09]

Ein weiterer Schritt ist das Annotieren der Klasse. Dabei werden Felder und Methoden der Klasse mit sogenannten *Annotations*, welche die Speicherung steuern, versehen. Eine zu speichernde Klasse muss mit der `@Entity`-Annotation versehen werden. Für die Felder der Klasse kann mittels der `@Column`-Annotation noch spezifiziert werden wie der Spaltenname lauten soll und von welchen Datentyp die Spalte sein soll, oftmals sind hier aber sinnvolle Standardwerte gesetzt. Außer diesen eben genannten Annotationen existieren noch weitere, die Abfragen von Objekten aus der Datenbank repräsentieren (`@NamedQueries`) oder die Konvertierung von Feldern steuern können (`@Convert`). Desweiteren besteht die Möglichkeit Beziehungen zwischen verschiedenen Entitäten herzustellen, dies kann mit den Annotations `@ManyToOne`, `@OneToMany` und `@ManyToMany` gesteuert werden. Mit

[1] Webseite der JPA
http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html

diesen Annotationen lassen sich selbst komplexe Datenmodelle abbilden. Eine Beispiel für eine mit JPA-Annotationen versehene Klasse zeigt Listing 2.1 auf Seite 26. [JPA09]

Implementierung

Im vorherigen Unterabschnitt wurde bereits auf die notwendigen Schritte um eine Klasse zu speichern eingegangen. Während der Implementierung wurden die Klassen aus Abbildung 4.9 auf Seite 53 mit den notwendigen Annotations versehen. Letztendlich war im Zuge der Umsetzung der Persistenz noch die Erstellung der entsprechenden `persistence.xml` und die Implementierung eines Konverters für die Konvertierung zwischen QNames und Strings notwendig.

Um die Filteranforderung umzusetzen gab es zwei unterschiedliche Ansätze, entweder den Ansatz alle Informationen im Speicher der Anwendung zu halten und die Filter auf diese Informationen anzuwenden oder die Daten soweit es geht nur in der Datenbank zu halten und alle Filteroperationen auf der Datenbank auszuführen. Letztendlich wurde sich für die Lösung entschieden die Datenbank zu nutzen um die Filter anzuwenden. Der Nachteil dieser Lösung ist, dass ohne die Zwischenspeicherung in der Datenbank die Daten öfters aus der Datenbank geladen werden müssen. Da sich bei OpenTOSCA diese Datenbank auf der selben Maschine wie der Container befindet, führt dies jedoch zu keinen erheblichen Leistungseinbußen. Die Vorteile der Lösung sind hingegen, dass dieser Ansatz zum einen nicht so einfach zu inkonsistenten Zuständen zwischen Anwendung und Datenbank führen kann und zum anderen die Filterung bequem mittels Structured Query Language (SQL) ermöglicht.

Das Filtern der unterschiedlichen Parameter war schnell durch eine SQL-Anweisung gelöst. Aus diversen Gründen, wie Validierung und Schutz vor böswilligen Anfragen, wurde ein `NamedQuery` eingesetzt. Es bereitete aber Schwierigkeiten, dass es sich bei einigen Parametern um optionale Parameter handelte, welche falls sie nicht gesetzt sind beim Filtern einfach ignoriert werden sollten. Eine einfache Abfrage (*Query*) mittels Gleichheits-Operator führte so also nicht zum Ziel.

Gelöst wurde das Probleme durch die `COALESCE`-Funktion. Sie nimmt beliebig viele Parameter an und liefert den ersten Parameter zurück der ungleich `NULL` ist. Die `COALESCE`-Funktion ermöglicht es indirekt einen optionalen Wert in einem SQL-Query anzugeben. Wie dies funktioniert zeigt Listing 5.1 auf der nächsten Seite. Durch das geschickte Einsetzen von `COALESCE` evaluiert die entsprechende

Listing 5.1 Pseudo SQL-Anweisung, die einen externen optionalen Parameter beinhaltet. Im Query nach der Auswertung der COALESCE-Funktionen wird ersichtlich, dass Name "ignoriert" wird

```
1  Query (vor Auswertung der COALESCE-Funktionen):
2  SELECT * FROM mitarbeiter WHERE
3    ID = COALESCE($id, ID) AND NAME = COALESCE($name, NAME);
4
5  Fall 1: alle Parameter gesetzt
6     externe Parameter: $id=1, $name = Marcus
7
8  Query nach Auswertung der COALESCE-Funktionen und Einsetzen der Variablen:
9  SELECT * FROM mitarbeiter WHERE
10   ID = 1 AND NAME = 'Marcus'; - verhält sich wie normaler Query
11
12
13 Fall 2: optionaler Parameter nicht gesetzt
14   externe Parameter: $id=1, $name = NULL
15
16 Query nach Auswertung der COALESCE-Funktionen und Einsetzen der Variablen:
17 SELECT * FROM mitarbeiter WHERE
18   ID = $id AND NAME = NAME;
19
20 NAME = NAME ist immer WAHR => Teil der Konjunktion ist immer erfüllt
```

Bedingung immer WAHR und wird so ignoriert. Wenn man es genau nimmt wird sie nicht ignoriert, die Anfrage liefert jedoch genau die gleichen Ergebnisse, wie wenn diese Bedingung nicht vorhanden gewesen wäre.

5.3 Erweiterung TOSCA-Engine

Die Erweiterung der Tosca-Engine wurde bereits im Entwurf in Abschnitt 4.6 Erweiterung der TOSCA-Engine betrachtet. Abbildung 4.8 auf Seite 52 im Entwurf zeigt die neu zu implementierenden Methoden. Im Unterschied zu der Betrachtung im Entwurf werden in dem folgenden Abschnitt Besonderheiten der Implementierung hervorgehoben. Die Methoden `doesNodeTemplateExist` und `getNameOfReference` werden in diesem Kapitel nicht ausführlich behandelt, da sie weitgehend bestehende Methoden nutzen oder Erweiterungen dieser darstellen.

Die Methode `getInstanceCountsOfNodeTemplatesByServiceTemplateID` bietet eine Möglichkeit die Kardinalitäten der in einem ServiceTemplate enthaltenen

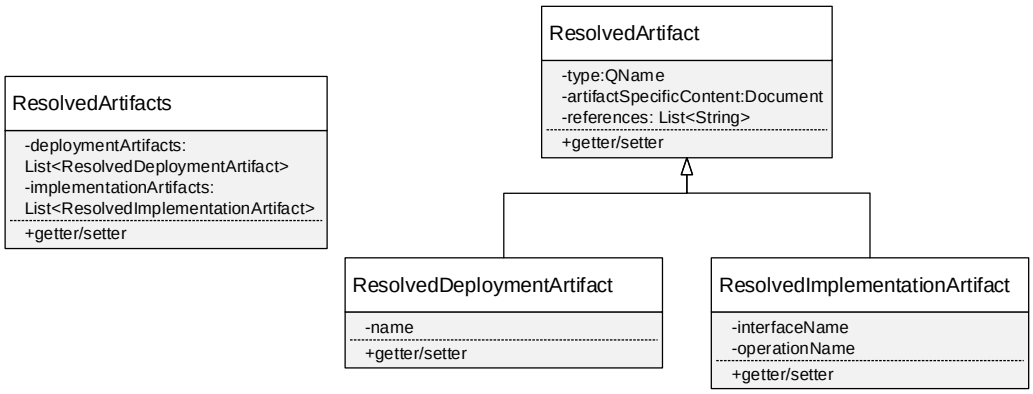

Abbildung 5.3: UML-Klassendiagramm der ResolvedArtifacts und beteiligten Klassen

NodeTemplates zu erfragen. Eine zu beachtende Besonderheit ist, dass im NodeTemplate die Mindestanzahl als `int` und die Maximalanzahl, da sie auch als unbegrenzt (`unbounded`) definiert werden kann, als `String` repräsentiert werden.
Die Methode nutzt einen speziellen Typ namens `NodeTemplateInstanceCounts` als Rückgabewert. Dieser Rückgabewert stellt eine Beziehung zwischen dem QName eines NodeTemplates und seiner Kardinalität her. Die Kardinalität wird durch eine interne Struktur repräsentiert, die Mindest- und Maximalanzahl kapselt. Mindest- und Maximalanzahl werden als `int` modelliert, wobei die Darstellung der unbeschränkten Maximalanzahl durch den Wert `-1` geschieht.

Bei der Implementierung der Methoden `getResolvedArtifactsOfNodeTemplate` und `getResolvedArtifactsOfNodeTypeImplementation` wurde die Dereferenzierung der Referenzen zu ArtifactTemplates wie in Abbildung 4.7 auf Seite 49 durchgeführt. Sie liefert also entweder den ArtifactSpecificContent eines Artefakts oder die ggf. vorhandenen und aufgelösten Referenzen des entsprechenden ArtifactTemplates. Der Rückgabewert dieser Methode bündelt diese Informationen und besteht aus einer speziellen Klasse namens `ResolvedArtifacts`. Diese Klasse enthält jeweils eine Liste für DAs und IAs. DAs und IAs haben viele Gemeinsamkeiten aber unterscheiden sich in einer Kleinigkeit: DAs besitzen lediglich ein name-Attribute, IAs hingegen besitzen ein `interfaceName` und ein `operationName`-Attribut.

Der Zusammenhang zwischen den einzelnen Klassen ist in Abbildung 5.3 in Form eines Unified Modeling Language (UML)-Klassendiagramms dar-

gestellt. Die beiden Listen der `ResolvedArtifacts`-Klasse beinhalten Elemente der `ResolvedDeploymentArtifact`- bzw. `ResolvedImplementationArtifact`-Klasse. Bei den drei in der Abbildung rechts befindlichen abgebildeten Klassen (`ResolvedArtifact`, `ResolvedDeploymentArtifact` und `ResolvedImplementationArtifact`) handelt es sich um interne Klassen der `ResolvedArtifacts`-Klasse.

5.4 Erweiterung der bestehenden REST-Schnittstelle

Die Erweiterung der REST-Schnittstelle wurde bereits in Abschnitt 4.8 auf Seite 53 sehr detailliert beschrieben, dort wurde auch ausführlich auf die grundlegenden Konzepte eingegangen. Dieser Abschnitt beschäftigt sich nun mit der zur Implementierung des Entwurfes eingesetzten Technologien und Verfahren. Dies unterteilt sich in die notwendige Erweiterung der Container-API und in die Erzeugung und Aufbereitung der Rückgabewerte der Operationen.

Erstellung der notwendigen REST-Pfade

Die benötigten REST-Pfade sind bereits im Abschnitt 4.8 im Entwurf auf Seite 53 identifiziert worden. Dieser Abschnitt wird aufzeigen, auf welche Weise die in Abbildung 4.10 auf Seite 54 aufgezeigte Struktur hergestellt wurde. Dies soll anhand eines Beispiels, exemplarisch für alle anderen Pfade, gezeigt werden.

Für die Implementierung der bisherigen REST-API kommt, wie im Entwurf bereits erwähnt, Jersey zum Einsatz. Jersey ermöglicht die Definitionen von Pfaden, die Verknüpfung von REST-Operationen zu Methoden, das Setzen von Rückgabetypen, und das Definieren von Parametern der REST-Operationen durch Annotationen. Pfade werden in Jersey mittels `@Path`-Annotation, die den Name des Pfades angibt, erstellt. Ein Ziel eines Pfades wird in der weiteren Arbeit als REST-Ressource bezeichnet. Die Annotationen `@GET`, `@POST`, `@DELETE`, `@PUT` und `@HEAD` verknüpfen die annotierten Methoden mit den entsprechenden Operationen der REST-Ressource des aktuellen Pfades. Außerdem kann für die Methoden noch der Rückgabe-Typ mittels der `@Produces`-Annotation definiert werden. Parameter werden in Jersey je nach Art des Parameters mittels `@QueryParam`-, `@PathParam`-, `@FormParam`- oder `@FormDataParam`-Annotation definiert. Mittels der `@Context`-Annotation kann noch der Kontext des Aufrufs, der Informationen wie beispielsweise die aktuelle URL des Aufrufs enthält, injiziert (*injected*) ,sprich zur Laufzeit eingefügt, werden.

Bei dem nun gezeigten Beispiel (siehe Listing 5.2 auf der nächsten Seite) handelt es sich um die Implementierung der `InstanceDataRoot`-Klasse, die als Einstiegspunkt für die Instanzdatenverwaltung innerhalb der REST-API dient. Dieser Einstiegspunkt musste zusätzlich in der `JerseyApplication`-Klasse, die Verweise auf alle Komponenten der REST-API enthält, hinzugefügt werden.

Dieses verkürzte Beispiel zeigt deutlich den Einsatz der Jersey-Annotationen. Außerdem können zwei weitere Konzepte in diesem kurzem Beispiel betrachtet werden. Bei der Implementierung der beiden Methoden `getNodeInstances` und `getServiceInstances` wird das Prinzip der Delegation genutzt, um den Aufruf an eine weitere Ressource weiterzuleiten und von dieser verarbeiten zu lassen. Das andere Prinzip ist ein spezielles Verhalten von Jersey um die Rückgabe von XML zu ermöglichen. Jersey ermöglicht die Verwendung von Java Architecture for XML Binding (JAXB)-annotierten Klassen, wie es die Klasse `InstanceDataEntry` in diesem Beispiel ist, um XML-Antworten zu generieren. Auf die Verwendung von JAXB und den dazugehörigen Annotationen wird im nächsten Abschnitt eingegangen.

Generierung der Rückgabewerte

Im Kapitel Entwurf (vgl. Abschnitt 4.8 auf Seite 53) wurde bereits sehr detailliert auf die gelieferten Ergebnisse der einzelnen Operationen eingegangen. Bei vielen der dort spezifizierten Rückgabetypen handelt es sich um anwendungsspezifisches XML. Im Entwurf wurde allerdings nicht festgelegt, wie diese XML-Typen erzeugt werden. Mit der für die Implementierung verwendeten Technologie wird sich dieser Unterabschnitt beschäftigen.

Die TOSCA-Engine selbst verarbeitet viele XML-Dateien beim Einlesen eines CSAR – die Definitionen sind in XML verfasst. In der TOSCA-Engine kam um aus dem, durch TOSCA definierten, Schema Java-Klassen zu generieren die JAXB-API[2] zum Einsatz. Die Konvertierung von XML zu Java-Klassen wird im Allgemeinen als *Unmarshalling* bezeichnet.

In der ContainerAPI soll nun JAXB genutzt werden um diese Konvertierung in die andere Richtung durchzuführen, es soll aus zur Laufzeit genutzten Java-Klassen XML generiert werden. Es soll also ein sogenanntes *Marshalling* der entsprechenden Klassen durchgeführt werden.

[2]Webseite des JAXB-Projekts: https://jaxb.java.net/

Listing 5.2 Gekürzte Implementierung der InstanceDataRoot-Klasse inklusive Jersey-Annotationen

```
1  @Path("/instancedata")
2  public class InstanceDataRoot {
3
4     @Context
5     UriInfo uriInfo;
6     @Context
7     Request request;
8
9     @GET
10    @Produces(MediaType.APPLICATION_XML)
11    public Response doGet() {
12       ...
13       InstanceDataEntry idr = new InstanceDataEntry(...);
14       return Response.ok(idr).build();
15    }
16
17    @Path("/nodeInstances")
18    public Object getNodeInstances() {
19       return new NodeInstanceListResource();
20    }
21
22    @Path("/serviceInstances")
23    public Object getServiceInstances() {
24       return new ServiceInstanceListResource();
25    }
26
27 }
```

Exemplarisch für alle anderen Konvertierungen wird an dieser Stelle die Erstellung des XMLs für die Rückgabe der NodeInstance-Liste (vgl. Listing 4.3 auf Seite 57) erläutert. JAXB arbeitet ebenso wie JPA mit Annotationen. Die Annotationen werden eingesetzt um die Struktur des zu generierenden XMLs zu definieren. Listing 5.3 auf der nächsten Seite zeigt die Implementierung der NodeInstance-Liste, welche bereits mit Annotationen versehen ist. Augenmerk soll bei der Betrachtung auf den Annotationen liegen, `@XmlRootElement` definiert den Namen des *Root-Elements* des XML-Dokuments wohingegen `@XmlElement` einzelne XML-Elemente definiert. `@XmlElementWrapper` ermöglicht es eine Liste von XML-Elementen mit einem umschließenden Element zu versehen.

JAXB unterstützt von Haus aus bereits viele Java-Datentypen, nicht unterstützte Typen oder Klassen können aber durch die Definition eines Bindings mittels

Listing 5.3 Implementierung der NodeInstanceList inklusive JAXB-Annotationen

```
@XmlRootElement(name = "NodeInstanceList")
@XmlType(propOrder = { "selfLink", "links" })
public class NodeInstanceList {

   private List<SimpleXLink> links;

   private SimpleXLink selfLink;

   public NodeInstanceList() {

   }

   public NodeInstanceList(SimpleXLink selfLink, List<SimpleXLink> links) {
      super();
      this.selfLink = selfLink;
      this.links = links;
   }

   @XmlElement(name = "self")
   public SimpleXLink getSelfLink() {
      return selfLink;
   }

   public void setSelfLink(SimpleXLink selfLink) {
      this.selfLink = selfLink;
   }

   @XmlElement(name = "link")
   @XmlElementWrapper(name = "nodeinstances")
   public List<SimpleXLink> getLinks() {
      return links;
   }

   public void setLinks(List<SimpleXLink> links) {
      this.links = links;
   }

}
```

Annotationen für diese ebenso umgewandelt werden. Ein nicht direkt ersichtliches Beispiel ist in Listing 5.3 die Klasse `SimpleXLink`, bei der es sich um eine JAXB-annotierte Klasse handelt.[JAX13]

Konvertierung der Links zu den Artefakten

Wie in Abschnitt 4.8 auf Seite 60 beschrieben liefert die TOSCA-Engine relative Referenzen zu den Artefakten zurück. Diese Pfade ermöglichen es einem Benutzer der API nicht ohne Zusatzaufwand, auf die jeweiligen Artefakte zuzugreifen. Es muss also eine Konvertierung zu absoluten Referenzen, die einen Zugriff auf diese Artefakte ermöglichen, durchgeführt werden.

Die REST-API bietet bereits einen Pfad an, der es ermöglicht den Inhalt einer CSAR-Datei zu traversieren und gezielt auf Dateien, die sich in dem CSAR befinden, zuzugreifen. Es handelt sich um den `\CSARs`-Pfad. Dieser Pfad kann an dieser Stelle genutzt werden um den Zugriff auf Artefakte zu erlauben. Um diesen Zugriff zu ermöglichen müssen lediglich die relativen Referenzen aus den Definitionen, der Form "`/Pfad/.../Datei.Endung`", in die Form "`http://.../CSARs/csarID/Content/Pfad/.../Datei.Endung`" umgewandelt werden.

Die Erzeugung des für die Operation notwendigen XMLs wurde wie nach dem im vorherigem Abschnitt "Generierung der Rückgabewerte" beschriebenen Prinzip implementiert.

6 Validierung des Konzepts und der Implementierung

Dieses Kapitel soll die Validierung des entworfenen Konzepts und der entwickelten Implementierung im Bezug auf die in Kapitel 3 definierten Anforderungen darstellen. Es wird betrachtet ob und in welchem Umfang die Anforderungen mittels der Erweiterung der Container-API erfüllt wurden.

Die ServiceInstance-spezifischen Anforderungen (vgl. Abschnitt 3.2 auf Seite 28) erfordern, dass eine ServiceInstance erstellt und gelöscht werden kann. Diese Anforderung werden vollständig durch die beiden Operationen `POST` und `DELETE` auf dem ServiceInstances- bzw. dynamischen ServiceInstanceID-Pfad der jeweiligen ServiceInstance realisiert.
Außerdem muss die Schnittstelle es ermöglichen, Informationen bezüglich einer ServiceInstance zu erhalten. Diese Anforderungen wird durch die `GET`-Operation auf dem ServiceInstanceID-Pfad umgesetzt.
Die letzte ServiceInstance-spezifische Anforderung ist es die ServiceInstances anhand spezieller Filter zu identifizieren. Die `GET`-Operation auf der ServiceInstance-Ressource ermöglicht dies, sie erlaubt es ServiceInstances anhand ihres serviceTemplateNames und der serviceTemplateID zu filtern. Durch den zusätzlichen Parameter serviceInstanceID erfüllt sie sogar die Anforderung der Existenzprüfung, da eine nicht existente serviceInstanceID als Filterparameter eine leere Liste zurückliefert.

Die NodeInstance-spezifischen Anforderungen (vgl. Abschnitt 3.3 auf Seite 31) sind denen der ServiceInstance-spezifischen sehr ähnlich. Ebenso müssen Erstellung und Löschung der NodeInstances ermöglicht werden. Diese beiden Anforderungen werden durch die `POST`- und `DELETE`-Operation auf dem NodeInstances- bzw. dynamischen NodeInstanceID-Pfad erfüllt. Die Prüfung der Existenz einer NodeInstance kann wie bei den ServiceInstances mittels der Operation zum Finden von NodeInstance-IDs genutzt werden, wobei eine nicht existierende NodeInstanceID eine leere Liste zurückliefert. Ebenso wie bei den ServiceInstances besteht auch die

Anforderung NodeInstances anhand von den in Abschnitt 3.3.6 auf Seite 33 definierten Parametern zu finden. Diese Anforderung wird mittels der GET-Operation auf der NodeInstance-Ressource umgesetzt. Besonderheit der NodeInstances im Vergleich zur Beschaffenheit der ServiceInstances ist das Vorhandensein von state und properties, diese beiden Informationen müssen einerseits abgefragt und geändert werden können. Hierfür hat jede dieser beiden Ressourcen einen Unterpfad relativ zum Pfad einer speziellen NodeInstance. Auf diesen Pfaden `/properties` und `/state` sind `GET`- und `PUT`-Operationen möglich um diese Informationen abzufragen bzw. zu ändern.

Dieser Absatz betrachtet die einzige NodeTemplate-spezifische Anforderung (vgl. Abschnitt 3.4 auf Seite 33). Diese Anforderung fordert eine Möglichkeit absolute Referenzen zu Artefakte eines NodeTemplates zu erhalten. Es muss also möglich sein mittels der gelieferten Referenzen auf die Artefakte zuzugreifen. In Abschnitt 4.5 auf Seite 48 wurde eine detaillierte Analyse durchgeführt, auf welche Art und Weise Artefakte in TOSCA dargestellt werden können. Die bei der Analyse gewonnenen Erkenntnisse sind in die spätere Implementierung eingeflossen und es wurde mit dem `/artifacts`-Pfad der Portability-API eine Möglichkeit geschaffen, Referenzen zu Artefakten eines serviceTemplates zu erhalten. Die in der dazugehörigen Anforderungen identifizierten Filtermöglichkeiten wurden umgesetzt.

Verbleibende Anforderungen sind die weiteren funktionalen Anforderungen Persistenz und Integration in die existierende Architektur des OpenTOSCA-Containers. Die Persistenz wurde mittels JPA umgesetzt. Die Implementierung speichert die Daten und Änderungen dieser direkt in der Datenbank. Dadurch kann garantiert werden, dass beim Herunterfahren oder Absturz des OpenTOSCA-Containers der Zustand der Anwendung nicht verloren geht. Der Verlust der Datenbank selbst wurde hier nicht betrachtet, da in diesem Fall der ganze TOSCA-Container nach einem Neustart nicht mehr wie gewohnt operieren kann.

Die Integration in die bestehende Architektur des OpenTOSCA-Containers bedeutet zwei Dinge. Auf der einen Seite muss der OpenTOSCA-Container nach der Erweiterung weiterhin eine konsistente Art des Zugriffs für den Benutzer bieten. Auf der anderen Seite sollten, um die Wartbarkeit zu gewährleisten, bei der Implementierung möglichst bereits innerhalb des OpenTOSCA-Containers eingesetzte Technologien zum Einsatz kommen. Im Nachhinein betrachtet kann man sagen, dass während der Implementierung diese Anforderungen erfolgreich umgesetzt wurden. Der Zugriff auf die REST-API erfolgt weiterhin in einer gewohnt einheitlichen Art, die neuen Funktionen wurden erfolgreich in die bestehende Schnittstelle

integriert. Bei der Implementierung wurde sich um diese Integration ebenso bemüht. Sowohl bei der Art des Projektsetup, als auch bei der Wahl der zur Implementierung genutzten Technologien, wurde sich an bestehenden Komponenten orientiert. Beide entwickelten Komponenten, der Dienst zur Instanzdatenverwaltung und die Portability-API, sind typische OpenTOSCA-Komponenten.

7 Zusammenfassung und Ausblick

Zur sinnvollen Verwaltung von Instanzen eines Cloud-Services ist es nötig, Laufzeitdaten bezüglich einzelner Komponenten eines Cloud-Services persistent zu speichern und bereit zu stellen. Instanzdaten alleine genügen aber nicht um eine Cloud-Anwendung aufzusetzen. Neben den bisherigen OpenTOSCA-Diensten, wird bei der Installation von einzelnen Instanzen ein Zugriff auf die in TOSCA definierten DAs benötigt. Vor Durchführung dieser Bachelorarbeit bot der OpenTOSCA-Container keine Möglichkeit an, um Instanzdaten zu verwalten. Der Zugriff auf die DAs war prinzipiell möglich, aber für die Anforderungen der Pläne nicht dynamisch genug.

Das Ziel der vorliegenden Arbeit war es deshalb, einen Dienst zur Verwaltung von Instanzdaten eines TOSCA-Cloud-Services zu entwerfen und umzusetzen. Eine der daraus resultierenden Aufgaben war es also theoretische Anforderungen an einen solchen Dienst zur Instanzdatenverwaltung und zur Bereitstellung von Artefakten zu identifizieren. Die Analyse wurden mit Rückblick auf bereits abgeschlossene Projekte und hinsichtlich neuer Anwendungsfälle durchgeführt. Das Ergebnis der Analyse liegt in Form von Anforderungen an den Dienst vor, welche in Kapitel 3 dargestellt sind.

Im Anschluss an die Analyse folgte der durch Kapitel 4 repräsentierte Entwurf eines Konzeptes für die Integration des neuen Dienstes in den bestehenden OpenTOSCA-Container. Er setzt sich mit für die Umsetzung relevanten Überlegungen auseinander. Neben Beschaffenheit der Artefakte, Definition sowohl der internen als auch externen REST Schnittstelle und der Interaktion der Komponenten untereinander wurde auch der Grundstein für die Umsetzung des für OpenTOSCA-typischen OSGi-Projektsetups gelegt.

Ein Ziel des Entwurfs war es, durch die bevorstehende Implementierung, den OpenTOSCA-Container nicht unnötig mit Ballast in Form von neuen Technologien zu beladen. Jede neu eingesetzte Technologie stellt zusätzliche Anforderungen an die zukünftigen Entwickler und erhöht so den Wartungs- und Weiterentwicklungsaufwand. Um dieses Ziels zu erreichen wurde analysiert, inwiefern der

OpenTOSCA-Container bereits Technologien verwendete, die für die spätere Implementierung sinnvoll genutzt werden konnten.

Als Ergebnis dieser Analyse wurden JPA und JAXB für die Umsetzung dieser Arbeit verwendet. Diese beide Technologien werden bereits von anderen Komponenten des Containers verwendet und stellen so kein neues notwendiges Wissen für die Wartung des zukünftigen OpenTOSCA-Containers dar. Neben den gerade genannten Technologien wurde zur Implementierung der REST-Schnittstelle des Dienstes, die bestehende ContainerAPI erweitert. Diese Tatsache hat zum einen zur Folge, dass neben Jersey kein weiteres Framework zur Implementierung einer REST-Schnittstelle benötigt wurde, als auch zum anderen, dass der neue Dienst unter dem bereits vorhandenen Pfad der Container-API erreichbar ist.

Das Kapitel 5 beschäftigt sich mit der Implementierung. Es erläutert neben den eingesetzten Technologien an dieser Stelle vor allem Feinheiten der Umsetzung des Entwurfs.

Die Validierung in Kapitel 6 führt eine Gegenüberstellung der in Kapitel 3 identifizierten Anforderungen und der Implementierung durch. Diese Gegenüberstellung bestätigt, dass die entwickelten Lösungen zum Zwecke der Instanzdatenverwaltung und zum Zugriff auf Artefakte eines Cloud-Services brauchbar sind.

Die OpenTOSCA-Laufzeit ist nun in der Lage Instanzdaten zu verwalten und ermöglicht mittels der bereits bestehenden REST-API den Zugriff auf diese. Neben dem Zugriff auf Instanzdaten können die Pläne nun ebenfalls auf die Artefakte eines ServiceTemplates zugreifen. Es ist zu erwarten, dass in Zukunft diese Zugriffe standardisiert werden und die Ergebnisse dieser Arbeit an diesen Standard angepasst werden müssen. Diese Standardisierung wäre aus Sicht der Kompatibilität zwischen verschiedenen TOSCA-Containern jedenfalls erstrebenswert. Die Ergebnisse dieser Arbeit sind dennoch sinn- und wertvoll, da die umgesetzten Funktionen der beiden Dienste benötigt werden und eine baldige Standardisierung der Schnittstellen nicht abzusehen war.

Literaturverzeichnis

[BBH+13] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, S. Wagner. OpenTOSCA – A Runtime for TOSCA-based Cloud Applications. In *11th International Conference on Service-Oriented Computing*, LNCS. Springer, 2013. (Zitiert auf den Seiten 23 und 25)

[BBLS12] T. Binz, G. Breiter, F. Leymann, T. Spatzier. Portable Cloud Services Using TOSCA. *IEEE Internet Computing*, 16(03):80–85, 2012. doi:10.1109/MIC.2012.43. URL http://doi.ieeecomputersociety.org/10.1109/MIC.2012.43. (Zitiert auf den Seiten 19 und 20)

[Fie00] R. T. Fielding. *REST: Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000. URL http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm. (Zitiert auf den Seiten 17 und 56)

[Fie08] R. T. Fielding. REST APIs must be hypertext-driven, 2008. URL http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven. (Zitiert auf Seite 56)

[IDC] IDC Forecasts Public IT Cloud Services Spending Will Approach $100 Billion in 2016 Generating 41% of Growth in Five Key IT Categories. URL http://www.idc.com/getdoc.jsp?containerId=prUS23684912. (Zitiert auf Seite 11)

[JAX13] JAXB Release Documentation, 2013. URL https://jaxb.java.net/2.2.7/docs/. Revision 20130802.c6cc023. (Zitiert auf Seite 73)

[JPA09] Java Persistence 2.0, Final Release. Technischer Bericht, Sun Microsystems, 2009. URL http://download.oracle.com/otndocs/jcp/persistence-2.0-fr-eval-oth-JSpec/. (Zitiert auf den Seiten 66 und 67)

[Mel10] I. Melzer. *Service-Orientierte Architekturen Mit Web Services: Konzepte - Standards - Praxis*. Spektrum Akademischer Verlag GmbH, 2010. URL http://books.google.de/books?id=e3qnVPngoUoC. (Zitiert auf Seite 63)

[MRV11] C. Metzger, T. Reitz, J. Villar. *Cloud Computing: Chancen und Risiken aus technischer und unternehmerischer Sicht*. Hanser, München, 2011. URL http://deposit.d-nb.de/cgi-bin/dokserv?id=3549075&prov=M&dok_var=1&dok_ext=htm. (Zitiert auf Seite 15)

[OAS07] OASIS. Web Services Business Process Execution Language Version 2.0. online, 2007. URL http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html. (Zitiert auf Seite 21)

[OAS13] OASIS. Topology and Orchestration Specification for Cloud Applications Version 1.0 - Candidate OASIS Standard 01, 2013. URL http://docs.oasis-open.org/tosca/TOSCA/v1.0/cos01/TOSCA-v1.0-cos01.html. (Zitiert auf den Seiten 12, 18, 19, 20, 21, 22, 48 und 50)

[OMG11] I. O. Object Management Group. Documents Associated With Business Process Model And Notation (BPMN) Version 2.0. online, 2011. URL http://www.omg.org/spec/BPMN/2.0/. (Zitiert auf Seite 21)

[PM11] T. G. P. Mell. The NIST Definition of Cloud Computing. Technischer Bericht, National Institute of Standards & Technology, 2011. URL http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf. (Zitiert auf Seite 15)

[SHI+13] B. Satzger, W. Hummer, C. Inzinger, P. Leitner, S. Dustdar. Winds of Change: From Vendor Lock-In to the Meta Cloud. *Internet Computing, IEEE*, 17(1):69–73, 2013. doi:10.1109/MIC.2013.19. (Zitiert auf Seite 11)

[Til11] S. Tilkov. *REST und HTTP: Einsatz der Architektur des Web für Integrationsszenarien*. dpunkt, Heidelberg, 2., aktualis. und erw. Auflage, 2011. URL http://deposit.d-nb.de/cgi-bin/dokserv?id=3678896&prov=M&dok_var=1&dok_ext=htm. (Zitiert auf Seite 18)

[WDMO10] N. Walsh, S. DeRose, E. Maler, D. Orchard. XML Linking Language (XLink) Version 1.1. W3C recommendation, W3C, 2010. URL

http://www.w3.org/TR/2010/REC-xlink11-20100506/. (Zitiert auf Seite 56)

[Wüt08] G. Wütherich. *Die OSGi-Service-Platform: eine Einführung mit Eclipse Equinox*. dpunkt, Heidelberg, 1. aufl. Auflage, 2008. URL http://deposit.d-nb.de/cgi-bin/dokserv?id=3067525&prov=M&dok_var=1&dok_ext=htm. (Zitiert auf Seite 64)

Alle URLs wurden zuletzt am 23. 10. 2013 geprüft.

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