Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 37

Konzept und Entwicklung eines generischen File Service für OpenTOSCA

Rene Trefft

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. Frank Leymann
Betreuer:	Dipl.-Inf. Sebastian Wagner
Beginn am:	2012-12-11
Beendet am:	2013-06-12
CR-Nummer:	E.5, H.3.2, H.3.5

Kurzfassung

Die Topology and Orchestration Specification for Cloud Applications (TOSCA) definiert eine Sprache, mit der Cloud-Anwendungen und deren Management portabel und interoperabel beschrieben werden können. Zur Verteilung einer TOSCA-Anwendung kommt das Cloud Service Archive (CSAR) zum Einsatz.

OpenTOSCA ist eine an der Universität Stuttgart entwickelte Laufzeitumgebung für TOSCA-Anwendungen, die als CSAR-Datei bereitgestellt werden. Der File Service, eine Komponente von OpenTOSCA, ist für die Speicherung, Verwaltung und den Zugriff auf übergebene CSAR-Dateien zuständig.

Im Rahmen dieser Bachelorarbeit wird der File Service um ein Plug-in-System erweitert, mit dem CSAR-Dateien in verschiedenen Umgebungen gespeichert werden können. Plug-ins werden für das lokale Dateisystem und den Cloud-Storage-Anbieter Amazon S3 bereitgestellt. Hierzu kommt die Multi-Cloud-Bibliothek jclouds zum Einsatz. Es werden Funktionen realisiert, mit denen CSAR-Dateien auf mehrere Umgebungen verteilt werden können. Auch wird eine Export-Funktion bereitgestellt, mit der eine gespeicherte CSAR wieder als CSAR-Datei abgerufen werden kann. Zur Speicherung und Verwaltung von Zugangsdaten, die Plug-ins benötigen, wird ein Credentials Service entwickelt.

Die neuen Funktionalitäten des File Service und Credentials Service werden über die Container API bereitgestellt. Die Container API stellt die externe REST-Schnittstelle von OpenTOSCA dar.

Dieses Dokument befasst sich im Wesentlichen mit der Konzeption und dem Entwurf für die angesprochene Weiterentwicklung von OpenTOSCA. Auch wird auf implementierungsspezifische Details eingegangen.

Inhaltsverzeichnis

Abkürzungsverzeichnis

AAR	Axis Archive
Amazon S3	Amazon Simple Storage Service
API	Application programming interface
Blob	Binary large object
BPEL	Business Process Execution Language
BPMN	Business Process Model and Notation
BSN	Bundle Symbolic Name
CPU	Central Processing Unit
CS	Committee Specification
CSAR	Cloud Service Archive
DA	Deployment Artifact
DVD	Digital Versatile Disc
FMC	Fundamental Modeling Concepts
FTP	File Transfer Protocol
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IA	Implementation Artifact
IaaS	Infrastructure as a Service
IAAS	Institut für Architektur von Anwendungssystemen
IDE	Integrated development environment
IPVS	Institut für Parallele und Verteilte Systeme
IT	Informationstechnik
JAR	Java Archive

Java EE	Java Platform, Enterprise Edition
JAXB	Java Architecture for XML Binding
JAX-RS	Java API for RESTful Web Services
JPA	Java Persistence API
JSP	Java Server Pages
NIST	National Institute of Standards and Technology
ORM	Object-relational mapping
PaaS	Platform as a Service
POM	Project Object Model
REST	Representational State Transfer
SaaS	Software as a Service
SLF4J	Simple Logging Facade for Java
SQL	Structured Query Language
TOSCA	Topology and Orchestration Specification for Cloud Applications
UI	User Interface
UML	Unified Modeling Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
US	United States
VALESCA	Visual Editor for TOSCA
WAR	Web Archive
WSDL	Web Services Description Language
WSO2 BPS	WSO2 Business Process Server
XML	Extensible Markup Language
XSD	XML Schema Definition

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

1 Einleitung

Die Entwicklungen der letzten Jahre zeigen, dass Cloud-Anwendungen immer beliebter werden. Viele IT-Unternehmen migrieren mit ihren bestehenden Produkte zu einem Cloud-Anbieter, da sie nicht mehr mit der Bereitstellung und Wartung von erforderlicher Hard- und Software konfrontiert sein möchten, die sich als teuer und kompliziert herausgestellt hat [sal]. Stattdessen werden diese Aufgaben vom Cloud-Anbieter übernommen, der hierfür spezialisiertes Personal und Software vorhält.

Aufgrund von strategischen Änderungen des IT-Unternehmens ist es oftmals erforderlich, dass mit einer Cloud-Anwendung zu einem anderen Cloud-Anbieter umgezogen werden muss. Aber auch tarifliche Änderungen des Cloud-Anbieters können einen Anlass für diese Maßnahme darstellen. Ein Umzug zwischen mehreren Anbietern ist jedoch mit einem hohen Aufwand für das IT-Unternehmen verbunden, da von der Anwendung und deren Aufbau in der Regel keine anbieterübergreifende Beschreibung existiert.

Aus diesem Problem heraus ist die Topology and Orchestration Specification for Cloud Applications (TOSCA) entstanden. Sie ermöglicht die portable und interoperable Beschreibung von Cloud-Anwendungen und deren Management. Anwendungen können so bei verschiedenen Anbietern betrieben und deren Management automatisiert werden. [TOS13]

OpenTOSCA ist eine Laufzeitumgebung für TOSCA-Anwendungen. Eine erste Version wurde im Rahmen eines Studienprojekts an der Universität Stuttgart entwickelt und soll im Rahmen dieser Bachelorarbeit nun in einem bestimmten Bereich weiterentwickelt werden, auf den im folgenden Abschnitt näher eingegangen wird.

1.1 Motivation und Aufgabenstellung

Das Cloud Service Archive (CSAR) ist ein Archivformat zur Verteilung von TOSCA-Anwendungen und wird von OpenTOSCA verstanden. Momentan kann der Inhalt einer solchen Datei, die OpenTOSCA übergeben wurde, jedoch ausschließlich auf dem lokalen Dateisystem gespeichert werden. Wünschenswert wären alternative Speicherorte, insbesondere Cloud-Storage-Anbieter. Diese bieten eine sichere und zuverlässige Infrastruktur für Daten. Die Speicherung erfolgt in der Regel repliziert,

um eine hohe Datensicherheit zu gewährleisten [Sei10]. Weiterhin steht bei den meisten Cloud-Storage-Anbietern für Unternehmen eine theoretisch unbegrenzte Speicherkapazität zu Verfügung.

Im Rahmen dieser Bachelorarbeit wird ein generischer File Service entworfen und implementiert, der eine einheitliche Schnittstelle bietet, um CSAR-Dateien in verschiedenen Umgebungen zu speichern und zu verwalten. Es werden zwei Implementierungen der Schnittstelle bereitgestellt: Eine Implementierung (Plug-in) ermöglicht weiterhin das Speichern auf dem lokalen Dateisystem, eine weitere Implementierung erlaubt die Speicherung auf dem Cloud-Storage-Anbieter Amazon Simple Storage Service (S3) (siehe Abschnitt 2.6). Der generische File Service, der in dieser Arbeit entsteht, stellt eine Weiterentwicklung des bisherigen File Service von OpenTOSCA dar.

1.2 Gliederung der Arbeit

In Kapitel 2 werden zunächst Begrifflichkeiten erläutert, die für das Verständnis dieser Arbeit relevant sind.
Im Anschluss werden in Kapitel 3 die Anforderungen genannt, die der generische File Service erfüllen muss. Dies impliziert auch Anforderungen, die nicht direkt der Aufgabenstellung zu entnehmen sind, sondern sich aus dieser ergeben haben.
Kapitel 4 beschreibt das Konzept des generischen File Service und erläutert, wie dieser implementiert wurde. Dabei wird in 4.1 einführend zunächst ein Überblick über die Architektur gegeben. Anschließend werden in den weiteren Abschnitten auf die einzelnen Komponenten und Datenmodelle eingegangen, die entwickelt bzw. weiterentwickelt worden sind. Abschnitt 4.7 thematisiert die Integration.
Den Abschluss bildet Kapitel 5, in dem die Arbeit zusammengefasst wird und Anregungen für konkrete Weiterentwicklungen gegeben werden, die im Zusammenhang mit der Entwicklung im Rahmen dieser Arbeit stehen.

2 Grundlagen

In diesem Kapitel werden grundlegende Begriffe erläutert, die in den weiteren Kapiteln als vorausgesetzt angenommen werden. Zunächst setzen wir uns in Abschnitt 2.1 mit dem Begriff „Cloud Computing“ auseinander, da dieser die Basis für die meisten der darauf folgenden Begriffe darstellt.

2.1 Cloud Computing

Bisher konnte sich für „Cloud Computung“ keine allgemeingültige Definition durchsetzen. In Fachkreisen wird jedoch meist die Definition der US-amerikanischen Standardisierungsstelle NIST (National Institute of Standards and Technology) herangezogen [Bun], im Folgenden aus dem Englischen übersetzt:

> Cloud Computing ist ein Modell, das es erlaubt bei Bedarf, jederzeit und überall bequem über ein Netzwerk auf einen geteilten Pool von konfigurierbaren Rechnerressourcen (z. B. Netze, Server, Speichersysteme, Anwendungen und Dienste) zuzugreifen, die schnell und mit minimalem Managementaufwand oder geringer Serviceprovider-Interaktion zur Verfügung gestellt werden können. [MG11]

Prinzipiell geht es also darum, IT-Ressourcen effizient in und über Netzwerke bereitzustellen [MÏ2]. Für einen Nutzer der IT-Ressourcen bleibt dabei die zugrunde liegende Infrastruktur (u. a. Hard- und Software) verborgen. Diese Einschränkung hat den Begriff der Cloud (deutsch „Wolke“) geprägt.

IT-Ressourcen werden durch Cloud-Computing als Dienste zur Verfügung gestellt. Man unterscheidet drei verschiedene Servicemodelle: Infrastructure as a Service (IaaS) stellt Rechenleistung, Datenspeicher und Netzwerkkapazität zur Verfügung. Auf einer virtuellen Rechnerinstanz kann bspw. ein Betriebssystem mit Anwendungen installiert und betrieben werden. Platform as a Service (PaaS) stellt eine Ausführungsumgebung für Anwendungen bereit, auf der mittels standardisierter Schnittstellen zugegriffen und Anwendungen installiert werden können [MG11]. Der Nutzer erhält keinen Zugriff auf die Infrastruktur (u. a. Server und Betriebssystem) [MG11]. Software as a Service (SaaS) stellt eine konkrete Cloud-Anwendung zur Verfügung. [Bun]

Meist werden Cloud-Dienste von Anbietern bereitgestellt, die sich auf diesem Gebiet spezialisiert haben. Dadurch wird eine hohe Zuverlässigkeit und Sicherheit der Dienste ermöglicht.

2.2 Topology and Orchestration Specification for Cloud Applications (TOSCA)

Die Topology and Orchestration Specification for Cloud Applications (TOSCA) ist ein Standard, mit dem Cloud-Anwendungen und deren Management portabel und interoperabel beschrieben werden können, also unabhängig von einem bestimmten Cloud-Anbieter oder einer Hosting-Technologie. Zur Repräsentation eines TOSCA-Modells kommt die Auszeichnungssprache bzw. das Datenformat XML zum Einsatz. [TOS13]

In Abschnitt 2.2.1 wird auf die wesentlichen Konzepte von TOSCA eingegangen. Das Cloud Service Archive (CSAR), das ebenfalls durch TOSCA spezifiziert ist, wird in Abschnitt 2.2.2 separat behandelt, da dieses eine zentrale Rolle in dieser Arbeit einnimmt.

Wir beziehen uns in diesem Dokument auf TOSCA in der Version CS01[1] (vom 2013-03-18).

2.2.1 Konzepte

In einem TOSCA-Modell wird die Struktur einer Cloud-Anwendung durch ein Topology Template beschrieben, das sich aus Node Templates und Relationship Templates zusammensetzt. Ein Node Template repräsentiert eine Komponente der Anwendung und ist durch ein Node Type typisiert. Ein Beziehung zwischen zwei Node Templates wird durch ein Relationship Template modelliert, das ein Relationship Type referenziert. [TOS13]

Sowohl Node Types als auch Relationship Types definieren (insbesondere) Schnittstellen mit Management-Operationen. Node Type Implementation bzw. Relationship Type Implementation repräsentiert die Implementierung eines referenzierten Node Type bzw. Relationship Type und definiert dazu Implementation Artifacts (IAs), welche die Schnittstellen realisieren. In einer Node Type Implementation können weiterhin Deployment Artifacts (DAs) definiert werden, die ein zugehöriges Node Template

[1] TOSCA Spezifikation Version CS01: http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-cs01.pdf

bzw. eine Komponente der Anwendung repräsentieren. Auch ist die Definition von DAs direkt in einem Node Template möglich. [TOS13]

Beispielsweise könnte es ein Node Template geben, das einen Application Server repräsentiert. Das zugehörige Deployment Artifact wäre dann das Image des Application Servers. Ein Implementation Artifact könnte eine Anwendung (z. B. WAR-Datei) sein, die Operationen bereitstellt, mit denen der Application Server gestartet und heruntergefahren werden kann. Mittels einem „hostedOn“-Relationship Template könnte der Application Server mit einem Linux, das ein weiteres Node Template darstellt, verbunden sein.

In einer Node Type Implementation, Relationship Type Implementation oder einem Node Template erfolgt die Definition eines Artefakts durch eine Referenz auf ein Artifact Template. In einem Artifact Template kann ein konkretes Artefakt direkt oder durch Referenzen spezifiziert werden. Ein Referenz ist dabei eine URI, die auf eine einzelne Datei oder einen Ordner verweist. In letzterem Falle sind Patterns erlaubt, mit denen Dateien an der Referenz ausgeschlossen werden können. Eine Komponente einer TOSCA-Laufzeitumgebung, die den Inhalt an einer Artefakt-Referenz verfügbar macht, muss folglich überprüfen, welche Dateien den Patterns entsprechen und lediglich diese zurückliefern. Ein Artifact Template referenziert ein Artifact Type, dass die Menge der erlaubten Artefakte durch Typisierung einschränkt. Beispielsweise wäre ein Artifact Type für WAR-Dateien denkbar. [TOS13]

Node Templates und Relationship Templates können auch Eigenschaften (Properties) besitzen, deren Struktur bzw. Schema im referenzierten Node Type bzw. Relationship Type definiert sein muss. Ein Node Template einer virtuellen Maschine könnte z. B. deren Hardwarezusammenstellung (Anzahl der CPUs, Größe der Festplatte etc.) als Eigenschaften enthalten. [TOS13]

Das Management einer Cloud-Anwendung wird in einem TOSCA-Modell durch Pläne (Plans) umgesetzt. Ein Plan ist ein Prozess-Modell, d. h. ein Workflow, der Management-Operationen, die durch Implementation Artifacts bereitgestellt werden, zu höherwertigen Management-Funktionalitäten orchestriert (kombiniert). Beispielsweise könnte es einen Plan geben, der die Cloud-Anwendung instanziiert (Build Plan). Pläne können direkt oder mittels einer Referenz (URI) spezifiziert werden. TOSCA definiert keine Sprache für Pläne, sondern erlaubt die Verwendung von existierenden Standards zur Beschreibung von Prozessen, insbesondere BPEL und BPMN. [TOS13]

Ein Service Template setzt sich aus einem Topology Template und Plänen zusammen. Durch Ausführung eines Build Plans wird das Service Template bzw. deren Topology Template instanziiert und repräsentiert damit einen konkreten Service. Definitions

besteht aus Service Templates und den angesprochenen Types, die zur Definition der Service Templates benötigt werden. [TOS13]

Alle angesprochenen Konstrukte von TOSCA sind auf XML-Elemente abgebildet und bilden nach beschriebener Hierarchie (Definitions ist das Wurzelelement) ein TOSCA Definitions-Dokument. [TOS13]

TOSCA sieht auch eine Import-Funktionalität vor, mit der die Beschreibung einer Cloud-Anwendung auf mehrere Dokumente verteilt werden kann. In einem Definitions-Dokument können dazu XML Schema Definitions (XSDs), WSDL Definitions oder weitere TOSCA Definitions-Dokumente spezifiziert werden, die importiert bzw. als Abhängigkeit deklariert werden sollen. Beispielsweise könnte es in einem Definitions-Dokument eine Referenz auf einen Node Type geben, das in einem anderen Definitions-Dokument definiert ist. Ersteres Definitions-Dokument müsste dann das Definitions-Dokument mit dem Node Type importieren. [TOS13]

VALESCA ist ein Modellierungswerkzeug für TOSCA, mit dem u. a. Topology Templates grafisch erstellt werden können [CLO]. Abbildung 2.1 zeigt eine Topologie, die mit VALESCA modelliert wurde.

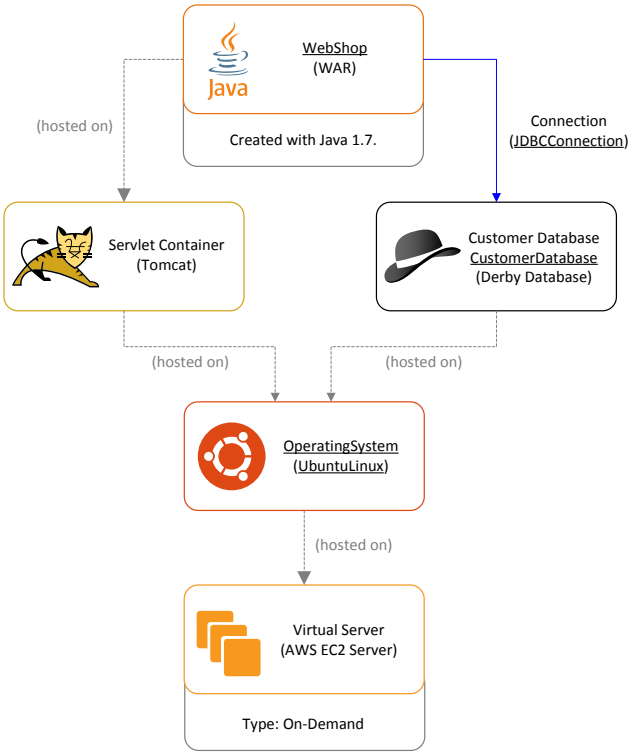

Abbildung 2.1: Beispiel einer TOSCA-Topologie, die mit VALESCA modelliert wurde. Node Templates werden durch abgerundete Rechtecke repräsentiert. Pfeile, die Node Templates miteinander verbinden, stellen Relationship Templates dar.

2.2.2 Cloud Service Archive (CSAR)

Ein Cloud Service Archive (CSAR) wird zur Verteilung einer TOSCA-Anwendung eingesetzt. Es handelt sich um eine ZIP-Datei (üblicherweise komprimiert) mit der Dateiendung „csar“, die alle Artefakte enthält, die zur Instanziierung und zum Management der Cloud-Anwendung bzw. Service benötigt werden. Dazu gehören Definitions-Dokumente, Implementation Artifacts, Deployment Artifacts und Pläne. In dieser Form kann eine Cloud-Anwendung einer TOSCA-Laufzeitumgebung übergeben werden. [TOS13]

Das Wurzelverzeichnis einer CSAR muss zumindest die Ordner „Definitions“ und „TOSCA-Metadata“ enthalten. In „Definitions“ liegen eine oder mehrere Definitions-Dokumente der Cloud-Anwendung. Beispielsweise wäre es denkbar, dass eine CSAR zur Wiederverwendung lediglich Node Types und Relationship Types spezifiziert. In weiteren CSARs könnten dann Node Templates bzw. Relationship Templates definiert sein, die auf diese Node Types bzw. Relationship Types referenzieren. In diesem Fall müssten die Definitions-Dokumente mit den Types importiert werden (siehe Abschnitt 2.2.1). Falls eine Cloud-Anwendung dagegen vollständig in einer CSAR verteilt wird, so muss mindestens ein Definitions-Dokument ein Service Template enthalten. [TOS13]

Der Ordner „TOSCA-Metadata“ enthält die TOSCA Metadatei „TOSCA.meta“ (auch CSAR Manifest genannt), in der Metadaten der CSAR als Schlüssel-Wert-Paare hinterlegt sind. Abbildung 2.1 veranschaulicht den Aufbau der TOSCA Metadatei.

```
1  TOSCA-Meta-Version: 1.0
2  CSAR-Version: 1.0
3  Created-By: string
4  Entry-Definitions: string ?
5  Topology: string ?
6
7  Name: <relFilePathToCSARRootOrPattern1>
8  Content-Type: type/subtype1
9  <key11>: <value11>
10 ...
11 <key1n>: <value1n>
12
13 ...
```

Listing 2.1: Aufbau einer TOSCA Metadatei. Der erste Block, der Metadaten über die CSAR selbst enthält, ist Pflicht.

Die Datei teilt sich in Blöcken auf, die durch Leerzeilen voneinander getrennt sind. Im ersten Block („block_0“) stehen Metadaten über die CSAR selbst, wie z. B. der Autor der CSAR. Unter dem optionalen Attribut „Entry-Definitions“ kann der relative Pfad

zum Haupt-Definitions-Dokument definiert werden, um einem TOSCA-Container eine effizientere Verarbeitung der Definitions-Dokumente zu ermöglichen. Auch “Topology“ ist optional und referenziert auf ein Bild, dass die Topologie der Cloud-Anwendung veranschaulicht. Letzteres Attribut ist dabei nicht durch TOSCA spezifiziert. Es wird lediglich vom TOSCA-Container OpenTOSCA (siehe Abschnitt 2.5) verstanden und verwendet.

In weiteren Blöcken können Metadaten zu Dateien in der CSAR definiert werden, z. B. der Hash einer Datei. Falls Metadaten zu einer Datei definiert werden sollen, so muss neben dem relativen Pfad der Datei mindestens ihr Content-Type[2] angegeben werden. Statt dem relativen Pfad der Datei können auch Pattern definiert werden, um Metadaten direkt mehreren Dateien zuzuweisen. [TOS13]

Die übrige CSAR-Struktur ist dem Ersteller der CSAR überlassen [TOS13]. Abbildung 2.2 zeigt ein Beispiel einer gültigen CSAR.

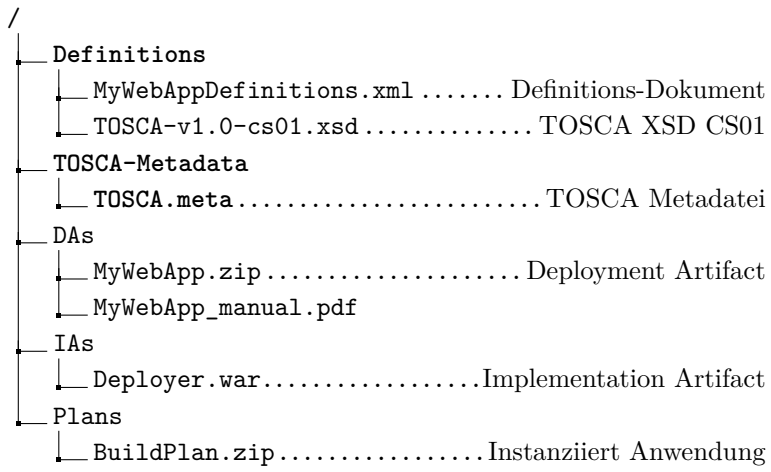

Abbildung 2.2: Beispiel einer gültigen CSAR. Vorgaben durch die Spezifikation sind fett gedruckt.

2.3 OSGi

OSGi spezifiziert eine dynamische Softwareplattform für Java, die als OSGi Service Platform bezeichnet wird. Im Wesentlichen stellt diese ein Komponenten- und Service-Modell für Java bereit. Komponenten und Services können zur Laufzeit dynamisch verwaltet („Hot Deployment“), d. h. installiert, gestartet, gestoppt, aktualisiert und deinstalliert werden. Dies ermöglicht insbesondere die Installation von Updates ohne einen Neustart der Anwendung. Auch kann eine nicht-primäre Funktionalität

[2] Der Content-Type muss der Typ/Subtyp-Struktur entsprechen. Ein existierender Content-Type ist bspw. „text/plain“.

ausfallen oder (temporär) deaktiviert werden ohne das die gesamte Anwendung nicht mehr funktionsfähig ist. Beides führt zu einer höheren Verfügbarkeit. Denkbar wäre es auch, dass man eine Anwendung in verschiedenen Varianten bereitstellt. Jede Variante zeichnet sich durch einen bestimmten Funktionsumfang aus, der sich aus einer Zusammenstellung von Komponenten ergibt. Weiterhin unterstützt die OSGi Service Platform Versionierung, d. h. mehrere Versionen einer Komponente können gleichzeitig in Betrieb sein. Komponenten, die Abhängigkeiten zu verschiedenen Versionen der selben Bibliothek haben, führen somit zu keinen Problemen. [WHKL09, Vog13, Hor12a, SC09]

Eine Komponente wird in OSGi durch ein Bundle repräsentiert. Dabei handelt es sich um eine klassische JAR-Datei mit zusätzlichen Metadaten, die in einem Bundle Manifest „MANIFEST.MF“ hinterlegt sind. Dieses muss im Ordner „META-INF“ relativ zum Wurzelverzeichnis der JAR abgelegt sein. Listing 2.2 zeigt beispielhaft ein Bundle Manifest. [Vog13]

```
1 Manifest-Version: 1.0
2 Bundle-ManifestVersion: 2
3 Bundle-Name: My Application.
4 Bundle-SymbolicName: org.example.myapp
5 Bundle-Version: 1.0.0
6 Bundle-RequiredExecutionEnvironment: JavaSE-1.7
7 Import-Package: org.example.myapp1,
8  org.example.myapp2
9 Export-Package: org.example.myapp
```

Listing 2.2: Beispiel einer OSGi Bundle Manifest.

In der Datei muss insbesondere der Bundle Symbolic Name (BSN) definiert werden, der die eindeutige Kennung des Bundles darstellt. Im Beispiel ist der BSN „org.example.myapp“. Gemäß einer Konvention sollten die Namen aller Pakete („packages“), die zu einem Bundle gehören, mit dem BSN beginnen. Weiterhin muss explizit angegeben werden, welche Pakete das Bundle veröffentlicht („Export-Package“) und importiert bzw. aus anderen Bundles verwendet („Import-Package“). Auf diese Weise erhält man eine effektive Kontrolle über die Schnittstelle und Abhängigkeiten des Bundle. Grundsätzlich setzen Bundles das Konzept der Modularisierung um. In der Entwicklungsumgebung Eclipse kann ein Bundle durch ein „Plug-in Project“ erstellt werden. [Vog13]

Ein Bundle kann Services (Dienste) mit einer definierten Schnittstelle bereitstellen (Service Provider) bzw. nutzen (Service Consumer). Zur Bereitstellung eines Service muss zunächst deren Schnittstelle in einem Bundle definiert werden. Ein weiteres Bundle stellt den Service bereit, indem es die Schnittstelle implementiert und die Service-Implementierung in der OSGi Service Plattform registriert. Letzteres

erfolgt über eine Component Definition-Datei (Declarative Services[3]). Diese XML-Datei enthält insbesondere die „fully qualified names“[4] der Schnittstelle und der Implementierungsklasse. Analog können weitere Services mit selbiger Schnittstelle bereitgestellt werden. Die Schnittstelle und Implementierungen werden somit durch separate Bundles[5] repräsentiert.

Eine Bundle bindet sich gegen die Schnittstelle, um auf alle Implementierungen bzw. Services mit dieser Schnittstelle zugreifen zu können. Hierzu muss ebenfalls eine Component Definition-Datei erstellt werden, die die „fully qualified names“ der Schnittstelle und der Klasse enthält, in der die Instanzen der Implementierungsklassen bereitgestellt werden sollen. Für letztere Klasse müssen weiterhin Namen für „bind“- und „unbind“-Methoden spezifiziert werden, die von der OSGi Service Plattform aufgerufen werden, falls ein Service mit der Schnittstelle verfügbar bzw. nicht mehr verfügbar ist. Wird ein Service verfügbar, so wird das entsprechende Objekt über einen Parameter der „bind“-Methode bereitgestellt und kann damit einer globalen Variable zugewiesen werden. Weiterhin muss das Paket, das die Schnittstelle enthält, im Bundle Manifest importiert werden. Die Verwendung von Services ermöglicht eine lose gekoppelte Architektur.

Die Referenz-Implementierung der OSGi-Spezifikation ist Eclipse Equinox [Vog13]. OpenTOSCA, auf das in Abschnitt 2.5 eingegangen wird, läuft auf dieser OSGi-Implementierung. Sie besteht momentan aus ca. 50 Bundles[6], die größtenteils mittels Services interagieren. Im Übrigen erfolgt der Zugriff auf Bundles direkt, d. h. durch Importieren des Pakets mit der benötigten Klasse. Eclipse stellt ein weiteres Beispiel dar, das auf Eclipse Equinox basiert.

2.4 Representational State Transfer (REST)

Representational State Transfer (REST) ist ein Architekturstil für Webanwendungen. Durch eine Anfrage an eine URI, die zu einer entsprechenden Anwendung gehört, können Daten zurückgeliefert und Änderungen hervorgerufen werden. Man redet hierbei auch von einer Ressource, die durch eine URI identifiziert und mittels einer Anfrage abgerufen, verändert oder erstellt werden kann. Im Normalfall kommt hierzu das Protokoll HTTP (Hypertext Transfer Protocol) zum Einsatz. [Hor12b, Rod08]

[3] Neben dem deklarativen Ansatz per XML kann auch der ServiceTracker verwendet werden, siehe [Hor12a].
[4] Der „fully qualified name“ der Java `File`-Klasse bspw. ist `java.io.file`.
[5] Die Verteilung der Schnittstelle und Implementierungen als separate Bundles stellt eine Konvention dar, die während der Entwicklung von OpenTOSCA eingeführt wurde.
[6] Exklusive Bundles, die zu benötigten Bibliotheken gehören.

Eine HTTP-Anfrage besteht aus zwei Teilen, dem Header (Nachrichtenkopf) und Body (Nachrichtenkörper). Es existieren verschiedene Anfrage-Methoden (Anfrage-Varianten), u. a. GET, POST und DELETE. Mittels einer GET-Anfrage wird eine Ressource angefordert. Die Ressource bzw. der Zustand der Anwendung sollte hierbei nicht verändert werden. Über eine POST-Anfrage kann eine neue Ressource erzeugt werden. Die Ressource wird unterhalb der Ressource erstellt, an deren URI die Anfrage gesendet wurde. Zurückgegeben werden sollte die URI der neuen Ressource. Auch kann POST für Operationen verwendet werden, die durch keine andere Methode abgedeckt werden. Mit einer DELETE-Anfrage kann eine Ressource gelöscht werden. [Hor12b, Rod08]

Im Body einer Anfrage können beliebige Daten übergeben werden, die z. B. zum Erstellen einer Ressource benötigt werden. Durch die Angabe eines „Content-Type“ im Header der Anfrage kann das Format des Body in der Typ/Subtyp-Struktur (z. B. „text/plain“) spezifiziert werden. [Hor12b, Rod08]

Weitere Methoden und Headerfelder sollen an dieser Stelle nicht angesprochen werden, da sie im Rahmen dieser Arbeit nicht relevant sind.

Die Funktionalitäten von OpenTOSCA (siehe Abschnitt 2.5) werden über eine REST-Schnittstelle, die von der Container API bereitgestellt wird, nach außen kommuniziert.

2.5 OpenTOSCA

OpenTOSCA[7] ist eine webbasierte Java-Implementierung einer Laufzeitumgebung für TOSCA-Anwendungen (TOSCA-Container). Cloud-Anwendungen bzw. Services, welche als Cloud Service Archive (CSAR; siehe Abschnitt 2.2.2) dem TOSCA-Container übergeben werden, können instanziiert und verwaltet werden.

Eine erste Version von OpenTOSCA wurde im Rahmen eines Studienprojekts der Institute IAAS und IPVS der Universität Stuttgart entwickelt und im Oktober 2012 fertiggestellt. OpenTOSCA soll demnächst als Open-Source-Software unter der Apache 2.0 Lizenz[8] veröffentlicht werden.

In Abschnitt 2.5.1 soll nun auf die Architektur und Features von OpenTOSCA eingegangen werden. Dabei wird der Stand des TOSCA-Containers vor der Entwicklung im Rahmen dieser Arbeit betrachtet. Die entwickelten bzw. weiterentwickelten Komponenten werden in Kapitel 4 angesprochen.

[7] Website von OpenTOSCA: http://www.iaas.uni-stuttgart.de/OpenTOSCA
[8] Apache 2.0 Lizenz: http://www.apache.org/licenses/LICENSE-2.0

2.5.1 Architektur und Features

Abbildung 2.3 veranschaulicht die Architektur von OpenTOSCA, die im Wesentlichen aus den Komponenten Container API, Control, TOSCA Engine, Implementation Artifact Engine (IA Engine), Plan Engine und Core besteht. Im Folgenden soll nun auf diese Komponenten und deren Zusammenhänge näher eingegangen werden. Dazu wird u. a. der Ablauf eines CSAR Deployments im Container beschrieben. Wie bereits

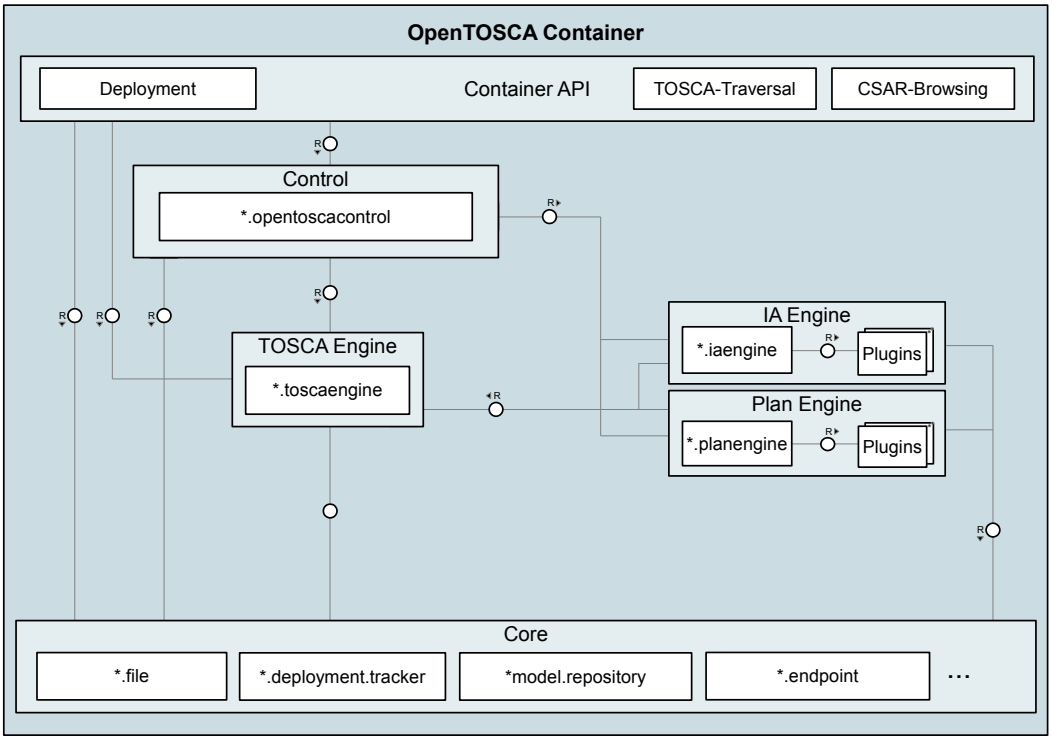

Abbildung 2.3: FMC[9]-Diagramm zur Architektur von OpenTOSCA.

in Abschnitt 2.4 erwähnt, stellt die Container API die externe REST-Schnittstelle von OpenTOSCA dar, mit der über HTTP-Anfragen kommuniziert werden kann. Sie ermöglicht den Aufruf aller Funktionen, die für den Benutzer vorgesehen sind. Eine CSAR muss über diese Schnittstelle zunächst gespeichert werden, damit sie im Container vorhanden ist. Das eigentliche Speichern erfolgt dabei im File Service der Core-Komponente, der die CSAR-Datei u. a. entpackt, damit auf deren Inhalt leichter zugegriffen werden kann. Das Löschen einer CSAR ist ebenfalls über den File Service möglich. Eine gespeicherte CSAR kann über die CSAR Browsing API durchsucht werden. Auch ist es möglich, eine Datei der CSAR herunterzuladen. Weiterhin können die Definitons-Dokumente der CSAR angezeigt und über diese traversiert werden (entsprechend der XML-Struktur; TOSCA Traversal API). Mittels

[9]Fundamental Modeling Concepts (Website: http://www.fmc-modeling.org)

der Deployment API können verschiedene Operationen (je nach Deployment-Status[10]) auf einer CSAR aufgerufen werden. Die möglichen Operationen sowie der Deployment-Status können ausgegeben werden. Wird eine Operation auf einer CSAR aufgerufen, so wird der Aufruf an die Control weitergereicht, die schließlich die zuständige Komponente aufruft. Weiterhin ist die Control für das Setzen des Deployment Status im Deployment Tracker der Core-Komponente zuständig. Der Deployment Tracker verwaltet die Deployment Status aller CSARs, die sich im Container befinden.

Die Verarbeitung der Definitions-Dokumente ist die erste Operation, die auf einer gespeicherten CSAR aufgerufen werden kann. Hierzu kommt die TOSCA Engine zum Einsatz. Die Verarbeitung besteht dabei in erster Linie aus dem Auflösen von Importen und Referenzen. Zur Vereinfachung wurde festgelegt, dass eine Datei, die in einem Definitions-Dokument importiert wird, im Ordner „IMPORTS“ in einer CSAR abgelegt sein muss. Die verarbeiteten Definitions-Dokumente werden im Model Repository der Core-Komponente gespeichert.

Anschließend können die Implementation Artifacts (IAs) (siehe Abschnitt 2.2.1) deployed werden, damit dessen Management-Operationen von Plänen aufgerufen werden können. Dieser Prozess wird durch die IA Engine umgesetzt, die auf einem Plug-in-System basiert. Jeder unterstützte Implementation Artifact-Typ wird durch ein Plug-in repräsentiert. Momentan existieren Plug-ins für WARs, die Java Servlets oder Java Server Pages (JSP) enthalten und AARs (Axis Archives). WARs werden auf einen lokalen Apache Tomcat[11], AARs auf einen lokalen Apache Axis2[12] deployed. Benötigte Daten aus Definitions und die Artefakte selbst werden von der TOSCA Engine bezogen, die wiederum mit dem File Service interagiert. Nach dem Deployment eines IAs wird der zugehörige Endpunkt im Endpoint Service der Core-Komponente gespeichert.

Nachdem die IAs deployt wurden, kann abschließend das Deployment der Pläne (siehe Abschnitt 2.2.1) initiiert werden. Dies stellt die Aufgabe der Plan Engine dar, die ebenfalls auf Plug-ins basiert. Momentan werden BPEL-Pläne unterstützt, die als ZIP-Datei bereitgestellt sind. Diese werden auf einem lokalen WSO2 Business Process Server (BPS) deployt. Beim Deployment eines Plans werden die benötigten Artefakte vom File Service bezogen. Anschließend werden die aufzurufenden Management-Operationen analysiert und gebunden, d. h. erhalten ihren korrekten Endpunkt. Hierfür werden die Endpunkt-Daten der IAs aus dem Endpoint Service benötigt.

Zur Speicherung und Verwaltung der Daten, die während der Nutzung von OpenTOSCA erzeugt werden, kommt eine lokale Derby-Datenbank zum Einsatz.

[10] Im ersten Deployment-Status „CSAR gespeichert“ bspw. kann (lediglich) die Verarbeitung der Definitions-Dokumente initiiert werden.
[11] Website von Apache Tomcat: http://tomcat.apache.org
[12] Website von Apache Axis2: http://axis.apache.org/axis2/java/core

Die Interaktion mit dieser Datenbank erfolgt über Eclipse Link[13], der Referenzimplementierung der Java Persistence API (JPA) 2.0[14] [Hor13]. Die Inhalte von CSAR-Dateien und BPEL-Plänen (ZIP-Dateien) werden direkt auf dem lokalen Dateisystem abgelegt.

Zu OpenTOSCA gehört neben dem Container auch eine grafische Benutzerschnittstelle, die auf Java Server Pages basiert (siehe Abbildung 2.4). Sie fungiert als REST Client, der mit der Schnittstelle der Container API kommuniziert. Container und GUI sind somit lose gekoppelt.

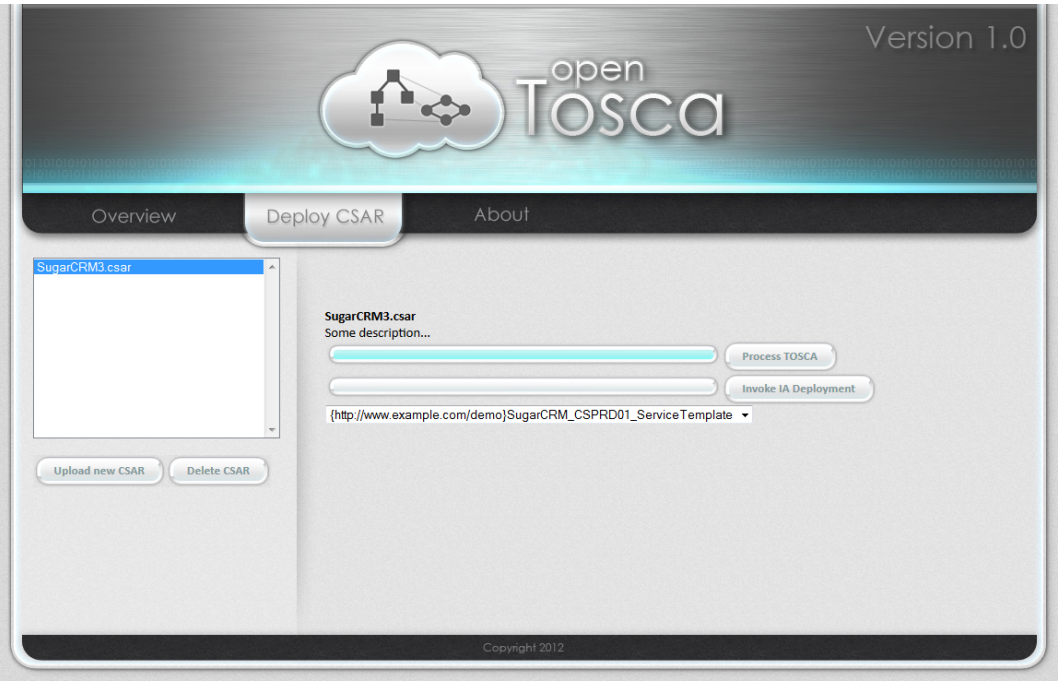

Abbildung 2.4: OpenTOSCA JSP UI. Die Definitions-Dokumente der gespeicherten CSAR wurden bereits verarbeitet. Nun kann das Deployment der IAs (eines bestimmten Service Template) initiiert werden.

2.6 Cloud Storage

Unter einem Cloud Storage versteht man einen Verband von verteilten Datencentern, die dem Benutzer eine Schnittstelle zur Verfügung stellen, um Daten bei diesen zu speichern. Die Basis bildet dabei Cloud-Computing (siehe Abschnitt 2.1). [Wil12, Sei10]

[13] Website von Eclipse Link: http://www.eclipse.org/eclipselink

[14] Die Java Persistence API (JPA) definiert eine Schnittstelle für objektrelationales Mapping (ORM) [Hor13]. Mit ORM können Objekte in einer Datenbank gespeichert und verwaltet werden [Hor13]. Die hierzu notwendigen SQL-Anweisungen werden von der ORM-Bibliothek generiert.

Um eine hohe Datensicherheit und Verfügbarkeit zu gewährleisten, werden die Daten repliziert und an verschiedenen Orten redundant abgelegt. Fällt ein Datencenter aus, so wird automatisch zu einem anderen gewechselt und der Dienst kann weiter genutzt werden. [Sei10]

Es existieren zurzeit eine Vielzahl von Anbietern, die einen Cloud Storage zur Verfügung stellen. Generell unterscheidet man zwischen zwei Arten von Anbietern: Dienste wie der Amazon Simple Storage Service (S3)[15] stellen ihre Ressourcen in erster Linie für Unternehmen (Business-to-Business) zur Verfügung. Da diese einen hohen Speicherbedarf haben, existiert in der Regel keine Begrenzung des Speicherplatzes. Die Abrechnung erfolgt nutzungsbasiert. Bei Amazon S3 bspw. unterscheidet man zwischen Speicher-, Anfrage- und Datenübertragungsgebühren [Ama]. Den anderen Bereich bilden Dienste, deren Kundenstamm vorzugsweise aus privaten Nutzern besteht (Business-to-Consumer). Diese bieten einen meist kostenlosen Basistarif mit begrenzter Speicherkapazität und u. U. weiteren Einschränkungen. Verschiedene Bezahlmodelle ermöglichen eine Erweiterung der Leistungen. Ein sehr bekannter Anbieter, der in diese Kategorie fällt, ist Dropbox[16]. [Wil12]

2.7 Blobstore

Ein Blobstore ist ein Key-Value-Store, d. h. ein Speicher, auf dem ein Schlüssel auf einen Wert abgebildet wird. Der Wert ist bei einem Blobstore ein Binary large object (Blob). Dabei handelt es sich um ein binäres Objekt, meist eine Datei. Die Schlüssel-Wert-Paare befinden sich in Containern, die eine Art von Namensraum für die Daten darstellen. Ein Container kann Ordner enthalten, wobei es sich um keine echten Ordner wie bei einem Dateisystem handelt. Beispielsweise befindet sich ein Blob mit dem Schlüssel `my-dir/my-sub-dir/my-picture.jpg` im virtuellen Ordner `my-dir/my-sub-dir`. Blobs können auch mit Metadaten versehen werden, bspw. können Zugriffsrechte definiert werden. Diese Metadaten werden ebenfalls durch Schlüssel-Wert-Paare repräsentiert. Grundsätzlich können auf Blobs eines Blobstores über HTTP-URLs zugegriffen werden. [jcl]

Ein Beispiel für einen Blobstore stellt der Cloud-Storage-Anbieter Amazon S3 dar, den wir bereits im vorherigen Abschnitt 2.6 angesprochen haben. Container werden dort als Buckets bezeichnet und müssen global eindeutig sein, d. h. ein Bucket-Namen darf Benutzerkonto-übergreifend nur einmal vorkommen [jcl]. Weiterhin dürfen pro Benutzerkonto maximal 100 Buckets erstellt werden [jcl]. Ein Blob wäre bei Amazon S3 bspw. unter der URL `http://opentosca.s3.amazonaws.com/my-dir/app.war`

[15] Website von Amazon S3: `http://aws.amazon.com/de`
[16] Website von Dropbox: `https://www.dropbox.com`

zu finden. In diesem Fall wäre `opentosca` der Name des Buckets und `my-dir/app.war` der Schlüssel des Blobs bzw. der Datei.

2.8 jclouds

jclouds ist eine Java-Bibliothek, welche die APIs von Cloud-Anbietern abstrahiert. Prinzipiell teilt sich jclouds in zwei Bereiche auf: Die Blobstore API, auf die im folgenden Abschnitt 2.8.1 näher eingegangen wird, ermöglicht den Zugriff auf Cloud-Storage-Anbieter, die einen Blobstore bereitstellen. Mit der Compute API können IaaS und PaaS Anbieter angesprochen werden. Es können u. a. Rechnerinstanzen erstellt und gestartet werden und Anwendungen auf diesen installiert werden. Da die Compute API für diese Bachelorarbeit nicht relevant ist, wird auf sie nicht weiter eingegangen. [jcl]

Die Anbieter, die zurzeit von jclouds unterstützt werden, können der jclouds Website entnommen werden[17]. Dabei sollte beachtet werden, dass jclouds zwischen Providers und APIs unterscheidet. Eine Provider-Implementierung nutzt eine API-Implementierung und enthält zusätzliche Eigenschaften, die nur für den Provider gelten (z. B. der Endpunkt). Beispielsweise wird die S3 API von den Providern Amazon S3 und Google Cloud Storage verwendet. Provider- und API-Implementierungen werden grundsätzlich in separaten JARs verteilt.

jclouds unterstützt seit Version 1.0 OSGi [Hig11]. Alle jclouds-JARs sind Bundles. Weiterhin werden Bundle Listeners mitgeliefert, die zum Benachrichtigen eingesetzt werden können, falls ein Provider- oder API-Bundle verfügbar bzw. nicht mehr verfügbar ist.

In dieser Arbeit wird jclouds zur Realisierung der Amazon S3- und Dateisystem-Storage-Plug-ins eingesetzt (siehe Abschnitt 4.5.2).

Alternativen zu jclouds sind Apache Deltacloud[18], Apache Libcloud[19], die Dasein Cloud API[20], JetS3t[21] und typica[22].

[17] Unterstützte Providers und APIs: `http://www.jclouds.org/documentation/reference/supported-providers`
[18] Website von Apache Deltacloud: `http://deltacloud.apache.org`
[19] Website von Apache Libcloud: `http://libcloud.apache.org`
[20] Website der Dasein Cloud API: `http://www.dasein.org`
[21] Website von JetS3t: `http://jets3t.s3.amazonaws.com`
[22] Website von typica: `http://code.google.com/p/typica`

Die Entscheidung für jclouds ist aus einer Evaluation hervorgegangen, in der jclouds und die genannten Alternativen auf folgende Anforderungen überprüft wurden:

(1) Java-Bibliothek, damit sie problemlos in OpenTOSCA eingesetzt werden kann.

(2) Unterstützung für das lokale Dateisystem, da entsprechend den Anforderungen (siehe Abschnitt 3) hierfür ein Storage-Plug-in bereitgestellt werden muss.

(3) Unterstützung für Amazon S3 aus selbigem Grund.

(4) JAR-Dateien der Bibliothek Bundles, da OpenTOSCA auf der OSGi Service Plattform läuft (andernfalls müssten aus den JARs selbst Bundles erzeugt werden, z. B. mit dem OSGi-Werkzeug bnd[23]).

(5) Implementierung von Bundle Listener, die benachrichtigen, falls ein unterstützter Anbieter verfügbar bzw. nicht mehr verfügbar ist.

(6) Unter Apache 2.0 Lizenz (oder kompatibler Lizenz) lizenziert, da OpenTOSCA ebenfalls unter dieser Lizenz veröffentlicht werden soll.

Tabelle 2.1 zeigt die Ergebnisse der Evaluation. Wie zu erkennen ist, erfüllt (ausschließlich) jclouds alle aufgestellten Anforderungen. Daher erübrigt sich die Aufzählung von Anforderungen, die lediglich wünschenswert wären bzw. nicht zwingend erfüllt sein müssen.

Bibliothek	**Version**	**(1)**	**(2)**	**(3)**	**(4)**	**(5)**	**(6)**
Apache Deltacloud [Apaa]	1.1.3	✗ (Ruby)	✗	✓	✗	✗	✓
Apache Libcloud [Apab]	0.12.4	✗ (Python)	✗	✓	✗	✗	✓
Dasein Cloud API [enS]	2013.04	✓	✗	✓	✗	✗	✓
jclouds [jcl]	1.6.0	✓	✓	✓	✓	✓	✓
JetS3t [Mur]	0.9.0	✓	✗	✓	✗	✗	✓
typica [Kav]	1.7.2	✓	✗	✗	✗	✗	✓

Tabelle 2.1: Ergebnisse der Evaluation von Multi-Cloud-Bibliotheken.

[23] Website des OSGi-Werkzeugs bnd: http://www.aqute.biz/Bnd/Bnd

Anmerkungen zur Evaluation:

- Apache Deltacloud stellt eine REST API zur Verfügung, über welche die Bibliothek genutzt werden kann [Apaa]. Folglich kann sie also auch in Java eingesetzt werden. Aus Gründen des Aufwands und da explizit eine Java-Bibliothek gefordert ist, definieren wir Anforderung (1) als nicht erfüllt.
- Die Blobstore API von jclouds unterstützt das lokale Dateisystem, da hierfür ein Dateisystem-basierter Blobstore bereitgestellt wird [jcl].
- Alle Bibliotheken, die nicht in Java entwickelt wurden, erfüllen die Anforderungen (4) und (5) implizit nicht (OSGi ist eine Softwareplattform für Java).

2.8.1 Blobstore API

Die Blobstore API von jclouds bietet eine portable Möglichkeit zur Verwaltung von Blobstores, die von Cloud-Storage-Anbietern bereitgestellt werden. Im Folgenden wird beispielhaft beschrieben, wie ein Blob bzw. eine Datei mit der Blobstore API gespeichert und wieder abgerufen werden kann. Wir gehen dabei davon aus, dass die jclouds Bibliotheken bereits im Java-Klassenpfad vorhanden sind. Falls dies noch nicht der Fall ist, wird an dieser Stelle an den jclouds Installation Guide verwiesen.

Für den Zugriff auf einen Blobstore muss zunächst ein `BlobStoreContext`-Objekt erzeugt werden, siehe Listing 2.3. Hierzu sind die Provider bzw. API ID[24] und die Zugangsdaten erforderlich.

```
1 BlobStoreContext context = new BlobStoreContextFactory().createContext("
      anbieterOderApiID", identitaet, credential);
```

Listing 2.3: Erzeugen eines jclouds `BlobStoreContext`-Objekts.

Die Zugangsdaten setzen sich aus `identitaet` und `credential` zusammen. Ersteres Attribut identifiziert ein konkretes Benutzerkonto auf dem Anbieter. Das `credential` bestätigt, das man berechtigt ist, auf dieses zugreifen zu dürfen. Im Falle von Amazon S3 bspw. sind die genannten Attribute der Access Key und Secret Access Key. Falls die entsprechende Provider- bzw. API-JAR und deren Abhängigkeiten nicht im Klassenpfad vorhanden sind, so wird eine entsprechende Exception geworfen.

Zur Verwaltung des Blobstore muss daraufhin ein `BlobStore`-Objekt[25] erzeugt werden. Mit diesem wird zunächst der Container erstellt. Beide Operationen sind in Listing 2.4 dargestellt.

[24] Unterstützte Providers und APIs: http://www.jclouds.org/documentation/reference/supported-providers (ID stimmt mit der „Maven Artifact ID“ überein)

[25] Neben dem `BlobStore` bietet die Blobstore API weitere Möglichkeiten zur Verwaltung eines Blobstore, siehe [jcl].

```
1 BlobStore blobStore = context.getBlobStore();
2 blobStore.createContainerInLocation(null, "my-container");
```

Listing 2.4: Erzeugen eines jclouds `BlobStore`-Objekts und Erstellen eines Containers.

Der erste Parameter der Methode `createContainerInLocation(...)` steht für den Ort des Containers. Ein Anbieter eines Blobstore ermöglicht in der Regel die Erstellung eines Containers an verschieden Orten bzw. Regionen (Bezeichnung bei Amazon S3). `null` steht für den Standard-Ort, der vom Anbieter vorgegeben wird. Falls der Container bereits von einem anderen Benutzerkonto erstellt wurde, wird eine Exception geworfen. Selbiges passiert, falls kein Internetzugang besteht.

Das Speichern einer Datei erfolgt durch Erzeugen eines `Blob`-Objekts, welches zusammen mit dem Container-Namen der Methode `putFile(...)` übergeben werden muss, siehe Listing 2.5.

```
1 File file = new File("C:\\test.txt");
2 InputStream fileInputStream = new FileInputStream(file);
3 Blob blob = blobStore.blobBuilder("my-dir/test.txt").payload(fileInputStream).
      contentLength(file.length()).build();
4 blobStore.putBlob("my-container", blob);
```

Listing 2.5: Speichern einer Datei mit jclouds.

`my-dir/test.txt` steht für den Schlüssel, unter dem der Blob gespeichert werden soll. Statt einem `InputStream` kann der Methode `payload(...)` auch ein `File`-Objekt übergeben werden. Die Angabe der Dateigröße (`contentLength`) ist dann nicht erforderlich.

Das Abrufen der Datei erfolgt mit der Methode `getBlob(...)`. Es kann der `InputStream` der Datei bezogen oder die Datei in einen `OutputStream` geschrieben werden. Beide Varianten sind in Listing 2.6 dargestellt.

```
1 Blob blob = this.blobStore.getBlob("my-container", "my-dir/test.txt");
2 Payload payload = blob.getPayload();
3 InputStream fileInputStream = payload.getInput();
4 OutputStream fileOutputStream = new FileOutputStream("C:\\fetchedTest.txt");
5 payload.writeTo(fileOutputStream);
```

Listing 2.6: Abrufen einer Datei mit jclouds.

3 Anforderungen

Im Rahmen dieser Arbeit soll ein generischer File Service für den TOSCA-Container OpenTOSCA (siehe Abschnitt 2.5) entworfen und implementiert werden, der eine Weiterentwicklung des aktuellen File Service darstellt.

Momentan können CSAR-Dateien bzw. deren Inhalt lediglich auf dem lokalen Dateisystem abgelegt werden. Der Aufwand für eine Erweiterung um zusätzliche Speicherorte ist hoch, da die eigentliche Logik zum Speichern nicht als Erweiterung bereitgestellt wird, sondern sich im File Service selbst befindet. Dieser architekturbedingte Nachteil soll durch den generischen File Service gelöst werden.

Analog zur IA Engine und Plan Engine soll er auf einem Plug-in System basieren. Ein Plug-in soll die Logik bereitstellen, die zum Speichern von Dateien in einer bestimmten Umgebung benötigt wird. Es soll eine Schnittstelle definiert werden, die von jedem Plug-in implementiert werden muss. Diese sollte insbesondere Methoden zum Speichern, Abrufen und Löschen von Dateien bereitstellen. Das Beziehen des `InputStream` einer Datei sowie Speichern einer Datei, die als `InputStream` gegeben ist, soll ebenfalls möglich sein. Auch soll die Größe einer Datei soll bestimmt werden können. Weiterhin sollen Methoden vorgesehen werden, mit denen Zugangsdaten im Plug-in hinterlegt bzw. aus dem Plug-in gelöscht werden können. Dies ist erforderlich, da die Plug-ins in den meisten Fällen die Speicherung bei externen Anbietern ermöglichen werden. Im Normalfall erfordern diese eine Authentifizierung.

Methoden zum Erstellen von Verzeichnissen sollen nicht vorgesehen werden. Es ist ausreichend, wenn die relativen Pfade der Verzeichnisse einer CSAR lokal als Metadaten vorliegen bzw. gespeichert sind. Zudem enthalten Dateipfade Verzeichnisnamen implizit, sodass Verzeichnisse in der Regel beim Speichern der Dateien mit angelegt werden. Eine Ausnahme stellt lediglich ein leeres Verzeichnis dar. Das Speichern von Verzeichnisattributen ist nicht erforderlich.

OpenTOSCA läuft auf der OSGi Service Platform. Diese stellt Konzepte bereit, mit der eine Plug-in-Architektur realisiert werden kann. Die Plug-in-Systeme der IA Engine und Plan Engine bspw. basieren bereits auf OSGi. Auch das Plug-in-System des generischen File Service soll auf Basis von OSGi entworfen werden. In einem Bundle soll die Schnittstelle der Plug-ins definiert werden. Die eigentlichen Plug-ins

sollen weitere Bundles darstellen, welche die Schnittstelle implementieren und ihre Implementierung als deklarativen Service bereitstellen (siehe auch Abschnitt 2.3).

Im Rahmen dieser Arbeit sollen zwei Plug-ins entwickelt werden: Eine Implementierung soll weiterhin das Speichern von CSAR-Dateien auf dem lokalen Dateisystem erlauben. Mit der zweiten Implementierung sollen CSAR-Dateien auf dem Cloud-Storage-Anbieter Amazon S3 (siehe Abschnitt 2.6) gespeichert werden können.

Zum Speichern einer CSAR gehören (insbesondere) auch die folgenden Vorgänge, die vom File Service weiterhin übernommen werden müssen:

- Entpacken der CSAR und Ermitteln der entpackten Dateien und Ordner.
- Validieren des Inhalts der CSAR.
- Validieren und Parsen der TOSCA Metadatei.
- Speichern der Metadaten der CSAR in der lokalen Datenbank.

Neben dem Speichern soll das Verschieben einer bereits gespeicherten CSAR zu einem anderen Plug-in bzw. Speicherort möglich sein. Auch soll lediglich eine einzelne Datei oder ein Ordner einer CSAR verschoben werden können. Eine CSAR soll folglich auf mehrere Speicherorte verteilt werden können. Weiterhin soll eine Export-Funktion realisiert werden, mit der eine CSAR wieder als CSAR-Datei abgerufen werden kann. Das Löschen einer sowie aller gespeicherten CSAR-Dateien soll (wie bisher) ebenfalls möglich sein.

Wird aktuell eine CSAR über den File Service abgerufen, so wird diese über ein Datenmodell bereitgestellt, das auf den lokal gespeicherten Metadaten der CSAR basiert. Dieses Datenmodell stellt Methoden bereit, mit denen strukturiert auf den Inhalt der CSAR zugegriffen werden kann. Beispielsweise gibt es eine Methode, die alle Definitions-Dokumente im „Definitions“-Ordner der CSAR zurückgibt. Für die Container API bzw. deren CSAR Browsing API muss dieses Datenmodell um Methoden erweitert werden, mit denen die CSAR vollständig durchsucht werden kann. Die CSAR Browsing API bezieht momentan den absoluten Pfad des Entpack-Ordners der CSAR und ermittelt anschließend selbstständig den Inhalt der Ordners, den der Nutzer über eine entsprechende HTTP-Anfrage anfordert. Diese Vorgehensweise ist nicht optimal, da Dateisystemoperationen nicht in den Zuständigkeitsbereich der Container API fallen. Durch den generischen File Service können CSAR-Dateien nun an prinzipiell beliebigen Orten abgelegt sein, sodass Dateisystemoperationen nicht mehr möglich sind. Aus diesem Grund werden Methoden zum Durchsuchen der CSAR benötigt.

Das CSAR-Datenmodell stellt eine Methode zur Verfügung, mit der auf Artefakt-Referenzen[26] zugegriffen werden kann. Aktuell werden relative Referenzen unterstützt, die auf eine Datei bzw. einen Ordner in der CSAR verweisen. Dieser Artefakt-Referenz-Typ muss auch weiterhin unterstützt werden. Eine Einschränkung ist, dass momentan lediglich Dateien, die sich an einer Artefakt-Referenz befinden, zurückgegeben werden. Die Ordnerstruktur geht folglich verloren. Diese Einschränkung soll durch die Realisierung eines entsprechenden Datenmodells zum Durchsuchen eines Artefakts aufgehoben werden. Das zu entwerfende Modell soll dabei nicht nur bei der Methode zum Zugriff auf Artefakte zum Einsatz kommen, sondern auch bei allen weiteren Methoden des CSAR-Datenmodells, die Dateien oder Ordner einer CSAR zurückliefern, z. B. den bereits angesprochenen Methoden zum Durchsuchen der CSAR. Die Dateien eines Artefakts bzw. einer CSAR müssen über dieses Datenmodell abgerufen bzw. heruntergeladen werden können.

Zur Verwaltung von Zugangsdaten für Plug-ins soll eine separate Komponente mit einem passenden Datenmodell entwickelt werden. Diese soll insbesondere Methoden zum Speichern, Abrufen und Löschen von Zugangsdaten vorsehen. Gespeicherte Zugangsdaten sollen in einem (verfügbaren) Plug-in gesetzt bzw. aus einem Plug-in gelöscht werden können. Zugangsdaten für Plug-ins, die nicht verfügbar sind, sollen ebenfalls gespeichert werden können.

Im Rahmen der Integration sollen (insbesondere) die neuen Funktionalitäten über die Container API bereitgestellt werden. Wichtige Methoden der entwickelten bzw. weiterentwickelten Komponenten sollen zusätzlich über OSGi Konsolen Kommandos aufgerufen werden können.

[26] In der Regel aus einem Artifact Template eines Definitions-Dokuments (siehe Abschnitt 2.2.1).

4 Konzept und Implementierung

Dieses Kapitel befasst sich mit der Realisierung der Anforderungen, die im vorherigen Kapitel definiert worden sind. Dazu setzen wir uns mit der Architektur der entwickelten bzw. weiterentwickelten Komponenten auseinander und erläutern deren Funktionalitäten sowie Zuständigkeiten. Zu wichtigen Methoden gehen wir auf die konkreten Abläufe in der Implementierung ein. Dadurch decken wir mit diesem Kapitel zugleich den Implementierungsteil ab.

Jeder der folgenden Abschnitte repräsentiert eine Komponente oder ein Datenmodell, wobei einführend in 4.1 zunächst ein Überblick über die Architektur gegeben wird. In Abschnitt 4.7 werden die Arbeiten erläutert, die im Rahmen der Integration durchgeführt wurden. Insbesondere gehören dazu die Erweiterungen und Anpassungen in der Container API, damit die neuen Funktionen über die externe Schnittstelle von OpenTOSCA aufgerufen werden können. Aus Gründen der Verständlichkeit und um Unklarheiten zu vermeiden, behandeln wir in diesem Kapitel auch Aufgaben und Abläufe, die bereits der bisherige File Service angeboten hat, wobei diese nicht den Schwerpunkt darstellen sollen.

In Abschnitt 4.1 kommt zur grafischen Veranschaulichung der Architektur ein FMC-Diagramm zum Einsatz. Mit Ausnahme der Datenmodelle sind alle dort dargestellten Akteure (aktive Komponenten) OSGi-Services. Zur Vereinfachung fassen wir die Schnittstelle eines OSGi-Service und deren Implementierungen in diesem Diagramm zusammen. Weiterhin kommen UML-Klassendiagramme zum Einsatz, um die Klassenhierarchie einzelner Komponenten zu veranschaulichen. Die Abläufe von Methoden und die damit verbundene Interaktion zwischen Komponenten werden durch UML-Sequenzdiagramme verdeutlicht. Sofern nicht anders angegeben, werden Fehlerfälle und verschiedene Abläufe einer Methode aus Gründen der Übersichtlichkeit nicht dargestellt. Diese können dem zugehörigen Text entnommen werden.

4.1 Überblick

Abbildung 4.1 veranschaulicht die Komponenten der Core-Komponente von OpenTOSCA, die im Rahmen dieser Arbeit entstanden bzw. weiterentwickelt worden sind. Die zentrale Komponente stellt der File Service dar, der wie bisher zur

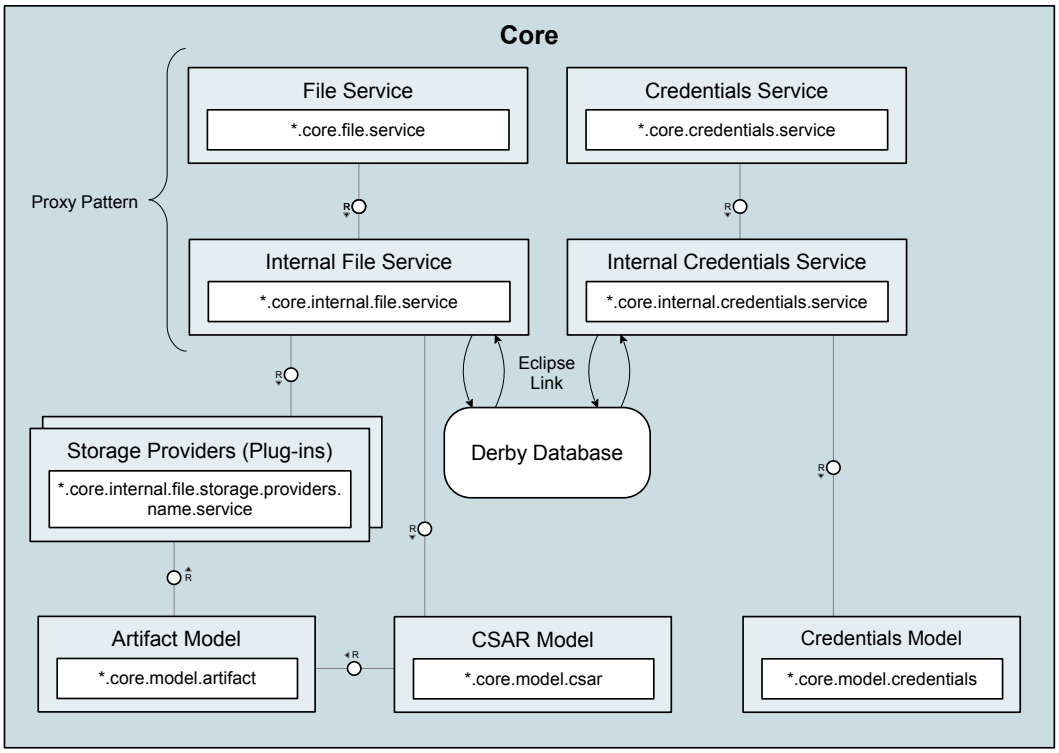

Abbildung 4.1: FMC-Diagramm zu den neuen und veränderten Komponenten der OpenTOSCA Core-Komponente.

Verwaltung von CSAR-Dateien verwendet wird, nun allerdings generisch realisiert ist und um zusätzliche Funktionen erweitert wurde. CSAR-Dateien können durch den Einsatz von Plug-ins, die im Folgenden Storage Provider genannt werden, in verschiedenen Umgebungen gespeichert werden. Die eigentliche Implementierung des File Service befindet sich im zugehörigen Internal File Service. Erstere Komponente weist lediglich die gleiche Schnittstelle auf und leitet alle Methodenaufrufe an den Internal File Service weiter (Proxy Pattern). Dieser Entwurf ermöglicht die Änderung der Schnittstelle des Internal File Service, ohne das die Schnittstelle des File Service, die für die Nutzung durch andere Komponenten gedacht ist, verändert werden muss. Der File Service kümmert sich lediglich um die Konvertierung zwischen den Schnittstellen. Dieser Ansatz kommt in allen Komponenten der Core-Komponente zum Einsatz und wurde daher im Rahmen dieser Arbeit beibehalten bzw. fortgeführt. Zur Vereinfachung führen wir beide Komponenten zusammen und bezeichnen diese im weiteren Verlauf der Arbeit als File Service.

Das CSAR Model repräsentiert die Metadaten einer CSAR und stellt Methoden bereit, mit denen strukturiert auf eine gespeicherte CSAR zugegriffen werden kann. Es wird vom File Service instanziiert und mittels Eclipse Link in der Datenbank von OpenTOSCA (siehe auch Abschnitt 2.5.1) gespeichert. Unter anderem kann über das CSAR Model auf Artefakte zugegriffen werden. Unterstützt werden weiterhin relative

Artefakt-Referenzen, die auf eine Datei oder einen Ordner in der CSAR verweisen (z. B. `IAs/deployer.war`). Bisher wurden jedoch lediglich die Dateien an der Artefakt-Referenz zurückgegeben. Die Ordnerstruktur ist folglich verloren gegangen. Durch das Artifact Model, das im Rahmen dieser Arbeit entstanden ist, können Artefakte nun unter Beibehaltung der Verzeichnisstruktur durchsucht werden. Es stellt abstrakte Klassen bereit, die vom einem Artefakt-Referenz-Typ implementiert werden müssen. Damit auch weiterhin relative Artefakt-Referenzen bzw. CSAR-Artefakte unterstützt werden, wurde hierfür eine entsprechende Implementierung bereitgestellt. Die Klasse, die eine Datei einer CSAR repräsentiert, stellt Methoden bereit, mit denen die Datei oder deren `InputStream` abgerufen werden kann. Dazu kommt der entsprechende Storage Provider zum Einsatz.

Auch stehen im CSAR Model u. a. Methoden bereit, mit denen z. B. eine CSAR durchsucht werden kann (von der Container API benötigt) oder lediglich alle Definitions-Dokumente zurückgegeben werden können. Generell werden alle Dateien und Ordner einer CSAR, die von Methoden des CSAR Model zurückgegeben werden, nun durch Klassen des Artifact Model bzw. derer Implementierung für CSAR-Artefakte repräsentiert.

Der Credentials Service dient der Speicherung und Verwaltung der Zugangsdaten, die ein Storage Provider zum Zugriff auf die entsprechende Umgebung benötigt. Analog zum File Service befindet sich deren Implementierung im zugehörigen Internal Credentials Service. Auch diese Komponenten sollen im Folgenden nur noch als Credentials Service bezeichnet werden. Zugangsdaten werden durch das Credentials Model repräsentiert und mittels Eclipse Link in der Datenbank gespeichert.

Für alle Methoden, die in diesem Kapitel vorgestellt werden, gilt, das diese mit einer entsprechenden Exception abbrechen, falls es zu einem Fehler kommt. Undo-Operationen wurde im Rahmen dieser Arbeit nicht realisiert. Ausnahmen stellen lediglich die Methoden zum Speichern und Exportieren einer CSAR dar (siehe Abschnitte 4.2.3 und 4.2.5).

Generell kommen für Dateisystem-Operationen nun Klassen aus der Java NIO.2-Bibliothek (Java-Paket `java.nio.file`) zum Einsatz, die seit Java 7 existiert. Im Rahmen der Entwicklung wurde u. a. die `File`-Klasse aus Java IO, die im bisherigen File Service verwendet wurde, durch die `Path`-Klasse ersetzt. Die Entscheidung zum Wechsel auf NIO.2 wurde in erster Linie deswegen getroffen, da diese deutlich mehr Methoden zur Manipulation von Pfaden anbietet. Besonders im File Service und Artifact Model werden entsprechende Hilfsmethoden benötigt. Weiterhin bietet sie eine höhere Performance, da jegliche Lese- und Schreiboperationen gepuffert und nicht blockierend ausgeführt werden [Jen].

4.2 File Service

Der File Service stellt Funktionalitäten zur Speicherung und Verwaltung von CSAR-Dateien zur Verfügung. Der Inhalt von CSAR-Dateien kann in Umgebungen gespeichert werden, für die entsprechende Storage Provider verfügbar sind. Neben dem Speichern sowie Löschen einer CSAR bzw. aller CSARs können auch bereits gespeicherte CSARs von einem Storage Provider zu einem anderen Storage Provider verschoben werden. Ebenso kann lediglich eine einzelne Datei oder ein Ordner verschoben werden. Weiterhin steht eine Export-Funktion zur Verfügung, mit der eine gespeicherte CSAR (wieder) als CSAR-Datei abgerufen werden kann.

In den folgenden Unterabschnitten soll nun näher auf die Funktionalitäten bzw. Methoden des File Service eingegangen werden, wobei in 4.2.1 und 4.2.2 zunächst erläutert wird, wie ein Storage Provider ausgewählt wird und welche Aufgaben vom Storage Provider Manager übernommen werden, der einen Teil des File Service darstellt.

Entsprechend den Anforderungen sind die meisten Methoden des File Service auch über OSGi Konsolen Kommandos aufrufbar.

4.2.1 Auswahl eines Storage Provider

Ein Storage Provider muss zunächst als aktiv definiert werden, damit er zum Speichern einer CSAR sowie als Ziel einer Verschiebe-Operation ausgewählt ist. Ein nicht existierender bzw. nicht verfügbarer Storage Provider kann grundsätzlich nicht als aktiv selektiert werden. Falls ein aktiver Storage Provider nicht mehr verfügbar ist, so ist kein Storage Provider als aktiv festgelegt. Wird er wieder verfügbar, so muss er erneut als aktiv definiert werden.

Damit der aktive Storage Provider schließlich auch verwendet wird, muss er zusätzlich einsatzbereit („ready“) sein. Dies bedeutet, dass alle Anforderungen des Storage Providers erfüllt sein müssen. Die Schnittstelle der Storage Providers (siehe Abschnitt 4.5.1) sieht zur Abfrage dieses Status eine entsprechende Methode vor. Ein Storage Provider auf Basis von jclouds (siehe Abschnitt 4.5.2) ist grundsätzlich einsatzbereit, falls er über den Credentials Service (siehe Abschnitt 4.6) Zugangsdaten erhalten hat und das benötigte jclouds Provider- bzw. API-Bundle verfügbar ist, wobei erstere Bedingung lediglich zutrifft, falls der Storage Provider Zugangsdaten benötigt. Ist der aktive Storage Provider nicht einsatzbereit, so wird auf den Default Storage Provider ausgewichen (durch entsprechende Logging-Meldung signalisiert[27]),

[27] Gegebenenfalls können zukünftig Klassen eingeführt werden, deren Instanzen die Ergebnisse von Operationen repräsentieren.

der fest definiert ist (lokales Dateisystem). Falls dieser nicht verfügbar oder ebenfalls nicht einsatzbereit ist, so schlägt die Operation fehl. Direkt nach dem Start des Containers ist kein Storage Provider als aktiv definiert. Es wird in diesem Fall direkt der Default Storage Provider gewählt.

Mit diesem zweistufigen Konzept soll das Risiko, dass eine Operation aufgrund eines fehlenden Storage Provider fehlschlägt, möglichst gering gehalten werden.

4.2.2 Storage Provider Manager

Im File Service erfolgt der Aufruf eines Storage Provider grundsätzlich über den Storage Provider Manager. Konkret handelt es sich dabei um eine Klasse des File Service, die sich gegen die Schnittstelle der Storage Providers bindet, um Zugriff auf die deklarativen Services bzw. Implementierungen der Storage Provider-Schnittstelle zu erhalten. Die Verwaltung der verfügbaren Storage Provider erfolgt in einer `Map`, in der die ID des Storage Providers auf die Storage Provider-Referenz (zeigt auf die Instanz der Implementierungsklasse) abgebildet ist.

Der Storage Provider Manager stellt entsprechend der Storage Provider-Schnittstelle (siehe Abschnitt 4.5.1) Methoden bereit, mit denen Dateien auf einem Storage Provider gespeichert, abgerufen und gelöscht werden können. Auch kann die Größe einer Datei zurückgegeben werden. Weiterhin existieren Methoden, mit denen z. B. der aktive Storage Provider gesetzt, der Default Storage Provider ausgegeben werden kann oder ermittelt werden kann, ob ein Storage Provider verfügbar und einsatzbereit ist. Hierbei sollte erwähnt werden, dass im Storage Provider Manager der aktive Storage Provider gespeichert und der Default Storage Provider definiert ist. Letztere Methoden sind auch über die (öffentliche) Schnittstelle des File Service zu erreichen. Die entsprechenden Methoden des File Service leiten hierzu Aufrufe an die (gleichnamigen) Methoden des Storage Provider Manager weiter (Proxy Pattern).

Abbildung 4.2 zeigt den Ablauf der Methode zum Speichern einer Datei. Neben dem absoluten Pfad muss der relative Pfad der Datei zum Wurzelverzeichnis der CSAR, die CSAR ID[28] sowie die ID des Storage Provider übergeben werden, auf dem die Datei gespeichert werden soll. Zunächst wird überprüft, ob der entsprechende Storage Provider verfügbar und einsatzbereit ist. Ist dies der Fall, so wird daraufhin der vollständige Pfad bestimmt, unter dem die Datei auf dem Storage Provider gespeichert werden soll. Der Pfad wird dabei wie folgt gebildet: „<csarID>/<relativePathToCSARRoot>“. Für eine WAR-Datei „IAs/deploy.war“ einer CSAR „myCSAR.csar“ würde sich folglich „myCSAR.csar/IAs/deploy.war“ ergeben. Mit dieser Information und dem

[28] Die CSAR ID identifiziert eine CSAR in OpenTOSCA. Es handelt es um eine Klasse, die (momentan) mittels dem Dateinamen der CSAR instanziiert wird.

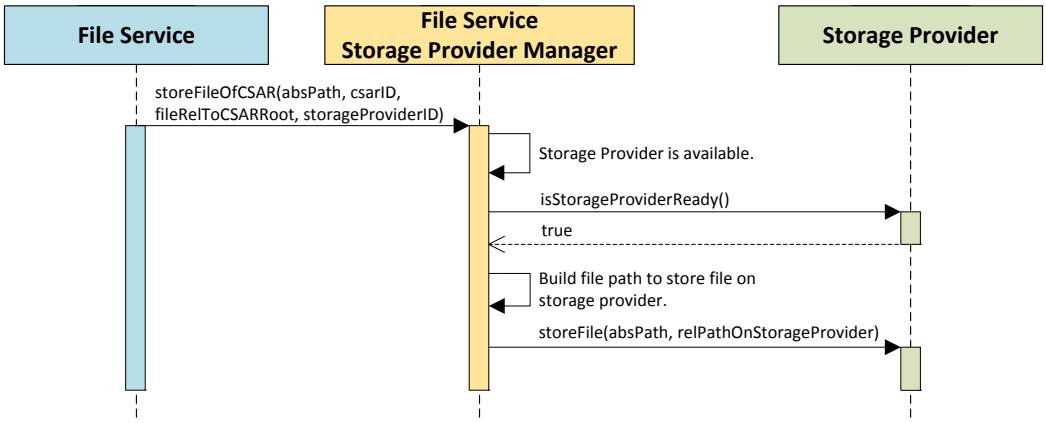

Abbildung 4.2: UML-Sequenzdiagramm zum Speichern einer Datei über den Storage Provider Manager.

absoluten Pfad der Datei wird schließlich die entsprechende Methode des Storage Provider aufgerufen. Die ersten beiden Schritte gelten analog für die weiteren Methoden des Storage Provider Manager zum Speichern (Datei gegeben als `InputStream`), Abrufen und Löschen einer Datei sowie Ermitteln der Dateigröße.

Zur Bestimmung des Storage Providers, auf dem eine CSAR gespeichert wird bzw. der das Ziel einer Verschiebe-Operation darstellt, kommt ebenfalls der Storage Provider Manager zum Einsatz. Abbildung 4.3 veranschaulicht den Ablauf der Methode, die das in Abschnitt 4.2.1 beschriebene Konzept zur Auswahl des Storage Providers umsetzt. Dargestellt ist der Fall, dass die ID des Default Storage Provider zurückgegeben wird, da der aktive Storage Provider gesetzt und verfügbar, allerdings nicht einsatzbereit ist.

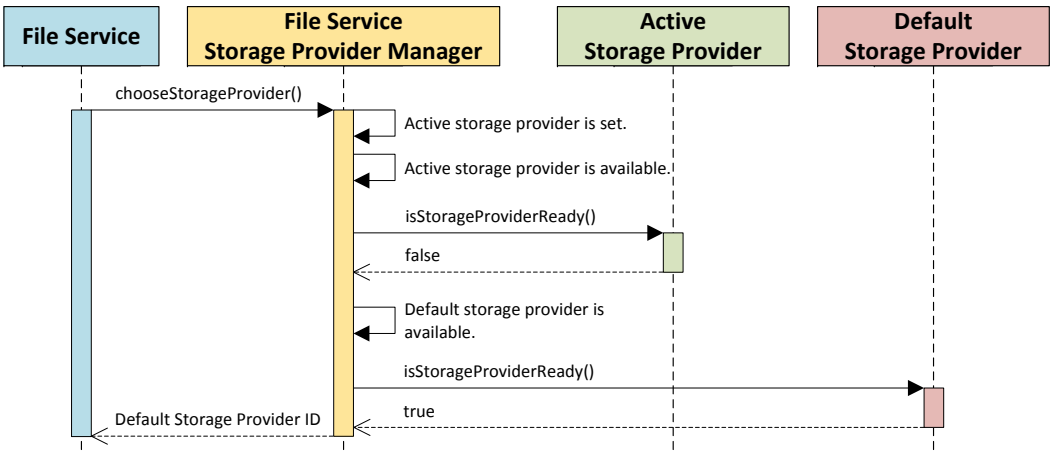

Abbildung 4.3: UML-Sequenzdiagramm zur Auswahl des Storage Providers über den Storage Provider Manager.

4.2.3 Speichern einer CSAR

Der Ablauf der Methode `storeCSAR(csarFile)` ist in Abbildung 4.4 dargestellt. Zunächst wird überprüft, ob die übergebende Datei existiert und eine Datei mit

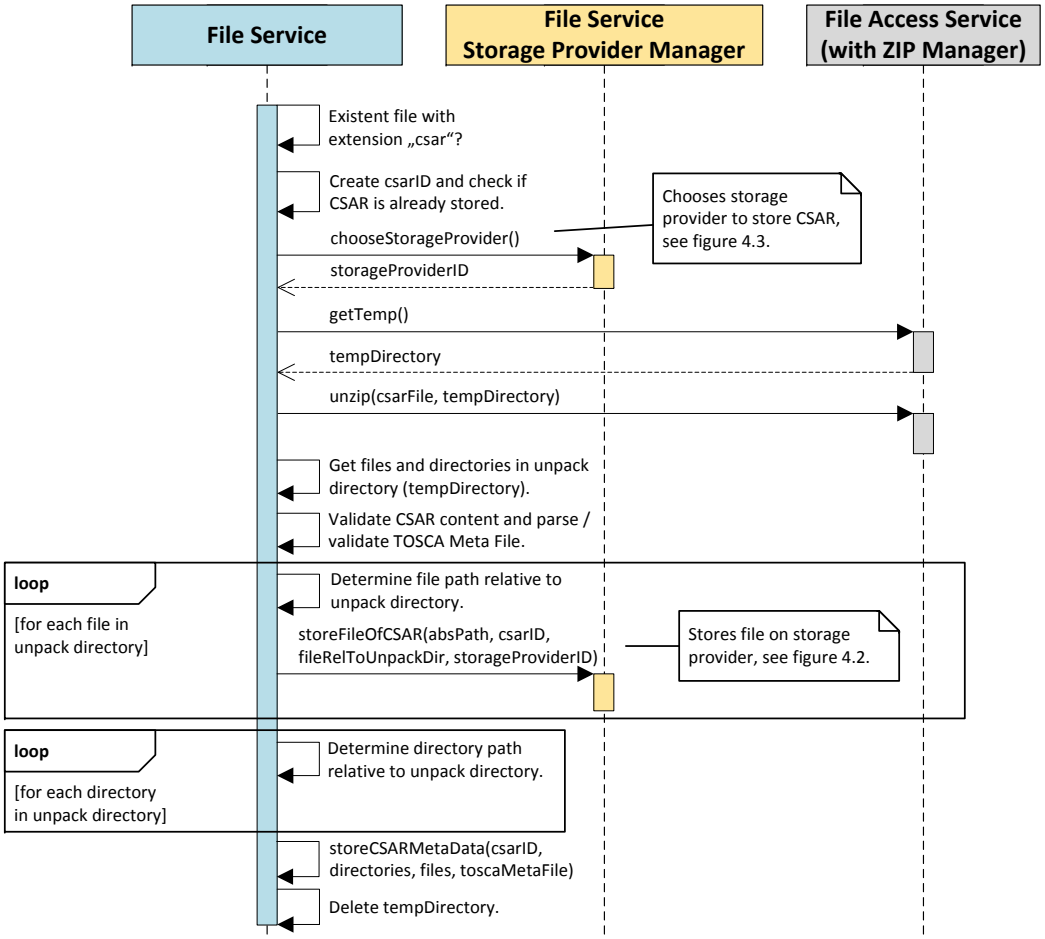

Abbildung 4.4: UML-Sequenzdiagramm zum Speichern einer CSAR.

der Endung `csar` ist. Ist dies der Fall, wird die CSAR ID (mittels dem Dateinamen der CSAR) erzeugt. Anschließend wird überprüft, ob die CSAR bereits gespeichert ist, indem ermittelt wird, ob bereits Metadaten unter der entsprechenden CSAR ID in der Datenbank gespeichert sind. Ist die CSAR nicht bereits gespeichert, so wird der Storage Provider ausgewählt, der zum Speichern der CSAR eingesetzt wird. Die Auswahl erfolgt mittels der in Abbildung 4.3 dargestellten Methode. Nach der Bestimmung des Storage Provider wird die CSAR-Datei in einem temporären Ordner entpackt und die Dateien und Ordner der CSAR ermittelt. Generell kommt für die Bereitstellung eines Temp-Ordners als auch dem Entpacken einer ZIP-Datei der File Access Service zum Einsatz. Zur rekursiven Bestimmung der Dateien und Ordner

wurde eine Implementierung des `SimpleFileVisitor`[29] (aus der NIO.2-Bibliothek) erstellt. Um das Speichern einer ungültigen CSAR (möglichst) zu verhindern, wird der Inhalt der CSAR validiert. Des Weiteren wird die TOSCA Metadatei geparst. Die Daten aus dieser Datei werden durch ein Objekt der Klasse `TOSCAMetaFile` repräsentiert. Ist die CSAR als auch deren TOSCA Metadatei gültig, so erfolgt schließlich das Speichern der Dateien über den zuvor ausgewählten Storage Provider. Jede Datei der CSAR[30] wird dazu zusammen mit der Storage Provider ID und der CSAR ID der entsprechenden Methode (siehe Abbildung 4.2) des Storage Provider Manager übergeben, der wiederum mit dem Storage Provider interagiert (siehe Abschnitt 4.2.2). Nachdem alle Dateien auf dem Storage Provider gespeichert wurden, werden die Metadaten der CSAR in der Datenbank abgelegt. Die Metadaten einer CSAR werden durch eine Instanz des CSAR Model bzw. deren Klasse `CSARContent` (siehe Abschnitt 4.3) repräsentiert und bestehen aus der CSAR ID, den Dateien und Ordnern der CSAR (relative Pfade zum Wurzelverzeichnis) und dem `TOSCAMetaFile`-Objekt. Jeder Datei wird auf die ID des Storage Providers abgebildet, auf dem diese gespeichert wird. Abschließend wird der nicht mehr benötigte Temp-Ordner gelöscht und die CSAR ID zurückgegeben.

Kommt es während des gesamten Vorgangs zu einem Fehler (z. B. CSAR-Datei bereits gespeichert oder ungültige TOSCA Metadatei), so wird zunächst der Temp-Ordner gelöscht (sofern erforderlich) und daraufhin eine entsprechende Exception geworfen.

4.2.4 Abrufen einer CSAR

Eine gespeicherte CSAR kann durch die Methode `getCSAR(csarID)` abgerufen werden. Hierbei wird die CSAR allerdings nicht heruntergeladen, sondern durch eine Instanz des CSAR Model bzw. deren Klasse `CSARContent` bereitgestellt, die beim Speichern der CSAR erzeugt und in der Datenbank gespeichert wurde. Folglich wird also lediglich die entsprechende `CSARContent`-Instanz aus der Datenbank abgerufen und zurückgegeben. `CSARContent` stellt Methoden bereit, mit denen strukturiert auf den Inhalt der CSAR zugegriffen werden kann. Näheres zum CSAR Model in Abschnitt 4.3.

4.2.5 Exportieren einer CSAR

Eine gespeicherte CSAR kann über den File Service exportiert bzw. als CSAR-Datei abgerufen werden. Der Ablauf der Methode `exportCSAR(csarID)` ist in Abbildung 4.5

[29] Weitere Informationen zum `SimpleFileVisitor`: http://docs.oracle.com/javase/tutorial/essential/io/walk.html

[30] Absoluter Pfad und relativer Pfad zum Wurzelverzeichnis der CSAR.

dargestellt. Zu Beginn werden die Metadaten zu den Dateien der CSAR (relative

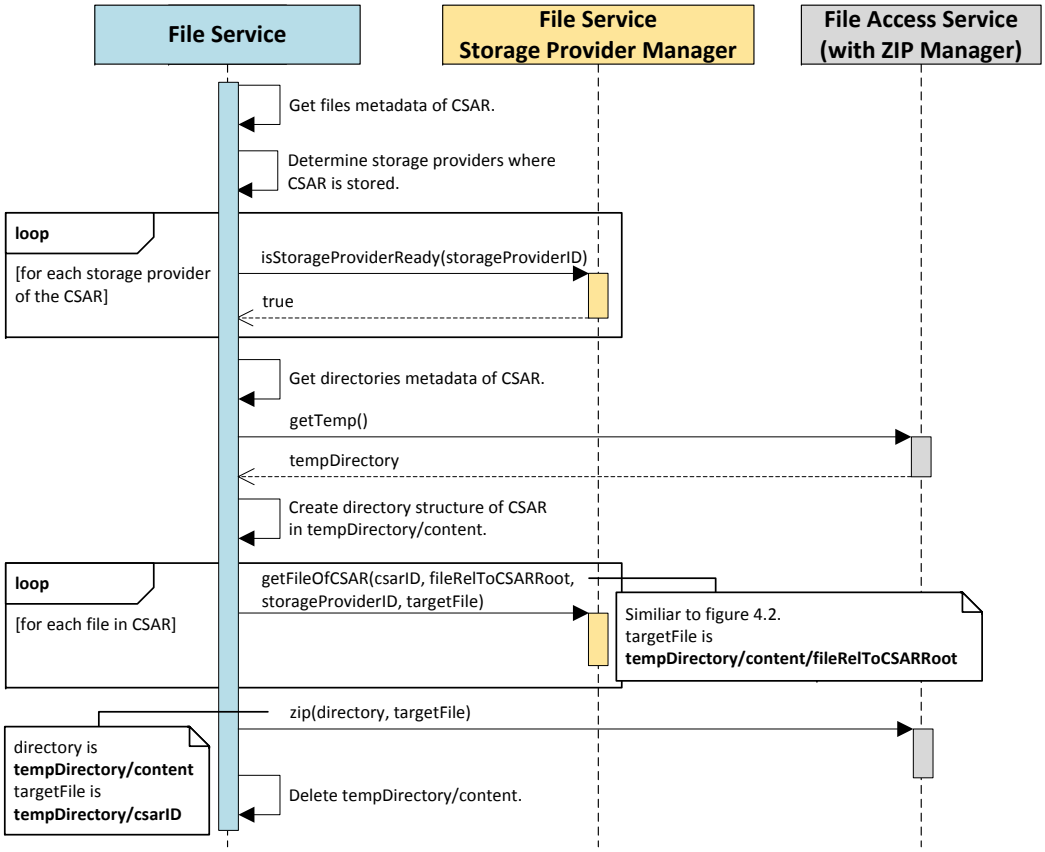

Abbildung 4.5: UML-Sequenzdiagramm zum Exportieren einer CSAR.

Pfade mit zugehöriger Storage Provider ID) aus der Datenbank abgerufen. Die Storage Provider IDs werden nacheinander dem Storage Provider Manager (siehe Abschnitt 4.2.2) übergeben, der überprüft, ob die Storage Providers, auf denen die CSAR verteilt ist, verfügbar und einsatzbereit sind. Hiermit soll das Risiko, dass das Abrufen der Dateien fehlschlägt, minimiert werden. Sind alle Storage Providers einsatzbereit, so werden auch die Metadaten zu den Ordnern der CSAR aus der Datenbank bezogen. Über den File Access Service wird ein Temp-Ordner geholt, in dem ein Unterordner „content“ angelegt wird. In diesem wird zunächst die Ordnerstruktur der CSAR angelegt und schließlich alle Dateien der CSAR über den Storage Provider Manager von den entsprechenden Storage Providern abgerufen. Nachdem dies erfolgreich abgeschlossen wurde, wird mit dem Inhalt des Ordners „content“ die CSAR-Datei (Dateiname ist die CSAR ID) direkt im Temp-Ordner erzeugt. Hierzu kommt erneut der File Access Service zum Einsatz. Abschließend wird der nicht mehr benötigte Ordner „content“ gelöscht und der absolute Pfad der CSAR-Datei als `Path`-Objekt zurückgegeben.

In einem Fehlerfall (z. B. benötigter Storage Provider nicht einsatzbereit) wird der Ordner „content“ ebenfalls gelöscht (sofern erforderlich) und eine entsprechende Exception geworfen.

4.2.6 Verschieben einer Datei oder Ordner einer CSAR

Abbildung 4.6 zeigt den Ablauf der Methode `moveFileOrDirectoryOfCSAR(csarID, relPathToCSARRoot)` zum Verschieben einer Datei oder Ordner einer CSAR zu einem anderen Storage Provider. Dargestellt ist das Verschieben eines Ordners, was den komplexeren Fall darstellt. In einem ersten Schritt wird über den Storage Provider

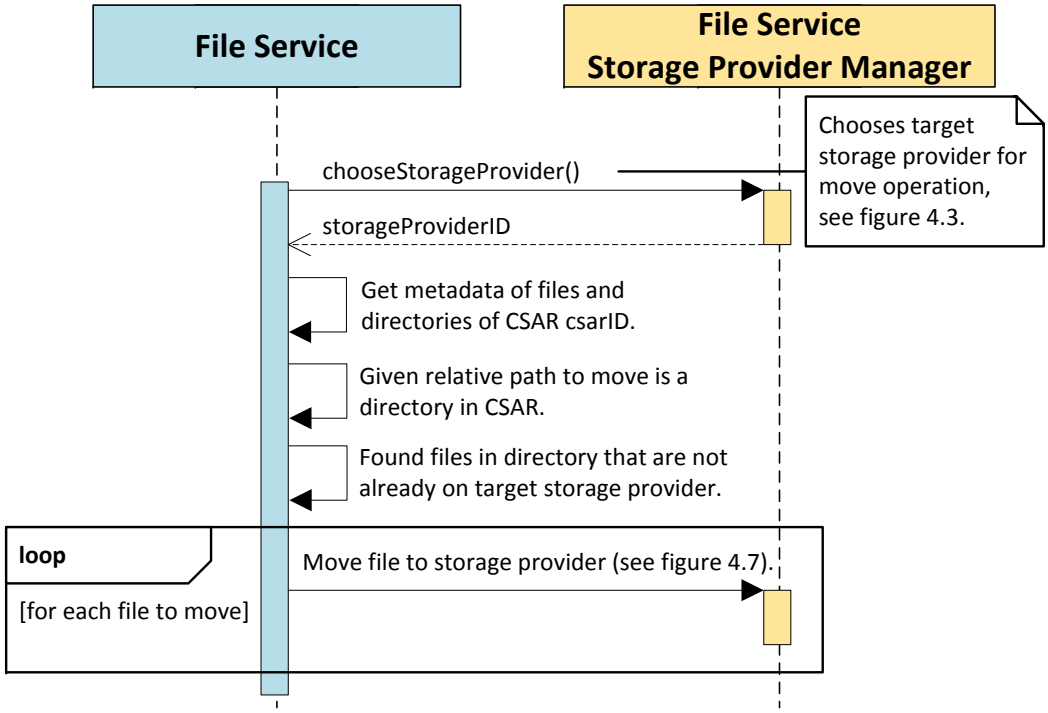

Abbildung 4.6: UML-Sequenzdiagramm zum Verschieben eines Ordners zu einem anderen Storage Provider.

Manager (siehe Abschnitt 4.2.2) der Storage Provider bestimmt, der das Ziel des Verschiebe-Vorgangs darstellt. Daraufhin werden die Metadaten der entsprechenden CSAR aus der Datenbank abgerufen. Es genügen die Metadaten zu den Dateien (relative Pfade mit zugehöriger Storage Provider ID) und Ordnern der CSAR (relative Pfade). Obwohl lediglich Dateien einer CSAR verschoben werden, sind dennoch die Metadaten der Ordner erforderlich, da wir andernfalls nicht unterscheiden können, ob der Nutzer der Methode einen nicht existierenden Ordner übergeben hat (Fehlerfall) oder ob es sich dagegen um einen Ordner handelt, der existiert, jedoch leer ist. Falls der Nutzer den relativen Pfad eines existieren Ordners übergeben hat, so werden

alle Dateien bestimmt, die verschoben werden müssen, d. h. jene Dateien, die sich im Ordner befinden und nicht bereits auf dem Ziel-Storage Provider abgelegt sind. Sofern eine oder mehrere Dateien verschoben werden müssen, so erfolgt dies im Folgenden für jede Datei nach der in Abbildung 4.7 dargestellten Vorgehensweise. Zunächst wird der `InputStream` und die Größe der Datei vom aktuellen Storage

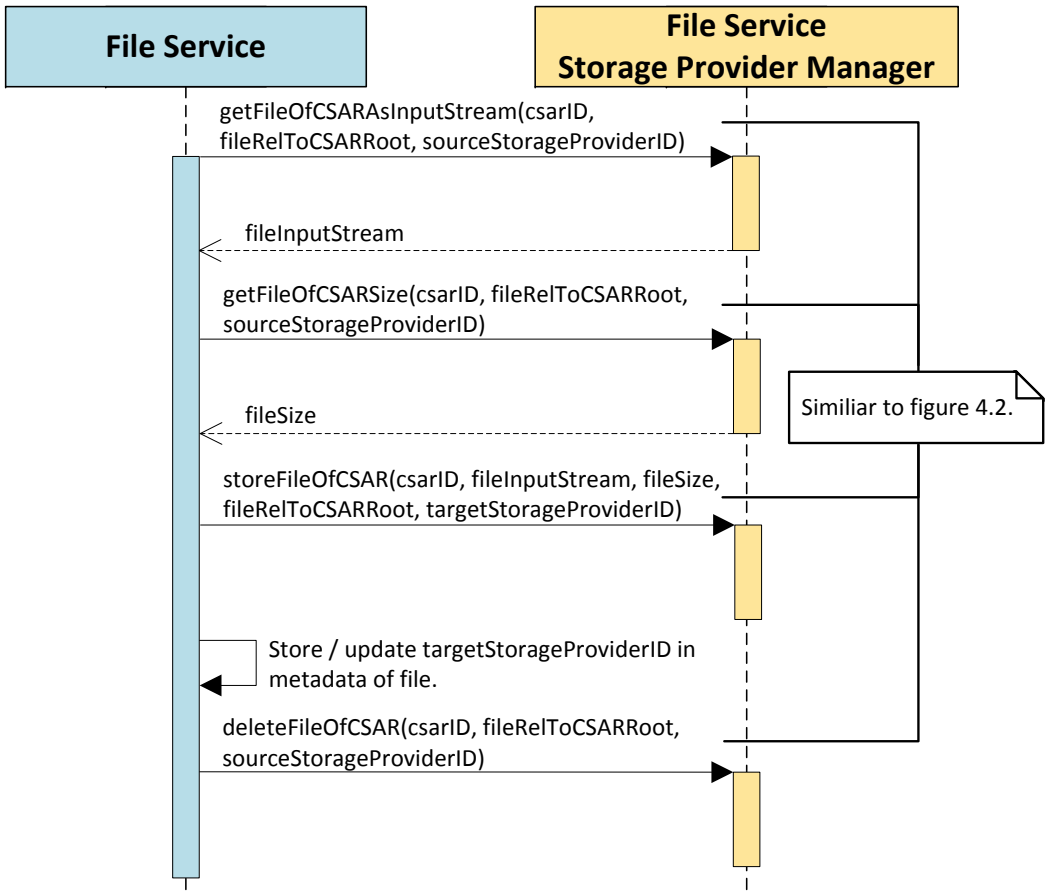

Abbildung 4.7: UML-Sequenzdiagramm zum `InputStream`-basierten Verschieben einer Datei zu einem anderen Storage Provider.

Provider abgerufen. Mit diesen Daten wird die Datei anschließend auf dem Ziel-Storage Provider gespeichert. Ist der Speichervorgang erfolgreich abgeschlossen, so werden die Metadaten der Datei in der Datenbank aktualisiert. Die ID des bisherigen Storage Provider wird hierbei mit der ID des neuen Storage Providers überschrieben. Abschließend wird die Datei auf dem bisherigen Storage Provider gelöscht. Alle Operationen, die auf einem Storage Provider ausgeführt werden, erfolgen über den Storage Provider Manager (siehe Abschnitt 4.2.2).

Alternativ wäre es auch möglich gewesen, die Datei zunächst komplett herunterzuladen, bevor sie schließlich auf dem neuen Storage Provider gespeichert wird. Für den `InputStream`-basierten Ansatz sprechen jedoch folgende Vorteile:

- Die Datei wird in einer geringfügig kürzeren Zeit verschoben.
- Verbrauch von weniger Speicherplatz.
- Es wird keine Datei auf dem Dateisystem erstellt, die nach Abschluss des Vorgangs gelöscht werden muss bzw. sollte.

Da das Löschen der Datei erst nach dem Speichern auf dem neuen Storage Provider erfolgt, kann es durch einen Ausfall der Internetverbindung während des Speichervorgangs nicht zu einem Verlust von Dateien der CSAR kommen. Dies trifft natürlich nur zu, falls der entsprechende Storage Provider Internetzugriff erfordert. Weiterhin wird das Aktualisieren der Metadaten vor dem Löschen ausgeführt. Das Fehlschlagen von Datenbank-Operationen kann also nicht dazu führen, dass eine Datei in einer CSAR aufgrund fehlerhafter bzw. veralteter Metadaten nicht mehr abrufbar ist.

4.2.7 Verschieben einer CSAR

Mit der Methode `moveCSAR(csarID)` kann mit einer kompletten CSAR zu einem anderen Storage Provider umgezogen werden. Ihr Ablauf entspricht im Wesentlichen der Methode zum Verschieben einer einzelnen Datei oder Ordner, auf die in Abschnitt 4.2.6 näher eingegangen wurde. Zunächst wird mittels des Storage Provider Managers der Storage Provider ermittelt, zu dem die CSAR bzw. deren Dateien verschoben werden sollen. Anschließend werden die Metadaten zu den Dateien der CSAR aus der Datenbank abgerufen. Die zu verschiebenden Dateien werden ermittelt und schließlich zum entsprechenden Storage Provider verschoben (siehe Abbildung 4.7).

4.2.8 Löschen einer CSAR oder aller CSARs

Der Ablauf der Methode `deleteCSAR(csarID)` zum Löschen einer CSAR ist in Abbildung 4.8 veranschaulicht. Zu Beginn werden die Metadaten zu den Dateien der CSAR (relative Pfade mit zugehöriger Storage Provider ID) aus der Datenbank abgerufen. Um möglichst zu vermeiden, dass der Löschvorgang fehlschlägt und damit ggf. nicht alle Dateien gelöscht sind, wird überprüft, ob alle benötigten Storage Provider verfügbar und einsatzbereit sind. Trifft dies zu, wird jede Datei auf ihrem jeweiligen Storage Provider gelöscht. Abschließend werden die Metadaten der CSAR gelöscht.

Sollte es während dem Löschen einer CSAR zum einem Fehler kommen (z. B. aufgrund eines Ausfalls der Internetverbindung), so kann die Methode für selbige CSAR nochmals aufgerufen werden, da die zugehörigen Metadaten nach wie vor vollständig vorhanden sind. Bereits gelöschte Dateien werden in diesem Fall ignoriert.

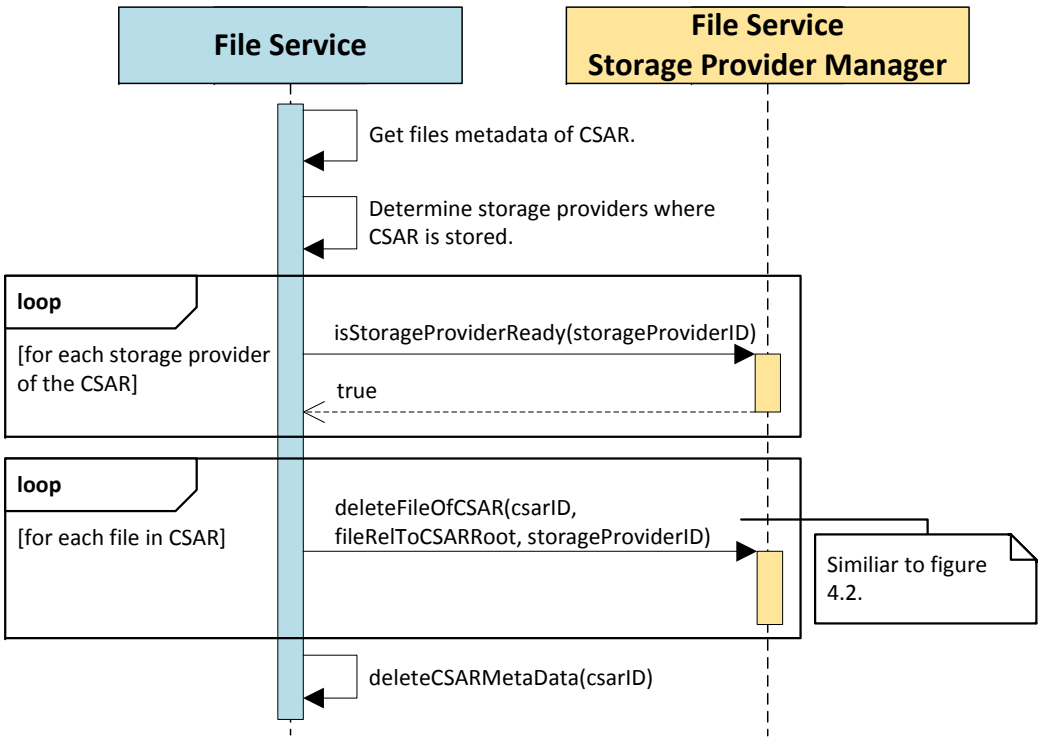

Abbildung 4.8: UML-Sequenzdiagramm zum Löschen einer CSAR.

Weiterhin wird eine Methode `deleteCSARs()` bereitgestellt, mit der alle CSARs gelöscht werden können. Diese ermittelt aus der Datenbank die IDs aller gespeicherten CSARs und ruft für jede ID die Methode zum Löschen einer CSAR auf.

4.3 CSAR Model

Das CSAR Model besteht im Wesentlichen aus den Klassen `CSARContent` und `TOSCAMetaFile`. Eine Instanz von `CSARContent` repräsentiert eine CSAR im Container und stellt Methoden bereit, mit denen strukturiert auf deren Inhalt zugriffen werden kann. Sie enthält hierzu die Metadaten der CSAR, die sich aus der CSAR ID, den relativen Pfaden aller Dateien und Ordner der CSAR sowie einem `TOSCAMetaFile`-Objekt, welches den Inhalt der TOSCA-Metadatei repräsentiert, zusammensetzen. Die relative Pfade der Dateien werden in einer `Map` gespeichert, die auf die ID des Storage Provider abbildet, auf dem die jeweilige Datei gespeichert ist. Dadurch kann prinzipiell jede Datei der CSAR auf einem anderen Storage Provider abgelegt sein. Die relativen Pfade der Ordner werden in einem `Set` verwaltet. Wie bereits angesprochen, wird `CSARContent` beim Speichern einer CSAR (siehe Abschnitt 4.2.3) instanziiert und in dieser Form in der Datenbank gespeichert. Wird eine gespeicherte

CSAR abgerufen, so wird die entsprechende `CSARContent`-Instanz aus der Datenbank geholt und zurückgegeben (siehe Abschnitt 4.2.4).

Im Rahmen dieser Arbeit wurde `CSARContent` um Methoden erweitert, mit denen die CSAR vollständig durchsucht werden kann. Zurückgegeben werden `AbstractFile`- bzw. `AbstractDirectory`-Objekte des Artifact Model (siehe Abschnitt 4.4), die Dateien bzw. Ordner repräsentieren. Erstere Klasse stellt Methoden bereit, mit denen die Datei (schließlich) heruntergeladen werden kann. Hierzu kommt der entsprechende Storage Provider zum Einsatz. Wie bereits erwähnt, werden die Methoden zum Durchsuchen der CSAR von der Container API bzw. deren CSAR Browsing API benötigt, die das Durchsuchen einer CSAR nach außen hin bereitstellt. Bisher hat die Container API den Entpack-Ordner der CSAR über eine Methode des entsprechenden `CSARContent`-Objekt geholt und anschließend mittels Dateisystemoperationen selbstständig den Inhalt des Ordners bestimmt, den der Benutzer über eine HTTP-Anfrage anfordert. Da eine CSAR nun jedoch nicht mehr lokal gespeichert sein muss, ist diese Vorgehensweise nicht mehr möglich. Dementsprechend musste die Methode, die den Entpack-Ordner der CSAR zurückgibt, entfernt werden.

Methoden, die dagegen bereits bisher von `CSARContent` angeboten wurden, stellen z. B. für die TOSCA Engine die Definitions-Dokumente im Definitions-Ordner bereit oder liefern jene Dateien zurück, die im Element `Import` eines Definitions-Dokument referenziert sind. Hierbei sollte erwähnt werden, dass während der Entwicklung von OpenTOSCA (zur Vereinfachung) festgelegt wurde, dass importierte Dateien grundsätzlich im Ordner „IMPORTS“ abgelegt sein müssen, sodass lediglich Dateien in diesem Ordner zurückgegeben werden müssen. Zusätzlich existieren Methoden, mit denen auf die Werte von Attributen aus der TOSCA Metadatei (z. B. Autor der CSAR) zugegriffen werden können. Generell gilt, das alle Methoden in `CSARContent`, die Dateien zurückliefern, diese nun als `AbstractFile`-Objekte zurückliefern. Bisher wurden `File`-Objekte zurückgegeben.

Weiterhin stellt `CSARContent` eine Methode bereit, mit der auf Artefakte zugegriffen werden kann. Hierzu muss die entsprechende Referenz übergeben werden, die in der Regel aus einem Artifact Template eines Definitions-Dokument stammt. Nach wie vor werden (ausschließlich) relative Artefakt-Referenzen unterstützt, die auf eine Datei oder Ordner in der CSAR verweisen. Bisher jedoch wurden lediglich die Dateien an einer Artefakt-Referenz zurückgegeben. Die Ordnerstruktur ist folglich verloren gegangen. Nun wird stattdessen ein `AbstractArtifact`-Objekt des Artifact Model (siehe Abschnitt 4.4) zurückgegeben, mit dem das Artefakt durchsucht werden kann. Falls die Artefakt-Referenz auf einen Ordner zeigt, sind entsprechend der TOSCA-Spezifikation zusätzlich Include Patterns und Exclude Patterns erlaubt, die zusammen mit der Artefakt-Referenz übergeben werden können. Alle Dateien

an der Artefakt-Referenz, die den Include Patterns entsprechen, werden in das Artefakt aufgenommen. Dateien, die zu den Exclude Patterns passen, werden aus dem Artefakt ausgeschlossen. Auf Ordner werden Patterns grundsätzlich nicht angewendet, sodass die Ordnerstruktur an einer Artefakt-Referenz unberührt bleibt. Da die TOSCA-Spezifikation (bisher) keine Notation für Pattern definiert, verwenden wir reguläre Ausdrücke[31]. Nehmen wir an, die Methode würde mit dem Include Pattern `^.*\.(war|WAR)$` und Exclude Pattern `^.*(deploy).*$` aufgerufen werden. In diesem Fall würde das Artefakt lediglich aus Dateien bestehen, die als Dateiendung „war“ oder „WAR“ besitzen und nicht in ihrem Dateinamen „deploy“ enthalten. Die Methode zum Zugriff auf Artefakte wird momentan von der IA Engine, Plan Engine und TOSCA Engine verwendet, wobei erstere diese nicht direkt, sondern über die TOSCA Engine aufruft.

4.4 Artifact Model

Grundsätzlich dient das Artifact Model, das im Rahmen dieser Arbeit entwickelt wurde, zum Durchsuchen ein Artefakts. Ein Artefakt kann dabei ein Ordner oder eine Datei sein, die sich prinzipiell an einem beliebigen Ort befinden kann. Falls Dateien aus dem Artefakt benötigt werden, so können diese direkt heruntergeladen bzw. abgerufen werden. Beim Entwurf, der in Abbildung 4.9 dargestellt ist, wurde auf Erweiterbarkeit geachtet. Für jeden Artefakt-Referenz-Typ (z. B. Ordner bzw. Datei auf einem FTP-Server), der unterstützt werden soll, muss eine Implementierung der Klassen `AbstractArtifact`, `AbstractDirectory` und `AbstractFile` bereitgestellt werden.

`AbstractArtifact` repräsentiert ein ganzes Artefakt und stellt das Wurzelverzeichnis des Artefakts dar. Dem Konstruktor von `AbstractArtifact` können neben der Artefakt-Referenz Include- und Exclude-Patterns übergeben werden. Wie bereits in Abschnitt 4.3 erwähnt, sind Patterns lediglich erlaubt, falls die Artefakt-Referenz auf einen Ordner verweist. Dementsprechend werden übergebene Patterns nur in diesem Fall beachtet. Weiterhin dürfen Patterns nur auf Dateien im Ordner angewendet werden. Zur Implementierung von `AbstractArtifact` müssen die Methoden `fitsArtifactReference(artifactReference)`, `getArtifactRoot()` und `isFileArtifact()` realisiert werden.
Erstere Methode ist statisch und soll `true` zurückliefern, falls die übergebene Artefakt-Referenz zu der Implementierung von `AbstractArtifact` passt. In der entsprechenden Methode der Klasse `CSARContent`, mit der auf Artefakte zugegriffen werden kann (CSAR Model, siehe Abschnitt 4.3), kann so bestimmt werden, welche

[31] Informationen zu regulären Ausdrücken: https://de.wikipedia.org/wiki/Regul%C3%A4rer_Ausdruck

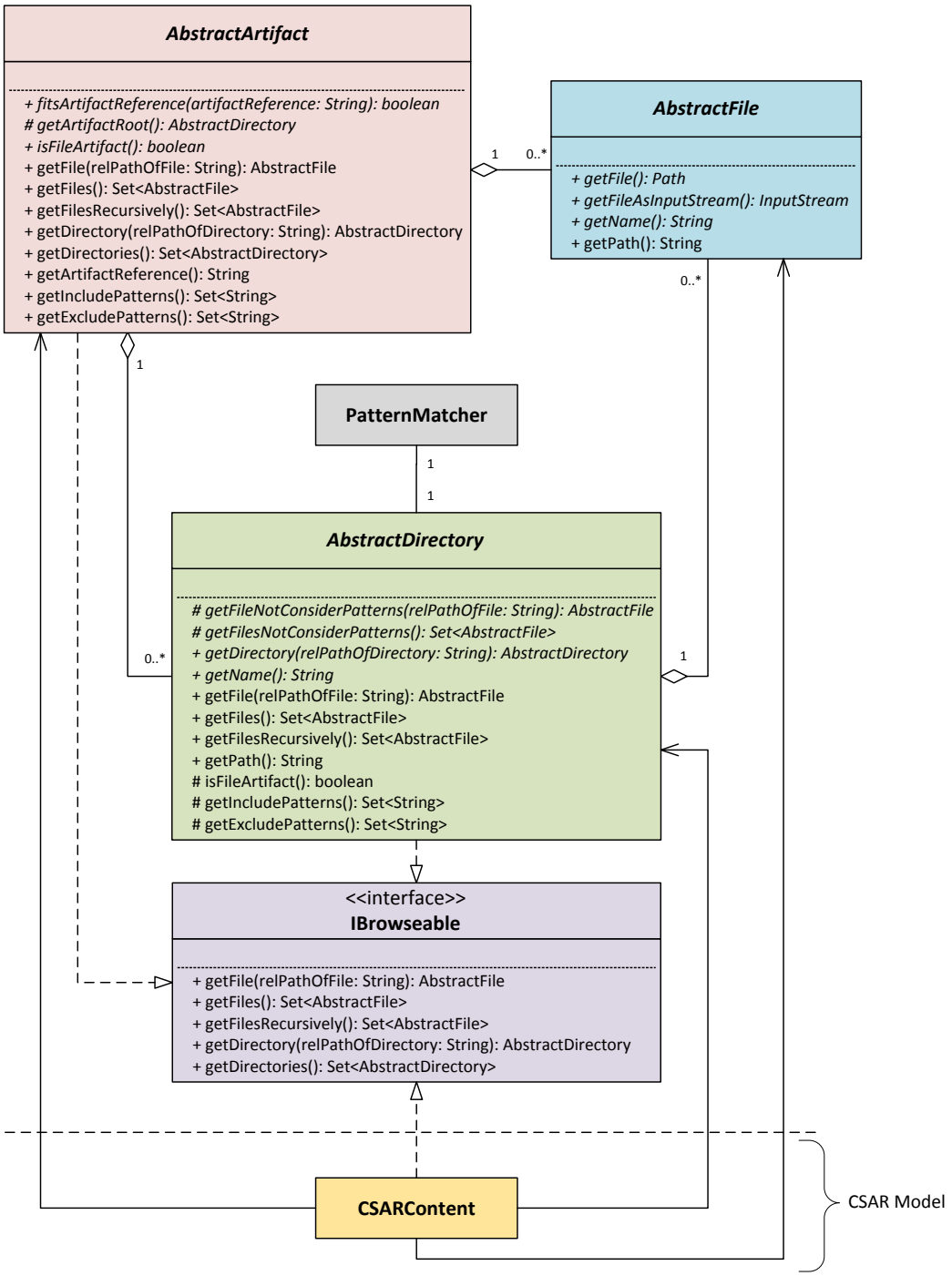

Abbildung 4.9: UML-Klassendiagramm zum Artifact Model und deren Beziehung mit `CSARContent` des CSAR Model. Die Methoden von `CSARContent` sind nicht dargestellt.

Implementierung von `AbstractArtifact` korrekt ist und damit instanziiert werden muss bzw. ob die Artefakt-Referenz nicht unterstützt wird. Falls eine passende Implementierung gefunden wurde, die Referenz jedoch auf eine Datei bzw. einen

Ordner verweist, der nicht existiert, so soll beim Erzeugen des Objekts eine Exception geworfen werden.
Die Methode `getArtifactRoot()` muss eine Instanz der zugehörigen `AbstractDirectory`-Implementierung zurückliefern, die das Wurzelverzeichnis repräsentiert. Jede Methode von `AbstractArtifact`, die zum Durchsuchen gedacht sind, delegiert an die gleichnamige Methode des `AbstractDirectory`-Objekts, das von `getArtifactRoot()` zurückgegeben wird, weiter. Ein `AbstractArtifact` soll also größtenteils durch ein `AbstractDirectory` realisiert werden, da wir davon ausgehen, dass die Logik zum Durchsuchen des Wurzelverzeichnisses und eines Ordners identisch ist.
Die Methode `isFileArtifact()` soll `true` zurückgeben, falls die Artefakt-Referenz auf eine Datei zeigt (Datei-Artefakt). In diesem Fall besteht das Artefakt lediglich aus dieser einen Datei.

Dem Konstruktor eines `AbstractDirectory` werden die Referenz des Ordners, Patterns (optional) und ein `boolean` übergeben, der definiert, ob die ursprüngliche Artefakt-Referenz auf eine Datei gezeigt hat. Letzterer Parameter ergibt sich durch die Methode `isFileArtifact()` von `AbstractArtifact` bzw. deren Implementierung.

Die Methoden zum Durchsuchen werden in einer Schnittstelle `IBrowseable` definiert, die von `AbstractDirectory`, `AbstractArtifact` und `CSARContent` implementiert werden (siehe Abbildung 4.9). Letztere Klasse implementiert die Schnittstelle zum Durchsuchen der kompletten CSAR (siehe Abschnitt 4.3). Folgende Methoden werden in `IBrowseable` definiert:

- `getFile(relPathOfFile)`: Liefert ein `AbstractFile`-Objekt zurück, das die Datei repräsentiert, die sich am übergebenen Pfad (relativ zum aktuellen Ordner) befindet. Existiert die Datei nicht, so wird `null` zurückgegeben. Im Falle eines Datei-Artefakts wird unabhängig vom übergebenen String das einzige `AbstractFile`-Objekt zurückgeliefert.
- `getFiles()`: Liefert eine Menge von `AbstractFile`-Objekten zurück, die Dateien repräsentieren, die sich im aktuellen Ordner befinden (nicht rekursiv). Im Falle eines Datei-Artefakts ist die Kardinalität der Menge eins.
- `getFilesRecursively()`: Liefert eine Menge von `AbstractFile`-Objekten zurück, die Dateien repräsentieren, die sich im aktuellen Ordner und deren Unterordnern befinden (rekursiv).
- `getDirectory(relPathOfDirectory)`: Liefert ein `AbstractDirectory`-Objekt zurück, das den Ordner repräsentiert, der sich am übergebenen Pfad (relativ zum aktuellen Ordner) befindet. Existiert der Ordner nicht, so

wird `null` zurückgegeben. Im Falle eines Datei-Artefakts wird immer `null` zurückgegeben.

- `getDirectories()`: Liefert eine Menge von `AbstractDirectory`-Objekten zurück, die Ordner repräsentieren, die sich im aktuellen Ordner befinden (nicht rekursiv). Im Falle eines Datei-Artefakts ist die Menge leer.

Eine weitere Methode `getDirectoriesRecursively()`, die rekursiv alle `AbstractDirectory`-Objekte ausgehend vom aktuellen Ordner zurückgibt, steht nicht zur Verfügung, da diese momentan nicht erforderlich ist. Falls sie zukünftig benötigt werden sollte, so kann sie mit geringem Aufwand realisiert werden (Implementierung überwiegend analog zu `getFilesRecursively()`).

Alle genannten Methoden mit Ausnahme von `getDirectory(...)` und `getDirectories()` sind bereits in `AbstractDirectory` implementiert und berücksichtigen Patterns. In einer Implementierung von `AbstractDirectory` müssen die protected-Methoden `getFileNotConsiderPatterns(...)` und `getFilesNotConsiderPatterns(...)` sowie die public-Methoden `getDirectory(...)` und `getDirectories()` realisiert werden, die keine Patterns beachten sollen. Weiterhin muss die Methode `getName()` implementiert werden, die den Dateinamen des Ordners zurückgibt. In der Oberklasse `AbstractDirectory` werden die zurückgegebenen `AbstractFile`-Objekte der beiden protected-Methoden dem `PatternMatcher` übergeben, der bestimmt, welche `AbstractFile`-Objekte den Patterns (reguläre Ausdrücke) entsprechen. Diese werden schließlich zurückgegeben. Falls dem Konstruktor von `AbstractDirectory` keine Patterns übergeben werden, so wird der `PatternMatcher` nicht aufgerufen.

Grundsätzlich wird das Pattern Matching also erst zum spätestmöglichen Zeitpunkt und nur für jene `AbstractFile`-Objekte durchgeführt, die der Nutzer anfordert. Würde man das Pattern Matching dagegen bspw. bereits beim Instanziieren einer Implementierung von `AbstractArtifact` ausführen, so müssten sofort alle `AbstractFile`-Objekte des Artefakts erzeugt werden, was u. U. unnötig ist, falls der Nutzer nur bestimmte `AbstractFile`-Objekte benötigt (ggf. unnötige Internetzugriffe).

Bisher wurde das Pattern Matching in der TOSCA Engine durchgeführt. Aus folgenden Gründen ist die Entscheidung gefallen, die Verarbeitung von Patterns nun im Artifact Model durchzuführen:

- Die IA Engine greift, wie bereits angesprochen, über die TOSCA Engine auf Artefakte zu. Letztere Komponente hat keine Informationen darüber, welche Dateien die IA Engine bzw. deren Plug-in aus einem Artefakt benötigt. Die TOSCA Engine müsste somit für das Pattern Matching immer alle `AbstractFile`-Objekte des Artefakts abrufen, was u. U. unnötig sein kann.

- Damit die TOSCA Engine `AbstractFile`-Objekte, die nicht den Patterns entsprechen, aus einem `AbstractArtifact`-Objekt löschen kann, müssten hierfür Methoden bereitgestellt werden.

`AbstractFile` repräsentiert eine Datei eines Artefakts, wobei das Instanziieren einer Implementierung von `AbstractFile` noch nicht mit dem Herunterladen der Datei verbunden ist. Dem Konstruktor von `AbstractFile` muss die Referenz der Datei übergeben werden. Wird die Klasse implementiert, so müssen Methoden zum Zurückgeben des Dateinamens und Herunterladen der Datei sowie Abrufen ihres `InputStream` realisiert werden.

4.4.1 CSAR Artefakte

Damit auch weiterhin auf relative Artefakt-Referenzen bzw. CSAR-Artefakte zugegriffen werden kann, musste im Artifact Model eine entsprechende Implementierung bereitgestellt werden, die durch die Klassen `CSARArtifact`, `CSARDirectory` und `CSARFile` (siehe Abbildung 4.10) repräsentiert wird. Da wir bereits die Metadaten

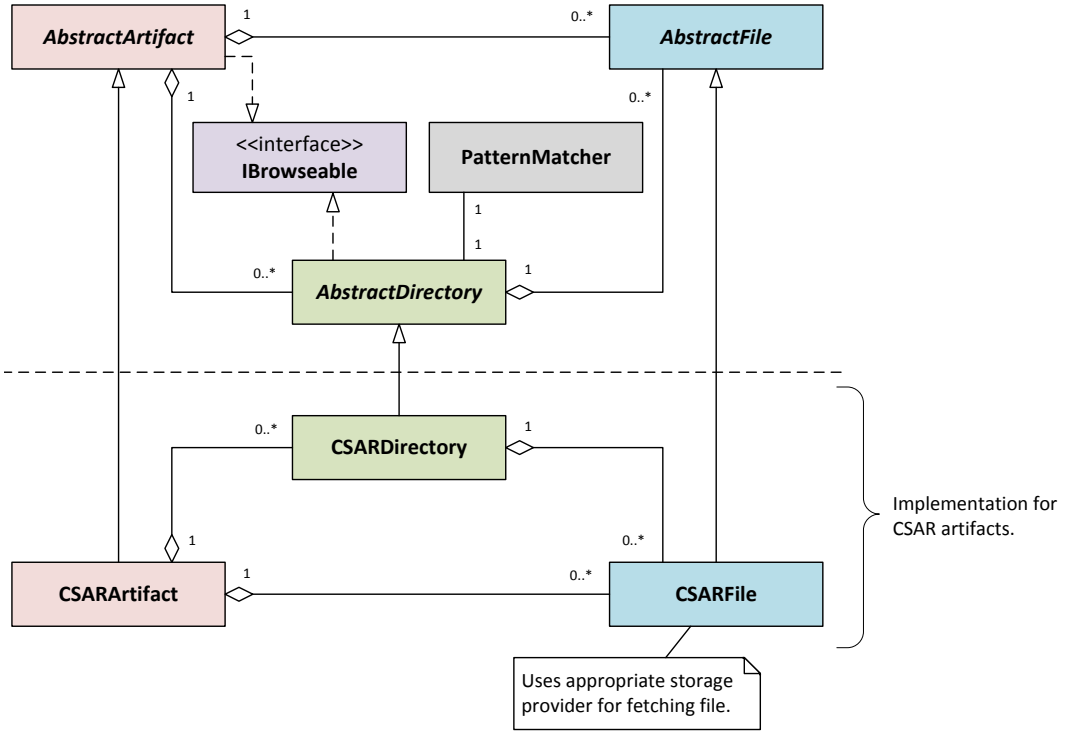

Abbildung 4.10: UML-Klassendiagramm zur Artifact Model-Implementierung für CSAR-Artefakte bzw. relative Artefakt-Referenzen.

einer CSAR lokal gespeichert haben, sollen diese auch zum Durchsuchen verwendet werden. Dementsprechend konnten die Konstruktoren von `CSARArtifact` und `CSARDirectory` nicht unverändert aus `AbstractArtifact` bzw. `AbstractDirectory`

übernommen werden, sondern mussten zusätzlich um entsprechende Parameter für CSAR-Metadaten erweitert werden. Dazu gehören die `Map`, die einen relativen Pfad einer Datei der CSAR auf die zugehörige Storage Provider ID abbildet, das `Set`, das relative Pfade von Ordnern der CSAR enthält und die CSAR ID (siehe Abschnitt 4.3). Der Konstruktor von `CSARFile` musste lediglich um die CSAR ID ergänzt werden.

Instanziiert man `CSARArtifact` mit allen Metadaten der entsprechenden CSAR und der Artifakt-Referenz, so werden im Konstruktor jene Dateien und Ordner aus den Metadaten herausgesucht, die sich an der Artefakt-Referenz befinden. Mit diesen Daten wird ein `CSARDirectory` erzeugt, das von der Methode `getArtifactRoot()` zurückgeliefert wird und damit das Wurzelverzeichnis des Artefakts repräsentiert. Die Implementierung der Methode `fitsArtifactReference(artifactReference)` überprüft, ob die URI relativ ist und damit passend zu diesem Artefakt-Referenz-Typ ist. Falls dies der Fall ist, die Referenz jedoch auf eine Datei bzw. einen Ordner verweist, der nicht in der CSAR existiert, so wird (wie im vorherigen Abschnitt gefordert) eine Exception bei der Instanziierung von `CSARArtifact` geworfen.

Die implementierten Methoden in `CSARDirectory` zum Durchsuchen eines Ordners suchen sich die entsprechenden Metadaten heraus und erzeugen entsprechend eine oder mehrere `CSARDirectory` bzw. `CSARFile`, die zurückgeliefert werden.

`CSARFile` bindet sich analog zum Storage Provider Manager des File Service (siehe Abschnitt 4.2.2) gegen die Schnittstelle der Storage Providers, damit die Datei mittels dem entsprechenden Storage Provider heruntergeladen bzw. deren `InputStream` abgerufen werden kann. Wird eine Methode zum Beziehen der Datei aufgerufen, so wird zunächst überprüft, ob der benötigte Storage Provider verfügbar und einsatzbereit ist. Falls dies zutrifft, wird der relative Pfad gebildet, unter dem die Datei auf dem Storage Provider abgelegt ist (siehe Abschnitt 4.2.2) und mit dieser Information schließlich die entsprechende Methode auf dem Storage Provider aufgerufen.

Die Implementierung für CSAR-Artefakte kommt in der Klasse `CSARContent` sowohl bei der Methode zum Zugriff auf Artefakte als auch bei allen weiteren Methoden zum Einsatz, die eine bzw. mehrere `AbstractDirectory`-Objekte oder `AbstractFile`-Objekte zurückliefern. Für letztere Methoden wird bei einer Instanziierung von `CSARContent` ein `CSARDirectory`-Objekt mit einem leeren String als Referenz (vollständige CSAR), allen Metadaten der CSAR, ihrer CSAR ID und keinen Patterns erzeugt. Der `boolean`, der definiert, ob es sich um ein Datei-Artefakt handelt, ist auf `false` gesetzt. Die Methoden zum Durchsuchen der CSAR bspw. delegieren (lediglich) an die gleichnamigen Methoden des erzeugten `CSARDirectory`-Objekts weiter.

4.5 Storage Providers

Die Storage Providers (Storage Plug-ins) bilden die Basis des File Service, da mit diesen der Inhalt von CSAR-Dateien in verschiedenen Umgebungen gespeichert werden kann. Entsprechend den Anforderungen musste zunächst eine Schnittstelle für Storage Providers definiert werden, auf die in Abschnitt 4.5.1 eingegangen wird. Implementierungen dieser Schnittstelle (Storage Providers) mussten für Amazon S3 und das lokale Dateisystem bereitgestellt werden. In Abschnitt 4.5.2 wird erläutert, wie diese mit der Bibliothek jclouds realisiert wurden. Abschnitt 4.5.3 erklärt, wie ein neuer Storage Provider entwickelt werden muss.

4.5.1 Schnittstelle

Abbildung 4.11 zeigt die Schnittstelle, die von einem Storage Provider implementiert werden muss. Beispielhaft ist zusätzlich ein Storage Provider dargestellt, der diese implementiert. Entsprechend den Anforderungen definiert die Schnittstelle Methoden, mit denen Dateien auf dem Storage Provider gespeichert, abgerufen und gelöscht werden können. Zum Speichern kann eine Datei entweder als `Path`-Objekt oder als `InputStream` übergeben werden. In letzterem Falle muss zusätzlich die Größe der Datei spezifiziert werden, da die meisten Anbieter diese Information benötigen. Dementsprechend gibt es eine Methode, mit welcher die Dateigröße bestimmt werden kann. Eine Datei kann ebenso als `Path`-Objekt heruntergeladen oder deren `InputStream` bereitgestellt werden. Falls eine der genannten Methoden fehlschlägt[32], so soll eine entsprechende Exception geworfen werden.

Für einen Storage Provider können Voraussetzungen definiert werden, die erfüllt sein müssen, damit dieser einsatzbereit ist (siehe auch Abschnitt 4.2.1). Sind alle Voraussetzungen erfüllt, so soll `isStorageProviderReady()` `true` zurückliefern. Voraussetzungen können bspw. Zugangsdaten sein, die im Storage Provider hinterlegt sein müssen oder das Vorhandensein eines (weiteren) Bundle, welches das Storage Provider Bundle benötigt.

Eine ID ist zur eindeutigen Identifizierung eines Storage Providers unerlässlich und muss daher im Storage Provider spezifiziert werden. Wie bereits angesprochen, wird in den Metadaten einer CSAR (siehe Abschnitt 4.3) der relative Pfad einer Datei zusammen mit der ID des Storage Providers gespeichert, auf dem die Datei abgelegt ist. Des Weiteren kann im File Service ein Storage Provider mittels seiner ID als aktiv definiert werden. Auch liefern einige Getter im File Service (z. B. zum Zurückgeben der ID des Default Storage Provider) und Credentials Service eine oder mehrere

[32] Beispielsweise aufgrund eines Ausfalls der Internetverbindung.

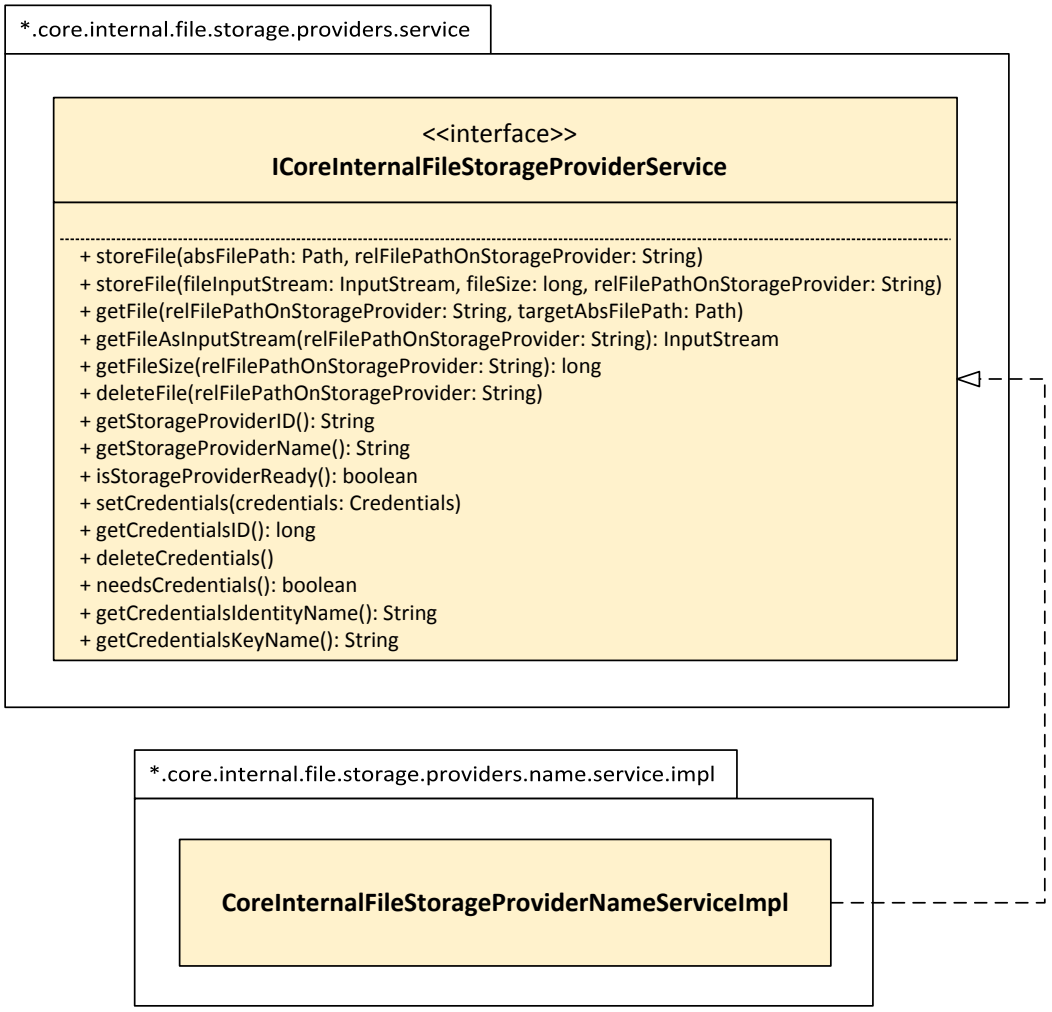

Abbildung 4.11: UML-Klassendiagramm zur Schnittstelle der Storage Providers.

Storage Provider IDs zurück. Neben der ID muss auch ein Name vergeben werden, der z. B. zukünftig zur Ausgabe auf der GUI verwendet werden kann. ID und Name des Storage Providers können über entsprechende Getter abgefragt werden.

Weiterhin stehen Methoden zur Verfügung, die ausschließlich vom Credentials Service (siehe Abschnitt 4.6) verwendet werden. Mit diesen können Zugangsdaten (Credentials) gesetzt bzw. aus dem Storage Provider gelöscht werden. Falls Zugangsdaten gesetzt sind, kann deren ID zurückgegeben werden. Die Credentials ID wird beim Speichern von Zugangsdaten im Credentials Service künstlich erzeugt und dient lediglich zur Identifizierung im Container. Für den Fall, das keine Zugangsdaten gesetzt sind, soll die entsprechende Methode `null` zurückliefern. Dementsprechend ist eine separate Methode `hasCredentials()` nicht erforderlich. Zugangsdaten bestehen neben Credentials ID, Storage Provider ID und einer optionalen Beschreibung aus Identität und Schlüssel (siehe Abschnitt 4.6). Damit z. B. auf der GUI statt

diesen generischen Begriffen die für den Storage Provider spezifischen Bezeichnungen ausgegeben werden können, müssen letztere im Storage Provider definiert werden. Zur Abfrage stehen entsprechende Methoden zur Verfügung. Des Weiteren kann zurückgegeben werden, ob der Storage Provider Zugangsdaten benötigt. Das Speichern von Zugangsdaten für einen Storage Provider, der keine benötigt, ist im Credentials Service nicht möglich.

Die Methodensignaturen der Schnittstelle enthalten keine CSAR ID. Der Aufrufer einer Methode ist folglich selbst dafür verantwortlich, aus der CSAR ID und dem relativen Pfad einer Datei der CSAR den Pfad zu bilden, unter dem die Datei auf dem Storage Provider abgelegt ist bzw. gespeichert werden soll. Diese Entwurfsentscheidung wurde aus folgenden Gründen getroffen:

- Konsistenz ist gegeben. Die Bildung des Pfads erfolgt für alle Storage Provider auf die gleiche Weise. Eine bestimmte Datei einer CSAR würde man folglich auf verschiedenen Storage Providern unter dem gleichen Pfad finden.
- Reduktion der Codemenge und ggf. Coderedundanz. Die Bildung des Pfads muss lediglich der Aufrufer eines Storage Provider durchführen und nicht jeder Storage Provider selbst.
- Falls zukünftig der File Service auch zum Speichern von Dateien eingesetzt werden soll, die nicht Teil einer CSAR sind, so ist zumindest die Schnittstelle hierfür bereits passend.

Gemäß einer Konvention, die während der Entwicklung von OpenTOSCA eingeführt wurde, sollen Schnittstelle und deren Implementierungen (in diesem Fall Storage Provider) in separaten Bundles verteilt werden (siehe auch Abschnitt 2.3). Diese Konvention wurde auch im Rahmen dieser Arbeit eingehalten. Damit auf einen Storage Provider zugegriffen werden kann, muss dieser einen deklarativen Service bereitstellen (siehe Abschnitt 4.5.3).

4.5.2 Realisierung mit jclouds

Entsprechend den Anforderungen mussten zwei Implementierungen der Schnittstelle für Storage Providers bereitgestellt werden: Eine Implementierung soll das Speichern auf dem lokalen Dateisystem ermöglichen. Mit einer weiteren Implementierung soll auf Amazon S3 gespeichert werden können.

Zur Verringerung des Implementierungsaufwands und der Codemenge wurde nach einer Bibliothek gesucht, mit der Dateien in verschiedenen Umgebungen gespeichert werden können. Im Rahmen einer Evaluation (siehe Abschnitt 2.8) wurden verschiedene Multi-Cloud-Bibliotheken miteinander verglichen. Die Wahl ist auf jclouds

gefallen, da diese die einzige Bibliothek ist, die alle aufgestellten Anforderungen erfüllt.

Da jclodus bzw. deren Blobstore API, die APIs von Blobstore-Anbietern abstrahiert, wurde eine abstrakte Klasse implementiert, welche die Storage Provider-Schnittstelle realisiert. In den implementierten Methoden der abstrakten Klasse werden die entsprechenden Methoden der Blobstore API aufgerufen. Im Rahmen dieser Arbeit kam jclouds in Version 1.6.0 zum Einsatz.

Ein Storage Provider, der das Speichern von Dateien in einer Umgebung erlauben soll, welche auch von jclouds unterstützt wird, kann so einfach durch Erben von der abstrakten Klasse realisiert werden. Dementsprechend wurden die Dateisystem- und Amazon S3-Storage Provider auf diese Weise realisiert. Abbildung 4.12 veranschaulicht den Entwurf mit jclouds. Die abstrakte Klasse implementiert alle Methoden

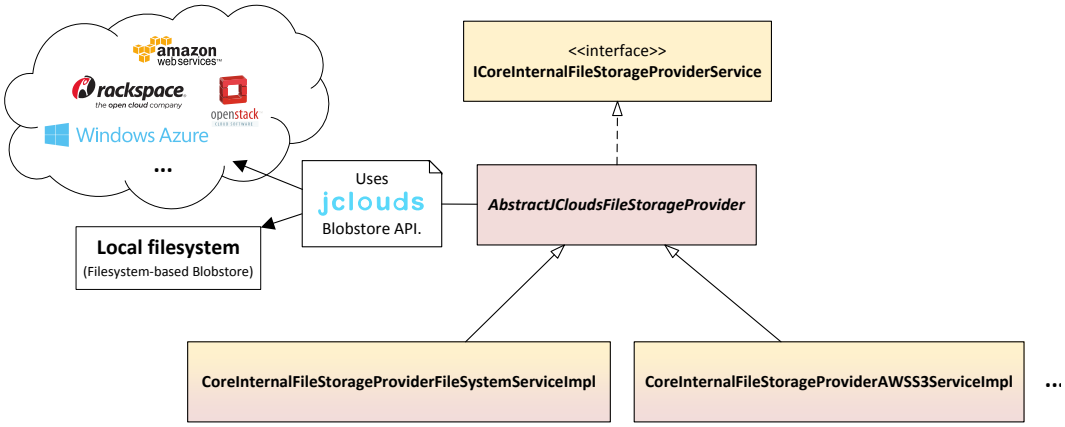

Abbildung 4.12: UML-Klassendiagramm zur Realisierung der Dateisystem- und Amazon S3-Storage Providers mittels einer abstrakten jclouds Implementierung.

der Schnittstelle bis auf die Getter zum Abfragen der ID und des Namens des Storage Provider, der spezifischen Bezeichnungen für Identität und des Schlüssel der Zugangsdaten sowie der Methode, die zurückgibt, ob der Storage Provider Zugangsdaten benötigt. Ein Storage Provider, der durch Ableiten von dieser Klasse realisiert werden soll, muss also lediglich die genannten Methoden implementieren. Dabei können die Rückgabewerte der Methoden frei gewählt werden mit Ausnahme der Methode, welche die Storage Provider ID zurückgibt. Diese muss mit der ID eines jclouds Provider oder einer jclouds API übereinstimmen. Falls kein jclouds Provider / API für den benötigten Anbieter existiert, so kann der Storage Provider nicht auf Basis von jclouds realisiert werden. In diesem Fall muss die komplette Schnittstelle implementiert werden.

In einer privaten Initialisierungs-Methode in der abstrakten Klasse wird der jclouds `BlobStoreContext` (siehe auch Abschnitt 2.8.1) mit der Storage Provider ID und den Zugangsdaten, die im Storage Provider gesetzt sind, erzeugt. Weiterhin wird der Container[33] auf dem entsprechenden Anbieter erzeugt. Als Name für den Container ist standardmäßig „org.opentosca.csars“ definiert. Durch Überschreiben einer Methoden in einer Implementierung der abstrakten Klasse kann ein anderer Container-Namen definiert werden. Näheres hierzu und zu weiteren Parametern, die verändert werden können, in Abschnitt 4.5.3. Falls die gesetzten Zugangsdaten falsch oder der Container bereits mit anderen Zugangsdaten erstellt wurde, wird eine entsprechende Exception geworfen und die Initialisierung schlägt fehl. Selbiges passiert, falls keine Internetverbindung vorhanden oder ein anderer Fehler auftritt (z. B. temporäres Problem mit dem Anbieter). Die Intialisierungs-Methode wird grundsätzlich aufgerufen, falls die Initialisierung nicht bereits stattgefunden hat und der Storage Provider einsatzbereit ist.

Ein Storage Provider, der auf Basis der abstrakten Klasse realisiert wurde, ist einsatzbereit, falls er Zugangsdaten besitzt und das benötigte jclouds Provider- bzw. API-Bundle verfügbar ist, wobei erstere Bedingung lediglich zutrifft, falls der Storage Provider Zugangsdaten benötigt. Zur Benachrichtigung, wenn ein entsprechendes jclouds Bundle verfügbar bzw. nicht mehr verfügbar ist, werden Bundle Listeners eingesetzt, die jclouds mitliefert (siehe auch Abschnitt 2.8). Die abstrakte Klasse implementiert hierzu die Schnittstellen `ProviderListener` und `ApiListener`. In den implementierten Methoden wird eine globale `boolean`-Variable entsprechend gesetzt, falls das benötigte Provider- bzw. API-Bundle verfügbar bzw. nicht mehr verfügbar ist. Die Bundle Listeners werden im Konstruktor der abstrakten Klasse gestartet.

Die Initialisierung wird aufgehoben bzw. der jclouds `BlobStoreContext` geschlossen, falls Zugangsdaten aus dem Storage Provider gelöscht, neue Zugangsdaten im Storage Provider gesetzt oder zur Laufzeit einer neuer Container-Namen definiert wird. Ist ein Storage Provider nicht mehr verfügbar, so ist implizit die Initialisierung verloren gegangen.

Zum Ändern des Container-Namens zur Laufzeit wurde in der abstrakten Klasse eine entsprechende Methode implementiert, die durch ein OSGi Konsolen Kommando aufgerufen werden soll. Leider konnte die Definition des Kommandos nicht in der abstrakten Klasse erfolgen. Eine Klasse, die OSGi Konsolen Kommandos bereitstellen soll, muss grundsätzlich einen deklarativen Service bereitstellen, dessen Schnittstelle mit dem OSGi Framework mitgeliefert wird. Die Klasse, die einen Service anbietet, wird vom OSGi Framework instanziiert, was jedoch mit einer abstrakten Klasse nicht möglich ist. Das angesprochene OSGi Konsolen Kommando musste daher in

[33] Bezeichnung für Amazon S3: Bucket

den Implementierungsklassen der Storage Providers bzw. den Klassen, die von der abstrakten Klasse erben, definiert werden.

In den Metadaten einer CSAR (CSAR Model, siehe Abschnitt 4.3) wird momentan zu einer Datei die Storage Provider ID gespeichert, nicht aber zusätzlich auch der Name des Containers, in dem die Datei gespeichert ist. Falls eine CSAR somit auf mehrere Container verteilt wurde und eine Datei dieser CSAR abgerufen werden soll, so muss hierzu folglich der Namen des Containers, in dem die Datei abgelegt ist, im entsprechenden Storage Provider gesetzt sein. Andernfalls wird die Datei nicht gefunden. Folgender Ablauf veranschaulicht diese Einschränkung (X und Y stehen für beliebige, verschiedene Dateien einer CSAR):

1. Speichern einer CSAR mit dem Dateisystem-Storage Provider.
2. Verschieben einer Datei X der CSAR mit dem Amazon S3-Storage Provider.
3. Ändern des Container-Namens im Amazon S3-Storage Provider.
4. Verschieben einer Datei Y der CSAR mit dem Amazon S3-Storage Provider.

Die Dateien X und Y befinden sich nun in verschiedenen Containern (Buckets) auf Amazon S3. Falls Datei X abgerufen wird, so schlägt dies fehl, da in jenem Container nach der Datei gesucht wird, in der Y abgelegt ist. Ändert man daraufhin den Namen des Containers wieder auf den ursprünglichen Namen, so wird X gefunden, jedoch nicht mehr Y.

Ebenfalls aufgrund fehlender Metadaten (in diesem Fall die Identität der Zugangsdaten) ist das Verschieben einer Datei bzw. eines Ordners zwischen verschiedenen Benutzerkonten eines Anbieters (mit selbigen Storage Provider) nicht möglich.

Da die Erweiterung der Metadaten mit größeren Anpassungen (insbesondere im File Service und Credentials Service) verbunden ist, konnte dies aus zeitlichen Gründen im Rahmen dieser Arbeit nicht mehr erledigt werden.

4.5.3 Entwicklung eines neuen Storage Providers

In Folgenden wird beschrieben, wie ein neuer Storage Provider entwickelt werden muss, damit er in OpenTOSCA zur Verfügung steht. Als Entwicklungsumgebung kommt Eclipse zum Einsatz. Kenntnisse mit diesem Werkzeug werden dementsprechend vorausgesetzt.

Wir gehen davon aus, dass bereits alle Projekte des TOSCA-Containers im Eclipse Workspace vorhanden sind. Zunächst muss ein neues „Plug-in Project“ angelegt werden, das den Storage Provider repräsentiert. Entsprechend der Konvention sollte als Projektnamen „org.opentosca.core.internal.file.storage.providers.x.service.impl“

gewählt werden, wobei „x“ durch eine selbstgewählte Bezeichnung des Storage Providers ersetzt wird. Unter „Target Platform“ wird „an OSGi framework“ und „standard“ ausgewählt. Im zweiten Schritt des Assistenten kann unter „Name“ ein Bundle-Name definiert werden. Eine Activator-Klasse wird nicht benötigt und sollte daher nicht erstellt werden. Nachdem das Projekt angelegt ist, öffnen wir deren Bundle Manifest „META-INF/MANIFEST.MF“. Im erscheinenden Plug-in Manifest Editor wechseln wir zur Registerkarte „Dependencies“ und fügen unter „Imported Packages“ folgende Pakete hinzu:

- `org.opentosca.core.internal.file.storage.providers.service`
- `org.jclouds.logging.config`
- `org.jclouds.domain`
- `com.google.inject`
- `org.opentosca.core.model.credentials`
- `org.opentosca.exceptions`

In einem Paket, das als Namen den zuvor gewählten Projektnamen erhält, wird eine Klasse `CoreInternalFileStorageProviderXServiceImpl` erstellt, welche die Implementierung des Storage Provider repräsentiert. X wird durch eine Bezeichnung des Storage Provider ersetzt.

Nun sollte ermittelt werden, ob der benötigte Anbieter von jclouds unterstützt wird und damit die abstrakte Klasse (siehe Abschnitt 4.5.2) zur Realisierung herangezogen werden kann. Hierzu existiert auf der Website von jclouds eine Liste der unterstützten Providers und APIs. Auf dieser Seite ist lediglich der Abschnitt „Blobstore API“ relevant. Die „Maven Artifact ID“ des entsprechenden jclouds Provider bzw. API wird zum Beziehen der Bundles benötigt. Falls jclouds den benötigten Anbieter nicht unterstützt, so muss die erstellte Klasse die Schnittstelle `ICoreInternalFileStorageProviderService` vollständig implementieren. Hierzu kann ggf. eine Anbieter-spezifische Bibliothek verwendet werden. Da sich die APIs solcher Bibliotheken von Anbieter zu Anbieter stark unterscheiden, kann an dieser Stelle keine allgemeingültige Anleitung zur Realisierung der Schnittstelle dargelegt werden. Stattdessen wird auf die Dokumentation verwiesen. Dropbox bspw. wird nicht von jclouds unterstützt, da dieser Anbieter keinen Blobstore bereitstellt. Falls die Realisierung nicht auf Basis der abstrakten jclouds Implementierung erfolgt, so können die folgenden zwei Abschnitte übersprungen werden.

Im Folgenden muss das benötigte jclouds Provider- bzw. API-Bundle heruntergeladen werden. Auch die zugehörige POM wird benötigt. Dazu gehen wir auf das zentrale Maven Repository, geben die „Maven Artifact ID“ in das Suchfeld ein und laden

die JAR und POM des Maven-Artefakts in Version 1.6.0 herunter. Anschließend wechseln wir in der Kommandozeile in den Ordner, in dem die beiden Dateien abgelegt sind und führen den in Listing 4.1 dargestellten Maven-Befehl aus, mit dem alle Abhängigkeiten des Bundle (weitere Bundles) heruntergeladen werden. Hierzu muss das Build-Management-Werkzeug Apachen Maven[34] installiert sein.

```
1 mvn dependency:copy-dependencies -DincludeScope=runtime
```

Listing 4.1: Maven-Befehl zum Beziehen der Abhängigkeiten eines Maven-Projekts.

Die heruntergeladenen JARs bzw. Bundles müssen in Eclipse zur Target Platform[35] hinzugefügt werden. Dazu wird das Projekt „org.opentosca.targetplatform.container“ geöffnet und die Dateien im Ordner „JClouds“ abgelegt. In diesem Ordner befinden sich bereits das Amazon S3 Provider-Bundle, das Dateisystem API-Bundle und deren Abhängigkeiten. Zu letzteren gehören auch die jclouds Hauptkomponenten. Diese wurden durch vorherigen Maven-Befehl erneut heruntergeladen, sodass beim Ablegen der Dateien eine Meldung erscheint, ob bereits existierende Dateien überschrieben werden sollen. Grundsätzlich sollte dies verneint werden. Damit Eclipse die neu hinzugefügten Bundles sofort erkennt, muss die Datei „OpenToscaTargetPlatform.target“ geöffnet und im erscheinenden Target Definition Editor auf „Set as Target Platform“ geklickt werden.

Im Storage Provider-Projekt öffnen wir anschließend die zuvor erstellte Klasse und lassen diese von der abstrakten Klasse `AbstractJCloudsFileStorageProvider` erben. Fünf Getter müssen nun noch implementiert werden. Die Methode `getStorageProviderID()` muss dabei die jclouds Provider bzw. API ID zurückliefern. Diese entspricht der „Maven Artifact ID“. Weiterhin können folgende Methoden überschrieben werden, die durch die abstrakte Klasse bereitgestellt werden:

- `getContainerName()`: Gibt den Namen des Containers zurück, in dem die Dateien abgelegt werden. Wird die Methode nicht überschrieben, so wird als Name „org.opentosca.csars“ verwendet.
- `getContainerLocation()`: Gibt den Ort zurück, an dem der Container angelegt werden soll. Bei Amazon S3 bspw. wird der Ort als Region bezeichnet. Wird die Methode nicht überschrieben, so wird der Standard-Ort verwendet, der vom Anbieter vorgegeben wird.

[34] Website von Apache Maven: http://maven.apache.org
[35] Die Target Platform besteht aus Bundles (Bibliotheken), die zur Entwicklung und Laufzeit einer Anwendung auf Basis von OSGi zur Verfügung stehen.

- `getJCloudsModules()`: Gibt jclouds Module zurück, die geladen werden sollen. Das SLF4J[36] Logging Modul wird immer geladen, da OpenTOSCA diese Logging Facade zusammen mit dem Logging Framework Logback[37] verwendet.
- `overwriteJCloudsProperties()`: Ermöglicht das Überschreiben von vordefinierten jclouds Properties. Für die jclouds Dateisystem-Implementierung bspw. muss ein Property überschrieben werden, in dem der Pfad hinterlegt wird, unter dem der Blobstore erstellt werden soll. Wird die Methode nicht überschrieben, so werden keine jclouds Properties verändert.

Nähere Informationen zu jclouds Module sowie Properties finden sich auf der jclouds Website[38].

Der letzte Schritt besteht darin, den Storage Provider als deklarativen Service bereitzustellen. Dazu wird im Storage Provider-Projekt zunächst ein Ordner „OSGI-INF“ angelegt, in dem anschließend eine Component Definition-Datei erstellt wird. Letzteres erfolgt über „New...“ → „Other...“ → „Plug-in Development“ → „Component Definition“. Im erscheinenden Assistenten wird unter „Component Definition Information“ → „Class“ die zuvor erstellte Klasse definiert, welche die Implementierung repräsentiert. Nach der Erstellung der Datei erscheint der Declarative Services Editor, in dem zur Registerkarte „Services“ gewechselt wird. Unter „Provided Services“ wird die Storage Provider-Schnittstelle `ICoreInternalFileStorageProviderService` hinzugefügt. Für OSGi Konsolen Kommandos muss zusätzlich die Schnittstelle `CommandProvider` in die Liste eingetragen werden. Analog zum Amazon S3- und Dateisystem-Storage Provider kann bspw. ein OSGi Konsolen Kommando zur Änderung des Container-Namens zur Laufzeit definiert werden. Die Vorgehensweise zum Erstellen eines entsprechenden Kommandos kann den Implementierungsklassen der angesprochenen Storage Providern entnommen werden.

4.6 Credentials Service

Entsprechend den Anforderungen musste eine Komponente zur Verwaltung von Zugangsdaten („Credentials“) für Storage Provider entwickelt werden. Zugangsdaten sollen mit dieser Komponente direkt in einem passenden Storage Provider gesetzt bzw. aus dem Storage Provider gelöscht werden können. Aus diesen Anforderungen heraus ist der Credentials Service entstanden, der Zugangsdaten in der Datenbank speichert. Zum Zugriff auf die verfügbaren Storage Provider bindet er sich gegen die Schnittstelle der Storage Providers.

[36] Website der Simple Logging Facade for Java (SLF4J): http://www.slf4j.org
[37] Website von Logback: http://logback.qos.ch
[38] Website von jclouds: http://www.jclouds.org

Zugangsdaten werden durch das Credentials Model bzw. deren Klasse `Credentials` repräsentiert und bestehen aus (1) Identität („identity“, z. B. Benutzername), (2) Schlüssel („key“, z. B. Passwort), (3) einer optionalen Beschreibung („description“), (4) der ID des Storage Provider, für den die Zugangsdaten gedacht sind sowie einer (5) ID, welche die Zugangsdaten identifiziert („credentialsID“). Im Falle von Amazon S3 bspw. wäre die Identität der Access Key und der Schlüssel der Secret Access Key. Bei Bedarf kann die Klasse zukünftig um weitere Attribute ergänzt werden.

Die Entscheidung zur Einführung von IDs für Zugangsdaten wurde aus folgenden Gründen getroffen:

- Eine ID ist stabiler gegenüber Änderungen an der Identifikation von Zugangsdaten. Nehmen wir an, Zugangsdaten würden über die Identität identifiziert werden. Nun kommt ein weiteres Attribut hinzu und Zugangsdaten sollen über dieses und der Identität identifiziert werden. Hierfür müssten dann ggf. größere Anpassungen am Credentials Service vorgenommen werden.
- Die gespeicherten Zugangsdaten sollen über die Container API des Containers abgefragt werden können. Es ist für einen Nutzer der REST API verständlicher, wenn Zugangsdaten unter `.../<id>/` statt z. B. unter `/<identity>;<key>/` zu finden sind. Bezugnehmend zum ersten Punkt ist ersteres URI-Design zudem stabiler.

Abbildung 4.13 zeigt den Ablauf der Methode `storeCredentials(credentials)`, mit der Zugangsdaten im Credentials Service gespeichert werden können. In einem

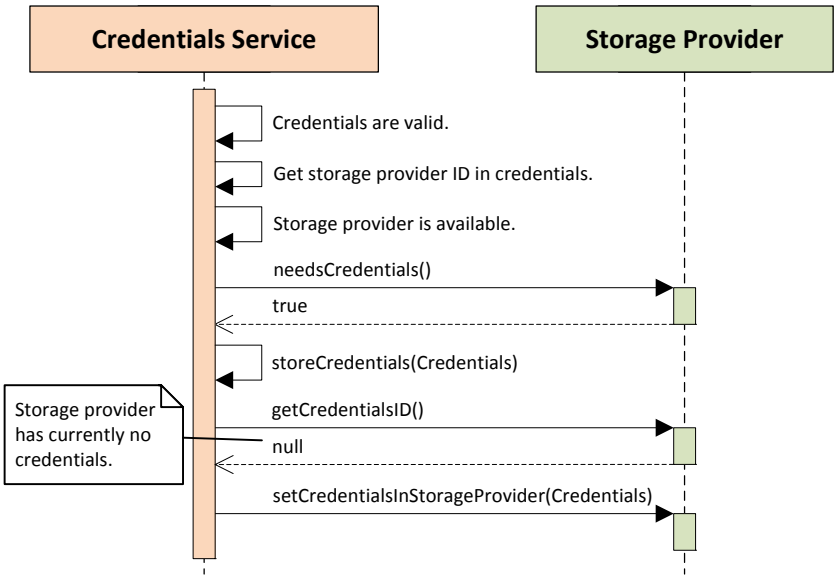

Abbildung 4.13: UML-Sequenzdiagramm zum Speichern von Zugangsdaten.

ersten Schritt wird überprüft, ob die übergebenen Zugangsdaten gültig sind. Ein

`Credentials`-Objekt, in dem kein Schlüssel, keine Identität oder keine Storage Provider ID definiert wurde, wird nicht akzeptiert. Falls die Zugangsdaten korrekt sind, so wird ermittelt, ob der entsprechende Storage Provider Zugangsdaten benötigt. Dies ist nur möglich, falls der Storage Provider verfügbar ist. Entsprechend den Anforderungen können im Credentials Service auch Zugangsdaten für Storage Provider gespeichert werden, die nicht verfügbar sind. Daraus folgt, das wir das Speichern von Zugangsdaten für einen Storage Provider, der keine Zugangsdaten benötigt und (gerade) nicht verfügbar ist, nicht verhindern können. Benötigt der Storage Provider Zugangsdaten bzw. ist dieser nicht verfügbar, so werden die Zugangsdaten daraufhin in der Datenbank gespeichert. Während diesem Vorgang wird überprüft, ob bereits Zugangsdaten mit gleichem Schlüssel und gleicher Identität existieren. Ist dies der Fall, so schlägt die Speicheroperation fehl. Hierzu wurden die genannten Attribute als „unique constraint“[39] im Credentials Model definiert. Weiterhin wird, wie bereits erwähnt, eine künstliche ID generiert und den Zugangsdaten zugewiesen. Falls der entsprechende Storage Provider verfügbar ist und nicht bereits Zugangsdaten besitzt[40], so werden die Zugangsdaten nach dem Speichern direkt im Storage Provider gesetzt (Fall, der in Abbildung 4.13 dargestellt ist). Abschließend wird die erzeugte ID der Zugangsdaten zurückgegeben.

Zugangsdaten werden nicht nur beim Speichern automatisch im Storage Provider gesetzt, sondern auch, falls ein Storage Provider verfügbar wird, für den bereits Zugangsdaten im Credentials Service gespeichert sind. Es sollte dabei beachtet werden, dass dies nur geschieht, falls nicht mehrere Zugangsdaten für den Storage Provider gespeichert sind und dieser auch Zugangsdaten benötigt. Erstere Bedingung wurde eingeführt, da wir andernfalls zufällig bestimmen müssten, welche Zugangsdaten gesetzt werden sollen.

Mit der Methode `deleteCredentials(credentialsID)` können Zugangsdaten gelöscht werden. Zunächst werden die Zugangsdaten mit der übergebenen ID aus der Datenbank abgerufen, um die ID des Storage Providers zu erhalten, für den die Zugangsdaten gedacht sind. Sofern der Storage Provider verfügbar ist, wird angefragt, ob er Zugangsdaten mit der übergebenen ID besitzt. Falls dies zutrifft, werden diese aus dem Storage Provider gelöscht, der dadurch nicht mehr einsatzbereit ist. Abschließend werden die Zugangsdaten aus der Datenbank gelöscht.

Mit `deleteAllCredentials()` können alle gespeicherten Zugangsdaten gelöscht werden. Hierzu wird `deleteCredentials(...)` wiederverwendet.

Mit den Methoden `setCredentialsInStorageProvider(credentialsID)` bzw. `deleteCredentialsInStorageProvider(credentialsID)` können bereits gespei-

[39] Ein „unique constraint“ definiert eine Menge von eindeutigen Spalten in einer Datenbanktabelle.
[40] Wir nehmen dabei an, dass das automatische Überschreiben von Zugangsdaten in einem Storage Provider unerwünscht ist.

cherte Zugangsdaten (manuell) in einem Storage Provider gesetzt bzw. aus dem Storage Provider gelöscht werden, sofern der entsprechende Storage Provider verfügbar ist. Erstere Methode überprüft, ob der Storage Provider Zugangsdaten benötigt und überschreibt bereits gesetzte Zugangsdaten. Wie bereits in Abschnitt 4.5.1 angesprochen, definiert die Schnittstelle der Storage Providers eine Methode, über die angefragt werden kann, ob Zugangsdaten erforderlich sind.

`getCredentials(credentialsID)` bzw. `getAllCredentials()` stellt Zugangsdaten bzw. alle gespeicherten Zugangsdaten bereit.

Weiterhin stehen Methoden für folgende Aufgaben zur Verfügung:

- Bereitstellen aller gespeicherten Zugangsdaten für einen bestimmten Storage Provider.
- Bestimmen der IDs aller gespeicherten Zugangsdaten.
- Ermitteln, ob ein Storage Provider Zugangsdaten bzw. Zugangsdaten mit einer bestimmten ID besitzt.
- Bestimmen, ob ein Storage Provider Zugangsdaten benötigt.
- Abfragen der Storage Provider-spezifischen Bezeichnungen für Identität und Schlüssel.

Entsprechend den Anforderungen können die meisten Methoden auch über OSGi Konsolen Kommandos aufgerufen werden.

Zurzeit kann der Credentials Service lediglich Zugangsdaten für Storage Provider verwalten. Zukünftig wäre es denkbar, dass er auch Zugangsdaten weiterer Komponenten verwaltet und somit eine zentrale Rolle in OpenTOSCA einnimmt.

4.7 Integration

4.7.1 Container API

Im Rahmen der Integration mussten insbesondere die neuen Funktionalitäten des File Service und Credentials Service über die Container API, der externen REST-Schnittstelle von OpenTOSCA, bereitgestellt werden. Die Container API wurde mittels der JAX-RS[41] Referenzimplementierung Jersey[42] realisiert und läuft auf dem HTTP-Webserver und Servlet/JSP-Container Jetty[43].

[41] Java API for RESTful Web Services (JAX-RS) spezifiziert eine Java-API zur Entwicklung von REST-basierten Anwendungen.
[42] Website von Jersey: https://jersey.java.net
[43] Website von Jetty: http://www.eclipse.org/jetty

Die Stammressource („root resource“) der Container API wird durch die URI `http://localhost:1337/containerapi` repräsentiert. Alle im folgenden genannten URIs, die Ressourcen identifizieren, werden relativ zu dieser URI angegeben. Abbildung 4.14 veranschaulicht die hierarchische Struktur der Container API, wobei lediglich Ressourcen dargestellt sind, die im Rahmen dieser Arbeit hinzugefügt wurden (blau umrandet) bzw. an denen Anpassungen vorgenommen wurden (schwarz umrandet). In den folgenden Unterabschnitten soll nun näher auf die dargestellten Ressourcen eingegangen werden.

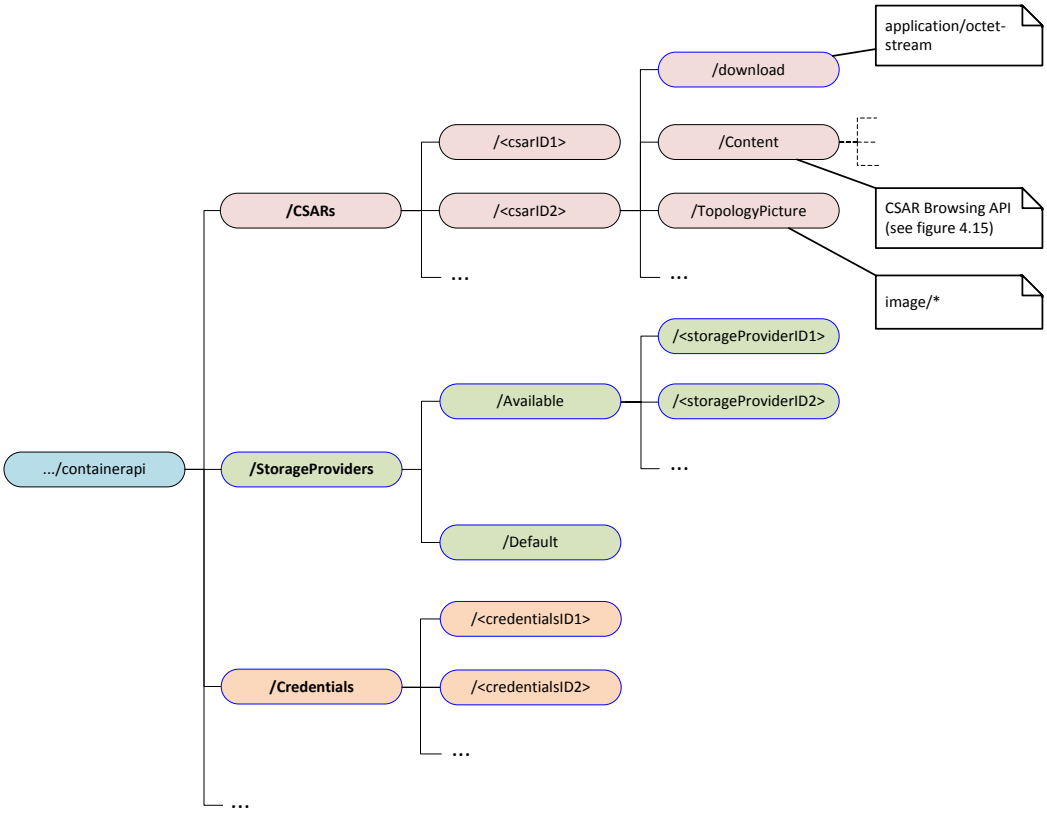

Abbildung 4.14: Neue und veränderte Ressourcen der Container API.

4.7.1.1 Storage Providers

Mittels GET-Anfragen an die URIs `/StorageProviders/Available` und `/StorageProviders/Default` werden Ressourcen bereitgestellt, die Informationen zu allen verfügbaren Storage Providern bzw. dem Default Storage Provider enthalten. Als Repräsentationsformat kommt XML zum Einsatz. Die zu einem Storage Provider zurückgegeben Daten bestehen aus (1) ID und (2) Namen des Storage Provider sowie den Informationen, ob der Storage Provider (3) als aktiv gesetzt ist, (4) einsatzbereit ist, der (5) Default Storage Provider ist, (6) Zugangsdaten benötigt

und (7) zurzeit Zugangsdaten besitzt. Auch werden die spezifischen Bezeichnungen für (8) Identität und (9) Schlüssel der Zugangsdaten zurückgegeben. Listing 4.2 zeigt beispielhaft XML-Code der Ressource `/StorageProviders/Available`.

```
<StorageProviders>
   <StorageProvider id="aws-s3" name="Amazon Simple Storage Service (S3)">
      <Active>false</Active>
      <Ready>true</Ready>
      <Default>false</Default>
      <NeedsCredentials>true</NeedsCredentials>
      <CredentialsIdentityName>Access Key ID</CredentialsIdentityName>
      <CredentialsKeyName>Secret Access Key</CredentialsKeyName>
      <HasCredentials>true</HasCredentials>
   </StorageProvider>
</StorageProviders>
```

Listing 4.2: XML-Repräsentation der Ressource `/StorageProviders/Available`. In diesem Fall ist lediglich der Amazon S3-Storage Provider verfügbar.

Die Unterteilung in zwei Ressourcen ist erforderlich, da es, wie bereits erwähnt, möglich sein kann, dass der Default Storage Provider (siehe Abschnitt 4.2.1) nicht verfügbar ist. In diesem Fall soll zumindest seine ID ausgegeben werden können.

In der Ressource `/StorageProviders/Available` stehen zusätzlich Filterfunktionen zur Verfügung:

- `.../Available?type=active`: Gibt Informationen zum aktiven Storage Provider zurück.
- `.../Available?type=default`: Gibt Informationen zum Default Storage Provider zurück (sofern verfügbar).
- `.../Available?type=ready`: Gibt Informationen zu Storage Providern zurück, die einsatzbereit und damit auch verfügbar sind.

Unter der URI `/StorageProviders/Available/<StorageProviderID>` werden die genannten Daten zu einem einzelnen, verfügbaren Storage Provider bereitstellt. Mittels einer POST-Anfrage an diese URI kann der Storage Provider als aktiv gesetzt oder Zugangsdaten aus diesem gelöscht werden. Hierzu muss der Body der Anfrage das Schlüsselwort „activate“ bzw. „unset“ enthalten. Besteht der Body aus einem anderen Inhalt, so wird der HTTP-Statuscode 400 (fehlerhafte Anfrage) zurückgeliefert.

Zur Repräsentation der Storage Provider-Informationen in XML musste ein JAXB-Modell entworfen und implementiert werden. Dieses besteht aus den Klassen `StorageProvidersJaxb` und `StorageProviderJaxb`. Ein `StorageProvidersJaxb`-Objekt enthält eine beliebige Anzahl von `StorageProviderJaxb`-Objekten, die jeweils

die genannten Daten zu einem Storage Provider enthalten. In einer `JaxbFactory`, die statische Methoden bereitstellt, werden die Klassen des JAXB-Modells instanziiert. Hierbei werden die benötigten Informationen aus dem File Service und Credentials Service abgerufen.

4.7.1.2 Credentials

Eine GET-Anfrage an die URI `/Credentials` liefert alle im Credentials Service gespeicherten Zugangsdaten zurück. Zu Zugangsdaten werden die Werte aller Attribute ausgegeben, die im Credentials Model (siehe Abschnitt 4.6) definiert wurden. Zusätzlich wird angegeben, ob Zugangsdaten in ihrem zugehörigen Storage Provider gesetzt sind. Als Datenformat kommt XML zum Einsatz.

Analog dazu liefert `/Credentials/<credentialsID>` die Zugangsdaten mit einer bestimmten ID zurück. Sendet man eine POST-Anfrage mit „set“ bzw. „unset“ im Body an diese URI, so werden die Zugangsdaten im zugehörigen Storage Provider gesetzt bzw. aus dem Storage Provider gelöscht. Falls das Setzen von Zugangsdaten fehlschlägt, da der entsprechende Storage Provider nicht verfügbar ist oder keine Zugangsdaten benötigt, so wird der HTTP-Statuscode 500 (interner Serverfehler) zurückgegeben. Der Code 400 wird zurückgeliefert, falls die Zugangsdaten nicht aus dem Storage Provider gelöscht werden, da dieser keine oder andere Zugangsdaten besitzt.

Mittels einer POST-Anfrage an `/Credentials` können Zugangsdaten gespeichert werden. Der Body der Anfrage muss die zu speichernden Zugangsdaten in XML enthalten, wobei die in Abbildung 4.3 dargestellte Syntax eingehalten werden muss. Weiterhin muss im Header das Attribut „Content-Type“ mit dem Wert „application/xml“ spezifiziert sein. Andernfalls wird der HTTP-Statuscode 415 (nicht unterstützter Medientyp) zurückgegeben. Im Erfolgsfall wird die URI der erzeugten Credentials-Ressource `/Credentials/<credentialsID>` zurückgeliefert. Kommt es dagegen während dem Speichern der Zugangsdaten zu einem Fehler (z. B. unvollständige Zugangsdaten), so wird stattdessen der Code 500 zurückgegeben.

```
<Credentials>
   <StorageProviderID>aws-s3</StorageProviderID>
   <Identity>amazon_access_key</Identity>
   <Key>amazon_secret_key</Key>
   <Description>Optional Description.</Description>
</Credentials>
```

Listing 4.3: XML-Repräsentation von Zugangsdaten.

Das Löschen von Zugangsdaten mit einer bestimmten ID erfolgt durch Senden einer DELETE-Anfrage an `/Credentials/<credentialsID>`. Auch ist das Löschen aller im Credentials Service gespeicherten Zugangsdaten möglich. Hierzu muss eine DELETE-Anfrage an `/Credentials` gesendet werden.

Für die Repräsentation von Zugangsdaten in XML musste ein weiteres JAXB-Modell erstellt werden. Ein `AllCredentialsJaxb`-Objekt repräsentiert eine Menge von Zugangsdaten. Es enthält eine beliebige Anzahl von `CredentialsJaxb`-Objekten, die einzelne Zugangsdaten darstellen. Da die Attribute von Zugangsdaten im Credentials Model (siehe Abschnitt 4.6) bzw. deren Klasse `Credentials` definiert sind, wurde `CredentialsJaxb` von `Credentials` abgeleitet. Zusätzlich musste `CredentialsJaxb` noch um ein Attribut erweitert werden, das angibt, ob die Zugangsdaten im zugehörigen Storage Provider gesetzt sind. Die Instanziierung des JAXB-Modells für Zugangsdaten erfolgt ebenfalls in einer `JaxbFactory`, welche die benötigten Informationen aus dem Credentials Service bezieht.

4.7.1.3 CSARs

Das Speichern einer CSAR erfolgt durch Senden einer POST-Anfrage an `/CSARs`. Der Body der Anfrage muss aus dem Quellpfad der CSAR-Datei bestehen. Wurde eine CSAR erfolgreich gespeichert, so wird die URI `/CSARs/<csarID>` der erzeugten Ressource zurückliefert, welche die gespeicherte CSAR repräsentiert. Falls es während dem Speichern zu einem Fehler kommt, der in den Verantwortungsbereich des Nutzers fällt (z. B. keine gültige CSAR), so wird der HTTP-Statuscode 400 zurückgegeben. Bei anderen Fehlern besteht die Antwort aus dem Code 500. Zum Löschen einer CSAR muss eine DELETE-Anfrage an `/CSARs/<csarID>` gesendet werden. Der Export einer CSAR erfolgt mittels einer GET-Anfrage an `/CSARs/<csarID>/download`. Zurückgeliefert werden Daten vom Medientyp „application/octet-stream“.

Eine gespeicherte CSAR kann über die Container API durchsucht werden (CSAR Browsing API). Die URI `/CSARs/<csarID>/Content` repräsentiert hierzu das Wurzelverzeichnis der CSAR. Eine GET-Anfrage liefert Referenzen (URIs) zu allen untergeordneten Ressourcen, die jeweils Dateien oder Ordner im Wurzelverzeichnis der CSAR repräsentieren. Analog dazu kann z. B. eine Anfrage an `/CSARs/<csarID>/Content/IA` gerichtet werden, um entsprechende Referenzen von Dateien und Ordnern zu erhalten, die sich im Ordner „/IA“ der CSAR befinden. Falls „/IA“ dagegen eine Datei ist, so wird lediglich eine Selbstreferenz zurückgegeben. Unter der URI `/CSARs/<csarID>/Content/IA/download` kann in diesem Fall die Datei heruntergeladen werden (Daten als „application/octet-stream“). Abbildung 4.15 veranschaulicht die Struktur der CSAR Browsing API.

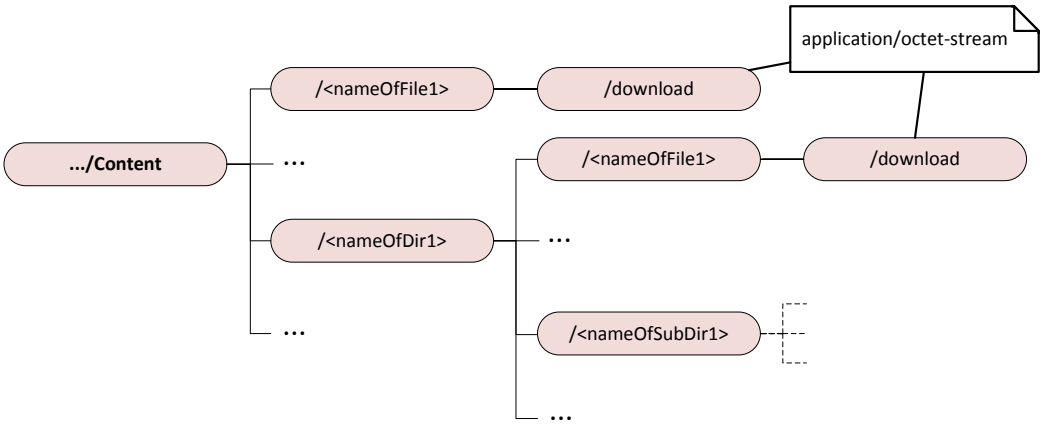

Abbildung 4.15: Struktur der CSAR Browsing API.

Zur Darstellung von Referenzen wird grundsätzlich XLink eingesetzt. Dabei handelt es sich um eine Syntax zur Darstellung von Links in XML. Listing 4.4 veranschaulicht diese Notation.

```
1 <References>
2   <Reference xlink:type="simple" xlink:href="http://localhost:1337/containerapi/
        CSARs/Test.csar/Content/IAs/EC2LinuxService" xlink:title="EC2LinuxService"/>
3   <Reference xlink:type="simple" xlink:href="http://localhost:1337/containerapi/
        CSARs/Test.csar/Content/IAs/EC2VMService" xlink:title="EC2VMService"/>
4   <Reference xlink:type="simple" xlink:href="http://localhost:1337/containerapi/
        CSARs/Test.csar/Content/IAs" xlink:title="Self"/>
5 </References>
```

Listing 4.4: XLink-Repräsentation einer CSAR Browsing API-Ressource. Die Ressource enthält zwei Referenzen zu untergeordneten Ressourcen, die Dateien oder Ordner repräsentieren können und eine Selbstreferenz.

Das Verschieben einer Datei bzw. eines Ordners zum aktiven Storage Provider (siehe Abschnitt 4.2.1) erfolgt ebenfalls über die CSAR Browsing API. Hierzu muss eine POST-Anfrage mit dem Schlüsselwort „move“ an die URI, welche die zu verschiebende Datei bzw. den zu verschiebenden Ordner repräsentiert, gesendet werden. Zum Verschieben einer kompletten CSAR muss selbige POST-Anfrage an `/CSARs/<csarID>/Content` gesendet werden.

Wie bereits erwähnt, muss zum Durchsuchen einer CSAR nun das Artifact Model eingesetzt werden. Dementsprechend mussten Anpassungen an der CSAR Browsing API vorgenommen werden, sodass statt Dateisystemoperationen nun die entsprechenden Methoden des Artifact Model verwendet werden. Falls der Nutzer eine Datei herunterlädt, so erfolgt dies direkt vom entsprechenden Anbieter (z. B. Amazon S3).

Unter `/CSARs/<csarID>/TopologyPicture` wird das Topologie-Bild der CSAR bereitgestellt. Sofern der relative Pfad in der TOSCA Metadatei spezifiziert ist (siehe Abschnitt 2.2.2) und auf eine existierende Datei verweist, werden Daten vom Medientyp „image/*“ zurückgeliefert. Andernfalls besteht die Antwort aus dem HTTP-Statuscode 404 (nicht gefunden). Das Abrufen des Bilds erfolgt nun analog über das Artifact Model.

4.7.2 Weitere Komponenten

Im Rahmen der Umstellung auf das Artifact Model (siehe Abschnitt 4.4) mussten weiterhin Anpassungen an der TOSCA Engine, IA Engine und Plan Engine vorgenommen werden.

Die TOSCA Engine ist u. a. für das Verarbeiten von Definitions-Dokumenten und Auflösen von Importen zuständig. Für den Zugriff auf die entsprechenden Dateien werden Methoden des CSAR Models bzw. deren Klasse `CSARContent` (siehe Abschnitt 4.3) verwendet. Wie bereits angesprochen, lieferten diese bisher `File`-Objekte zurück. Nun werden stattdessen `AbstractFile`-Objekte zurückgegeben, sodass entsprechende Anpassungen vorgenommen werden mussten.

Weiterhin stellt die TOSCA Engine eine Methode bereit, die alle Dateien zurückliefert, die sich an Artefakt-Referenzen eines Artifact Template befinden. Diese Methode hat bisher alle Dateien an den Artefakt-Referenzen über `CSARContent` geholt, Pattern Matching durchgeführt und schließlich die zutreffenden Dateien zurückgeliefert. Nun liefert `CSARContent` nach Übergabe einer Artefakt-Referenz keine Dateien mehr zurück, sondern ein `AbstractArtifact`-Objekt. Dementsprechend musste die Signatur und Implementierung der Methode auf `AbstractArtifact` angepasst werden. Zurückgegeben werden nun eine Menge von `AbstractArtifact`-Objekten, die die Artefakt-Referenzen eines Artifact Template repräsentieren. Weiterhin wurde das Pattern Matching entfernt, da dieses nun vom Artifact Model übernommen wird. Verwendet wird die Methode der TOSCA Engine von der IA Engine, sodass auch in dieser Komponente Anpassungen vorgenommen werden mussten.

`AbstractArtifact`-Objekte werden in der IA Engine bis zum Plug-in weitergereicht. Momentan existieren in der IA Engine Plug-ins zum Deployment von WAR-Dateien und AAR-Dateien. Beide Plug-ins benötigen lediglich die Dateien eines Artefakts und keine Informationen über dessen Verzeichnisstruktur, sodass wir in jedem Plug-in direkt die Methode `getFilesRecursively()` auf allen `AbstractArtifact`-Objekten aufrufen können. Diese Methode liefert rekursiv alle `AbstractFile`-Objekte zurück. Mit der Methode `getFile()` auf einem `AbstractFile`-Objekt kann die Datei schließlich heruntergeladen werden. Um ein unnötiges Herunterladen zu vermeiden, wird

diese Methode erst aufgerufen, nachdem überprüft wurde, ob die Endung der zu deployenden Datei korrekt ist.

Analog zur IA Engine mussten auch in der Plan Engine Anpassungen für `AbstractArtifact` vorgenommen werden. Die Plan Engine kommuniziert im Gegensatz zur IA Engine direkt mit dem CSAR Model bzw. `CSARContent`, um benötigte Dateien zu erhalten.

5 Zusammenfassung und Ausblick

Bisher konnten CSAR-Dateien, die der TOSCA-Laufzeitumgebung OpenTOSCA übergeben werden, ausschließlich auf das lokale Dateisystem gespeichert werden. Damit eine Speicherung in verschiedenen Umgebungen möglich ist, wurde der File Service um ein Plug-in-System erweitert. Zunächst musste hierfür eine Schnittstelle definiert werden. Implementierungen der Schnittstelle bzw. konkrete Plug-ins wurden daraufhin für das lokale Dateisystem sowie den Blobstore-Anbieter Amazon S3 bereitgestellt.

Die Realisierung der Plug-ins erfolgte mittels der Multi-Cloud-Bibliothek jclouds bzw. deren Blobstore API. Da diese die APIs von Blobstore-Anbietern abstrahiert, wurde eine abstrakte Implementierung der Schnittstelle auf Basis der jclouds Blobstore API bereitgestellt. Ein Plug-in, welches das Speichern auf einem Blobstore-Anbieter, der von jclouds unterstützt wird, ermöglichen soll, kann so einfach und mit minimalen Aufwand durch Verwendung der abstrakten Implementierung realisiert werden. Die angesprochenen Plug-ins für Amazon S3 und das lokale Dateisystem (Dateisystem-basierter Blobstore) wurden auf diese Weise erstellt.

Eine gespeicherte CSAR kann mit dem File Service in eine andere Umgebung, für die ein entsprechendes Plug-in existiert, verschoben werden. Auch ist es möglich, lediglich eine Datei oder ein Ordner einer CSAR zu verschieben. Folglich kann eine CSAR also auf mehrere Umgebungen verteilt werden. Weiterhin wurde eine Export-Funktion realisiert, mit der eine gespeicherte CSAR wieder als CSAR-Datei abgerufen werden kann.

Das Verschieben zwischen verschiedenen Benutzerkonten des selben Anbieters ist momentan nicht möglich. In den Metadaten zu einer Datei wird lediglich die ID des Plug-ins hinterlegt, mit dem die Datei gespeichert wurde. Damit jedoch ein Verschieben zwischen verschiedenen Benutzerkonten möglich wird, müsste zusätzlich auch die Identität der entsprechenden Zugangsdaten gespeichert werden. Da die Erweiterung der Metadaten mit größeren Anpassungen (insbesondere im File Service und Credentials Service) verbunden ist, konnte dies aus zeitlichen Gründen im Rahmen dieser Arbeit nicht mehr erledigt werden.

Falls es während einer Operation zu einem Fehler kommt, so wird diese aktuell mit einer entsprechenden Exception sofort abgebrochen. Wiederholungen nach z. B.

einer fehlgeschlagenen Speicheroperation werden vom File Service selbst bisher nicht durchgeführt. jclouds allerdings sieht mehrere Wiederholungsversuche vor, die auch ausgeführt werden. Des Weiteren ist weitestgehend keine Logik vorgesehen, die bereits durchgeführte Änderungen nach einem Fehler wieder zurücknimmt bzw. zu zurücknehmen versucht (Undo-Operationen oder transaktionale Konzepte). Eine Ausnahme bilden lediglich Datenbankoperationen, die mittels Eclipse Link (JPA-Implementierung) innerhalb von Transaktionen ausgeführt werden.

Zur Speicherung und Verwaltung von Zugangsdaten für Plug-ins wurde der Credentials Service entwickelt. Gespeicherte Zugangsdaten können über diese Komponente direkt im entsprechenden Plug-in gesetzt werden. Momentan bestehen Zugangsdaten aus Identität und Schlüssel. Für alle Anbieter, die von jclouds unterstützt werden, reichen diese Informationen aus. Falls jedoch Plug-ins entwickelt werden sollten, die zusätzliche bzw. andere Parameter zur Authentifizierung benötigen, müsste das Datenmodell der Zugangsdaten entsprechend erweitert werden. Grundsätzlich wäre es denkbar, dass der Credentials Service zukünftig auch von weiteren Komponenten zur Verwaltung von Zugangsdaten eingesetzt wird.

Das Artifact Model wurde für den Zugriff auf Artefakt-Referenzen entwickelt. Momentan werden relative Referenzen unterstützt, die gemäß TOSCA-Spezifikation auf eine Datei oder Ordner in einer CSAR verweisen. Das Durchsuchen erfolgt in diesem Fall auf Basis der lokal gespeicherten Metadaten der CSAR. Zum Herunterladen einer Datei kommt das entsprechende Plug-in zum Einsatz. Da beim Entwurf der Komponente auf Erweiterbarkeit geachtet wurde, kann eine Unterstützung für weitere Arten von Artefakt-Referenzen leicht hinzugefügt werden. Das Artifact Model ersetzt das bisherige Datenmodell, mit dem lediglich die Dateien an einer Artefakt-Referenz zurückgegeben werden konnten. Die Verzeichnisstruktur ist folglich verloren gegangen.

Über die Container API bzw. REST API von OpenTOSCA kann eine CSAR durchsucht und Dateien der CSAR heruntergeladen werden. Bisher führte die Container API hierzu selbstständig Dateisystemoperationen aus. Eine CSAR muss jetzt jedoch nicht mehr lokal gespeichert sein, sodass dieser Ansatz nicht mehr möglich ist. Zum Durchsuchen der gesamten CSAR kommt nun ebenfalls das Artifact Model zum Einsatz.

Im Rahmen der Integration wurden u. a. die neuen Funktionalitäten des File Service und Credentials Service über die Container API bereitgestellt. Die GUI von OpenTOSCA wurde noch nicht erweitert. Dies müsste zukünftig noch erledigt werden.

Literaturverzeichnis

[Ama] Amazon Web Services, Inc. Amazon S3 – Preise. URL http://aws.amazon.com/de/s3/pricing. Abgerufen am 2013-04-19.

[Apaa] Apache Software Foundation. Apache Deltacloud Website. URL http://deltacloud.apache.org. Abgerufen am 2013-05-30.

[Apab] Apache Software Foundation. Apache Libcloud Website. URL http://libcloud.apache.org. Abgerufen am 2013-05-30.

[Bun] Bundesamt für Sicherheit in der Informationstechnik. Cloud Computing Grundlagen. URL https://www.bsi.bund.de/DE/Themen/CloudComputing/Grundlagen/Grundlagen_node.html. Abgerufen am 2013-04-17.

[CLO] CLOUDCYCLE. VALESCA - Visual Editor for TOSCA. URL http://www.cloudcycle.org/valesca. Abgerufen am 2013-04-26.

[enS] enStratus Networks, Inc. Dasein Cloud API Website. URL http://www.dasein.org. Abgerufen am 2013-05-30.

[Hig11] R. Hightower. Adrian Cole Announces JClouds 1.0 Release, 2011. URL http://www.infoq.com/news/2011/07/jclouds_release_1_0. Abgerufen am 2013-04-22.

[Hor12a] T. Horn. OSGi: Dynamisches Komponentensystem für Java, 2012. URL http://www.torsten-horn.de/techdocs/java-osgi.htm. Abgerufen am 2013-04-28.

[Hor12b] T. Horn. RESTful Web Services mit JAX-RS und Jersey, 2012. URL http://www.torsten-horn.de/techdocs/jee-rest.htm. Abgerufen am 2013-05-17.

[Hor13] T. Horn. JPA (Java Persistence API), 2013. URL http://www.torsten-horn.de/techdocs/java-jpa.htm. Abgerufen am 2013-06-07.

[jcl] jclouds, Inc. jclouds Website. URL http://www.jclouds.org. Abgerufen am 2013-04-25.

[Jen] J. Jenkov. Java NIO vs. IO. URL http://tutorials.jenkov.com/java-nio/nio-vs-io.html. Abgerufen am 2013-05-12.

[Kav] D. Kavanagh. typica Website. URL http://code.google.com/p/typica. Abgerufen am 2013-05-30.

[MÏ2] M. Müller. *Sichere Nutzung von Cloud-Storage in Datenbanken.* Diplomarbeit, Technische Universität Dresden, 2012. URL http://www.rn.inf.tu-dresden.de/uploads/Studentische_Arbeiten/Diplomarbeit_M%C3%BCller_Mario.pdf. Abgerufen am 2013-04-22.

[MG11] P. M. Mell, T. Grance. SP 800-145. The NIST Definition of Cloud Computing. Technischer Bericht, National Institute of Standards & Technology, Gaithersburg, MD, United States, 2011. URL http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf. Abgerufen am 2013-04-21.

[Mur] J. Murty. JetS3t Website. URL http://jets3t.s3.amazonaws.com. Abgerufen am 2013-05-30.

[Rod08] A. Rodriguez. RESTful Web services: The basics, 2008. URL http://www.ibm.com/developerworks/webservices/library/ws-restful. Abgerufen am 2013-05-18.

[sal] salesforce.com Germany. Was ist Cloud Computing? URL http://www.salesforce.com/de/cloudcomputing. Abgerufen am 2013-04-18.

[SC09] C. Schmidt-Casdorff. OSGi: Anwendungsszenarien, Auswahlkriterien und Ausblick, 2009. URL http://www.iks-gmbh.com/files/pdf/03_OSGi_Anwendungsszenarien_Ausblick. Abgerufen am 2013-06-01.

[Sei10] R. Seiger. *Entwurf eines mehrseitig sicheren Cloud Storage Dienstes.* Studienarbeit, Technische Universität Dresden, 2010. URL http://www.rn.inf.tu-dresden.de/uploads/Studentische_Arbeiten/Belegarbeit_Seiger_Ronny.pdf. Abgerufen am 2013-04-22.

[TOS13] TOSCA Technical Committee. Topology and Orchestration Specification for Cloud Applications (TOSCA) – Committee Specification 01. Technischer Bericht, OASIS, 2013. URL http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-cs01.pdf. Abgerufen am 2013-04-26.

[Vog13] L. Vogel. OSGi Modularity – Tutorial, 2013. URL http://www.vogella.com/articles/OSGi/article.html. Abgerufen am 2013-04-28.

[WHKL09] G. Wütherich, N. Hartmann, B. Kolb, M. Lübken. Einführung in die OSGi Service Platform, 2009. URL http://www.it-agile.de/fileadmin/docs/Vortragsfolien/OSGi-Powerworkshop-JAX2009.pdf. Abgerufen am 2013-04-28.

[Wil12] K. Wilhelmi. *CloudRaid – Ein sicherer Raid-Manager für freie Cloud Storages*. Bachelorarbeit, Technische Universität Darmstadt, 2012. URL http://www.informatik.tu-darmstadt.de/fileadmin/user_upload/Group_SIT/Publications/Thesis/CloudRaid-final.pdf. Abgerufen am 2013-04-21.

A Anhang

A.1 Verwendete Software

Folgende Software wurde für die Erstellung dieses LaTeX-Dokuments und der darin enthaltenen Diagramme eingesetzt:

- TeXstudio 2.5.2[44] (LaTeX-Editor)
- JabRef 2.10b[45] (Literaturverwaltung für BibTeX)
- LibreOffice 3.6.4.3[46] mit den FMC OpenOffice Templates 1.0[47] zur Erstellung der FMC-Diagramme
- Microsoft Visio 2013[48] zur Erstellung aller weiteren Diagramme

Im Rahmen der Entwicklung wurden folgende Software bzw. Werkzeuge verwendet:

- Eclipse IDE for Java EE Developers Juno (4.2) SR2[49]
- Subclipse 1.8.20[50] (Subversion-Client für Eclipse)
- Apache Maven 3.0.5[51] (Build-Management-Werkzeug) zum Herunterladen der jclouds Bundles und deren Abhängigkeiten

[44] Website von TeXstudio: http://texstudio.sourceforge.net
[45] Website von JabRef: http://jabref.sourceforge.net
[46] Website von LibreOffice: http://de.libreoffice.org
[47] Download der FMC Stencils: http://www.fmc-modeling.org/fmc_stencils
[48] Website von Microsoft Visio: http://office.microsoft.com/de-de/visio
[49] Website von Eclipse: http://www.eclipse.org
[50] Website von Subclipse: http://subclipse.tigris.org
[51] Website von Apache Maven: http://maven.apache.org

A.2 Inhalt der DVD

Abbildung A.1 zeigt die Struktur der mitgelieferten DVD, die im Wesentlichen alle Dokumente und praktischen Ergebnisse enthält, die im Rahmen dieser Bachelorarbeit entstanden sind. Hierzu gehört auch eine lauffähige Version von OpenTOSCA mit Dokumentation, erforderlicher Software („Requirements“) und CSAR-Dateien zum Testen.

```
/
├── CSARs ..... CSAR-Dateien
├── Implementation ..... Quellcode (Eclipse-Projekte)
├── Implementation-Test ..... Quellcode der Testfälle (Eclipse-Projekte)
├── OpenTOSCA ..... OpenTOSCA Container und JSP UI
├── OpenTOSCA-Docs ..... OpenTOSCA Dokumentation
│   ├── Credentials_internal.pdf ..... Erforderliche Zugangsdaten
│   ├── Deploying_a_CSAR.pdf ..... How-To: CSAR-Deployment mit der JSP UI
│   ├── Installation_Guide.pdf ..... Installationsanleitung
│   └── Uninstalling_Guide.pdf ..... Deinstallationsanleitung
├── OpenTOSCA-Requirements ..... Erforderliche Software für OpenTOSCA
├── Generischer_File_Service_fuer_OpenTOSCA.pdf ..... Diese Ausarbeitung
├── abstract.txt ..... Kurzfassung der Ausarbeitung
└── readme.txt
```

Abbildung A.1: Struktur der mitgelieferten DVD.

Im Ordner `Implementation` befinden sich die Eclipse-Projekte von allen Komponenten von OpenTOSCA, die im Rahmen dieser Arbeit entstanden oder wesentlich weiterentwickelt worden sind. Dazu gehört auch die Container API, da diese während der Integration erheblich erweitert und angepasst wurde.

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