Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3602

Einheitliches Auffinden, Erfassen und Ablegen von DevOps-Artefakten

Thorsten Jonas

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. Frank Leymann
Betreuer/in:	Dipl.-Inf. Johannes Wettinger
Beginn am:	11. Dezember 2013
Beendet am:	12. Juni 2014
CR-Nummer:	C.2.4, D.2.11, K.6

Kurzfassung

In der heutigen Zeit sind Unternehmen darauf angewiesen, einerseits eine Vielzahl von Arbeitscomputern konfigurieren und andererseits ihre Server-Infrastruktur so skalieren zu können, dass durchgehend genügend Ressourcen zur Verfügung stehen. Gerade Web-Unternehmen, die Dienste im Internet anbieten, müssen ihre Server-Struktur so anpassen können, dass ihre Dienste auch bei stark steigenden Benutzerzahlen zur Verfügung stehen und dabei trotzdem schnell und stabil sind. Beide Aufgaben erforderten bisher Administratoren, die meistens mit Hilfe von Skripten die Server und Arbeitscomputer passend konfigurierten. Je nach Größe des Unternehmens werden dadurch hohe Kosten verursacht. Abhilfe schaffen sogenannte Konfigurationsmanagement-Tools, die im Zuge der DevOps-Bewegung, bei der Software-Entwicklung und Software-Betrieb näher zusammen rückten, entstanden sind.
Mit diesen Tools lassen sich, einmal installiert, die erforderlichen Computer gemeinsam konfigurieren und auch neue Server zur bestehenden Infrastruktur hinzufügen, um das Gesamtsystem zu entlasten oder entfernen, um die Kosten zu reduzieren. Oft bieten diese Tools auch grafische Benutzerschnittstellen an, welche das Konfigurationsmanagement weiter erleichtern.
Die Software-Artefakte, mit denen ein Computern eingerichtet werden kann, hängt von dem jeweiligen Konfigurationsmanagement-Tool ab. Diese Artefakte sind in Online-Repositories gespeichert und frei zugänglich. Jedoch gibt es keine Möglichkeit, Artefakte von verschiedenen Tools miteinander zu kombinieren oder zu vergleichen.
Die Arbeit beschäftigt sich daher mit dem Auslesen und Speichern von Artefakten aus diesen Repositories der Konfigurationsmanagement-Tools Chef, Juju und Puppet. Ziel ist es, alle Artefakte in einem einheitlichen Format zu speichern und zum Beispiel über das TOSCA-CSAR-Format, ein Standard im Bereich des Konfigurationsmanagements, exportieren zu können.
Um dies zu erreichen, müssen sogenannte Crawler implementiert werden, welche die Online-Repositories der einzelnen Tools durchforsten und deren Inhalte in einer Datenbank speichern. Dazu wird über ein GUI-Frontend und eine API die Möglichkeit gegeben, die Artefakte durchsuchen, speichern und exportieren zu können.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

1 Einleitung

Schon 1970 kritisierte Dr. Winston Royce in einer wissenschaftlichen Publikation [Roy70] die gängige klassische Softwareentwicklung, wie sie zum Beispiel durch das Wasserfallmodell beschrieben wird. In diesem Modell werden Software-Projekte anhand eines Treppenverfahrens entwickelt, d.h. einzelne Aufgaben werden schrittweise nacheinander abgearbeitet. Dr. Royce kritisierte die wenig vorhandene Flexibilität dieses Verfahrens und das Fehlen der Kommunikation zwischen den Abteilungen, welche die verschiedenen Arbeitsschritte durchführten.
Diese Art der Softwareentwicklung wird seit dem Ende des 20. Jahrhunderts immer mehr von den agilen Methoden zur Softwareentwicklung abgelöst. Deren Ziel ist es, [BBB+01] durch die Konzentration auf eine gut funktionierende Arbeitsweise und die daraus resultierende, möglichst schnell entwickelte, aber gut funktionierende Software, die bestehenden Hürden bei der Entwicklung von Software zu minimieren und so die Zeit bis zum fertigen Produkt zu verkürzen. Eine Software sollte in kurzen Zyklen, in enger Absprache mit dem Kunden entwickelt werden.[Net12]
Sowohl bei klassischer als auch bei agiler Softwareentwicklung sind die reinen Entwickler in verschiedenen Teams oder Abteilungen untergebracht, wie die Systemadministratoren, die meist für das Bereitstellen und den Betrieb der Software zuständig sind. Dies führt dazu, dass die Softwareentwickler oft nicht direkt mit den Problemen des laufenden Betriebs der Software zu tun haben und ihren Fokus ganz auf die schnelle Entwicklung neuer, vom Auftraggeber gewünschter, Funktionen legen. Der IT-Betrieb hingegen ist vor allem an Stabilität und Sicherheit der aktuellen Software interessiert. Dies führt bei Fehlern nicht selten zum sogenannten „Blame Game“ [Pes12], bei dem sich die verantwortlichen Seiten gegenseitig die Schuld zuweisen. Die Umstellung der IT-Entwicklung auf agile Methoden hat diesen Interessenkonflikt noch verstärkt, da die Entwickler immer kürzere Zeiten bis zur Fertigstellung neuer Software-Releases benötigen, aber sich die Zeit zum Testen und Veröffentlichen der Software vom IT-Betrieb kaum verändert hat. Die IT-Betriebsabteilung wurde so zum Flaschenhals in der Softwareentwicklung.
Bei Treffen von Beschäftigten aus dem IT-Betrieb, die mit dieser Arbeitssituation nicht zufrieden waren [MWGK] und anschließend Softwareentwickler zu den Treffen einluden, entstand schließlich die DevOps-Bewegung, um beide, an der Softwareentwicklung beteiligten Abteilungen, näher zusammenzubringen.

1.1 Die DevOps-Bewegung

Der Begriff DevOps setzt sich aus Developer und Operations zusammen, steht für das Zusammenspiel zwischen Softwareentwicklern und den Betreibern der Software und wurde maßgeblich von Patrick

Debois geprägt, als er 2009 die ersten DevOpsDays[1] in Gent (Belgien) veranstaltete. Die DevOps Bewegung wird als Weiterentwicklung der agilen Softwareentwicklung betrachtet, da in vielen Unternehmen damals schon die Umstellung auf agile Methoden begonnen und die Verbreitung zur Zeit dieser Treffen gerade am wachsen war. Die grundlegenden Ziele von DevOps sind Zusammenarbeit, Automatisierung und Prozesse.

Zusammenarbeit

Bei der Zusammenarbeit geht es um die zwischenmenschlichen Aspekte in der IT-Arbeitswelt. Dazu zählt der Respekt vor der Arbeit des anderen Teams und die bessere Kommunikation untereinander. Die Vorurteile sollen abgebaut und die Zusammenarbeit schon während der Entwicklung intensiviert werden. Eine dadurch verbesserte Kommunikation führt zu einer besseren Unternehmenskultur, was sich schließlich in besser und schneller entwickelter Software auszahlt.

Automatisierung

Die immer schneller Software produzierende IT-Entwicklung benötigt beim Veröffentlichen der Software auch oft Anpassungen an die Software-Umgebung, was zu Änderungen im Veröffentlichungsprozess führt, die der IT-Betrieb umsetzen muss. Anstatt das Veröffentlichen manuell mit immer neuen Befehlen umzusetzen, wie es immer noch in vielen Unternehmen der Fall ist [HF10], setzen DevOps auf Automatisierung, d.h. die versionierte Bereitstellung der benötigten Befehle in Skripten. Das vermeidet nicht nur Fehler, die durch manuelle Ausführung entstehen, sondern erhöht auch noch die Geschwindigkeit und kann von jedem ausgeführt werden. Außerdem können die Skripte als „ausführbare Dokumentationen“ verwendet werden, da oft die einzelnen Arbeitsschritte eines System Administrators nicht, oder nur schlecht, dokumentiert werden. Bei der Automatisierung der Veröffentlichungsmechanismen wird man durch **Tools zum Konfigurationsmanagement** unterstützt, mit denen die gesamte Infrastruktur durch Skripte aufgesetzt und verändert werden kann. Da die Skripte aus Code-Zeilen einer Programmiersprache bestehen, spricht man von „Infrastructure as code“.[inf12] Ein weiterer Vorteil ist, dass die gleichen Skripte im Betrieb und in der Entwicklung genutzt werden können und so die Kompatibilität der Software sichergestellt ist. Die Entwickler und Tester einer Software haben dadurch dieselbe Umgebung wie später die Produktivumgebung.

Prozesse

Die DevOps-Methoden sollten mittels Prozessen in das Unternehmen eingeführt werden. Diese Prozesse müssen individuell an die vorhandene Struktur des Unternehmens angepasst und ausgeführt werden. Beispielsweise kann im Veröffentlichungsprozess die Betriebsabteilung mehr Verantwortung an die Entwicklungsabteilung abgeben. Der sogenannte „Shift left“-Prozess [Mic13], d.h. Aufgaben des Betriebs an die Entwickler zu übertragen, hilft dabei, der Entwicklungsabteilung mehr Einblick in

[1] http://www.devopsdays.org/events/2009-ghent

die relevanten Kriterien zur Veröffentlichung zu geben und verstärkt somit die Integration beider Abteilungen.

1.2 Problembeschreibung

Im nachfolgenden Kapitel werden die grundlegenden Funktionen dreier Konfigurationsmanagement-Tools erläutert. Diese Tools helfen Administratoren bei der Installation und Konfiguration von Software-Artefakten sowohl auf den lokalen Arbeitsrechnern der Mitarbeiter eines Unternehmens als auch auf den Servern, auf denen die Leistungen eines Unternehmens angeboten und bei Web-Unternehmen auch ausgeführt werden.
Die Software-Artefakte beschreiben zum Beispiel Dienste wie Datenbanken oder Server, die auf den Servern ausgeführt werden können. Die zu installierenden Artefakte liegen dabei in Online Repositories auf Servern des jeweiligen Tools und können daraus, meist auch über eine grafische Benutzerschnittstelle, passend zusammengestellt und anschließend parallel auf vielen Rechnern installiert und konfiguriert werden.
Die in den Online Repositories gespeicherten Artefakte sind bei jedem Tool unterschiedlich. Auch in der Art und Weise, wie die Artefakte in den Repositories gespeichert und aufgelistet sind, unterscheiden sich die Tools untereinander. Ein Vergleich der zu installierenden Artefakte ist daher kaum möglich. Außerdem kann man, wenn man sich für ein Konfigurationsmanagement-Tool entschieden hat, keine Artefakte aus anderen Tools direkt in das System integrieren.
Hier setzt diese Arbeit an, in dem ein einheitlicher „**Artifact Store**“ geschaffen werden soll, der die Artefakte aus den Repositories von drei Konfigurationsmanagement-Tools enthält. Die Artefakte sollen dabei von **Crawlern** aus den Repositories ausgelesen, zwischengespeichert und in den Artifact Store geladen werden.
Außerdem soll es möglich sein, die Artefakte in das Standardformat für Konfigurationsmanagement, TOSCA-CSAR, zu exportieren, um die Artefakte mit TOSCA laden und weiterverarbeiten zu können.
Um eine Übersicht über alle gespeicherten Artefakte zu erhalten, soll es eine graphische Benutzerschnittstelle geben. Mit ihr sollen die Artefakte nach vorgegebenen und eigenen Kriterien sortierbar sein, außerdem soll darüber auch der Export und das Herunterladen der Artefakte erfolgen.

Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Grundlagen: Hier werden werden die Grundlagen dieser Arbeit und die aktuelle Entwicklung beschrieben.

Kapitel 3 – Anforderungen: In diesem Kapitel werden die Anforderungen an die Crawler, den Artifact Store und das User Interface definiert.

Kapitel 4 – Architektur und Design: Die grundlegende Struktur für die Architektur der Software-Komponenten wird hier festgelegt.

Kapitel 5 – Implementierung: Dieses Kapitel skizziert die tatsächliche Umsetzung mit Code-Beispielen.

Kapitel 6 – Evaluation: Die Ergebnisse der Arbeit werden in einem Überblick dargestellt.

Kapitel 7 – Zusammenfassung und Ausblick: Das letzte Kapitel fasst die Leistung der Arbeit zusammen und stellt mögliche Weiterentwicklungen in diesem Bereich vor.

2 Grundlagen

2.1 Konfigurationsmanagement-Tools

Zu einer Umsetzung der in Kapitel 1.1 genannten DevOps-Methoden gehört auch, die Infrastruktur durch wiederverwendbare Skripte bereitzustellen, um die Automatisierung der Veröffentlichung voranzutreiben. Gerade bei Cloud-basierten Anwendungen kann dies schnell zur großen Belastung für die Entwickler werden, da man Software-Umgebungen, die online ausgeführt werden, oft nicht mit einem Skript installieren kann, oder erst ein eigenes Programm entwickelt werden muss, das die Installation übernimmt. Möchte man diese automatisierten Veröffentlichungsprozesse für viele unterschiedliche Services im Web nutzen, kostet das viel Entwickler-Zeit. In der Praxis führt ein Entwickler oder Administrator häufig diese Schritte noch von Hand aus. Selbst wenn die Administratoren solche Skripte schon geschrieben haben, kommt es in der heutigen Zeit oft dazu, dass die Anforderungen an die Services skalierbar, z.B. nach Nutzerzahlen, sein müssen. Möchte man nun sein System an eine steigende Nutzerzahl anpassen, müssten die Administratoren alle Skripte anpassen, was viel Arbeit verursacht und etliche Fehlerquellen entstehen lässt.
Um diese Automatisierung von Deployments in Cloud-basierten Anwendungen zu vereinfachen, wurden daher Tools zum Konfigurationsmanagement entwickelt, die schon fertige Skripte oder Container liefern, mit denen Software-Services in der Cloud installiert werden können. Diese Tools benutzen dazu meist domänenspezifische Sprachen (DSLs) mit deren Hilfe Entwickler plattformunabhängig die Konfiguration einer Softwareumgebung beschreiben können. Die Sprachen sind dabei so abstrakt gehalten, dass sie schnell verstanden und einfach benutzt werden können [GHS10]. Die Tools übernehmen die Übersetzung der Vorgaben in eine Serverkonfiguration oder installieren die gewünschten Services automatisch. So werden Serverausfälle durch Fehler beim Konfigurieren und Veröffentlichen von Software minimiert.

2.1.1 Puppet

Puppet ist ein Open Source Framework, das entwickelt wurde, um die Infrastruktur in Unix- und Windows-basierten Softwareumgebungen automatisch zu gestalten. Es wurde von Puppet Labs, das 2005 von Luke Kanies gegründet wurde, mit Ruby entwickelt [KVHK+13]. Puppet hilft Administratoren, die sich wiederholende Arbeit zu automatisieren. Es setzt auf eine eigene deklarative, domänenspezifische Sprache, mit der Adminisitratoren den Sollzustand der Softwareumgebung beschreiben können. Nach dem Evaluieren der so definierten Konfiguration installiert Puppet alle nötigen Pakete und startet die ausgewählten Dienste [Lab]. Diese Konfiguration, die in sogenannten „Manifeste“ gespeichert wird, kann man auf beliebig vielen Systemen, auch mit verschiedenen Betriebssystemen replizieren. Puppet passt die Installation der Dienste automatisch entsprechend an.

Das Veröffentlichen der Infrastruktur mit Puppet folgt immer der gleichen Reihenfolge:

1. Manifeste definieren
2. Infrastruktur emulieren und Konfiguration anpassen
3. Installation durchführen
4. Unterschiede zwischen Ist- und Sollzustand aufzeigen

Beim Definieren der Manifeste kann man auf ein Verzeichnis mit über 1000 vorgefertigten, wiederverwendbaren Module zurückgreifen, das sogenannte Puppet Forge.

Client-Server-Modus

Dieser Modus ist dann sinnvoll, wenn mehrere Server verwaltet werden müssen. Es werden die ausführbaren Dienste *puppetmasterd* als Server und *puppetd* als Client erzeugt. Nach der Installation der Konfiguration sendet der Puppet-Dienst auf dem Client automatisch Daten, die er zuvor über den Client-Rechner gesammelt hat, an den Puppet-Master. Mit deren Hilfe wird vom Server ein Katalog erzeugt und zurück an den Client geschickt. In dem Katalog stehen die Änderungen, die erforderlich sind, um den Sollzustand auf dem Client herzustellen. Der Client passt daraufhin seine Konfiguration so an, dass der Ist-Zustand erreicht wird.
Aus diesem Vorgehen resultieren folgende Vorteile:

- Alle Konfigurationen sind zentral gespeichert und anpassbar
- Die Clients bekommen nur die nötige, vorkompilierte Software und keinen Sourcecode
- Zertifikate und Verbindungen sichern die Kommunikation
- Komplizierte Konfigurationen können mit einen Dashboard zusammengestellt werden

Ein Nachteil im Client-Server-Modus ist, dass der Server ausreichend CPU-Leistung zum Kompilieren der Kataloge benötigt. Diese hängt davon ab, wie viele Clients es gibt und wie häufig die Nachfragen vom Client kommen.

Standalone-Modus

Obwohl der im vorigen Abschnitt beschriebene Fall am häufigsten eintritt, muss Puppet nicht unbedingt im Client-Server-Modus genutzt werden. Mit der ausführbaren *puppet*-Datei kann man Konfigurationskataloge sowohl kompilieren als auch lokal installieren. Man kann Puppet also auch zum „application bootstrapping“, d.h. Software die automatisch weitere, noch komplexere Software installiert, benutzen, um schnell eine ganze Reihe an Diensten und Software auf einem Computer zu installieren.

Modus für hochskalierbare Systeme

Der Geschwindigkeits-Nachteil, der im Client-Server-Modus durch den Flaschenhals beim Server durch das Vorkompilieren der Konfigurationen entsteht, kann durch die Verwendung der Standalone Variante vermieden werden. Um trotzdem viele Clients mit der gleichen Konfiguration zu versehen, wird diese durch ein zentrales Lager mit allen Maschinen synchronisiert. Jeder Client kompiliert dabei seinen Software-Katalog lokal. Der gewonnene Geschwindigkeitsvorteil geht hier mit einem Verlust an Sicherheit und Konfigurationsmöglichkeit einher.

Die Puppet-DSL

Der Kern der Puppet-DSL liegt im Deklarieren von **Ressourcen** [pup]. Es gibt viele Arten von Ressourcen, die deklariert werden können, z.B. Dateien, Services und Packages. Durch Attribute wie z.B. Name und Pfad werden diese Ressourcen genau beschrieben. Mehrere Ressourcen können zu **Klassen** zusammengefasst werden. Mit ihnen können ganze Anwendungen beschrieben werden, mit allen dazugehörenden Konfigurationsdateien, Diensten und Aufgaben. Kleinere Klassen können zu größeren Klassen kombiniert werden, um z.B. ganze Datenbank-Server zu beschreiben. Als drittes können mit der Puppet-DSL **Knoten** („Nodes“) definiert werden. Ihnen werden Klassen zugeordnet. Code-Beispiel für einen Knoten:

```
# /etc/puppetlabs/puppet/manifests/site.pp
      node 'www1.example.com' {
              include common
              include apache
              include squid
       }
```

Die Manifeste, werden mit der Endung .pp gespeichert.

Code-Beispiel für das Einbinden eines Apache2 Dienstes in einem Puppet-Manifest

```
      class myapache{
            package "apache2"
            service "apache2":
            ensure=>running,
            require=>Package["apache2"]
      }
```

Mit dem `puppet-module build`-Befehl kann aus den Manifesten und Dateien ein fertiges Puppet-Paket generiert werden. Dabei wird auch die metadata.json Datei erstellt, die alle wichtigen Informationen über das Puppet Paket bereit hält.

2.1.2 Chef

Das Open Source Framework Chef der Firma Chef (ehemals Opscode) orientiert sich vom Aufbau an Puppet und kann ebenfalls Software-Kataloge auf Server übertragen und installieren. Chef wurde ursprünglich in Ruby geschrieben, aber ab Version 11 wurde zur Programmiersprache Erlang gewechselt, was zusammen mit dem Datenbank-Wechsel von couchDB zu PostgresSQL eine bessere

Skalierbarkeit bei einer großen Anzahl an Servern ermöglichen soll [cheb].
Die sogenannten Recipes für die Software-Konfiguration werden in einer Kombination aus Ruby und einer domänenspezifischen Sprache verfasst. Im Vergleich zu Puppet, bei dem die Programme nach ihren Abhängigkeiten sortiert ausgeführt werden, wird hier der Code am Ende einfach hintereinander ausgeführt, was bei nicht berücksichtigten Abhängigkeiten der Software untereinander zu Fehlern führen kann.
Die Chef-Architektur besteht aus drei speziellen Komponenten: Dem **Chef-Server**, mindestens einem **Knoten** („Node“) und der **Workstation**. Eine extra Rolle spielen die „Cookbooks“, die die Recipes mit Attributen und Konfigurationen enthalten. Diese werden von der Workstation erzeugt, an den Server übergeben und vom Knoten abgerufen. Ein Knoten kann eine physikalische oder virtuelle Maschine sein, auf der ein Chef-Client läuft, der die Konfiguration übernimmt. Chef kann vier verschiedene Arten von Knoten [chea] verwalten:

- Physikalischer Knoten
  der ein beliebigen Gerät, das im Netzwerk aktiv ist, sein kann und einen Chef-Client installiert hat.
- Virtueller Knoten
  verhält sich wie der physikalische Knoten, existiert aber nur als Software Implementierung.
- Cloud-basierter Knoten
  ist ein Knoten, der in einem Cloud-basierten Dienst, wie Amazon Virtual Private Cloud, OpenStack, Rackspace, Google Compute Engine, Linode oder Windows Azure gehostet ist. Chef bietet zu diesen Diensten die passenden Pugins an.
- Netzwerk-Knoten
  ist ein Netzwerkgerät, wie Switch oder Router.

Der Chef-Client, der auf den Knoten läuft, verhält sich ähnlich wie der Puppet-Client und ist für die Konfiguration des Knotens zuständig. Dafür

- registriert er den Knoten beim Chef-Server
- erzeugt er ein Knoten Objekt
- synchronisiert er die Cookbooks
- kompiliert er die in den Recipes der Cookbooks enthaltenen Ressourcen
- konfiguriert er den Knoten und löst automatisiert Fehler auf

Die Abbildung 2.1 zeigt die Chef-Architektur im Überblick. Der Chef-Client benötigt das **ohai**-Tool, um die Eigenschaften des Knotens zu erkennen[oha]. Dieses Tool wird automatisch mitinstalliert, kann aber auch als Standalone-Werkzeug verwendet werden. Das Tool analysiert Daten wie die Prozessor- und Netzwerkauslastung des Knotens und stellt sicher, dass diese Daten nach der Installation der Chef Cookbooks wieder hergestellt wurden.
Die **Workstation** ist ein spezieller Knoten und stellt die Maschine dar, mit der die Administratoren mit den Knoten interagieren[Ewa14]. Hier werden eigene Cookbooks und Recipes erstellt. Auf der Workstation läuft das Chef-Administrator Tool **knife**, das die Interaktion der Workstation mit dem Chef-Server übernimmt. Außerdem enthält die Workstation eine Kopie des **chef-repo**, dem Ort, an

dem alle Cookbooks gespeichert sind. Das knife-Tool ist für die Synchronisation der Cookbooks im chef-repo zuständig und lädt die entsprechenden Cookbooks auf den Chef-Server, damit sie der Chef-Client dort finden und bei Bedarf aktualisieren kann.
Der **Chef-Server** ist der Verteiler für die Konfigurationsdaten der Knoten. Auf ihm werden die aktuell benutzen Cookbooks, die Regeln („policy“) für die einzelnen Knoten und die Metadaten („node object“) der Knoten gespeichert. Der Chef-Client fragt beim Server nach den Konfigurationsdetails und vollbringt die Arbeit auf dem Knoten selbst, wodurch auch Chef gut skalierbar wird. Zu den auf dem Server gespeicherten Metadaten über die Knoten gehört die **run-list**. Dort ist die Reihenfolge festgelegt, nach der ein Knoten die Cookbooks und Recipes ausführen soll. Diese run-list wird mit knife generiert.

```
run_list "recipe[apache2]",
         "recipe[apache2::mod_ssl]",
         "role[monitor]"
      }
```

Die run-list startet zuerst das Apache2 Recipe, fügt die Unterstützung für SSL hinzu und setzt den Knoten in die „Monitor Role“ [run].

Cookbooks

Ein Cookbook beschreibt und beinhaltet alles, was benötigt wird, um einen bestimmten Dienst oder Service zu installieren und konfigurieren. Die Cookbook-Dateistruktur besteht aus den Verzeichnissen:

- Resources
  Eine Ressource ist Teil eines Recipe und definiert die Aktionen näher, die bei der Installation einzelner Pakete ausgeführt werden sollen, z.B. die Benutzergruppen für bestimmte Dienste.
- Providers
  Ein Provider führt die von Ressourcen bestimmten Aktionen auf dem Client aus.
- Attributes
  Attributes sind die Eigenschaften für die Knoten, die bei Bedarf überschrieben werden
- Definitions
  In den Definitionen werden schon vorhanden Ressourcen kombiniert
- Files
  Das Verzeichnis enthält spezielle Ressourcen
- Libraries
  Mit Ruby-Bibliotheken können die Cookbooks um Ruby-Code erweitert werden
- Recipes
  Das Verzeichnis enthält die Recipes, den Hauptbestandteil der Cookbooks
- Templates
  Enthält komplexe Konfigurations-Szenarien

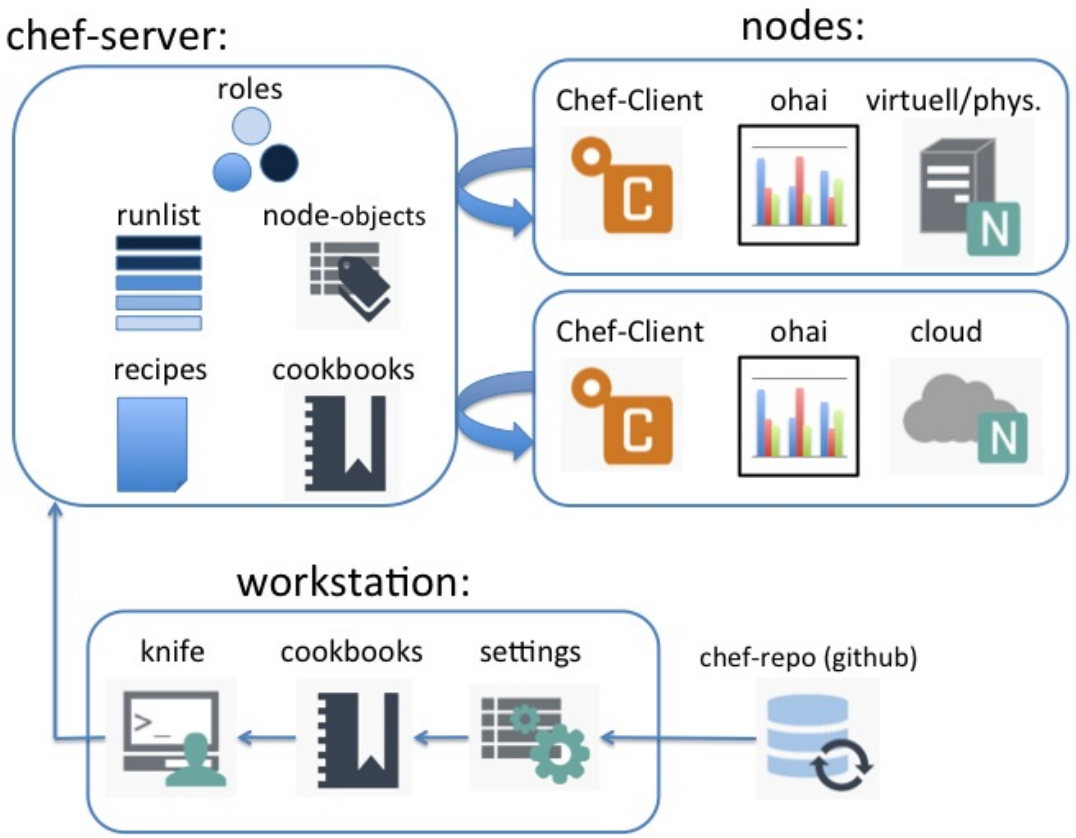

Abbildung 2.1: Chef-Architektur

und der

- Metatadata.rb-Datei
  die Informationen, wie Abhängikeiten und Versionen über die Recipes bereitstellt.

Damit sich Cookbooks dynamisch an die Knoten anpassen, können sie mit Attributen parametrisiert werden. Diese Attribute können direkt im Cookbook gespeichert, über die Umgebungsvariablen angepasst, als Rolle mitgegeben oder für einen Knoten direkt vorausgewählt werden.

Recipes

Ein Recipe ist der wesentliche Bestandteil einer Chef-Konfiguration. Es handelt sich um Programmcode in Ruby, der hauptsächlich aus einer Hintereinanderausführung von Ressourcen besteht. Bevor das Recipe vom Chef-Client ausgeführt wird, muss es der run-list hinzugefügt werden.

Code-Beispiel für ein Recipe, das Apache2 startet

```
package "apache2"
      service "apache2" do
      action [:enable,:start]
end
```

2.1.3 Juju

Das Konfigurationsmanagement-Tool Juju wurde von Canonical entwickelt und liegt mit dem Fokus im Vergleich zu Puppet und Chef mehr auf dem einfachen Zusammenspiel der einzelnen Cloud-Services. Canonical hat ebenfalls das auf Linux basierte Betriebsystem Ubuntu entwickelt.
Juju teilt sich auf in einen lokal installierten Client, der auf Linux-, Mac- und Windows-Maschinen installiert werden kann, und einen „bootstrap“ Knoten, der auf dem Cloud-Server läuft. Der Cloud-Server muss bestimmte Voraussetzungen erfüllen, um den bootstrap-Knoten installieren zu können. So kann Juju aktuell mit vorgefertigten Konfigurationen auf Cloud-Infrastrukturprovidern wie Amazon Web Services, Windows Azure und HP Cloud, sowie auf OpenStack oder MAAS Installationen oder lokal auf einem Linux Container (LXC) installiert werden.
Nach der Installation auf einem der Cloud Infrastrukturprovider startet Juju eine Server Maschine mit einem „State Server“, der die gesamte Umgebung kontrolliert. Sobald die Maschine gestartet ist, können über eine grafische Benutzeroberfläche oder die Konsole weitere Dienste installiert werden. Die Perspektive von Juju auf die Umgebung befindet sich auf einem hohen Level [juja], die mit der eines Menschen vergleichbar ist. Mit Juju können Dienste aus einem Repertoire an fertigen Software Containern, sogenannten „Charms“ ausgewählt, installiert und miteinander verbunden werden. Juju kümmert sich selbstständig um die richtige Verbindung zwischen den einzelnen Charms. Man kann beispielsweise relativ einfach mit dem folgenden Code eine Installation und Verbindung zwischen MySQL und dem Content-Management System Wordpress[1] herstellen:

```
$ juju deploy cs:precise/wordpress
$ juju deploy cs:precise/mysql
$ juju add-relation wordpress mysql
```

Die Befehle kommunizieren mit dem State-Server, indem sie nur den gewünschten Zustand des Systems übermitteln. Der State-Server versucht nun, die Unterschiede im aktuellen und gewünschten Zustand zu beseitigen. In diesem Beispiel werden zuerst zwei neue Maschinen so instanziiert und konfiguriert, dass sie die entsprechenden Dienste ausführen können. Als nächstes werden die Dienste direkt auf den Maschinen installiert.
Ein Vorteil von Juju gegenüber einfacheren Konfigurationsmanagement-Tools wie Puppet oder Chef ist der Austausch der erforderlichen Informationen, um eine Verbindung der Dienste zu gewährleisten. Es werden in diesem Beispiel automatisch Login-Informationen und Adressen der Datenbank an Wordpress weitergegeben, damit der Dienst ausgeführt werden kann. Voraussetzung für das erfolgreiche Verbinden zweier Dienste sind gemeinsame Schnittstellen.
Juju kann mit speziellen Befehlen angewiesen werden, die Dienste optimal zu konfigurieren, was meist schon eine Verbesserung gegenüber der Standard-Installation ist [jujb]. Kommt es bei der

[1] http://de.wordpress.org/

Wordpress-Instanz zu einer Last, kann man mit dem Befehl

```
$ juju add-unit wordpress
```

eine weitere Maschine mit Wordpress dem Netzwerk hinzufügen. Die Maschine ist automatisch so konfiguriert, dass sie die gleiche Datenbank wie die erste Maschine benutzt. Die Last kann so auf zwei Knoten verteilt werden, z.B. durch das zusätzliche Intstallieren von „haproxy“ und dem „S3-Plugin“. Durch die Eingabe von

```
$ juju set wordpress tuning=optimized
```

fügt das System bei Last automatisch neue Knoten hinzu und entfernt diese auch wieder. Dazu wird als automatischer Lastverteiler „nginx“ installiert und konfiguriert. Die Neukonfigurierung erfolgt immer während der Laufzeit des Gesamtsystems und geht von dem Charm des jeweiligen Dienstes selbst aus.

Charms

Ein Charm ist die eigentliche Service-Einheit bei Juju und besteht aus Metadaten und sogenannten „Hooks“, die in der Konsole ausführbare Skripte sind. In den Metadaten, bestehend aus den *metadata.yaml*- und *config.yaml*-Dateien, werden der Name des Charms, seine Schnittstellen und die möglichen Konfigurationen für die eigentlichen Dienste des Charms festgelegt. Im Charm ist selbst festgelegt, wie es auf Veränderungen der Umgebung reagieren soll.
Der Inhalt der **metadata.yaml**-Datei im Wordpress-Charm[2]:

```
name: wordpress
summary: "WordPress is a full featured web blogging tool, this charm deploys it."
maintainer: Marco Ceppi <marco@ceppi.net>
description: ...
categories: ["applications"]
requires:
 db:
  interface: mysql
 nfs:
  interface: mount
 cache:
  interface: memcache
provides:
 website:
  interface: http
peers:
 loadbalancer:
  interface: reversenginx
```

[2]https://manage.jujucharms.com/charms/oneiric/wordpress

Hooks

Die Hooks sind Skripte, die in einer beliebigen Sprache geschrieben werden können, solange sie in der Konsole ausführbar sind. Wichtig ist die richtige Bezeichnung der Skripte. So gibt es z.B. „install“-, „start“-, „stop“- oder „config-change“-Hooks, die genau die im Namen festgelegte Aktion für den Dienst ausführen, bzw. bei Veränderungen entsprechend reagieren. Im obigen Beispiel werden beim Hinzufügen der Verbindung zwischen Wordpress und MySQL bei beiden Charms das „db-relation-joined“-Hook aufgerufen, das weitere Hooks, die die nötigen Verbindungseinstellungen vornehmen, auslöst. Beide Seiten wissen dabei nicht genau, mit welchem Dienst sie kommunizieren. Solange die Schnittstellen implementiert sind, könnte hier z.B. auch eine andere Datenbank über die MySQL Schnittstelle von Wordpress angesprochen werden.

Auszug aus dem **„install“-Hook**[3] von Wordpress:

```
#!/bin/bash
set -xe

add-apt-repository ppa:charmers/charm-helpers
apt-get update && apt-get -y upgrade

apt-get -y install php5-memcache mysql-client pwgen php5 php5-fpm \
      php-apc mailutils php-mail sysstat php5-mysql php5-mcrypt php5-memcache \
      charm-helper-sh php5-curl rsync nfs-common git-core mktemp
......
juju-log "Installing wp-cli to make this charm’s life a little easier ..."
git clone https://github.com/wp-cli/wp-cli.git /usr/src/wp-cli
# other hooks call hooks/install and don’t expect the CWD to change so
# run this section in a subshell
(
      cd /usr/src/wp-cli
      # 20120926: v0.6.0 is confirmed working, so use it rather than alpha.
      git reset --hard v0.6.0
      git submodule update --init
      utils/dev-build
)
```

2.2 Standardisierungsbestrebungen im Bereich Cloud Management

Einheitliche Standards im Bereich Cloud Management sind aus Benutzersicht wichtig, um die vielfältigen Angebote an unterschiedlichen Werkzeugen zusammenzufassen und vergleichbar zu machen. Außerdem benötigt es standardisierte Schnittstellen im Cloud Computing, um die Daten zwischen den Werkzeugen austauschen zu können.

[3] https://bazaar.launchpad.net/ charmers/charms/oneiric/wordpress/trunk/files/head:/hooks/

2.2.1 Portierbarkeit der DevOps-Module

Das vorige Kapitel befasste sich mit den drei Konfiguration-Management Tools Puppet, Chef und Juju. Auf dem IT-Markt befinden sich noch eine Reihe weiterer Tools, die bisher nicht genannt wurden, aber auch von vielen Administratoren eingesetzt werden. Dazu gehören zum Beispiel noch Ansible[4] und Salt[5], die wieder eine andere Art des Cloud Management verfolgen und etwas näher an den typischen Administrator-Skripten sind.
Alle vorgestellten Tools setzen entweder auf gänzlich verschiedene Programmiersprachen wie Ruby, Python und YAML[6] oder auch auf eigene Sprachen zur Konfigurationsbeschreibung.
Zur Ausführung wird meist auf dem Client die eigene, proprietäre Laufzeitumgebung der Tools benötigt. Dieser Umstand führt dazu, dass einzelne Module der Konfigurationsmanagement-Tools nicht oder nur sehr schwer in andere Tools importiert werden können. Ist man auf bestimmte Module, die es nur bei einem Tool gibt, angewiesen, ist man also gezwungen, komplett auf dieses System zu setzen. Eine Kombination verschiedener Systeme ist so gut wie nicht möglich. Was bei einem Wechsel des eingesetzten Systems erhebliche Kosten verursachen würde.

2.2.2 Vendor Lock-In

Dienste im Bereich Cloud Computing werden von vielen verschiedenen Unternehmen bereitgestellt. Möchte ein Anwender diese Dienste nutzen, muss er sich für ein Unternehmen entscheiden. Das Einrichten der Dienste ohne Konfigurationsmanagement-Tools macht es für den Kunden nahezu unmöglich, den Anbieter der Dienste zu wechseln, ohne ein weiteres Mal die eingesetzten Programme neu zu konfigurieren. Dies erfordert einen Kostenaufwand, den der Anwender bei einem Wechsel des Cloud-Dienstanbieters zu tragen hat. Das Einsetzen oben genannter Tools verringert zwar diesen Aufwand, beseitigt ihn aber nicht, da die Tools auch für jeden Dienstanbieter neu konfiguriert werden müssen. Das sogenannte „Festsitzen“ bei einem speziellen Dienstanbieter wird „Vendor Lock-In“ genannt.
Ein Lösungsansatz sind sogenannte „Cloud provider abstraction libraries“ wie libcloud[7], fog[8] und jclouds[9] [SHI+13], die APIs bereitstellen, die dem Anwender direkten und universellen Zugang zu den Produkten verschiedener Cloud-Dienstanbietern liefern.
Um jedoch auch die Module der verschiedenen Tools untereinander austauschen zu können, bedarf es zusätzlich des Einsatzes sogenannter Meta-Cloud-Standards. Erst standardisierte Meta-Cloud-Modelle, die Applikationen unabhängig vom Dienstanbieter und Management-Tools installieren und betreiben, können das Problem der Portierung wirklich lösen. Ein Ansatz in dieser Richtung ist „Topology and Orchestration Specification for Cloud Applications“ (TOSCA)[10]

[4] http://www.ansible.com/
[5] http://www.saltstack.com/
[6] http://www.yaml.de/
[7] http://libcloud.apache.org
[8] http://fog.io
[9] www.jclouds.org
[10] https://www.oasis-open.org/committees/tosca/

2.2.3 TOSCA

Eine Umsetzung der Portabilität einzelner Programme auf funktionalem Level wurde schon zum Beispiel in Projekten wie Cmotion [BLS11] erforscht. Was hier jedoch noch fehlte, war die Portabilität ganzer Management Systeme und deren operationalen Aufgaben [BBLS12].
Hier setzt „Topology and Orchestration Specification for Cloud Applications“ (TOSCA) an und definiert einen Standard für Cloud Management-Systeme und deren ausführbare Programme. TOSCA definiert hierfür ein technisches Meta-Modell der Konfigurationstopologie der eingesetzten Dienste in XML und setzt bei der Beschreibung der Programmabläufe auf bekannte Sprachen wie BPEL und BPMN. Durch die Einhaltung dieses Standards können sowohl die Programmabläufe, als auch deren Konfigurationsmanagement über Cloud-Dienstanbieter hinweg portiert werden. Voraussetzung hierfür ist eine TOSCA kompatible Laufzeitumgebung, welche die Ausführung und Konfiguration der Dienste übernimmt.

Modellebenen

In TOSCA gibt es, wie in Abbildung 2.2 dargestellt, grundsätzlich drei Ebenen: Typen, Templates und Instanzen. Ein Node Template ist eine physikalische oder virtuelle Maschine, die durch ihren Node Type definiert wird. Diese Node Types sind allgemein wiederverwendbare Konfigurations-Einheiten, die die Node Templates beschreiben. Zu der Beschreibung des Node Types gehören die Requirements, die Capabilities und das Interface des Node Templates.
Die Templates beschreiben die Topologie durch das Einsetzen konkreter Werte in die abstrakten Eigenschaften ihrer Typen. Daraus werden zur Laufzeit Instanzen gebildet. In TOSCA wird nur das Metamodell für Typen und Templates definiert. Damit lassen sich alle möglichen Arten von Diensten beschreiben.
Ein Beispiel für ein Topology Template könnte ein Webserver sein, auf dem Apache2 als Software läuft, das wiederum auf einem Linux Betriebssystem installiert ist. Ein Node Template wäre in diesem Fall der „Webserver Node Type“ mit den speziellen Attributen, wie Name und Konfigurationseigenschaften. Ein weiteres Node Template wäre das Betriebsystem Linux, das aus einem „OS Node Type“ besteht. Die Beziehung zwischen den Templates kann durch einen Relationship Type, wie „hostet on“ dargestellt werden [Lip13].

Service Templates

Node Templates und Relationship Templates definieren zusammen ein Topology Template als einen gerichteten Graphen. Die Templates wurden als Instanzen aus den jeweiligen Typen gebildet. Aus dem Node Type wurde so durch nähere Spezifizierung das Node Template als eine Einheit im Service Template. Mehrere Node Templates werden durch Relationship Templates miteinander verbunden. Die Abbildung 2.4 stellt diesen Zusammenhang dar.
Ein Node Template bekommt von dem instanziierenden Node Type „Implementation Artifacts“ und „Deployment Artifacts“ übergeben.

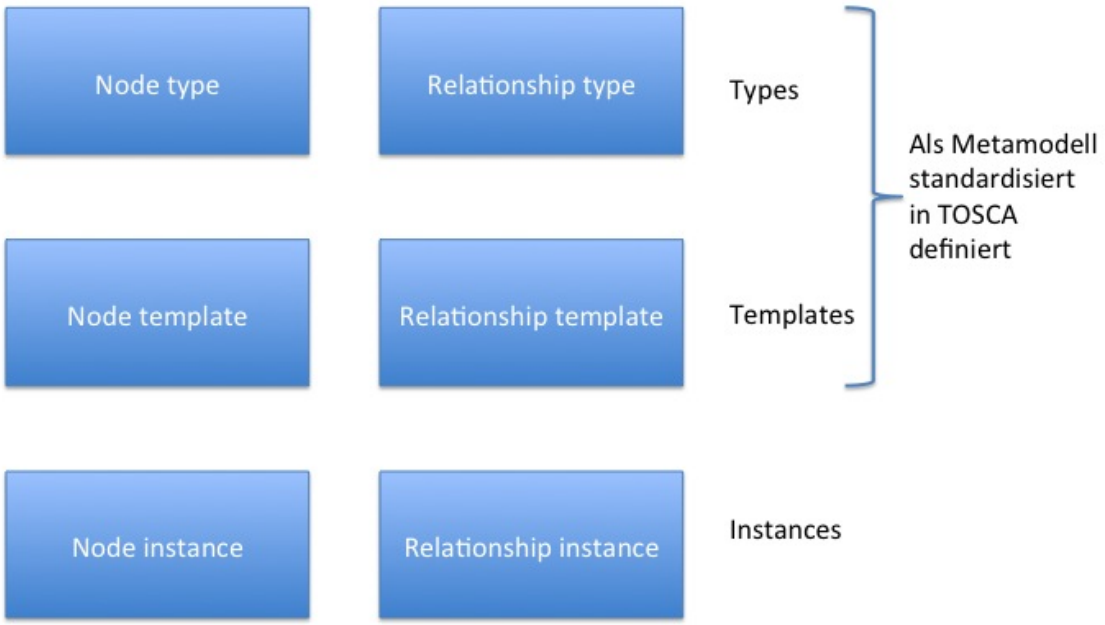

Abbildung 2.2: Modellebenen in TOSCA [TCAT]

Implementation Artifacts enthalten Informationen über die Implementierung der Management-Funktionen, wie zum Beispiel „install“ und „deploy“, des Node Types. Die einzelnen Funktionen der Node Types können dadurch zusammengefasst zur Beschreibung der Funktionen des Service Templates hinzugefügt werden.
Die Deployment Artifacts enthalten Bedingungen, die zum Beispiel vom Betriebssystem oder TOSCA Container erfüllt sein müssen, um ein Node Template aus einem Node Type zu instanziieren.
Ein Service Template beschreibt einen kompletten Dienst durch die Zusammenfassung der einzelnen Node und Relationship Templates, der Typen inklusive deren Artefakte und Ablaufplänen.
Während im Service Template die Struktur des Dienstes beschrieben wird, sind die Ablaufdiagramme für die Erstellung, Verwaltung und Terminierung der eigentlichen Instanzen aus den Templates zuständig. Für die Beschreibung der Pläne setzt TOSCA auf die bewährten Sprachen zur Prozessbeschreibung BPEL und BPMN.
Die Node-Instanzen enthalten eine Reihe an Eigenschaften, die das Ablaufdiagramm benötigt, um sie starten, stoppen und konfigurieren zu können. Diese Eigenschaften sind in TOSCA im XML-Format beschrieben. Zu den Eigenschaften gehört zum Beispiel auch der Zustand der Implementation Artifacts.

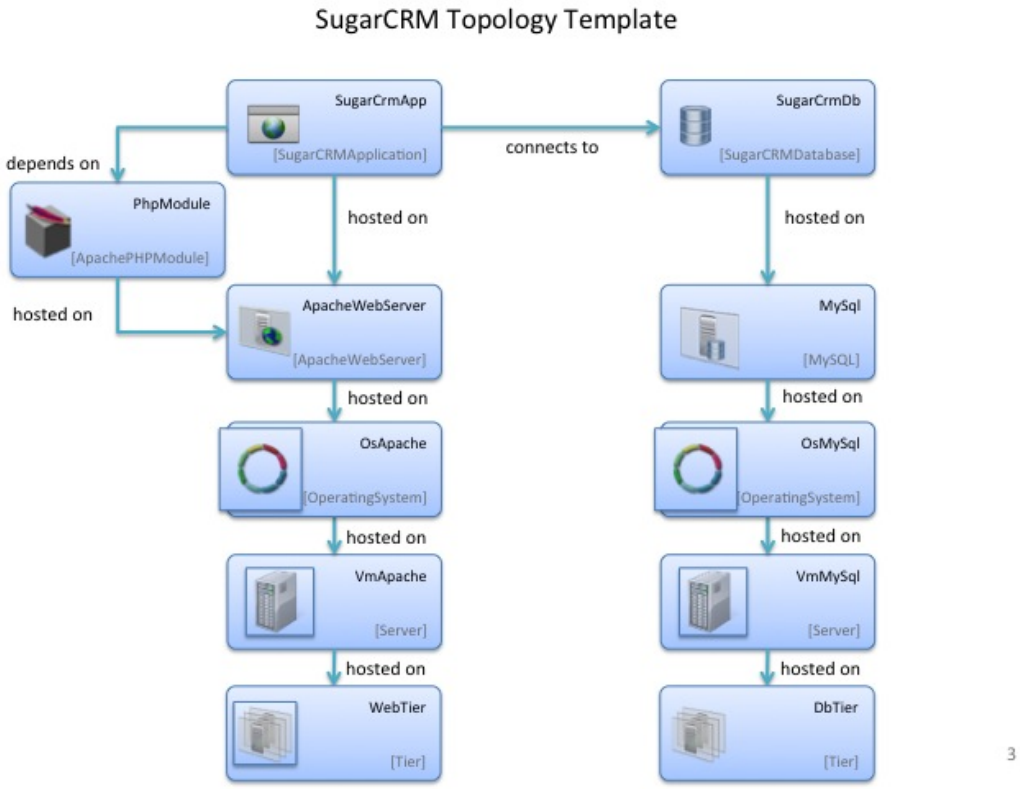

Abbildung 2.3: Topology Template von SugarCRM

bestehend aus verschiedenen Node Templates, welche aus Node Types (unterer, grauer Text im Node Template) instanziiert wurden. Die Pfeile repräsentieren die Relationship Types. [Rut]

Hinzu kommt der „instance state“, der den aktuellen Zustand der Instanz, allerdings nur in Textform, beschreibt.

Cloud Service Archive (CSAR)

Das Service Template und alle darin enthaltenen Dateien, wie Skripte, Artifacts, Beschreibungen und Ablaufdiagramme werden in einem ZIP-Archiv zusammengefasst. Dieses sogenannte „Cloud Service Archive“ (CSAR) enthält somit alle zur Ausführung des Cloud Dienstes nötigen Dateien und Informationen. Man bezeichnet es dadurch als „self-contained“. Eine Software, die CSARs ausführen kann, benötigt die TOSCA-Laufzeitumgebung, genannt TOSCA-Container. Ein CSAR muss die Verzeichnisse „TOSCA-Metadata“ und „Definitions“ beinhalten, damit es ausgeführt werden kann. In der Metadaten-Datei ist der restliche Aufbau der CSAR-Datei beschrieben. Das Definitions-Verzeichnis kann mehrere .tosca-Dateien enthalten, die die einzelnen Elemente des Services näher beschreiben.

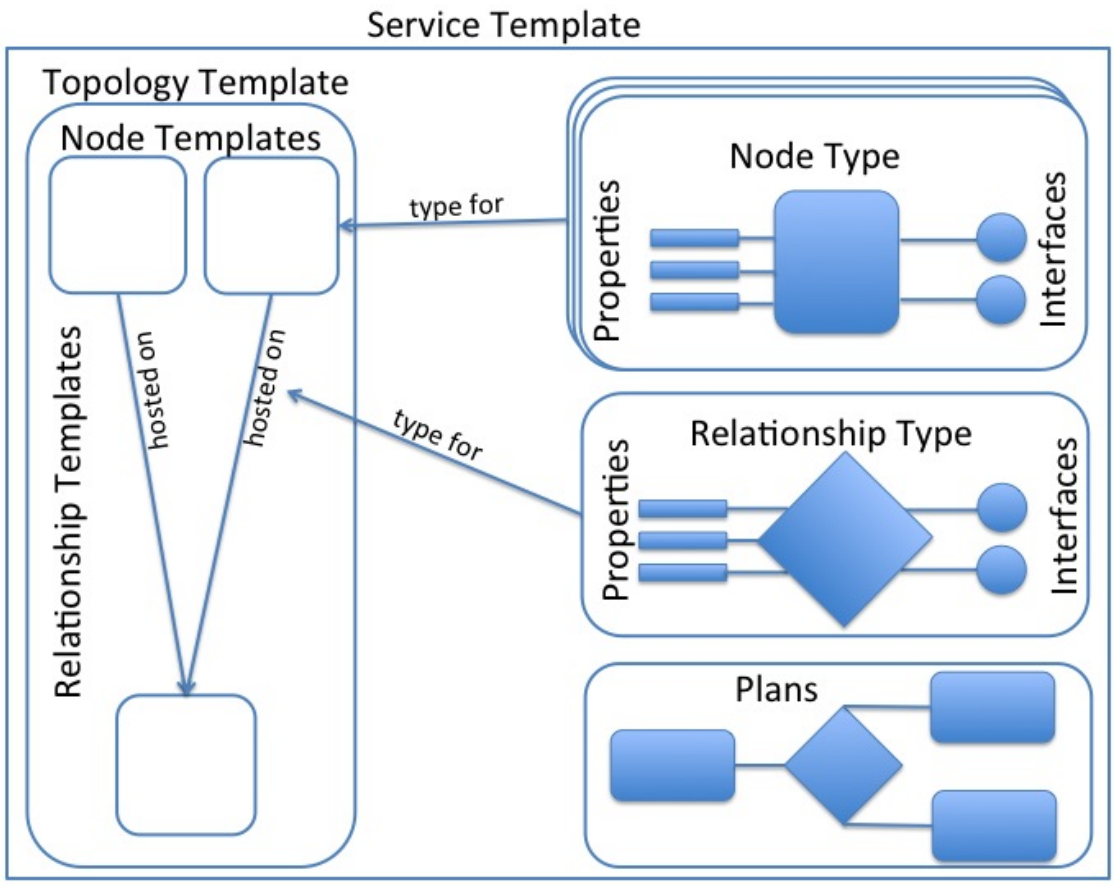

Abbildung 2.4: Service Templates in TOSCA [TCAT]

Die genaue Verzeichnisstruktur ist in Abbildung 2.5 dargestellt. Eine .tosca-Datei wird im XML-Format angelegt und muss mindestens eines der TOSCA-spezifischen Elemente wie zum Beispiel ein Service Template oder einen Node Type definieren. (Andere TOSCA-spezifische Elemente die definiert werden können sind: Node Type Implementation, Relationship Type, Relationship Type Implementation, Requirement Type, Capability Type, Artifact Type, Artifact Template, Policy Type oder Policy Template.)

OpenTOSCA

OpenTOSCA stellt eine Laufzeitumgebung für mit TOSCA generierte Service Templates zur Verfügung. Es kann also die in den CSARs enthaltenen Ablaufdiagramme, ausgehend von einem Startknoten, durchführen und somit die darin beschriebenen Cloud-Dienste starten und stoppen. Der Ablauf der Ausführung beginnt mit dem Core, der das CSAR-Archiv entpackt und die Dateien im entsprechenden Dateisystem speichert. Die im CSAR enthaltenen .tosca-XML-Dateien werden von der „OpenTOSCA Control“-Einheit an die TOSCA Engine übergeben, welche sie überprüft und verarbeitet. Die

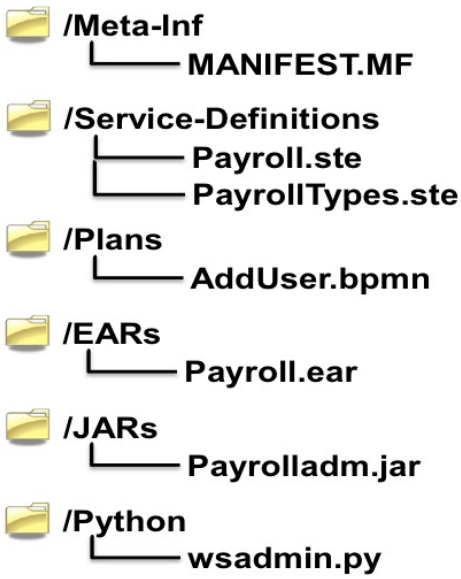

Abbildung 2.5: Beispiel CSAR-Dateistruktur aus [TCAT]

„OpenTOSCA Control"-Einheit ruft anschließend die Implementation Artifact und Plan Engine auf. Die IAEngine installiert die in den .tosca Dateien enthaltenen Implementation Artifacts auf einem Server und speichert den Endzustand in einer Datenbank [BBH+13]. Die Plan Engine verbindet und installiert die enthaltenen Ablaufpläne, deren Endzustände auch in einer eigenen Datenbank gespeichert werden.
Schließlich kann das Service Template ausgeführt werden, in dem durch das UI oder eine Schnittstelle der „Build-Plan" gestartet wird. Die Abbildung 2.6 stellt die beschriebene Architektur bildlich dar.

2.3 Bestehende Ansätze für Crawling

Ein Ziel der Diplomarbeit ist das Finden und Speichern vorhandener Module der verschiedenen Konfigurationsmanagement-Tools. Dazu müssen sowohl JSON-Schnittstellen, als auch menschenlesbare Webseiten durchforstet und deren Inhalt gespeichert werden. Dieses Kapitel stellt verschiedene Möglichkeiten einzelner Programmiersprachen vor.

2.3.1 Perl

Zur Programmiersprache Perl gibt es auf www.cpan.org[11] ein reichhaltiges Angebot an Modulen, die man mit einfachen Mitteln in sein Programm integrieren kann, sobald man CPAN einmal installiert

[11] http://www.cpan.org/

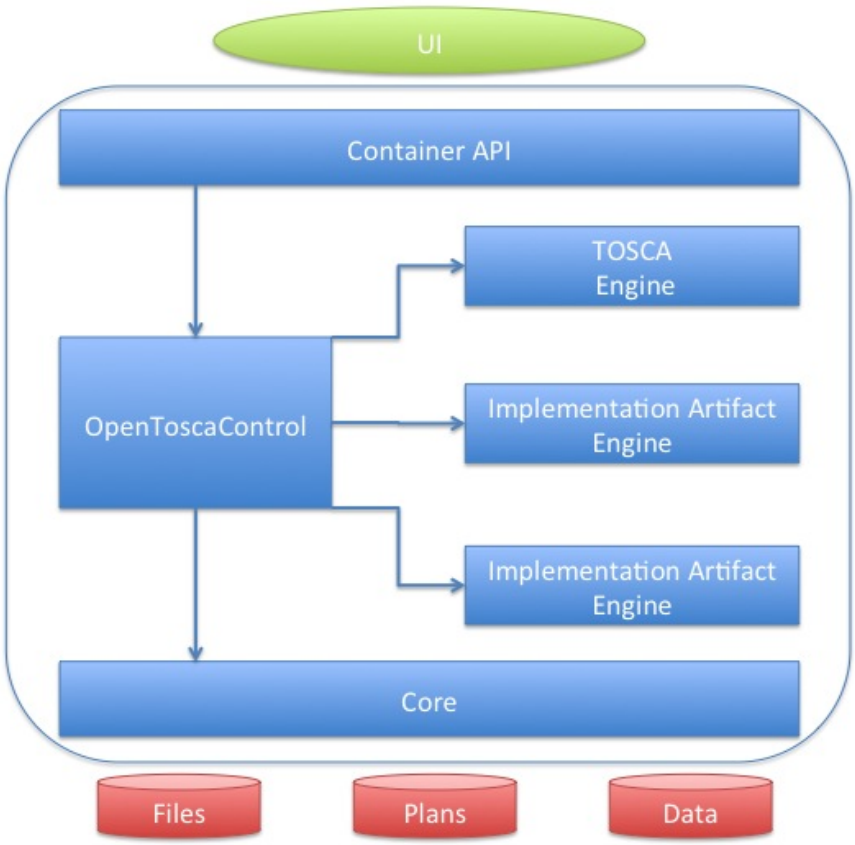

Abbildung 2.6: OpenTOSCA Architektur [BBH+13]

hat. So gibt es schon einen vorgefertigten JSON Parser[12] und eine WWW-Library[13] mit deren Hilfe man Internetseiten aufrufen und auslesen, bzw. den JSON Code parsen kann. Zum gezielten Auslesen einer JSON-API benötigt man so nur wenige Zeilen Code:

```
use LWP::UserAgent;
use JSON qw( decode_json );

$ua = LWP::UserAgent->new;
$ua->agent("$0/0.1 " . $ua->agent);
# $ua->agent("Mozilla/8.0") # pretend we are very capable browser

my $total=10;
my $start=0;
```

[12]http://search.cpan.org/ makamaka/JSON-2.90/lib/JSON.pmund
[13]http://search.cpan.org/ gaas/libwww-perl-6.05/lib/LWP.pm

```
$req = HTTP::Request->new(GET =>
    'https://cookbooks.opscode.com/api/v1/cookbooks?start='.$start.'&items='.$total);
$req->header('Accept' => 'text/HTML');
# send request
$res = $ua->request($req);
 # check the outcome
     if ($res->is_success) {
           #print $res->decoded_content;
            my $json=$res->decoded_content;
            my $decoded = decode_json($json);
            my @items = @{ $decoded->{'items'} };

            foreach my $f ( @items ) {
                        $cname=$f->{"cookbook_name"};
                        $cdescr=$f->{"cookbook_description"};
                         print "Name: ".$cname. "\n";
                         print "Beschreibung: ".$cdescr . "\n";
                }
     }else {
            print "Error: " . $res->status_line . "\n";
     }
```

Das oben gezeigte Code Beispiel durchsucht die ersten zehn Chef Cookbooks und gibt deren Namen und Beschreibungstext in der Konsole aus. Die WWW-Library für Perl baut eine Verbindung zur Chef JSON-API auf und übergibt den Inhalt an den JSON-Dekoder. Mit seiner Hilfe kann man durch die einzelnen Items navigieren. Zum Auslesen der zusätzlichen Informationen eines einzelnen Cookbooks müssen noch Verbindungen zu weiteren Teilen der Schnittstelle hergestellt werden.
Eine Speicherung der Informationen in einer Datenbank, wie z.B. MongoDB kann mit Hilfe des MongoDB Treibers[14] für Perl vorgenommen werden. Die WWW-Library erlaubt auch das Herunterladen der Cookbook Dateien, die mit dem MongoDB-Treiber direkt in die Datenbank kopiert werden können.
Für Multithreading-Anwendungen sollte man am besten die WWW-Library durch die WWW::Curl Bibliothek[15] ersetzen. Um eine JSON-API mit wenig Einarbeitungszeit auszulesen, ist Perl demnach gut geeignet.
Um normale, mit HTML generierte Webseiten zu durchforsten, muss man von einer Startseite ausgehend allen Links folgen und den Inhalt mit Perl nach dem gewünschten Ausdruck durchsuchen. Links, die von einer Seite ausgehen, können automatisch mit dem WWW::Mechanize[16] gefunden und verfolgt werden. Für das Finden eines bestimmten Wortes im Quellcode der Seite setzt Perl vor allem auf reguläre Ausdrücke. Diese müssen stark angepasst werden, damit genau alle Module eines DevOps Tools gefunden werden.
Fazit:
Die Aufgabe kann durchaus mit Perl bewältigt werden. Es setzt beim crawlen der HTML-Webseiten aber stark auf reguläre Ausdrücke, was den Programmcode schnell recht unübersichtlich und kaum allgemein wiederverwendbar macht.
Vorteile:

[14]http://search.cpan.org/ friedo/MongoDB/lib/MongoDB.pm
[15]http://search.cpan.org/ szbalint/WWW-Curl-4.17/lib/WWW/Curl.pm
[16]http://search.cpan.org/ ether/WWW-Mechanize-1.73/lib/WWW/Mechanize.pm

- CGI Schnittstelle für HTML
- JSON Parsing integriert
- Viele Module schon vorhanden
- Kurze Einarbeitungszeit
- Wenige Codezeilen
- Keine Zusatzprogramme nötig

Nachteile:

- Kommandozeilen Code
- HTML-Crawler müssen von Hand genau angepasst werden

2.3.2 Apache Nutch

Nutch[17] ist ein Open Source Web Crawler, der aus dem Apache-Lucene-Projekt hervorgegangen ist [Apa]. Er basiert in seiner ersten Version auf der Hadoop-Dateistruktur, was ihn gerade für den Multithreading-Einsatz, also das gleichzeitige Crawlen mit verschiedenen Instanzen, prädestiniert. Um Nutch effektiv nutzen zu können, müssen zuvor andere Apache-Dienste wie HBase[18] , Solr[19] oder Gora[20] installiert und konfiguriert werden. Dieser Umstand lässt die Projekt Komplexität schnell extrem anwachsen.
Nach dem Herunterladen und Installieren kann Nutch über Konfigurationsdateien so angepasst werden, dass nur bestimmte URLs verfolgt werden und eine maximale Tiefe beim Crawlen nicht überschritten wird. Das Ergebnis des Crawlers kann direkt in einer Datenbank gespeichert werden. Dort müssen die Daten weiter bearbeitet werden.
Fazit:
Vorteile:

- Crawlt automatisch URLs
- Extrem leistungsstark

Nachteile:

- Benötigt Solr, hbase, Gora die wiederum andere Anwendungen benötigen
- Nicht einfach einzurichten
- Hadoop als Dateistruktur in diesem Projekt eher hinderlich
- Allgemein für dieses Projekt überdimensioniert

[17] http://nutch.apache.org/
[18] https://hbase.apache.org/
[19] https://lucene.apache.org/solr/
[20] http://gora.apache.org/

2.3.3 Python

Ähnlich wie Perl setzt auch Python auf einfach gehaltene Befehle im Komandozeilen-Stil. Mit dem Pip-Installer[21] lassen sich Pakete zu Python hinzufügen. Zum Beispiel kann man nach dem Einbinden des requests-Paketes[22] HTTP-Requests abschicken und das Ergebnis bearbeiten. Eine Bibliothek, um JSON-Objekte zu laden und bearbeiten, ist in der Standard Bibliothek von Python bereits enthalten.

```
import json, requests, pprint

url = 'https://cookbooks.opscode.com/api/v1/cookbooks?start=1'

params = dict()
data = requests.get(url=url, params=params)
binary = data.content
output = json.loads(binary)

# test to see if the request was valid
#print output['status']

# output all of the results
#pprint.pprint(output)

# step-by-step directions
for item in output['items']:
      print "Name: "+item['cookbook_name']
      print "Descr: "+item['cookbook_description']
      url2 = item['cookbook']
      #print url
      output2=requests.get(url2).json()
      url3= output2['latest_version']
      output3=requests.get(url3).json()
      print "URL: "+output3['file']
      print '*******'
```

Der obige Code lädt den Inhalt der Chef JSON-API in ein JSON-Objekt. Anschließend wird jedes „item“ untersucht und ein zweiter Request zur jeweiligen Detailseite abgeschickt. Der Inhalt wird einfach ausgegeben.
Mit der MongoDB Bibliothek pymongo[23] könnten die Dokumente und Dateien in einer Datenbank gespeichert werden.
Um menschenlesbare HTML-Webseiten zu durchforsten, gibt es auch schon fertig entwickelte Open Source Lösungen, wie zum Beispiel Scrapy. Damit können automatisch Links verfolgt und bestimmte Strukturen im Quellcode, wie z.B. <DIV> Elemente erkannt und deren Inhalt gespeichert werden. Die Erkennung funktioniert wie bei Perl mit regulärereren Ausdrücken.
Fazit:

- Module requests, um Webseiten aufzurufen

[21] http://www.pip-installer.org/en/latest/
[22] http://docs.python-requests.org/en/latest/
[23] http://api.mongodb.org/python/2.7rc0/

- Zusatz-Module einfach nachinstallierbar: "pip intall requests"
- JSON-Module schon standardmäßig installiert
- Einfaches JSON-Parsen mit output[Śchlüsselname]́
- Scrapy als HTML Crawler, der automatisch Links erkennt

2.3.4 Java (crawler4j)

Die Programmiersprache Java bietet viele Möglichkeiten, um die im nachfolgenden Kapitel definierten Anforderungen umzusetzen. Mit der Open Source Java-Entwicklungsumgebung Eclipse[24] und dem Build-Management Tool Maven[25] lassen sich über die integrierte Suche in globalen Maven-Repositories auf einfache Art Module und Bibliotheken zu einem Projekt hinzufügen.
Einmal installiert, kümmert sich Maven um das Bereitstellen der hinzugefügten Bibliotheken. Mit dem so installierten Open-Source Crawler crawler4j[26] lassen sich Inhalte von menschenlesbare Webseiten speichern und automatisch weiterführende Links verfolgen. Der Crawler unterstützt dabei auch Multi-Threading, d.h. man kann parallel mit mehreren Prozessen arbeiten. Um gezielt Informationen der Webseiten auszulesen, muss man den Inhalt mit regulären Ausdrücken untersuchen.
Zum Ansprechen und verarbeiten einer RestFulApi gibt es in Java eine Reihe nützlicher Bibliotheken. Mit der Erweiterung org.json[27] kann man direkt JSON-Objekte anlegen und verarbeiten. Der Java - Code lädt den Inhalt einer Webseite in ein neues JSONObject:

```
private static JSONObject readJsonFromUrl(String url) throws IOException, JSONException {
   InputStream is = new URL(url).openStream();
   try {
      BufferedReader rd = new BufferedReader(new InputStreamReader(is, Charset.forName("UTF-8")));
      String jsonText = readAll(rd);
      JSONObject json = new JSONObject(jsonText);
      return json;
   } finally {
      is.close();
   }
}
```

Mit der obigen Funktion wird aus der RESTful API von Chef ein JSONObject erzeugt, die einzelnen Inhalte (hier: Items) in ein JSONArray transformiert und durchforstet. Der Name und die URL eines einzelnen Items werden in der Konsole ausgegeben:

```
String url ="https://cookbooks.opscode.com/api/v1/cookbooks?start=0&items=10;
JSONObject allJSONObjects = readJsonFromUrl(url);
int gecrawlt=0;
JSONArray nestedArray = allJSONObjects.getJSONArray("items");
for (int i=0; i<nestedArray.length(); i++) {
```

[24] https://www.eclipse.org/
[25] http://maven.apache.org/
[26] http://code.google.com/p/crawler4j
[27] http://www.json.org/java/

```
            JSONObject item = nestedArray.getJSONObject(i);
            String name = item.getString("cookbook_name");
            String curl = item.getString("cookbook");
            System.out.println("Name: " + name);
            System.out.println("URL: " + curl);
}
```

Fazit:

- Mächtige Programmiersprache
- Auch komplexe Anforderungen umsetzbar
- Maven
- Viele Bibliotheken
- Große Community
- JSON Verarbeitung mit eigener Library
- MongoDB Verarbeitung mit eigenem Treiber.
- HTML Crawling mit crawler4j
- Folgt automatisch allen Links zu Unterseiten
- Multi-threading beim Crawlen

2.3.5 Vergleich

Sowohl mit Perl, Python als auch Java lassen sich Webcrawler einsetzen, die automatisch Links verfolgen und den Webseiten-Inhalt untersuchen können. Alle genannten Programmiersprachen benötigen dabei ähnlichen Aufwand und unterscheiden sich letztlich nur darin, welche Bibliotheken es schon gibt. Für diese Arbeit macht das crawler4j-Modul für Java letztlich den positiven Unterschied aus, da diese Lösung schon direkt einsetzbar ist und sofort Ergebnisse liefert.

3 Anforderungen

3.1 Werkzeug-spezifische Anforderungen

Eine Anforderung der Diplomarbeit ist es, Informationen über Module aus den online verfügbaren Chef-, Juju- und Puppet-Repositories auszulesen und deren ausführbare Dateien zu speichern. Bei jedem der drei Konfigurationsmanagement-Tools muss dabei auf andere Weise vorgegangen werden.

3.1.1 Chef-Cookbooks

JSON-API ansprechen

„Chef“ bietet eine RESTful API an, die JSON-Objekte als Ergebnis liefert. Der Einstiegspunkt um Informationen zu allen Modulen zu erhalten bietet diese URL[1] zur API. Man kann durch Erhöhen des Startwertes in Zehnerschritten das gesamte verfügbare Verzeichnis durchforsten und erhält einen kurzen Überblick über die angebotenen Module:

```
{
   "total": 1393,
   "start": 20,
   "items": [
      {
         "cookbook_name": "android-sdk",
         "cookbook": "http://cookbooks.opscode.com/api/v1/cookbooks/android-sdk",
         "cookbook_maintainer": "gildegoma",
         "cookbook_description": "Installs Google Android SDK"
      },
      {
         ...
      }
   ]
}
```

Weitere Informationen zu einzelnen Cookbooks kann man durch das Aufrufen der unter „cookbook“ enthaltenen URL[2] erhalten:

[1] https://cookbooks.opscode.com/api/v1/cookbooks?start=1
[2] http://cookbooks.opscode.com/api/v1/cookbooks/android-sdk

```
{
   "versions": [
      "http://cookbooks.opscode.com/api/v1/cookbooks/android-sdk/versions/0_1_1",
      "http://cookbooks.opscode.com/api/v1/cookbooks/android-sdk/versions/0_1_0",
      "http://cookbooks.opscode.com/api/v1/cookbooks/android-sdk/versions/0_0_0"
   ],
   "description": "Installs Google Android SDK",
   "maintainer": "gildegoma",
   "name": "android-sdk",
   "average_rating": null,
   "updated_at": "2014-04-01T06:01:15Z",
   "latest_version": "http://cookbooks.opscode.com/api/v1/cookbooks/android-sdk/versions/0_1_1",
   "created_at": "2013-08-08T14:29:46Z",
   "category": "Programming Languages",
   "external_url": "https://github.com/gildegoma/chef-android-sdk"
}
```


Eine URL zu einer aktuellen Version der ausführbaren Dateien bekommt man schließlich durch das weitere Aufrufen der URL[3] unter „latest version“.

JSON verarbeiten

Um Chef Cookbooks aus dem Online Verzeichnis herunter zu laden und die Informationen in einer Datenbank zu speichern, muss der Crawler also durch JSON-Objekte navigieren und Verweise zu weiter verschachtelten Objekten verfolgen können. Durch ein einheitliches JSON-Format reicht es, den entsprechenden „Key“ im JSON-Objekt zu suchen und den „Value“ als neue URL für das Crawlen der weiteren Informationen zu verwenden.

Dateien herunterladen und verarbeiten

Außerdem muss der Crawler Dateien, wie zum Beispiel .tgz-Archiv-Dateien, herunterladen, speichern und verarbeiten können. Die erforderliche URL zur Datei des Cookbooks liegt in der dritten Ebene der JSON-API. Der Crawler muss diese Datei herunterladen und so speichern, dass sie mit den Informationen des Cookbooks verknüpft werden kann. In dem Cookbook-Archiv ist außerdem die Metadaten-Datei enthalten, deren Informationen ebenso direkt in die Datenbank gespeichert werden sollen.

[3] http://cookbooks.opscode.com/api/v1/cookbooks/android-sdk/versions/0_1_1

3.1.2 Juju-Charms

Links auslesen und JSON-API ansprechen

Die Module für Juju (Charms) sind in einem online Repository[4] gespeichert. Zugang zu den Informationen über einzelne Charms erhält man auch hier über eine RESTful API[5], die JSON-Objekte zurückliefert. Jedoch kann man nicht durch diese API iterieren, da maximal 20 Objekte zurück geliefert werden.
Um an alle Objekte zu gelangen, muss man die Webseite des Charm Browsers[6] aufrufen, die direkte Links zu allen Charms enthält. Mit dem Namen aus dem Link muss man die REST-API abfragen, deren Ausgabe zum Beispiel so aussieht:

```
{
  "result": [
    {
      "charm": {
       "code_source": {
        "location": "lp:~charmers/charms/precise/cinder/trunk",
        "revision": "29",
        "type": "bzr"
                           },
         "maintainer": {
        "email": "adamg@canonical.com",
        "name": "Adam Gandelman"
       },
       "name": "cinder",
       ....
       }
       .. }]
 }
```

JSON verarbeiten

In den Informationen, die die JSON-API zurückliefert, ist meistens ein Verweis auf ein „Bazaar"-Verzeichnis [7] auf www.launchpad.net enthalten. Dort befinden sich die ausführbaren Skripte und eine Metadaten Datei.

Dateien aus einem Bazaar-Branch von Launchpad herunterladen und verarbeiten

Diese Metadaten-Datei im YAML-Format muss vom Crawler untersucht und deren Inhalt in der Datenbank gespeichert werden können, da die API nicht alle Informationen über das Modul bereitstellt. Das restliche Archiv muss so gespeichert werden, dass es mit den Informationen über das Charm

[4] https://jujucharms.com/
[5] https://manage.jujucharms.com/api/2/charms?limit=5
[6] http://manage.jujucharms.com/charms
[7] https://launchpad.net/bzr

verknüpft werden kann. Der Crawler muss also in der Lage sein, durch JSON-Objekten einer API zu navigieren und Dateien aus einem Bazaar-Branch speichern, entpacken und durchsuchen zu können.

3.1.3 Puppet-Forge

Crawlen einer menschenlesbaren Webseite

Um Informationen über Module in dem sogenannten „Puppet-Forge“[8], dem online Repository von Puppet, zu erhalten, bietet Puppet ebenso eine API[9] an, die JSON-Objekte zurück liefert. Die API eignet sich allerdings nicht dazu, alle Module des Puppet-Forge zu durchsuchen, da die Suche über alle Module nicht limitiert werden kann und so zu langsam antwortet, was zumindest bei Browsern einen Timeout-Fehler hervorruft.Zum späteren genauen Spezifizieren eines bekannten Moduls, ist die API jedoch geeignet.
Möchte man also Informationen über alle verfügbaren Module erhalten, muss der Crawler die normale, menschenlesbare Webseite[10] durchsuchen und den weiterführenden Verweise folgen.

Detailseite erkennen und Dateien speichern

Eine Anforderung an die Entwicklung bei der Erstellung von Puppet-Modulen ist das Hinzufügen des Moduls als tar.gz-Archivdatei. Diesen Umstand kann sich der Crawler zu Nutzen machen. Um an die ausführbaren Dateien zu gelangen, muss man den Inhalt der Unterseiten also nach tar.gz Archiven durchsuchen. Diese Dateien können direkt gespeichert werden und der Crawler erkennt so, dass er sich auf einer Detailseite zu einem Modul befindet..

JSON-API ansprechen und verarbeiten

Detaillierte Informationen könnte man hier auch auslesen, allerdings ist das Benutzen der API komfortabler, wenn man mit der Suchfunktion nach dem gewünschten Modul sucht. Somit erhält man nur das gewünschte Modul und umgeht den Makel der nicht vorhandenen Limitierung der API. Die Informationen des JSON-Objektes müssen ausgelesen und in den Artifact Store gespeichert werden.

Dateien verarbeiten

Das heruntergeladene Archiv enthält eine Metadaten-Datei, deren Inhalt auch in der Datenbank gespeichert werden soll. Außerdem muss das gesamte Archiv so gespeichert werden, dass es mit dem Eintrag in der Datenbank verknüpft werden kann.

[8] https://forge.puppetlabs.com/
[9] http://forge.puppetlabs.com/modules.json?q=test
[10] "http://forge.puppetlabs.com/

3.1.4 Zusammenfassung

Um Informationen und Dateien der Module finden und speichern zu können, muss ein Crawler für das Finden und Speichern aller Module der drei Konfigurationsmanagement-Tools also

- JSON verarbeiten
- Links erkennen
- Dateien speichern und entpacken
- Archive erstellen
- Bazaar-Verzeichnisse speichern
- Menschenlesbare Webseiten durchsuchen und Verweisen folgen
- HTML-Quelltext nach regulären Ausdrücken durchsuchen

können.

3.2 Speicherformat

Die Informationen über einzelne Module liegen größtenteils im JSON-Format vor. Viele Felder, wie Name, Beschreibung, Author, usw. der JSON-Objekte haben bei den Modulen verschiedener Konfigurationsmanagement-Tools die ähnliche Bedeutung. Zudem gibt es jeweils ein Paket mit ausführbaren Skripten, das auch heruntergeladen und gespeichert werden muss.
Als Speichermedium eignet sich eine Dokumenten-basierte Datenbank, da diese sowohl Daten als auch Dateien als unabhängige Container aufnehmen kann.
Die Daten müssen von einer Webseite durchsucht und angezeigt werden können. Zudem sollte eine Sortierung nach bestimmten Kategorien möglich sein. Um die verschiedenen Namen der Kategorien unterschiedlicher Tools zu berücksichtigen, sollten Benutzer eigene Kategorien anlegen und den Modulen zuweisen können. Die gesamte Funktionalität muss über die UI zur Verfügung gestellt werden.

3.3 User-Interface

Das User-Interface ist eine von Menschen bedienbare Webseite. Es muss Möglichkeiten zum Durchsuchen und Sortieren der Module geben. Zusätzlich müssen die ausführbaren Skripte der einzelnen Module speicherbar sein und eine Möglichkeit zum Exportieren des gesamten Moduls ins TOSCA-Format gegeben sein. Die UI muss auch die Möglichkeit bieten, Kategorien anzulegen und den Modulen zuzuweisen.

3.4 Export ins TOSCA Format

Die Export-Funktion muss ein TOSCA-lesbareres CSAR-Archiv erstellen. Darin enthalten sind die ausführbaren Dateien und alle Informationen über das Modul. Diese Informationen müssen im XML-Format vorliegen.
Das erstellte CSAR-Archiv muss ein TOSCA-Service Template und ein Node Template enthalten. Die erforderlichen Node Types müssen ebenfalls erstellt werden.
Außerdem muss eine Metadatei erstellt werden, die den Überblick über alle im CSAR-Archiv enthaltenen Dateien bereitstellt.

4 Architektur und Design

Der Aufbau des gesamten Projektes sollte möglichst modular sein. Das bedeutet, dass die Crawler, die Datenbank, in der gespeichert wird, und das User Interface so aufgebaut sind, dass sie möglichst unabhängig von einander sind. Es sollte also möglich sein, zum Beispiel neue Crawler hinzuzufügen oder Module auszutauschen, ohne die anderen Module bearbeiten zu müssen.

4.1 Auswahl und Aufbau der Crawler

Um die Informationen der einzelnen Module aus den Online Repositories der Konfigurationsmanagement-Tools auszulesen und auch deren ausführbare Dateien zu speichern, wird für die Diplomarbeit die Programmiersprache Java benutzt.
Den Ausschlag gab letztlich die fertig implementierte Crawling-Lösung „crawler4j“[1], die als Modul dem Projekt hinzugefügt werden kann und sich um das Auslesen der menschenlesbaren Internetseiten kümmert. Als Attribute müssen die Startadresse und Ausschlusskriterien für das Weiterverfolgen der Links eingetragen werden. Der Crawler sucht eigenständig nach weiterführenden Links auf der Startseite und verfolgt diese, wenn sie mit den vorher eingegeben Attributen nicht in Konflikt stehen. Ein Ausschlusskriterium für das Weiterverfolgen von Links sind zum Beispiel Style-Dateien wie *.css, oder Bild-Dateien wie *.jpg. Außerdem sollte eine Basis-Adresse eingegeben werden, mit der jeder Link verglichen wird und die verhindert, dass der Crawler auch fremde Webseiten untersucht.
Um die RESTful API der Tools anzusprechen und auszulesen kommt eine Standart-JSON-Bibliothek zum Einsatz, die direkt JSON Objekte aus den Daten der API erzeugt. Das unkomplizierte Einfügen dieser Module erfolgt über das Software-Management-Tool Apache Maven[2].
Der Crawler ist durch obige Module in der Lage, Informationen sowohl aus den RESTful APIs als auch direkt von der menschenlesbaren Webseite auszulesen.
Das Speichern erfolgt durch eine eigens entwickelte REST-API. An diese müssen vom Crawler sowohl die Daten als JSON-Objekt als auch die Dateien mittels HTTP-Aufruf gesendet werden.


```
{
 "summary": "Installs 1password",
 "author": "Joshua Timberman",
 "artifactType": "chef",
 "dependencies": {},
 "description": "OWN DESCRIPTION",
 "name": "2password111",
 "categories": ["applications"],
```


[1] http://code.google.com/p/crawler4j/
[2] http://maven.apache.org/

```
 "operations": ["1password/recipes/default.rb"],
 "metaData": {"maintainer_email":"cookbooks@housepub.org","groupings":{},"recipes":{},
               "replacing":{},"platforms":{"mac_os_x":">= 0.0.0"},"providing":{},"suggestions":{},
               "maintainer":"Joshua Timberman","version":"1.0.6","long_description":"Description",
               "dependencies":{},"conflicting":{},"description":"Installs 1password",
               "name":"1password1","recommendations":{},"attributes":{},"license":"Apache 2.0"},
 "version": "1.1.3",
 "url": "http://cookbooks.opscode.com/api/v1/cookbooks/1password",
 "file": "1password-1.0.6.tgz"
}
```


Beispiel für das JSON-Format, das zur REST-API gesendet wird.

4.2 Artifact Store

Als Speichermedium für die Daten aus den Online Verzeichnissen eignet sich die mongoDB[3]. Da die Daten aus den RESTful APIs schon als JSON Dokument vorliegen, macht es hier besonders Sinn, eine dokumentenbasierte Datenbank zu benutzen. Diese kann selbst verschachtelte Dokumente direkt speichern und bei einem Zugriff das gesamte Dokument zurück liefern.
Außerdem lassen sich auch die ausführbaren Dateien direkt in der Datenbank speichern. Dies ist bei mongoDB aus Programmierer-Sicht auf relativ einfache Art möglich.
Um ein einheitliches Datenformat bei den Dokumenten zu bekommen, muss man die Schlüsselwerte der Metadaten aus den Modulen der Konfigurationsmanagement-Tools zusammenfassen. Tabelle 4.1 zeigt die Felder, die in der mongoDB gespeichert werden, und welchen Feldern der Daten der Konfigurationmanagement-Tools sie jeweils entsprechen. Die Verwaltung der Daten und Dateien in der Datenbank erfolgt über eine eigens entwickelte RESTful API.
Die Daten werden vom Crawler über einen HTTP-Request an diese RestAPI gesendet. Dafür müssen die Daten in einem bestimmten JSON Format vorliegen, damit sie adäquat verarbeitet werden können. Zusätzlich zu den gemeinsamen Attributen wird das komplette Metadaten-Dokument der einzelnen Module gespeichert. Dadurch sind auch alle übrigen Felder aus Tabelle 4.1, die nicht in der mongoDB zusammengefasst wurden, trotzdem abrufbar. Bei der Programmierung der Controller der RestAPI wurde auf das „node.js“[4]-Framework gesetzt. Dieses basiert auf der JavaScript Runtime von Chrome[5], damit lassen sich in kurzer Zeit Serverapplikationen erstellen. Damit die API bei verschiedenen Pfaden in der Adressleiste verschiedene „Controller“ aufruft, muss das node.js-Webapplikationen Framework express benutzt werden. Dadurch lassen sich zum Beispiel Abfragen nach dem Namen oder dem Typ über die URL steuern.

[3] www.mongodb.org
[4] http://nodejs.org/
[5] http://code.google.com/p/v8/

mongoDB	Puppet	Chef	Juju
name	full_name	name	name
version	version	version	revision
summary	summary	description	summary
description	description	long-description	description
author	author	maintainer	maintainer
dependencies	dependencies	dependencies	requires
categories	tag-list	categories	categories
url			
artifactType			
file (in mongoDB)			
filetype			
usertags			
als Metadaten:	types	attributes	config
		recommendations	provides
	checksums	suggestions	peers
	source	conflicting	
	project-page	providing	
		replacing	
		groupings	
		recipes	
		maintainer-email	
		platforms	

Tabelle 4.1: Zusammenfassung der Metadaten-Modelle

Die linke Spalte gibt das JSON-Objekt an, das in der Datenbank gespeichert wird. Der Wert des Feldes, der gespeichert wird, ergibt sich aus dem Wert der Informationen eines Modules, der aus dem Feld in der gleichen Zeile kommt. Hinzu kommen generische Daten, wie der Dateiname des Archivs oder der Artifakttyp (zum Beispiel „chef“ oder „puppet“). Alle weiteren Daten eines Artefaktes werden in der Metadaten-Datei, die direkt übernommen wird, gespeichert.

4.3 Artifact Management User Interface

Das User Interface muss die gespeicherten Informationen und Dateien dem Benutzer bereitstellen können. Es bekommt von der RESTful API die Informationen als Listen oder Metadaten-Modell und muss diese auf einer Webseite anzeigen.
Als Sprache kommen HTML, JavaScript sowie diverse Frameworks zum Einsatz. Das Laden der Daten über die RestAPI erfolgt zum Beispiel asynchron über Ajax[6] Abfragen.
Über das User Interface können zudem eigene Attribute an die gespeicherten Module angehängt

[6] http://api.jquery.com/jquery.ajax/

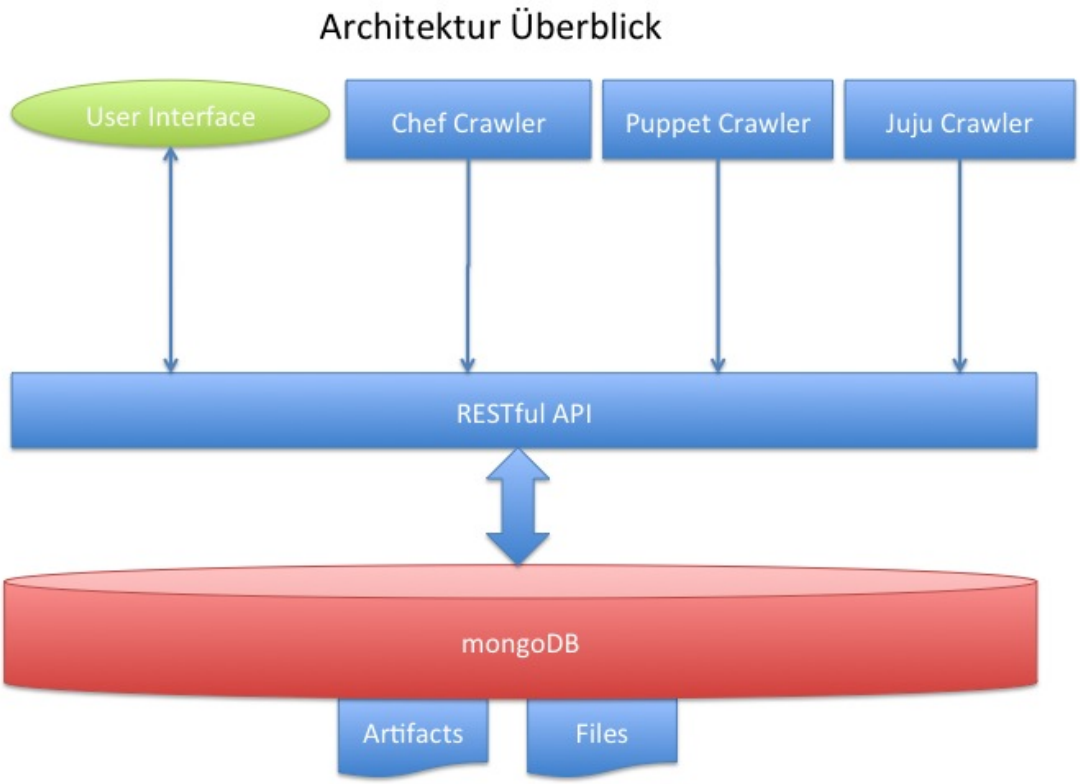

Abbildung 4.1: Projekt-Architektur

werden, die bei der Suche und Sortierung der Module in der Datenbank hilfreich sein können. Dafür müssen also auch Daten an die REST-API gesendet werden können.

5 Implementierung

5.1 Artifact Crawler

Die Crawler sind als einzelne, unabhängige Module in einem Java-Projekt implementiert. Die Projektstruktur wurde vom Build-Management-Tool Maven vorgegeben, das sich auch um die Verwaltung der Abhängigkeiten von externen Software-Projekten kümmert, die hier verwendet werden.
Im Projekt werden die externen Software-Module crawler4j[1], JSON[2], Apache httpclient[3], bazaar[4], Apache commons IO[5], JRuby[6], Jsoup[7], Apache Commons Compress[8] und Snakeyaml[9] eingesetzt. Im Internet gibt es für Maven frei verfügbare Repositories, die viele Software-Projekte beinhalten. Maven sucht automatisch in diesen Repositories nach dem erforderlichen Projekt, lädt es bei Bedarf selbstständig herunter und bindet die Projektdatei in das eigene Software-Projekt mit ein.
Die Crawler sind einzeln ausführbare Programme, die lokal als Java-Applikation gestartet oder als JAR- oder WAR-Datei auf einem Java-fähigen Server bereitgestellt werden können.

5.1.1 Chef Crawler

Crawling

Das Crawler-Modul für Chef Artefakte beginnt das Crawling in einer von Chef bereitgestellten JSON-API[10]. Mit dieser API lassen sich alle im Chef-Verzeichnis gespeicherten Cookbooks finden. Man kann der API sowohl einen Startwert als auch eine maximale Anzahl an Ergebnissen übergeben und so durch die komplette Liste an Cookbooks navigieren. Die schon in Kapitel 2 vorgestellte Funktion *readJsonFromUrl()*

[1] https://code.google.com/p/crawler4j/
[2] http://www.json.org/java/
[3] http://hc.apache.org/httpclient-3.x/
[4] https://launchpad.net/bzr-java-lib
[5] http://commons.apache.org/proper/commons-io/
[6] http://jruby.org/
[7] http://jsoup.org/
[8] http://commons.apache.org/proper/commons-compress/
[9] http://www.snakeyaml.org
[10] https://cookbooks.opscode.com/api/v1/cookbooks

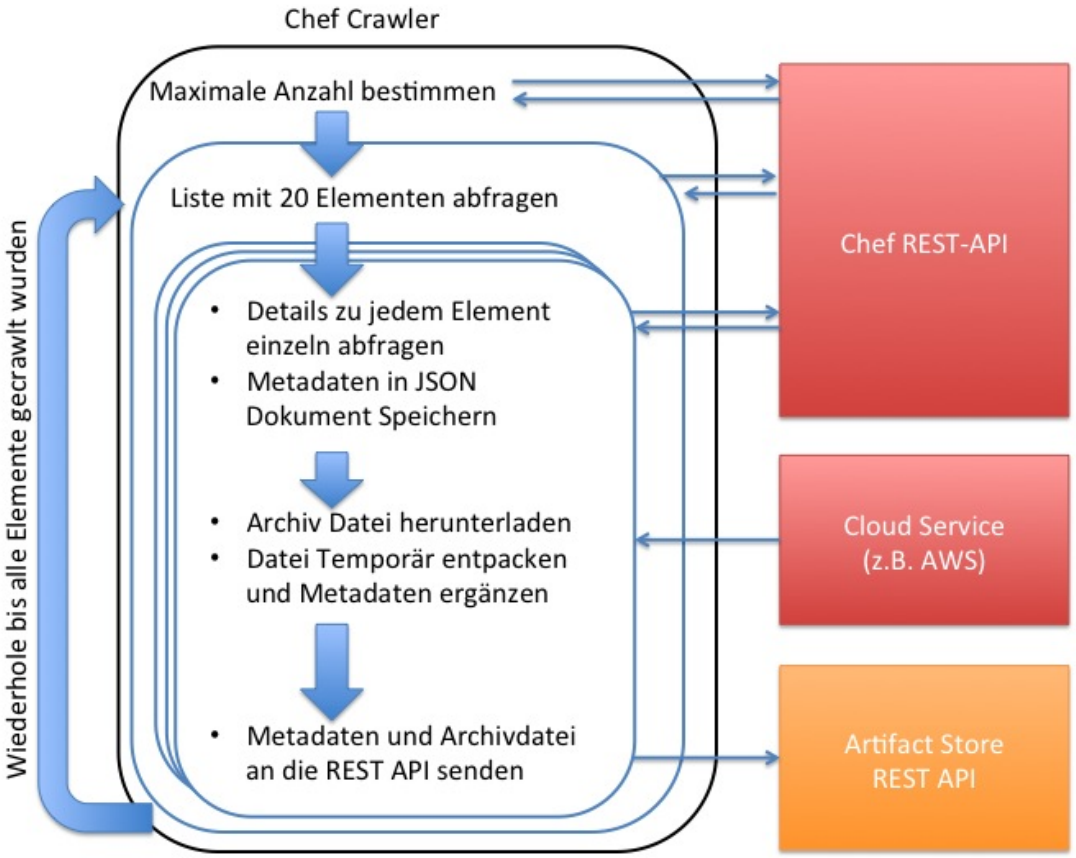

Abbildung 5.1: Chef Crawler Ablauf Diagramm

```
private static JSONObject readJsonFromUrl(String url) throws IOException, JSONException {
  InputStream is = new URL(url).openStream();
  try {
    BufferedReader rd = new BufferedReader(new InputStreamReader(is, Charset.forName("UTF-8")));
    String jsonText = readAll(rd);
    JSONObject json = new JSONObject(jsonText);
    return json;
  } finally {
    is.close();
} }
```

Listing 5.1: JSON aus einer URL auslesen

verwandelt zuerst den Inhalt einer URL in einen *Java.io.InputStream*. Dieser Inputstream wird an einen *java.io.BufferedReader* übergeben, um ihn schließlich mit der *readAll()*-Funktion Zeichen für Zeichen in einen String zu kopieren.

```
private static String readAll(Reader rd) throws IOException {
   StringBuilder sb = new StringBuilder();
```

```
   int cp;
   while ((cp = rd.read()) != -1) {
      sb.append((char) cp);
   }
   return sb.toString();
}
```

Die *readAll()*-Funktion benutzt dazu die *java.lang.StringBuilder*-Klasse, mit der Strings mittels *append()* zusammengesetzt werden können. Das Ergebnis ist ein String, der den Inhalt der kompletten Webseite enthält. Dieser String wird direkt in ein *org.json.JSONObject* konvertiert und schließlich zurückgegeben.

```
public static void main(String[] args) throws Exception {
            String url = "https://cookbooks.opscode.com/api/v1/cookbooks?start=0&items=0";
           JSONObject getTotalNumber = readJsonFromUrl(url);
            int total=getTotalNumber.getInt("total");
           int start=0;
```

Der Chef Crawler beginnt das Speichern der Informationen, indem zuerst die maximale Anzahl an verfügbaren Cookbooks ausgelesen wird. Dazu wird eine Abfrage auf eine eigentlich leere JSON-Liste von Cookbooks gemacht. Diese enthält jedoch die maximale Anzahl der verfügbaren Cookbooks als Element und kann somit in eine Variable gespeichert werden.
Die Funktion *getInt()* eines JSONObjects sucht dazu im gesamten JSONObject nach dem entsprechenden Schlüssel und liefert diesen als Wert zurück.

```
           while (start<total){
                 String url2
                     ="https://cookbooks.opscode.com/api/v1/cookbooks?start="+start+"&items=20";
                 JSONObject allJSONObjects = readJsonFromUrl(url2);
                 int gecrawlt=0;
                 JSONArray nestedArray = allJSONObjects.getJSONArray("items");
                 for (int i=0; i<nestedArray.length(); i++) {
                       ...
                       gecrawlt++;
                 }
                 start=start+gecrawlt;
           }
      }
      ...
```

Das eigentliche Auslesen der Informationen findet in der *while*-Schleife statt. Es werden hier immer 20 Cookbooks pro Aufruf ausgelesen. Der Startwert erhöht sich nach jedem Durchgang um die Anzahl der gespeicherten Artefakte. Eine *for*-Schleife durchsucht die einzelnen Elemente schließlich genau und speichert die Inhalte entweder als String oder JSONArray-Variablen:

```
      ...
for (int i=0; i<nestedArray.length(); i++) {
                    JSONObject item = nestedArray.getJSONObject(i);
                    String name = item.getString("cookbook_name");
                    String curl = item.getString("cookbook");
                    JSONObject innerJSONObject = readJsonFromUrl(curl);
                    ...
```

Um mehr Informationen über einzelne Artefakte zu erhalten, muss man den Wert im Schlüssel „cookbook“ als neue URL aufrufen und das JSONObject speichern.

```
...
 String latest_version = innerJSONObject.getString("latest_version");
 JSONObject latestVersionJSONObject = readJsonFromUrl(latest_version);
...
```

Da auch Informationen, wie die Version und der Dateiname, gespeichert werden sollen, muss man in der JSON-API von Chef noch eine Ebene tiefer suchen und den Wert im Schlüssel „latest_version“ als neue URL aufrufen und als eigenes JSONObject speichern.

```
...
 String file = latestVersionJSONObject.getString("file");
 String version=latestVersionJSONObject.getString("version");
 JSONArray categories=new JSONArray();
 if (!innerJSONObject.isNull("category")){
    categories.put(innerJSONObject.getString("category"));
 }

 URL localurl=new URL(file);
 String fileName=getLastBitFromUrl(localurl.getFile());
 String saveName=fileName.substring(0, fileName.indexOf('.'));
 String extension=fileName.substring(fileName.indexOf('.'),
      fileName.length());
...
```

Alle Informationen aus der REST-API von Chef sind jetzt in Variablen zwischengespeichert und bereit zur weiteren Verarbeitung.

```
...
  ReturnType ret= saveChefFileToREST(file,version);
...
```

Bevor diese Daten an die REST-API zum Speichern übergeben werden, muss die Datei mit den ausführbaren Dateien heruntergeladen und analysiert werden. In ihr befinden sich Informationen über die möglichen Operationen des Cookbooks. Dies erfolgt über die *saveChefFileToREST()* Funktion, die weiter unten beschrieben wird. Zurück liefert diese Funktion einen Variable, bestehend aus den Operationen und den Metadaten des Artefaktes.

```
...
  JSONObject metajson=null;
  try{
   metajson = new JSONObjectIgnoreDuplicates(ret.metaData);
      String summary=null;
      if (metajson.has("description")){
            summary=metajson.getString("description");}
      String description=null;
      if(metajson.has("long_description")){
            description=metajson.getString("long_description");};
      String author=metajson.getString("maintainer");
      JSONObject dependencies=metajson.getJSONObject("dependencies");
      String dependenciesString=dependencies.toString();
```

```
                ...
```

Aus dieser Variable werden die letzten wichtigen Informationen gewonnen, die für die spätere Verarbeitung wichtig sind. Der Konstruktor *JSONObjectIgnoreDuplicates(ret.metaData)* erzeugt aus dem Metadaten-String ein neues JSONObject und ignoriert dabei doppelt vorkommende Werte bei den Schlüsseln im Metadaten-Objekt. Andernfalls würde bei einem Fehler in der Ausgangsdatei eine Ausnahmebehandlung greifen und das Crawling beenden.

```
                ...
                   JSONObject allWorking=new JSONObject().
                                   put("name",name).
                                   put("summary", summary).
                ...
                                   put("metaData",new JSONObject(ret.metaData));

                    int result=saveJSONtoREST(allData);
                   gecrawlt++;
               } catch (JSONException e) {
                       continue;
                              }
                }
         start=start+gecrawlt;
}
```

Listing 5.2: Chef Crawler

Abschließend wird ein neues JSONObject angelegt und mit den Informationen aus den vorher gespeicherten Variablen gefüllt. Dieses Objekt wird über die *saveJSONtoREST()*-Funktion an die REST-API gesendet. Der Zähler wird entsprechend erhöht.

Dateien laden, analysieren und speichern

Die Funktion *saveChefFileToREST()* lädt von einer URL die Artefakt-Datei herunter, übergibt sie an die REST-API und durchsucht sie schließlich nach den enthaltenen Operationen und der Metadaten-Datei.

```
private static ReturnType saveChefFileToREST(String filename, String version) throws IOException{
       URL url=new URL(filename);
       ReturnType rType=new ReturnType();
       ArrayList<String> operations=new ArrayList<String>();
       String fileName=getLastBitFromUrl(url.getFile());
       String saveName=fileName.substring(0, fileName.indexOf('.'));
       String extension=fileName.substring(fileName.indexOf('.'), fileName.length());
       File dir=new File(saveName+"-"+version+""+extension);
       FileUtils.copyURLToFile(url, dir);
```

Der Dateiname wird der Funktion als vollständige URL übergeben, daraus muss zum Speichern der eigentliche Name extrahiert werden. Die *org.apache.commons.io.FileUtils*-Klasse hilft beim Speichern einer Datei aus dem Internet.

```
   CloseableHttpClient httpClient = HttpClientBuilder.create().build();
   HttpPost httpPost = new HttpPost("http://localhost:3000/files");
   try
      {
      MultipartEntityBuilder multipartEntityBuilder = MultipartEntityBuilder.create();
      multipartEntityBuilder.setMode(HttpMultipartMode.BROWSER_COMPATIBLE);
      multipartEntityBuilder.addPart("file", new FileBody(dir));
      httpPost.setEntity(multipartEntityBuilder.build());
      HttpResponse httpResponse = httpClient.execute(httpPost);
      if(httpResponse != null) {
                           System.out.println("Verbindung erfolgreich");
      } else {
                           return null;
       }
} catch (IOException e) {
  return null;
} finally {
   httpClient.close();
}
```

Listing 5.3: Datei an die REST-API senden

Nach dem Herunterladen der Artefakt-Datei wird diese direkt an die eigene REST-API übergeben. Mit der *org.apache.http.client.methods.HttpPost*-Klasse können HTTP-Verbindungen zu Servern aufgebaut werden. Um eine Datei zu senden, muss allerdings eine „Multipart-Entity“, also eine Nachricht, die verschiedene Typen an Informationen enthält[mul], an den Server gesendet werden. Die zu sendende Nachricht muss hier die Datei, aber auch Informationen über den Dateityp und den Dateinamen enthalten. Diese Informationen werden mit dem *org.apache.http.entity.mime.MultipartEntityBuilder* der zu sendenden Entity hinzugefügt. Der HTTP-POST, mit der Datei als Inhalt, wird schließlich mit einem *org.apache.http.impl.client.CloseableHttpClient* versendet und es kann mit der Analyse begonnen werden.

```
TarInputStream tarInput=null;
if (extension.equalsIgnoreCase(".tar") ){
      tarInput = new TarInputStream(new FileInputStream(dir));
}else{
      tarInput=new TarInputStream(new GZIPInputStream(new FileInputStream(dir)));
}
```

Abhängig von der Dateiendung wird entschieden, wie der *org.apache.commons.compress.tar.TarInputStream* erstellt wird.

```
[caption={Artefakt-Archiv analysieren}]
TarEntry entryx = null;
String actualDir="";
String jsonTxt = null;
while((entryx = tarInput.getNextEntry()) != null) {
      System.out.println(entryx.getName());
   if (entryx.isDirectory()) {
      // entryx ist ein Verzeichnis, durchforste weiter..
   }
   else {  //entryx ist eine Datei:
        if ( entryx.getName().endsWith(".rb") && entryx.getName().contains("recipes")) )
```

```
       // im "recipes" Verzeichnis
             {   System.out.println("Recipes: "+entryx.getName());
            operations.add(entryx.getName().toLowerCase());
       }else
            if ( entryx.getName().endsWith("metadata.json")){
                        FileOutputStream fout = new FileOutputStream("tmp.json");
                        tarInput.copyEntryContents(fout);
                  fout.close();
                        InputStream is = new FileInputStream("tmp.json");
                        jsonTxt = IOUtil.toString(is);
     }
   }
}
```

Der erstellte TarInputStream kann wie ein normales Verzeichnis durchforstet werden. Man iteriert mit einer *while*-Schleife über alle Elemente, die sich im Artefakt-Archiv befinden. Alle Elemente, die sich im „recipes“-Verzeichnis befinden und mit Ruby ausführbar sind, werden als Operationen gespeichert und so später in die Datenbank eingefügt. Diese Werte sind wichtig für das spätere Exportieren des Artefaktes.
Desweiteren wird nach der „metadata.json“-Datei, die jedes Cookbook enthalten muss, gesucht und deren Inhalt in einer Variablen gespeichert.

```
tarInput.close();
dir.delete();
rType.operations= operations;
rType.metaData=jsonTxt;
return rType;
}
```

Beide Werte werden zusammen zurückgegeben.

An die REST-API senden

Nachdem das JSONObject generiert wurde, kann es mit einem *HttpPost* an die REST-API gesendet werden.

```
CloseableHttpClient httpClient = HttpClientBuilder.create().build();
HttpPost request = new HttpPost("http://localhost:3000/artifacts");
StringEntity params = new StringEntity(json.toString());
request.addHeader("content-type", "application/json; charset=utf-8");
request.setEntity(params);
httpClient.execute(request);
```

Listing 5.4: JSON Dokument an die REST-API senden

Der Header des *HttpPosts* muss dabei den richtigen „Content-Type“ haben, damit die REST-API die Daten auch verarbeiten kann.

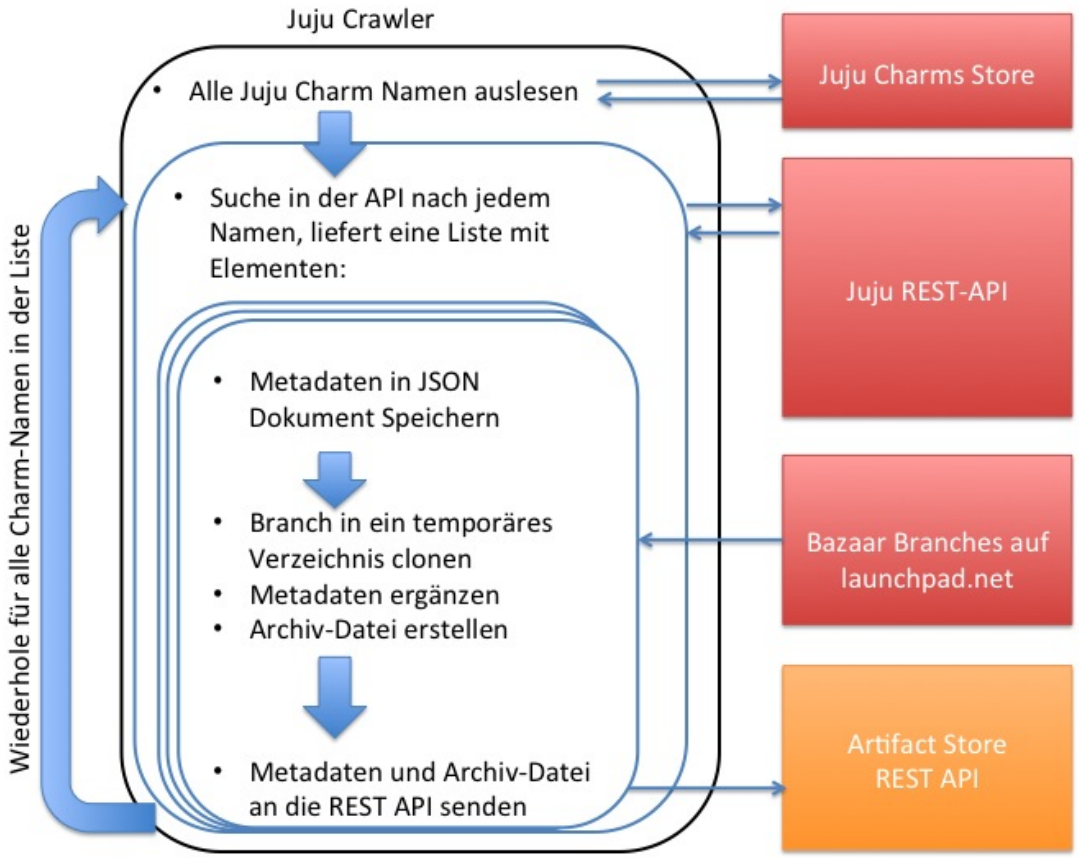

Abbildung 5.2: Juju Crawler Ablauf Diagramm

5.1.2 Juju Crawler

Crawling

Um an alle Juju Charms zu gelangen, muss man die Startseite des Charms Browsers[11] aufsuchen. Dort sind Links zu allen Charms aufgelistet. Da man keine Links verfolgen muss, reicht das Parsen und Analysieren der Webseite. Der Java-HTML-Parser jSoup[12] erstellt aus dem HTML-Quellcode der Seite ein Dokument, das nach Elementen durchsucht werden kann:

```
Document doc = Jsoup.parse(new URL("https://manage.jujucharms.com/charms"), 2000);
            Elements links = doc.select("a[href]");
            for (Element link : links) {
               String charm= link.text();
            ...}
```

[11]http://manage.jujucharms.com/charms
[12]http://jsoup.org/

Eine Schleife durch alle Links übergibt deren Namen an die Juju Charms REST-API[13], die nach allen Elementen mit dem entsprechenden Namen sucht und sie als JSON-Objekte zurück liefert. Durch diese iteriert man auf die gleiche Weise wie beim Chef Crawler. Allerdings muss man hier nach einer Abfrage der API keine weiteren Abfragen vornehmen, um nach Informationen zu suchen, da diese schon nach dem ersten Abrufen der Seite enthalten sind. Auch die Operationen, die bei Chef aus den Dateinamen abgeleitet werden, sind in dem JSON-Objekt der REST-API enthalten.
Die Informationen werden entsprechend der Tabelle 4.1 in ein JSON-Objekt für die Datenbank gespeichert.

Dateien laden, analysieren und speichern

Im Gegensatz zum Crawling der Chef-Artefakte liegen die Dateien der Charms nicht direkt zum Download auf einer Internetseite, sondern sind in dem „Code-Module“ auf Launchpad[14] zu finden. Launchpad ist ein von Canonical verwalteter Dienst, der vor allem das gemeinsame Entwickeln an Ubuntu-Modulen unterstützen soll. Auf Launchpad gibt es, ähnlich zu Github[15], eine Sektion, um Code versioniert zu verwalten. Dafür benutzt Launchpad das von Canonical selbst entwickelte Versionsverwaltungssystem Bazaar[16]. Hier sind auch die Charms gespeichert.
Um die Dateien speichern zu können, muss man also eine Verbindung zu dem entsprechenden „Bazaar Branch“ herstellen und die Dateien auschecken:

```
public static String bazaar(String path,String fileName,String revisionString) throws IOException
    {
      ...
            final String BZR_BIN_PATH = "/usr/local/Cellar/bazaar/2.6.0/libexec/bzr";
      ...
```

Um mit Java einen Bazaar Branch zu speichern, muss ein Bazaar Client auf dem Rechner installiert sein.

```
      ...
            File dir = new File(BRANCH_LOCAL_PATH+fullPath);
            File newDir= new File(BRANCH_LOCAL_PATH+lastDirOfPath);
            if (dir.isDirectory()) {
                  deleteRecursive(dir); //loesche die zwischengespeicherten Branches
                  newDir.mkdirs();
                  BRANCH_LOCAL_PATH=BRANCH_LOCAL_PATH+fullPath;
            }else{
                  newDir.mkdirs();
              BRANCH_LOCAL_PATH=BRANCH_LOCAL_PATH+fullPath;
              }
      BazaarClientPreferences.getInstance().set(BazaarPreference.EXECUTABLE, BZR_BIN_PATH);

      try {
            CommandLineClientFactory.setup(false);
```

[13] https://manage.jujucharms.com/api/2/charms?limit=20
[14] http://launchpad.net/
[15] https://github.com/
[16] http://bazaar.canonical.com/en/

```
        BazaarClientFactory.setPreferredClientType(CommandLineClientFactory.CLIENT_TYPE);
        IBazaarClient client =
            BazaarClientFactory.createClient(BazaarClientFactory.getPreferredClientType());
        BranchLocation branchLocation = new BranchLocation(URI.create(BRANCH_URL));
        BazaarRevision revision = client.revno(branchLocation);
        client.branch(branchLocation, new File(BRANCH_LOCAL_PATH), revision, new Option(""));
...
```

Es wurde ein neues Verzeichnis angelegt, in das die Java-Bazaar-Bibliothek[17] den Branch von der Launchpad URL speichern kann.

```
        try{
              String tarGzPath = "archive.tar";
              fOut = new FileOutputStream(new File(tarGzPath));
              bOut = new BufferedOutputStream(fOut);
              gzOut = new GzipCompressorOutputStream(bOut);
              tOut = new TarArchiveOutputStream(gzOut);
              addFileToTarGz(tOut, BRANCH_LOCAL_PATH, "");
               } finally {
              tOut.finish();
              tOut.close();
              gzOut.close();
              bOut.close();
              fOut.close();
               }
         File archive=new File("archive.tar");
         File file2 = new File(fileName+"-"+revisionString+".tar");
         if(file2.exists()) throw new java.io.IOException("file exists");
           // Rename file (or directory)
         boolean success = archive.renameTo(file2);
         try
         {
...
              HttpResponse httpResponse = httpClient.execute(httpPost);
              if(httpResponse != null) {
                     file2.delete();
              }
```

Das heruntergeladene Verzeichnis wird in eine Datei mit dem Artefaktnamen als Dateiname gepackt, mit der gleichen Funktion wie beim Chef-Crawler an die eigene REST-API zum Speichern in der Datenbank gesendet und bei erfolgreicher Übermittlung gelöscht. Das Komprimieren in ein TAR-GZ-Archiv erfolgt über einen *GzipCompressorOutputStream*, der einem *TarArchiveOutputStream* übergeben wird. Alle Streams müssen nach jedem Artefakt wieder geschlossen werden.

```
                        } catch (IOException e) {
            e.printStackTrace();
        }
        String metadata=findYamlFile("metadata.yaml",new File(BRANCH_LOCAL_PATH));
        return metadata;
    } catch (BazaarClientException e) {
```

[17] https://launchpad.net/bzr-java-lib

```
            e.printStackTrace();
        }
        return null;
}
```

Listing 5.5: Juju Crawler

Abschließend wird in dem temporären Verzeichnis mit dem Branch-Inhalt nach der Metadaten Datei („metadata.yaml“), die hier im YAML-Format vorliegt, gesucht und deren Inhalt als String zurückgeliefert. Das temporäre Verzeichnis kann jetzt ebenfalls gelöscht werden.

An die REST-API senden

Das Senden an die REST-API erfolgt wie beim Crawler für Chef über einen HTTP-POST mit einem JSON-Objekt, das alle Informationen über das Charm enthält, als „Body-Teil“.

```
Yaml yaml = new Yaml();
Map<String, Object> object = (Map<String, Object>) yaml.load(metaData);
```

Beim Einfügen des YAML-formatierten Strings in das JSON-Objekt, das zur REST-API geschickt wird, muss man den Inhalt zuerst in eine *java.util.map* verwandeln. Die Java Bibliothek Snakeyaml[18] hilft hier bei der Umwandlung.

5.1.3 Puppet Crawler

Crawing

Um an Informationen und Dateien der Artefakte für Puppet zu gelangen, muss man die menschenlesbare Seite https://forge.puppetlabs.com/modules und deren Unterseiten vollständig durchsuchen. Puppet bietet auch eine REST-API an, die JSON Dokumente zurück liefert, allerdings eignet sich diese bisher nur zur Vervollständigung der Informationen über einzelne, ausgewählte Artefakte, da es dort nicht möglich ist, durch die API zu iterieren und der einmalige Aufruf aller Artefakte aktuell zu einem Timeout im Browser führt.
Um also ausgehend von der Startseite an alle Puppet-Artefakte zu gelangen, muss man auf jeder Seite nach weiterführenden Links suchen und diese verfolgen. Das muss solange wiederholt werden, bis jeder Link einmal besucht wurde.
In Java gibt es genau für diesen Anwendungsfall das Software-Modul crawler4j[19]. Dieses Modul sucht und verfolgt selbstständig die Links auf jeder Seite. Es lässt sich entweder als Bibliothekdatei dem Projekt hinzufügen oder man setzt in Maven eine Abhängigkeit.
Um eine crawler4j-Instanz zu implementieren, muss man sowohl eine Unterklasse der Klasse „edu.uci.ics.crawler4j.crawler.WebCrawler“, die den eigentlichen Crawling-Prozess steuert, als auch eine Controller-Klasse, welche die Crawler startet und konfiguriert, erstellen und bearbeiten.

[18] http://www.snakeyaml.org
[19] https://code.google.com/p/crawler4j/

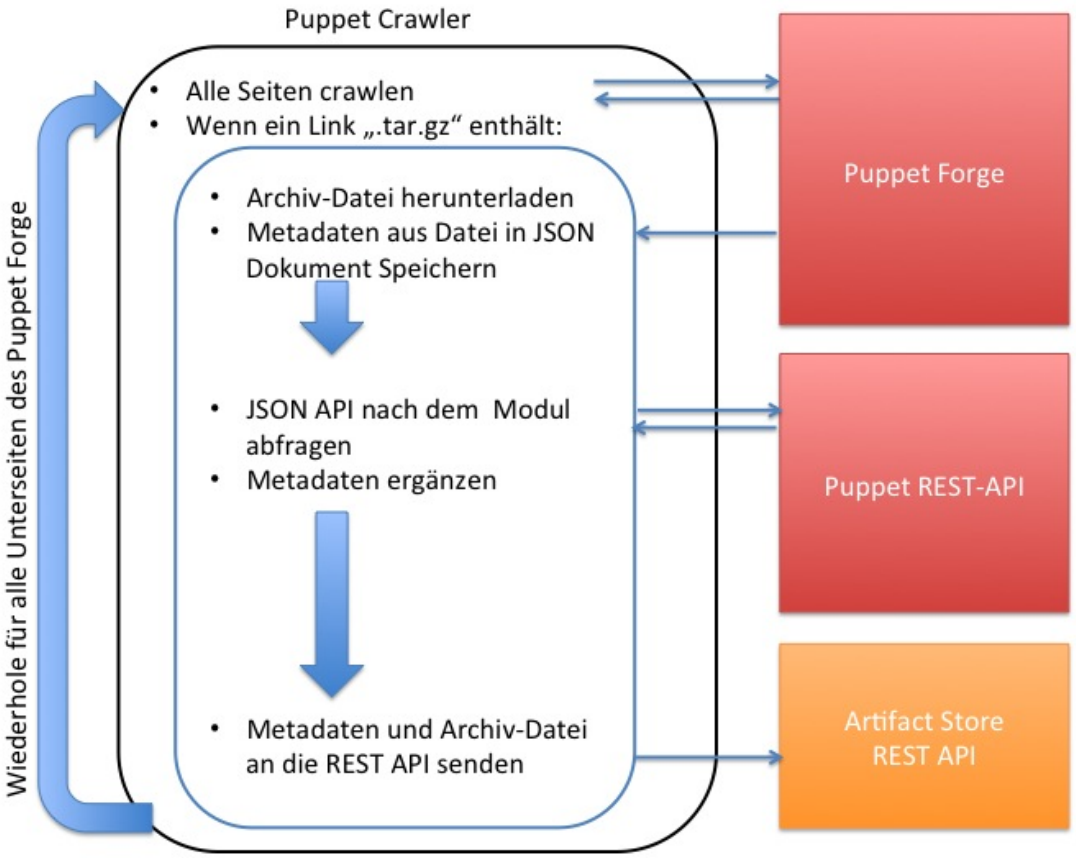

Abbildung 5.3: Puppet Crawler Ablauf Diagramm

```
public class PuppetCrawlerController {

      public static void main(String[] args) throws Exception {
      String crawlStorageFolder = "/Temp/";
      int numberOfCrawlers = 1;

      CrawlConfig config = new CrawlConfig();
      config.setCrawlStorageFolder(crawlStorageFolder);
      config.setIncludeHttpsPages(true);
      PageFetcher pageFetcher = new PageFetcher(config);

      RobotstxtConfig robotstxtConfig = new RobotstxtConfig();
      robotstxtConfig.setUserAgentName("Mozilla");
      RobotstxtServer robotstxtServer = new RobotstxtServer(robotstxtConfig, pageFetcher);
      CrawlController controller = new CrawlController(config, pageFetcher, robotstxtServer);

      controller.addSeed("https://forge.puppetlabs.com/modules?sort=rank");
      controller.start(MyPuppetCrawler.class, numberOfCrawlers); }}
```

Listing 5.6: Puppet Crawler Controller

Im Controller müssen das temporäre Verzeichnis zum Zwischenspeichern, die Anzahl an gleichzeitig aktiven Crawling-Threads, der User-Agent, mit dem sich der Crawler identifizieren lässt, und die Startseite, bei der das Crawling beginnen soll, konfiguriert werden.
Anschließend startet der Controller den Webcrawler. Die Unterklasse von *WebCrawler.java* sollte die beiden Funktionen *shouldVisit()* und *visit()* überschreiben. Die Funktion *shouldVisit()* ist für das Begrenzen des Crawling-Prozesses zuständig. Dazu bedient sie sich einerseits einem regulären Ausdruck, der verschiedene Datei-Endungen enthält, deren Seiten nicht gecrawlt werden sollen und andererseits dem generellen Anfang der URLs, die gecrawlt werden sollen:

```
public class MyPuppetCrawler extends WebCrawler {
      private final static Pattern FILTERS = Pattern.compile(".*(\\.(css|js|bmp|gif|jpe?g"
         + "|png|tiff?|mid|mp2|mp3|mp4"
         + "|wav|avi|mov|mpeg|ram|m4v|pdf"
         + "|rm|smil|wmv|swf|wma|zip|rar|gz))$");

@Override
public boolean shouldVisit(WebURL url) {
String href = url.getURL().toLowerCase();
return !FILTERS.matcher(href).matches() && href.startsWith("https://forge.puppetlabs.com/");
}

@Override
public void visit(Page page) { }
}
```

Sind die Bedingungen erfüllt, wird die *visit()*-Funktion aufgerufen und die bereits geparste Webseite übergeben.

```
String url = page.getWebURL().getURL();
if (page.getParseData() instanceof HtmlParseData) {
HtmlParseData htmlParseData = (HtmlParseData) page.getParseData();
String text = htmlParseData.getText();
String html = htmlParseData.getHtml();
List<WebURL> links = htmlParseData.getOutgoingUrls();
}
```

Die Variablen „text“ und „html“ enthalten den Inhalt der Webseite, allerdings ist für unsere Zwecke nur die Liste mit den ausgehenden Links von Bedeutung. Jede Beschreibungsseite eines Puppet-Artefakts enthält einen Link auf die gepackte Artefaktdatei. Es genügt somit, in den ausgehenden Links nach diesen Datei-Endungen zu suchen und die URL zu speichern:

```
for (int i=0; i<links.size();i++)
{
      if(links.get(i).toString().contains("tar.gz")){
            String file=links.get(i).toString();
            try {
                  URL localurl=new URL(file);
                  ReturnType rt=savePuppetFileToREST(file);
                  JSONObject metajson=new JSONObject(rt.metaData);
                  }
      }
}
```

Mit einer ähnlichen *savePuppetFileToREST(file)*-Funktion wie beim Chef Crawler wird die Datei heruntergeladen und analysiert. In der *while*-Schleife beim Analysieren der Datei wird zusätzlich zu den .rb-Ruby-Dateien wie bei Chef auch nach .sh-Dateien gesucht und diese werden als Operationen gespeichert und zurück geliefert. Außerdem befinden sich hier alle ausführbaren Dateien in einem „files“-Unterordner.
Neben den Operationen wird auch nach der metadata.json-Datei gesucht und deren Inhalt als String von der Funktion zurückgeliefert. Aus dem String wird ein JSON-Objekt konstruiert, das fast alle wichtigen Informationen über das Artefakt enthält.
Neben der Suche nach den Dateien wird die gepackte Datei auch an die REST-API zur Speicherung gesendet. Dies erfolgt auf die gleiche Weise wie bei den zuvor beschriebenen Crawlern.

```
JSONObject dependencies=new JSONObject();
for(int j = 0; j < metajson.getJSONArray("dependencies").length(); j++)
{
  JSONObject depObject = metajson.getJSONArray("dependencies").getJSONObject(j);
  String depName = depObject.getString("name");
  String depVer = depObject.getString("version_requirement");
  dependencies.put(depName, depVer);
}
```

Die JSON Daten müssen teilweise noch angepasst werden, da hier zum Beispiel die Abhängigkeiten als JSON-Array aus der Metadaten-Datei kommen, aber während der Implementierung der ersten beiden Crawler, das Speicherformat für Abhängigkeiten als eine Liste von JSON-Objekten festgelegt wurde.

```
JSONObject dataFromApi=getJSONData(name,author1,author2);
JSONArray categories=new JSONArray();
if (dataFromApi!=null){
categories=dataFromApi.getJSONArray("tag_list");}
```

Die Kategorien, die dem Artefakt zugeordnet werden, sind nicht in der Metadaten-Datei enthalten. Hier muss man die Daten über die REST-API von Puppet abfragen. Dies erfolgt über einen ähnlichen Zugriff wie schon beim Chef (Kapitel 5.1.1) und Juju (Kapitel5.1.2) Crawler gezeigt, allerdings müssen hier, bedingt durch teilweise inkonsistente Benennung des Autors, der Autor Name aus dem Dateinamen abgeleitet und der Autor Name aus den Metadaten bei der Suche berücksichtigt werden. Da die Suche in der REST-API wahrscheinlich mehrere Ergebnisse liefert und aus dem JSON-Array mit den Ergebnissen das richtige JSON-Objekt noch bestimmt werden muss, vergleicht man in der Funktion *getJSONData(name, author1, author2)* sowohl den Namen, als auch die beiden möglichen Autor Namen mit dem Objekt. Zurück liefert die Funktion ein JSON-Objekt, das man nach der „tag_list“ durchsuchen kann.

An die REST-API senden

```
JSONObject allWorking=new JSONObject().
            put("name",name).
      ...
            put("file", fileName).
            put("metaData",new JSONObject(rt.metaData.replace(".", "dot")));
```

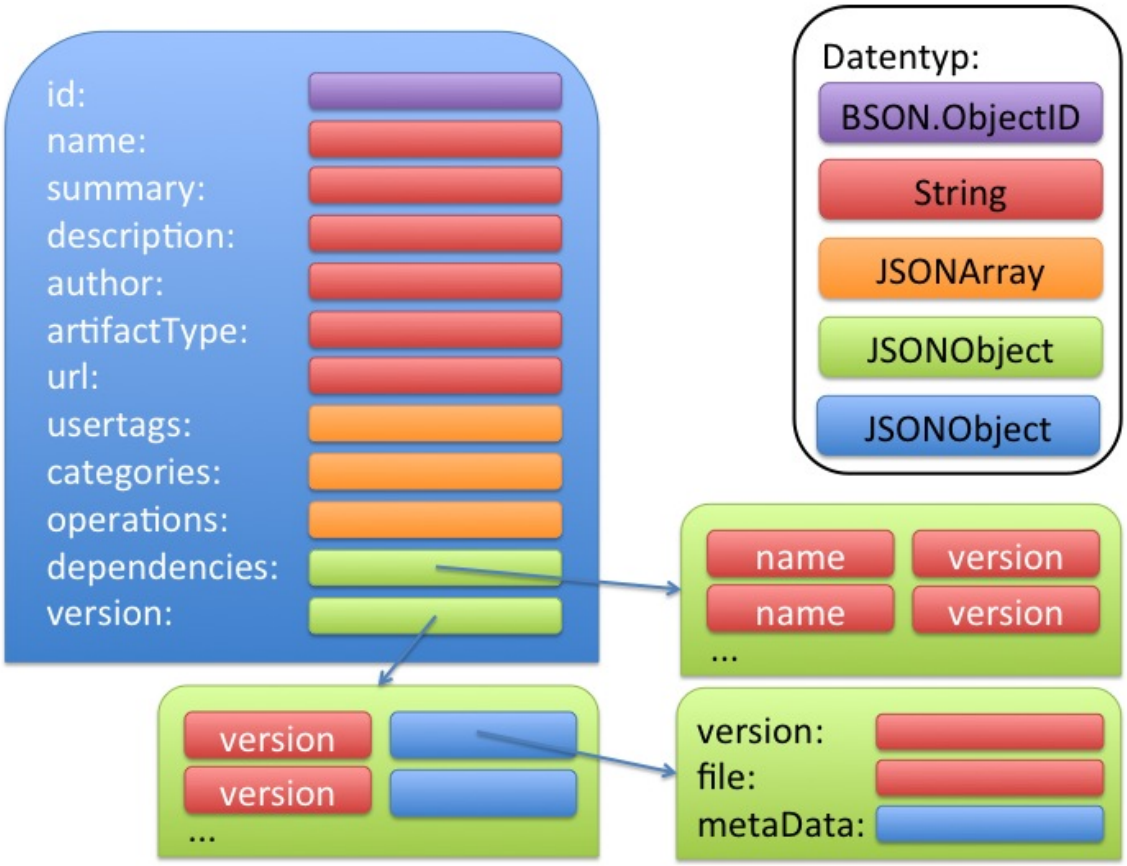

Abbildung 5.4: Datenstruktur eines Objektes im Artifact Store

```
int result=saveJSONtoREST(allWorking);
```

Listing 5.7: Puppet Crawler

Das fertig erstellte JSON-Objekt wird schließlich wie bei den zuvor beschriebenen Crawlern mit der Funktion *saveJSONtoREST()* an die REST-API übergeben.

5.2 Artifact Store

Die gecrawlten Artefakte der Konfigurationsmanagement-Tools werden in einer Datenbank gespeichert. Die Anbindung erfolgt über eine selbst erstellte REST-API. Diese liefert JSON Dokumente als Suchergebnisse zurück und kann sowohl die Dateien der Artefakte speichern, als auch deren Informationen aus JSON Dokumenten in der Datenbank ablegen.
Angesteuert wird die REST-API über HTTP-Requests.

5.2.1 Datenbank

Die Datenbank ist eine mongoDB[20]. Diese speichert die Informationen über alle Artefakte als Dokumente in einer „artifacts-Collection“. Das hat den Vorteil, dass man eine konsistente Dokumentstruktur in der Datenbank hat, was die spätere Ausgabe im User Interface erleichtert. Außerdem muss man bei der Suche nach bestimmten Kategorien keine Mapping Konstrukte erstellen, was einem SQL-Join nahe kommt.
Zusätzlich zur „artifacts-Collection“ gibt es mit GridFS[21] eine „artifacts-files-Collection“, um die Dateien zu speichern. Diese Collection besteht dabei eigentlich aus zwei getrennten Collections, da GridFS die Dateien, in sogenannte „chunks“ zerlegt, speichert, auf die von der files-Collection zugegriffen wird.
Die Installation erfolgt geräteabhängig. Nach dem Start ist der mongoDB-Server standardmäßig über localhost:27017 erreichbar. Die Datenbank und die Collections werden über die REST-API angelegt.

5.2.2 REST-API

Die Implementierung der REST-API erfolgte mit node.js[22] und express[23]. Node.js ist ein Framework, das JavaScript serverseitig einsetzt. Es basiert auf der für den Chrome-Browser[24] entwickelten JavaScript Engine „V8“, wodurch es Ressourcen sparen und schnell arbeiten soll.
Node.js wurde so konzipiert, dass man auf schnelle Weise eine einfache Server-Applikation erstellen und betreiben kann. Der Server wartet auf Client Anfragen und ist besonders auf bidirektionale Kommunikation zwischen Client und Server ausgelegt. Der Client kann also über Änderungen am Server informiert werden, was besonders dynamische Webappliaktionen in Kombination mit HTML5[25] ermöglicht.
In vorliegenden Fall benutzen wir lediglich die Server Funktionalität, um Anfragen an die Datenbank und Antworten an das User-Interface zu senden.
Das Node.js Framework besitzt den Node-Paketmanager npm[26], der eine schnelle Installation verschiedenster Pakete, wie zum Beispiel dem mongoDB Treiber oder Archivierungsmodulen ermöglicht.
Das „Express“-Modul wird ebenfalls über den Node-Paketmanager installiert. Dieses Modul ermöglicht, dass die Server-Applikation über unterschiedliche Pfade der URL verschiedene Funktionen ausführen kann. So liefert zum Beispiel „http://localhost:3000/artifacts/type/chef/“ alle Chef-Artefakte, während „http://localhost:3000/categories?filter=best“ die häufigsten Kategorien der Artefakte zurückliefert.
Bei der Erstellung der Server-Applikation kann man dem Express-Modul über den „app.get()“ Befehl Pfaden eine Funktion zuweisen:

[20] http://www.mongodb.org/
[21] http://docs.mongodb.org/manual/core/gridfs/
[22] http://nodejs.org/
[23] http://expressjs.com/
[24] https://www.google.de/intl/de/chrome/browser/
[25] http://www.w3.org/TR/html5/
[26] https://www.npmjs.org/

```
var express = require('express');
saveSkript=require('./routes/save');
allcolls= require ('./routes/allcolls');
exportSkript=require('./routes/export');
var app = express();

app.configure(function(){
  app.use(express.bodyParser({limit: '50mb'}));
  app.use(app.router);
});
app.get('/artifacts/type/:type/', allcolls.findByType);
```

Listing 5.8: Die node.js REST-API konfigurieren

Die „require"-Funktion importiert Funktionen aus anderen Dateien, damit die Programmierung nicht in nur einer Datei erfolgen muss. Bei der Konfiguration der Express-App muss darauf geachtet werden, den „Body-Parser" zu aktivieren und die Datei-Größe heraufzusetzen, da die REST-API auch große Dateien speichern muss, die als Body in einem HTTP-POST-Request gesendet werden.

Suchen

Die Such-Funktionen werden über HTTP-GET-Funktionen aufgerufen. Dabei werden die Parameter teilweise über Variablen im Pfad zum Beispiel :type als Name des Typs oder über Parameter am Ende der URL mitgegeben, zum Beispiel ?name=Test.
Die Rest-API unterstützt bisher folgende Abfragen:

Pfad	Funktion
GET localhost/artifacts/type/{at}/	Alle Elemente eines bestimmten Typs {at}
GET localhost/artifacts/type/{at}/?name={n}	Ein Element mit Name {n} und Typ {at}
GET localhost/artifacts/type/{at}/?id={id}	Ein Element mit Id {id} und Typ {at}
GET localhost/collections/	Alle Collections der Datenbank
GET localhost/files?name={name}&type={at}	Datei eines Artifacts zum Download
GET localhost/tags?filter=best/	Liefert alle Tags (Filter nach Häufigkeit)
GET localhost/categories?filter=best	Liefert alle Kategorien (Filter nach Häufigkeit)
GET localhost/artifacts?filter={tag}	Alle Elemente, filtert nach {tag}
GET localhost/artifacts/searches/{search}	Alle Elemente mit {search} im Namen

Tabelle 5.1: Die REST-API Funktionen zum Suchen

Vor der Ausführung einer Suche in der Datenbank muss node.js eine Verbindung mit der mongoDB-Datenbank aufbauen. Dazu muss der mongoDB-Treiber für node.js eingebunden und die Instanz konfiguriert werden. Die erstellte Verbindung wird einmal für alle Suchanfragen aufgebaut und von den einzelnen Funktionen wieder verwendet.

```
exports.findByTypeAndName = function(req, res) {
     res.header("Access-Control-Allow-Origin", "*");
     res.header("Access-Control-Allow-Headers", "X-Requested-With");
```

```
            var artifactType = req.params.artifactType;
            var name = req.query.name;
            db.collection('artifacts', function(err, collection) {
            collection.findOne({'name':name,'artifactType':artifactType}, function(err, item) {
            res.send(item);
                  });
            });
};
```

Listing 5.9: REST-API Artefaktsuche

In den Node.js-Modulen, die mit *require()* eingebunden werden, müssen Funktionen, die vom Hauptprogramm zugänglich sein sollen, ein „exports“ voran gestellt werden. Die Funktion *exports.findByTypeAndName* wird von der Express-Applikation aufgerufen, wenn der Pfad die Form „localhost/type/:artifacttype/“ hat, wobei *:artifacttype* beliebige Werte annehmen kann. Diese Werte werden als Parameter in der „req“-Variablen übergeben.
Anschließend kann eine Datenbank-Abfrage gestartet werden. Hierzu wird ein Datenbankobjekt im BSON-Format[27] angelegt, das alle Werte enthält, welche die gesuchten Dokumente enthalten sollen. In diesem Beispiel müssen „name“ und „artifactType“ mit den Variablen übereinstimmen. Da hier nur ein Element gefunden werden soll, kann *collection.findOne* benutzt werden.
Über die Express-Applikation wird das Ergebnis zurück gesendet. Da die REST-API nicht nur direkt über den Browser, sondern auch von Elementen außerhalb des eigenen Servers angesprochen werden kann, muss der Header des Ergebnisses entsprechend konfiguriert werden.
Das Ergebnis ist ein JSON-Objekt, das die zuvor definierte Struktur der Datenbank-Collection hat.

Beispiel für die Suche nach einer Liste von Elementen:

```
var tag = req.query.filter;
..
   collection.find( { $or: [{ 'usertags': { $in: [ new RegExp('.*'+tag+'.*')] } },
                            { 'categories': { $in: [ new RegExp('.*'+tag+'.*')] } } ]} ).
                            toArray(function(err, docs) {
```

Bei dieser Suche wurde das zu suchende Objekt nicht über den Pfad „localhost/artifacts“, sondern als Teil eines HTTP-GET Aufrufs übergeben. Über den „query“-Teil des Aufrufs, erhält man alle so übergebenen Variablen.
Da hier gleichwertig nach Dokumenten, die den Tag-Parameter entweder in dem „categories“-Feld, oder im „usertags“-Feld stehen haben, gesucht werden soll, muss die Abfrage ein „or“ enthalten. Der Tag-Parameter wird in einem regulären Ausdruck zum Suchen benutzt, damit auch Dokumente gefunden werden, die den Parameter nur als Teil des Feld-Wertes enthalten.
Das Ergebnis der Suche wird als Array zurückgeliefert.

```
    if(!err){
        intCountDoc = docs.length;
        gesamt=intCountDoc;
            if(intCountChef > 0){
              for(var i=0; i<intCountChef;){
```

[27] http://bsonspec.org/

```
                    strJson += '{"Name":"' + docs[i].name + '", "Type":"'+ docs[i].artifactType
                        +'"}'
                    i=i+1;
                  if(i<intCountDoc){strJson+=',';}
                      }
          strJson = '{"Suche":"'+tag+'","Gefunden":'+gesamt+',"Artefakte":[' + strJson + "]}";
          res.send("",JSON.parse(strJson));
    }
    }
     else{onErr(err,res);}
    })
```

Listing 5.10: REST-API Artefakte nach Kategorie suchen

Die Rest-API soll bei dieser Suche eine Zusammenfassung der Dokumente bereitstellen, bekommt aber von der mongoDB als Ergebnis alle Dokumente, die den gesuchten Begriff enthalten. Deshalb muss man das Ergebnis selbst als JSON-Objekt erstellen.

Beispiel für die Suche und das Herunterladen von Dateien:

```
db.collection('artifacts-files.files').find({ 'filename' : name }).toArray(function(err, files) {
   if (err) throw err;
   files.forEach(function(file) {
```

Über die *db.collection*-Funktion wird die Collection ausgewählt, in der die Informationen zu den Dateien gespeichert sind. Um an den Inhalt einer Datei zu gelangen, wird die eindeutige „_id“ benötigt, welche man über eine Abfrage nach dem Dateinamen erhält. Diese „ _id“ ist in der „file“-Variable im Ergebnis gespeichert.

```
      var Grid = require('gridfs-stream');
      var gfs = Grid(db, mongo);
      var options = {_id: file._id, root: 'artifacts-files'};
      try{
            var readstream=gfs.createReadStream(options);
            readstream.on('open', function () {
                  readstream.pipe(res);
      });
      readstream.on('end', function() {
      });
   } catch(err){
      console.log(err);
   } });
   req=null;
});
```

Listing 5.11: Datei aus der mongoDB herunterladen

Der Zugang zu dem Inhalt der Dateien erfolgt über das „gridfs-stream“-Modul für node.js. Dazu muss es erst installiert und instanziiert werden. Über die Optionen kann ein sogenannter „ReadStream“ aufgebaut werden, der den Inhalt als Stream ausgeben kann. Da die Datei hier direkt heruntergeladen werden soll, wird der ReadStream den Inhalt an die res-Variable der Express-Applikation senden. Diese Variable wird direkt als Antwort im Browser angezeigt, bzw. bei Archiven als Download angeboten.

Speichern und verändern

Um Inhalte wie Artefakt-Dateien und JSON-Dokumente über die REST-API in der Datenbank zu speichern, müssen diese über eine HTTP-POST-Abfrage an die API gesendet werden. Beim Verwenden der POST-Abfrage muss das Dokument als Body, bzw. die Datei oder der Tag als Form-Data-Element in der Abfrage übermittelt werden.

Pfad	Funktion
POST localhost/artifacts/{at}/?name={n}&tag={tag}	Fügt einen User-Tag hinzu
DELETE localhost/artifacts/{at}/?name={n}&tag={tag}	Entfernt einen User-Tag
POST localhost/artifacts	Speichert ein JSON-Dokument
POST localhost/files	Speichert eine Datei

Tabelle 5.2: Die REST-API Funktionen zum Speichern und Bearbeiten

User-Tags verändern

Das Hinzufügen und Löschen der Usertags, erfolgt über *collection.update()*-Funktionen:

```
 collection.update({'name':name,'artifactType':coll} ,{$push:{usertags:tag}}, {w:1},
     function(err, result) {
res.send(200);
});
```

$push fügt ein neues Element in das Array der User-Tags im Dokument ein.

```
 collection.update({'name':name,'artifactType':coll}, {$pull:{usertags:tag}}, {w:1},
     function(err, result) {
res.send(200);
});
```

$pull entfernt ein Element aus dem Array im entsprechenden Dokument.
Die Express-Applikation sendet anschließend den HTTP-Statuscode 200, der „ok“ symbolisiert, an den Browser.

JSON-Dokumente speichern

```
for(var attributename in artifact){
      if (attributename=="name"){
            name=artifact[attributename];
      }
      ...
}
```

Da die REST-API auch überprüfen muss, ob schon Dokumente zu einem Artefakt in der Datenbank gespeichert sind, muss ein Mapping der JSON-Schlüsselwerte mit lokalen Variablen stattfinden. Das ankommende JSON-Dokument hat dabei eine flachere Struktur als später das Dokument in der Datenbank. In der Datenbank wird die Version zusammen mit den Metadaten und dem Link zur Datei in ein JSON-Objekt eingetragen, da die REST-API auch verschiedene Versionen eines Artefaktes

ausliefern soll.

```
var versionRep=version.replace(/\./g,'');
collection.find( { 'name': name , 'artifactType': artifactType} ).toArray(function(err, docs) {
    if(!err){
    if (docs.length>0){
      console.log("name und type gefunden");
```

Mit den extrahierten Variablen kann in der Datenbank nach schon vorhandenen Dokumenten zu dem Artefaktnamen und Typ gesucht werden. Ist ein Artefakt schon vorhanden, muss überprüft werden, ob die neueste Version des Artefakts im Array mit allen gespeicherten Versionen zu finden ist:

```
      if (typeof(docs[0].version) != "undefined" && docs[0].version[versionRep]!=null){
            console.log("aktuelle version gefunden);
                  collection.update({'_id':docs[0]._id}, {
            'name':name,
            ....
            'version':docs[0].version
      });
```

Ist die neueste Version schon vorhanden, werden nur die übrigen Elemente des Artefakt-Dokuments erneuert.

```
    } else {
            console.log("keine aktuelle version gefunden");
            if (typeof(docs[0].version) != "undefined"){
                  var versionDBObject=docs[0].version;
                  var objToJson = { };
                  objToJson.version = version;
                  objToJson.file = file;
                  objToJson.metaData = metaData;
                  versionDBObject[versionRep]=objToJson;
```

Ist die neueste Version nicht im entsprechenden Array zu finden, muss ein neues JSON-Dokument mit der aktuellen Version (ohne Punkte) als Schlüssel und den Unterelementen Version, Datei und Metadaten erstellt werden. Dieses JSON-Dokument wird als neues Element in das vorher extrahierte JSON-Array mit den schon gespeicherten Versionen eingefügt.

```
                        collection.update({'_id':docs[0]._id}, {
                  'name':name,
                   ...
                  'version':versionDBObject
            });
```

Das Dokument wird anschließend so aktualisiert, dass es die neuesten Werte aus dem übermittelten JSON-Dokument enthält und im Versions-JSON-Objekt ein neues Element hinzugekommen ist.

```
     }
    }
     }else{
      console.log("name und type nicht gefunden");
      var objToJson = { };
            objToJson.version = version;
```

```
            objToJson.file = file;
            objToJson.metaData = metaData;
            var versionDBObject={};
      versionDBObject[versionRep]=objToJson;
      collection.insert({
            'name':name,
            ...
```

Wurde in der Datenbank kein Dokument mit dem Namen und Typ des JSON-Dokuments gefunden, wird ein neues Dokument eingefügt. Vor dem Einfügen muss auch hier ein Unterobjekt mit den Informationen zur Version und der bereinigten Version als Schlüssel erzeugt und in ein JSON-Array eingefügt werden.

```
     }, {safe:true}, function(err, result) {
            if (err) {
            res.send({'Save error':'An error has occurred'+err});
            } else {
            res.send(result[0]);
            }
            });
    }}
  else{onErr(err,res);}
});
```

Listing 5.12: JSON Dokument in der mongoDB speichern

Je nach Erfolg beim Einfügen sendet die REST-API über die Express-Applikation ein „ok“ oder den Fehlertext zurück.

Dateien speichern

Zu Beginn des Speichervorgangs muss überprüft werden, ob überhaupt eine Datei mitgesendet wurde:

```
var file = req.files.file;
if(!file) return res.send({result: 'NO_FILE_UPLOADED'});
var tempfile = req.files.file.path;
var origname = req.files.file.name;
var found=0;var file_id=0;
db.collection('artifacts-files.files').find({ 'filename' : origname }).toArray(function(err,
     files) {
   if (err) throw err;
   files.forEach(function(file) {
      console.log(found);
      found++;
      file_id=file._id;
   })
```

Analog zum Herunterladen einer Datei aus dem gridFS-Store wird hier zuerst nach dem Dateinamen gesucht und gegebenenfalls die file._id gespeichert.

```
      if (found==0){
            var writestream = GridFS.createWriteStream({
            filename: origname,
```

```
                mode:'w',
                root: 'artifacts-files'
      });}else{
           var writestream = GridFS.createWriteStream({
           _id: file_id,
                filename: origname,
                mode:'w',
                root: 'artifacts-files'
  });
  }
```

Listing 5.13: Datei in der mongoDB speichern

Abhängig vom Ergebnis der Suche wird ein „GridFS-WriteStream“, der Daten in die Datenbank schreiben kann, mit oder ohne file._id angelegt. Ist die file._id dabei, wird genau diese Datei in der Datenbank überschrieben.
Nachdem ein WriteStream angelegt wurde, wird über das node.js File-System-Modul aus der mitgeschickten Datei ein ReadStream erzeugt, dessen Inhalt an den WriteStream gesendet wird. Wurde die Datei vollständig übertragen, wird ein „ok“ als Ergebnis zurück geliefert:

```
    fs.createReadStream(tempfile)
    .on('end', function() {
      res.send('OK'); })
    .on('error', function() {
      res.send('ERR');})
    .pipe(writestream);
});
```

Exportieren ins TOSCA Format

Um ein gespeichertes Artefakt über die REST-API ins TOSCA-CSAR-Format zu exportieren, müssen einerseits aus den Daten des Artefakts Informationen in XML-Dateien gespeichert werden und andererseits noch die Paket-Datei des Artefakts mit in das CSAR-Archiv gepackt werden.
Um XML-Dateien mit node.js nicht durch String Operationen erstellen zu müssen, kann man Module wie den xmlbuilder[28] für node.js zu Hilfe nehmen.
In die XML-Dokumente müssen Informationen über den Namen, die enthaltenen Operation, den Dateinamen sowie den allgemeinen Typ gespeichert werden, da jedes Konfigurationsmanagement-Tool eine andere Vorgehensweise beim späteren ausführen in der TOSCA Umgebung benötigt. Diese Informationen werden aus der Datenbank geholt und in Variablen gespeichert.
Die Tabelle 5.3 gibt die XML-Dateien an, die von der REST-API erstellt werden müssen.
In [WBB+13] wurden Vorschläge gemacht, wie man einen TOSCA Artifact-Type für Chef-Artefakte konstruieren kann. Es muss für jedes Konfigurationsmanagement-Tool, dessen Artefakte mit TOSCA ausgeführt werden sollen, ein eigener Artifact-Type erstellt werden. Hierin muss geregelt sein, wie TOSCA die Artifact-Dateien später ausführt.
Ein „Artifact Template“, das die Funktionen des jeweiligen Artefakts beschreibt, muss einen Verweis auf den Dateinamen des gesamten Artefakts, sowie Informationen über die möglichen Operationen

[28] https://github.com/oozcitak/xmlbuilder-js

XML-Datei	Erklärung
NodeType-Definition	Allgemeine Angaben zum Knoten z.B. zum Input/Output
NodeType-Implementierung	Eigentlicher Knoten, enthält einen Verweis auf das ArtifactTemplate
ArtifactType	Pro Konfigurationsmanagement-Tool ein Type
ArtifactTemplate	Informationen über das Artefakt
TopologyTemplate	Mehrere Nodes können zusammengefasst werden

Tabelle 5.3: XML-Definitions-Dokumente für den TOSCA-Export

enthalten. Die Grundstruktur ist dabei für jeden Typ identisch, jedoch müssen die „Properties“ jeweils angepasst werden.
Die REST-API muss also aus den Daten, die in der Datenbank zu jedem Artefakt verfügbar sind, den Properties-Teil im Artifact Template so anpassen, dass es später ausgeführt werden kann.
Mit dem node.js-Xmlbuilder Softwarepaket kann man einen Properties-Teil für jeden Typ einzeln erstellen und anschließend in das Artifact Template einfügen:

```
if (artifactType=="chef"){
 var properties = require('xmlbuilder').create("Properties")
      .ele(artifactType+":"+artifactType+"artifactTypeProperties")
      .att("xmlns:chef","http://docs.oasis-open.org/tosca/ns/2011/12/chef")
      .att("xmlns","http://docs.oasis-open.org/tosca/ns/2011/12/chef")
       .ele("Cookbooks")
```

Zuerst wird ein Properties Element angelegt und der XML-Namespace als Attribut gesetzt. Im Falle eines Chef-Artefakts ist das weitere Element eine Liste von Cookbooks.

```
 if (item.operations!=null){
      item.operations.forEach(function(entry) {
                  properties.ele("Cookbook").att("name",name+"-"+entry).att("location","files/"+file)
                      .ele("RunList")
                          .ele("Include")
                       .ele("RunListEntry").att("cookbookName",name).att("recipeName",entry).up()
                   .up()
                .up();});}
      properties.up();
}else if (artifactType=="puppet"){
                  ...
```

Listing 5.14: XML Dokument generieren

Diese Liste wird über eine for-each-Schleife erzeugt, die alle Einträge des Array mit den Operationen durchforstet und den jeweiligen Wert eines Eintrages als Variable „entry“ liefert. Es wird hier für jede Operation des Cookbooks ein eigenes Cookbook angelegt, in dessen „RunList“ die einzelne Operation steht. Die *up()*-Funktion lässt das als nächstes erzeugte Element eine Ebene höher im XML-Pfad entstehen und das Element in der aktuellen Ebene schließen.
Die weiteren Definitions-Dateien für TOSCA werden auf ähnliche Art erstellt und über einen „node.js-WriteStream“ in eine Datei geschrieben. Die vollständig generierte XML Datei, das ein Artifact Template eines Chef-Artefakts darstellt, sieht so aus:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<tosca:Definitions id="winery-defs-for_ns3-at-ant-chef-Script"
    targetNamespace="http://www.example.org/ant/tosca"
    xmlns:tosca="http://docs.oasis-open.org/tosca/ns/2011/12"
    xmlns:winery="http://www.opentosca.org/winery/extensions/tosca/2013/02/12"
    xmlns:ns1="http://opentosca.org/self-service">
 <tosca:Import importType="http://www.w3.org/2001/XMLSchema"
     location="Definitions/tbt__chef.tosca"
     namespace="http://docs.oasis-open.org/tosca/ns/2011/12/ToscaBaseTypes"/>
  <tosca:ArtifactTemplate name="ant-chef-artifact" id="ant-chef-artifact" type="tbt__chef"
      xmlns:tbt="http://docs.oasis-open.org/tosca/ns/2011/12/ToscaBaseTypes">
      <Cookbooks>
            <Cookbook name="ant-default.rb" location="files/chef-ant-1.0.2.tgz">
                  <RunList>
                        <Include>
                              <RunListEntry cookbookName="ant" recipeName="default.rb"/>
                        </Include>
                  </RunList>
            </Cookbook>
            <Cookbook name="ant-install_package.rb" location="files/chef-ant-1.0.2.tgz">
                  <RunList>
                        <Include>
                                    <RunListEntry cookbookName="ant"
                                        recipeName="install_package.rb"/>
                        </Include>
                  </RunList>
            </Cookbook>
      </Cookbooks>
      <tosca:ArtifactReferences>
            <tosca:ArtifactReference reference="files/chef-ant-1.0.2.tgz"/>
      </tosca:ArtifactReferences>
  </tosca:ArtifactTemplate>
</tosca:Definitions>
```

Der „pretty:true“-Ausdruck in der *toString*-Funktion des XML-Builder Elements sorgt dafür, dass die Zeilen korrekt eingerückt werden.
Mit Hilfe eines WriteStreams kann auch Zeile für Zeile die TOSCA.meta-Datei erstellt werden, die Verweise auf alle Dateien im CSAR-Archiv enthält.
Wenn alle Dateien im Archiv-Verzeichnis erstellt sind, wird die Artefakt-Datei aus der Datenbank geladen und in das „files“ Verzeichnis im Archiv-Verzeichnis kopiert. In diesem Verzeichnis befindet sich schon ein „imports“-Ordner, der die XML-Schema-Definitionen der TOSCA-Basistypen enthält, die in jedem CSAR-Archiv enthalten sein müssen.
Enthält das Artefakt Abhängigkeiten zu weiteren Artefakten, werden diese auch in das CSAR-Archiv gespeichert und in den XML-Dateien verknüpft. Das Verzeichnis mit allen enthaltenen Dateien wird schließlich mit dem archiver[29]-Modul für node.js in ein ZIP-Archiv gepackt und anschließend in eine .csar-Datei umbenannt.
Alle erstellten Dateien und Verzeichnisse, außer dem import-Verzeichnis, können jetzt gelöscht werden. Um die Datei vom node.js-Server herunterladen zu können,wird sie mit einem ReadStream

[29]https://www.npmjs.org/package/archiver

eingelesen und der Inhalt an die Ergebnis-Variable der Express App gesendet:

```
var readStream = fs.createReadStream(name+'.csar');
readStream.on('open', function () {
    res.setHeader('Content-disposition', 'attachment; filename=' + name+'.csar');
    res.setHeader('Content-type', 'application/zip');
    readStream.pipe(res);
});
```

5.3 Artifact Management User Interface

5.3.1 Graphische Benutzeroberfläche und Workflow

Das User Interface ist eine Webseite, über die der Zugang zu allen, von den Crawlern gespeicherten, Artefakten der verschiedenen Konfigurationsmanagement-Tools ermöglicht wird.
Auf der linken Seite befinden sich in der graphischen Benutzeroberfläche die Sortiermöglichkeiten. Man kann die Elemente einerseits nach dem Typ des Konfigurationsmanagement-Tools, von dem sie stammen, oder nach der Kategorie, zu der sie gehören, filtern lassen.
Die Liste der ausgewählten Elemente erscheint nach der Auswahl des Filters in der Mitte des User Iterface. Hier kann der Benutzer das einzelne Element aus der Liste auswählen und sich die Detailinformationen anzeigen lassen.
Hat der Benutzer ein Element angeklickt, erscheinen auf der rechten Seite die Interaktionsmöglichkeiten mit dem User Interface: Man kann das Artefakt als Service Template in einem TOSCA-Archiv exportieren oder die Original-Dateien direkt herunterladen. Außerdem gibt es die Möglichkeit dem Element neue eigene Usertags hinzuzufügen, damit man die Filter Möglichkeiten entsprechend erweitert. Die einzelnen Usertags können bei der Detailansicht eines Elements jederzeit gelöscht werden.

5.3.2 Implementierung

Das User Interface besteht grundsätzlich aus HTML-Code, dessen Inhalt beim Navigieren durch das Interface durch JavaScript Befehle ständig angepasst wird. Die Navigation findet also durch Austauschen des Inhalts der HTML-Elemente statt, was im Vergleich zum Navigieren durch Blättern auf verschiedene Seiten, den Vorteil hat, dass der Inhalt dynamisch verändert und ohne Nachladen der ganzen Seite angezeigt werden kann.
Der dynamische Inhalt wird durch asynchrone Abfragen der REST-API erzeugt. Diese Abfragen werden mit dem AJAX-Framework[30] ausgeführt und das Ergebnis mit dem jQuery-Framework[31] in die HTML-Seite geladen.
Das jQuery-Framework ermöglicht zum Beispiel über die ID den Zugriff auf alle im HTML-Dokument

[30] http://api.jquery.com/jquery.ajax/
[31] http://jquery.com/

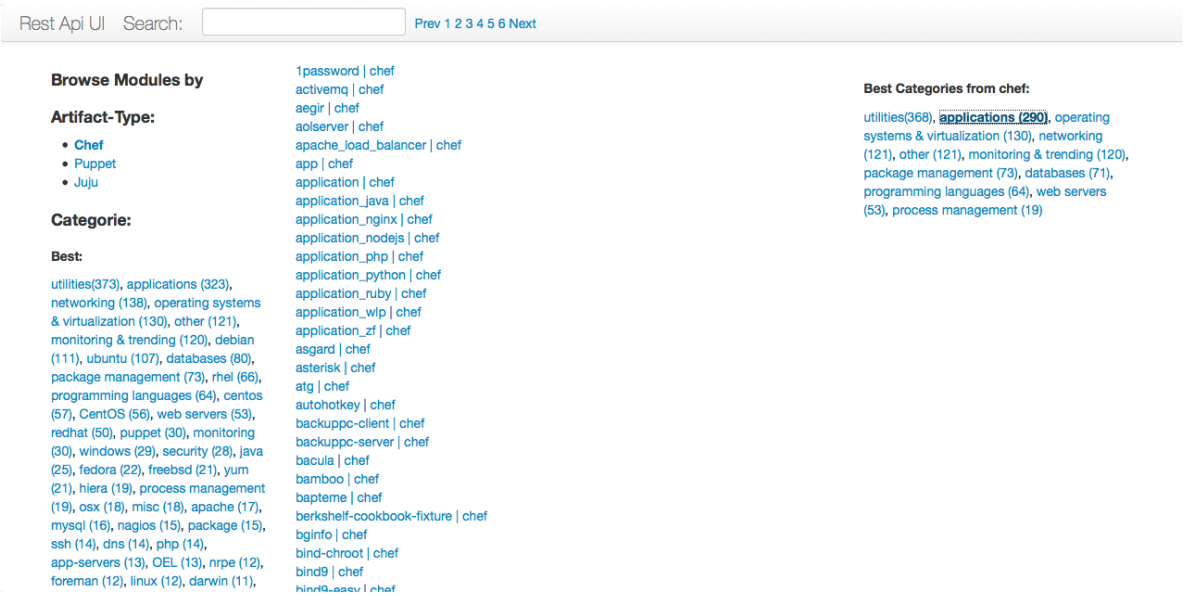

Abbildung 5.5: User Interface mit der Liste an Chef Cookbooks und der Kategorie „Applikationen“

(DOM) befindlichen Elemente.

```
$('.collection').on("click",function(){
      findAll( $(this).attr("id"));
      ..})
```

Beim Auswählen zum Beispiel eines Types, nach dem die Datenbank durchsucht werden soll, wird die ID des ausgewählten Elements an die Suchfunktion übergeben.

```
jQuery.support.cors = true;
function findAll(type){
$.ajax({
      type:"GET",
      url: "http://localhost:3000/artifacts/"+type+"/",
      dataType:"json",
      crossDomain:"true",
            success: function(data){
                        renderList(data,collection);
      },
      error: function(msg){ alert("error in findAll \n" + JSON.stringify(msg)); }
   });
}
```

Listing 5.15: REST API mit JavaScript ansprechen

Die Such-Funktion *findAll(type)* ruft die Suche nach dem Artefakttyp in der REST-API auf. Das jQuery Framework muss dabei so konfiguriert sein, dass es auch Antworten außerhalb des eigenen URL-Pfades empfangen kann. Dies erfolgt durch die Einstellungen *jQuery.support.cors = true;* und *crossDomain:"true"*.

Bei erfolgreichem Aufruf erhält man als Antwort von der REST-API die als JSON-Array formatierte Liste mit allen Elementen, die den gewählten ArtifactType haben. Diese Liste mit den Elementen befindet sich im „data“-Element, das an die Funktion zum Darstellen der Liste übergeben wird.
Die Elemente sollen in einem bestimmten Bereich der HTML-Seite angezeigt werden. Dieser Bereich wurde z.B. durch das Aufzählungs-Element „ul“ mit der ID „Liste“ festgelegt. Diesem Aufzählungselement werden die Elemente durch den JavaScript Befehl

```
$('#List2').append('<li><a href="#" class="clickelement">'+element.name+'</a></li>');
```

einzeln hinzugefügt und einem „click-Listener“

```
$('.clickelement').on("click",function(){...});
```

, der auf das Anklicken der Elemente reagiert, zugeordnet.
Ein Anklicken ruft die nächste Funktion auf, die die Details eines Elements anzeigt und die ausgeblendeten Interaktionselemente an der rechten Seite des graphischen Benutzerinterfaces erscheinen lässt.
Um die Beschreibungstexte mancher Artefakte, die im Markdown-Format geschrieben sind, in HTML Code umzuwandeln, wird eine Instanz des Markdown[32] Converters[33]pagedown benötigt. Dieser Instanz wird der Beschreibungstext der einzelnen Artefakte übergeben.
Um an die Unterelemente des JSON-Objekts zu gelangen, das die verschiedenen Versionen eines Artefakts enthält, benötigt man die *Object.keys()* Funktion, die alle Schlüsselwerte eines Elements zurückliefert. Durch das Array mit den Schlüsselwerten kann man mit einer Schleife iterieren und den Inhalt an die Liste mit den Elementinformationen anhängen.

[32] http://daringfireball.net/projects/markdown/
[33] https://code.google.com/p/pagedown/

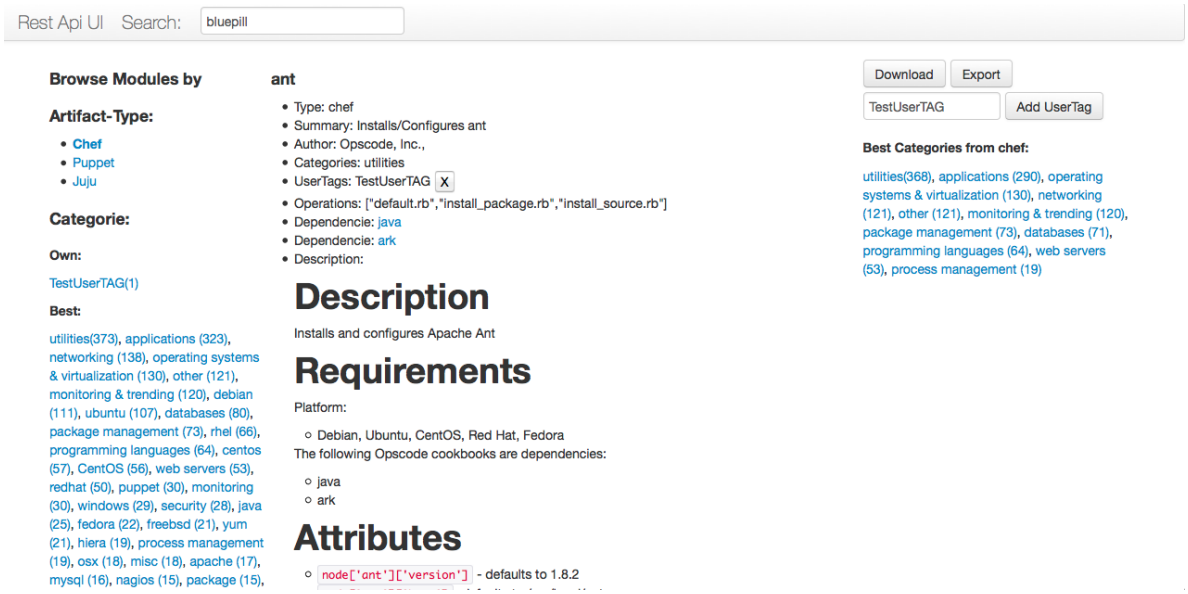

Abbildung 5.6: User Interface mit ausgewähltem Chef Element

6 Evaluation

6.1 Crawling-Leistung

Die Crawler wurden so konzipiert, dass sie möglichst fehlerverzeihend arbeiten. Das heißt, dass sie bei Unregelmäßigkeiten, wie fehlenden Informationen zu einem einzelnen Modul, den Crawling-Vorgang nicht abbrechen. Je nach Wichtigkeit des Feldes, in dem ein Fehler vorkommt, oder den Informationen, die fehlen, wird diese Stelle im JSON-Objekt leer gelassen, oder das gesamte Modul nicht die Datenbank eingetragen.
Da in Objekten des Types org.json.JSONObject nach den Werten bestimmter Schlüssel gesucht wird, muss vor den Abfragen der Felder, die nicht von entscheidender Bedeutung sind, mit *.has()* überprüft werden, ob der Schlüssel im JSON-Dokument existiert. Andernfalls würde bei der Abfrage eine Ausnahmebehandlung greifen, die den Crawling-Vorgang unterbricht, mit dem nächsten Element weitermacht, oder ganz stoppt.
Dies ist zum Beispiel der Fall, wenn der Name des Modules nicht in den Feldern beschrieben wurde, da dieser essentiell für das weitere Finden und Bearbeiten im Artifact Store ist.

6.1.1 Besonderheiten des Chef Crawlers

Positiv

Alle Module sind über die REST-API erfassbar. Man muss keine HTML-Webseite crawlen und nach besonderen regulären Ausdrücken oder Elementen im DOM[1] suchen, um alle Namen der Cookbooks zu erhalten. Die API kann zudem limitiert und so gezielt durchforstet werden.

Schwierigkeiten

Die JSON-API ist mehrstufig aufgebaut, so dass man drei Abfragen benötigt, um Informationen über ein Cookbook zu bekommen. Das damit generierte JSON-Objekt muss anschließend noch mit Informationen aus der Archiv-Datei eines Cookbooks ergänzt werden. Dazu muss das Archiv durchforstet werden, was mit sogenannten „InputStreams“ gemacht wird. Allerdings muss man für jedes Kompressionsverfahren (ZIP, TAR, BZ2) unterschiedliche InputStreams auswählen, was nur über das Erkennen der Endung einer Datei geschehen kann.
Vielen Cookbooks fehlen Einträge in den Metadaten. Diese müssen durch die oben erwähnte Strategie

[1] http://www.w3.org/DOM/

entweder leer gelassen werden oder zu einem Abbruch des Crawling-Vorgangs des Cookbooks führen. Manche Cookbooks benutzen in ihren Metadaten Zeichen wie „\“ und „/“, die nicht automatisch vom JSON-Parser gelesen werden können und so zu einer Ausnahmebehandlung führen. Um auch diese Elemente in ein gültiges JSON-Dokument einzufügen, müssten im Metadaten-String diese Zeichen codiert werden. Allerdings muss bei der Anzeige der Artefakte darauf geachtet werden, die Codierung wieder aufzuheben.

Ergebnisse

Von den 1456 Cookooks, die es aktuell auf der Chef Webseite gibt, wurden davon 1430 in der Datenbank gespeichert. Das ergibt eine Quote von ca 99%. Die nicht gespeicherten Cookbooks haben hauptsächlich die oben beschriebenen Fehler in der Metadaten-Datei, oder kein gültiges JSON-Format, wie zum Beispiel das „accounts“-Cookbook[2]. Für das speichern benötigte der Crawler ca 1,5 Stunden.

6.1.2 Besonderheiten des Puppet Crawlers

Das Crawling wird mit dem Open Source Tool crawler4j ausgeführt, das sicherstellt, dass alle Seiten, die über weiterführende Links gefunden werden können, untersucht werden.

Positiv

Durch das einheitliche Kompressionsverfahren (.tar.gz), das alle Puppet Artifacts benutzen, ergeben sich zwei Vorteile. Einerseits lässt sich dadurch herausfinden, ob die aktuell untersuchte Seite eine Artefakt-Seite ist, und andererseits erleichtert es die Implementierung, da die Endung der Archiv-Datei nicht noch extra überprüft werden muss, um das Verfahren zum Entpacken der Datei zu definieren.

Schwierigkeiten

Während der Arbeit an diesem Projekt wurde man beim Aufruf des Puppet Forge auf die „HTTPS“-Seite umgeleitet. Daraufhin musste auch der Crawler angepasst werden. Da nur auf Unterseiten der Haupt-URL gesucht wird, muss diese mit dem Anfang jeder URL identisch sein.
Etwa ein fünftel der Puppet Artefakte haben eine inkonsistente Autor oder Namen Bezeichnung. Beispielsweise wird bei dem „openshift/openshift_origin“ Modul[3] in der heruntergeladenen Metadaten-Datei der Autor mit „Mojo Lingo, Red Hat“ bezeichnet. Sucht man allerdings in der REST-API von Puppet nach diesem Modul[4], um alle Daten zu vervollständigen, wird bei diesem Modul der Autor mit „openshift“ bezeichnet. Da man über den Dateinamen des .tar.gz-Archivs auch an diese Autor Bezeichnung gelangen kann, sucht der Crawler in der API schließlich nach Modulen mit beiden Namen.

[2] http://community.opscode.com/cookbooks/accounts
[3] https://forge.puppetlabs.com/openshift/openshift_origin
[4] http://forge.puppetlabs.com/modules.json?q=openshift-openshift_origin

Das oben genannten Beispiel enthält zudem mehrere „metadata.json“-Dateien, da es die Artefakte, von denen es abhängig ist, direkt im Archiv integriert hat. Der Crawler musste also auch so angepasst werden, dass die richtige Metadaten-Datei im Archiv gefunden wird.
Wie bei den Chef Cookbooks fehlen auch bei den Puppet Artifacts einige Felder, was berücksichtigt werden muss. Da das Crawling über das crawler4j-Modul ausgeführt wird, der alle Links von der Startseite ausgehend betrachtet, muss der Crawling-Vorgang im Fehlerfall komplett neu gestartet werden und kann nicht von einem bestimmten Punkt aus fortgesetzt werden.

Ergebnisse

Von den im Puppet Forge angegebenen 2423 Modulen werden aktuell über 2190 gecrawlt, was einer Quote von über 90% entspricht. Der Crawling Vorgang benötigt dafür ca 2,5 Stunden.

6.1.3 Besonderheiten des Juju Crawlers

Die Seite https://manage.jujucharms.com/charms listet Verweise zu allen Juju Charms auf, die mit jSoup ausgelesen werden. Der Name des Charms steht im HTML Inhalt des Links. Mit dem Namen kann die Juju-API benutzt werden, um alle Informationen zu erhalten.

Positiv

Die Namen aller Charms stehen direkt nach dem Auslesen der ersten Seite in einer Liste. Man erhält dadurch einen genauen Überblick über den Crawling-Vorgang. Da die Suche in der Juju API nach einem Charm Namen mehrere Ergebnisse liefert, die sich überschneiden, ist es sinnvoll, eine Liste mit den schon gespeicherten Charms zu führen und das aktuelle Charm in der Liste zu suchen, bevor es erneut gespeichert wird.

Schwierigkeiten

Die Charms-Dateien liegen nicht in einem Archiv, sondern auf www.launchpad.net in einem Bazaar Branch. Dies führt bei manchen Charms zu sehr langen Pfad und Dateinamen, was beim Komprimieren in ein Archiv zu Fehlern führen kann. Gängige Kompressionsverfahren erlauben maximal 100 Byte pro Namen. Fehlende Felder in den Metadaten müssen auch bei den Juju Charms entsprechend behandelt werden.

Ergebnisse

Da die Liste mit den angegeben Charms direkt übernommen wird, werden außer den Artefakten mit zu langen Dateinamen, ca 95% der Charms in der Datenbank gespeichert. Der Vorgang dauert ca 1 Stunde, da es auch Charms gibt, die deutlich größer als die meisten anderen Artefakte sind.

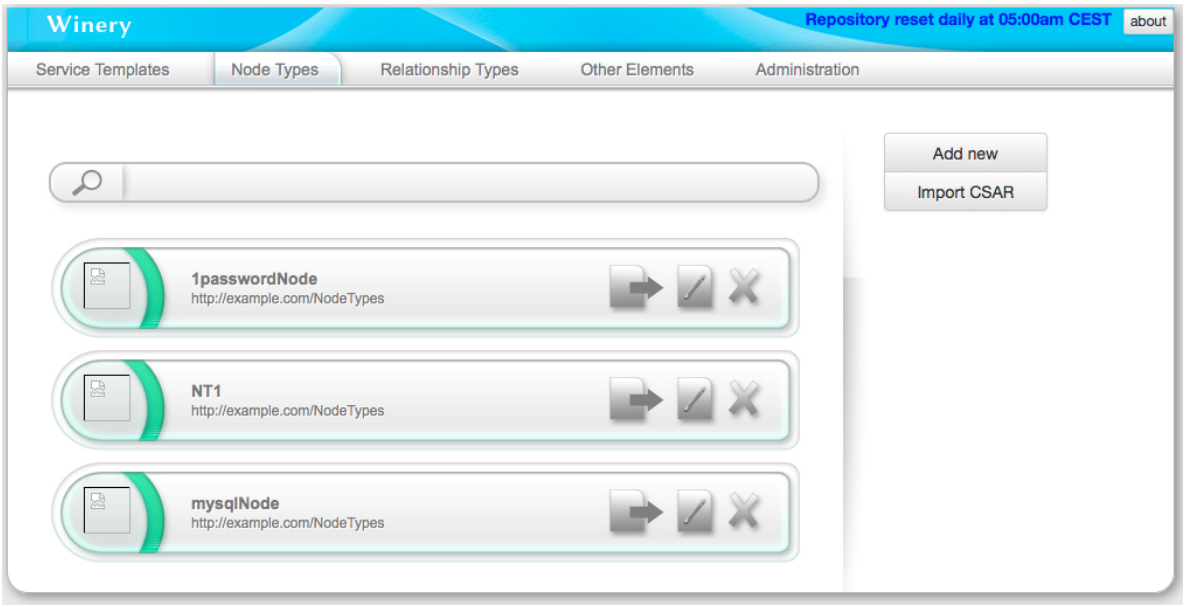

Abbildung 6.1: Open-TOSCA Winery mit den importierten Node Types der 1password und mysql Artefakte

6.2 TOSCA-Export

Der Export eines Artefakts ins TOSCA Format erfolgt über eine HTTP-GET-Anfrage an die REST-API. An den Browser wird eine CSAR-Datei zurückgeliefert, die im ZIP-Kompressionsverfahren gepackt ist. Das resultierende Service Template enthält einen Knoten, der die entsprechenden Operationen als Interfaces beinhaltet. Hängt das Artefakt von anderen Artefakten und deren Node Types ab, werden diese gesucht und gegebenenfalls dem CSAR hinzugefügt.
Zur Überprüfung, ob ein Export funktioniert, kann man die exportierten CSAR-Artefakte in das Modellierungs-Tool für TOSCA[KBBL13], die sogenannte Winery, als neues Service Template importieren. Anschließend stehen einem die darin enthaltenen Elemente, wie zum Beispiel Node Types mit den entsprechenden Operationen, zur Verfügung. Die hierfür benutzte Winery gehört zu dem OpenTOSCA[5]-Projekt der Universität Stuttgart.

Positiv

Es gibt für alle Artifact Types in der REST-API nur eine Implementierung, was die Programmierung vereinheitlicht und beschleunigt.

[5]http://www.iaas.uni-stuttgart.de/OpenTOSCA/indexE.php

Schwierigkeiten

Die XML-Dateien können in node.js mit dem XML-Builder Zeile für Zeile erstellt werden, dabei muss jedoch beachtet werden, dass Zweige eines XML-Baumes geschlossen werden und am Ende der XML-Erstellung das Dokument abgeschlossen wird. Beim Import der CSAR Dateien in die Winery von TOSCA erschweren fehlende Informationen bei Fehlern die genaue Diagnose und Fehlerbeseitigung.

7 Zusammenfassung und Ausblick

Die Arbeit beschäftigte sich mit der DevOps-Bewegung und den daraus entstandenen Werkzeugen, welche die Installation und Konfiguration von Software-Artefakten erleichtern. Es wurden die Konfigurationsmanagement-Tools Chef, Puppet und Juju vorgestellt, ihre Funktionsweise genau beschrieben und deren Unterschiede aufgezeigt. Außerdem wurde auf das Problem eingegangen, dass es kaum einheitliche Standards im Umfeld des Konfigurationsmanagement gibt und wie TOSCA dieses Problem lösen kann.
Diese Arbeit unterstützt das Bestreben, einen Standard für die Konfigurationsmanagement-Tools zu schaffen, in dem ein eigenes Repository erzeugt wurde, das die Artefakte der oben genannten Tools vereint. Diese Artefakte wurden aus den Repositories der unterschiedlichen Tools ausgelesen und in einem einheitlichen Format gespeichert, was ein Vergleichen der Artefakte auch unterschiedlichen Artefakttyps ermöglicht.
Außerdem ist es dadurch möglich, über eine einheitliche Exportfunktion die Artefakte in das TOSCA-Format zu überführen. Die Artefakte können anschließend zum Beispiel in die TOSCA Winery[1] importiert und miteinander kombiniert werden.
Die Export-Funktion und alle weiteren Funktionen sind über eine API abrufbar. Diese stellt den einzigen Zugang zur Datenbank her, in der die Artefakte gespeichert sind, was einen späteren Austausch erleichtert.
Die Arbeit beschreibt, wie man Crawler passend für die Artefakte der oben genannten Konfigurationsmanagement-Tools entwirft und implementiert. Dazu wurde erst ein Überblick über die aktuellen Möglichkeiten verschiedenster Werkzeuge zum Durchforsten von Internetseiten und JSON-APIs gegeben.
Die für die Konfigurationsmanagement-Tools entwickelten Crawler durchforsten auf jeweils unterschiedliche Art die Repositories, in denen die Artefakte gespeichert sind. Sie sammeln die Daten und senden diese zusammen mit den jeweils zugehörigen Archiv-Dateien der Artefakte an eine API.
Das Senden der Artefakte erfolgt über HTTP-Requests mit JSON-Dokumenten und den zugehörigen Archivdateien als Formularinhalt. Die API speichert den Inhalt der Requests in der Datenbank. Die Logik hinter der API kann dabei selbst feststellen, ob ein Artefakt eine neue Version enthält, und fügt diese an entsprechender Stelle der Datenbank hinzu.
Die gespeicherten Artefakte können über eine grafische Benutzeroberfläche abgerufen werden. Diese ermöglicht die zweidimensionale Sortierung nach Typ des Konfigurationsmanagement-Tools und nach Kategorien der Artefakte. Außerdem enthält sie eine Freitextsuche, die auch benutzt werden kann, um mit einem Mausklick zu den jeweiligen Artefakten zu gelangen, von denen ein Artefakt abhängt.
Über die Benutzeroberfläche können einem Artefakt eigene Kategorien, sogenannte „Usertags“ hinzu-

[1] http://www.eclipse.org/proposals/soa.winery/

gefügt werden. Diese werden daraufhin auch als Sortierkriterium angezeigt.
Außerdem gibt es in der Benutzeroberfläche die Möglichkeit, die originale Artefakt-Datei herunterzuladen oder zuerst ins TOSCA-CSAR-Format exportieren zu lassen und anschließend herunterzuladen. Die Benutzeroberfläche leitet dabei alle Anfragen an die API weiter, deren Implementierung auch hier sämtliche Logik zur Generierung der Artefakt-Listen und die nötigen Sortierfunktionen enthält. Die Benutzeroberfläche sorgt anschließend für die Darstellung.

Ausblick

Die Evaluation in Kapitel 6 hat gezeigt, dass die Crawler in der Lage sind, nahezu alle Artefakte aus den Repositories der jeweiligen Konfigurationsmanagement-Tools in den Artifact Store zu laden. Um auch an Artefakte der Tools, die in anderen Repositories, wie zum Beispiel GitHub[2], gespeichert sind, zu gelangen, könnte man den Puppet Crawler, der das crawler4j Modul für Java einsetzt, so anpassen, dass als Einstiegspunkt für das Crawling eine Suche auf GitHub nach zum Beispiel den Chef Cookbooks gewählt wird. Der Crawler wird daraufhin alle weiterführenden Links untersuchen. Um passende Cookbooks zu erkennen, könnte der Crawler den Seiteninhalt auf Stichworte untersuchen und die Datei anschließend über den „Download as Zip"-Link speichern und untersuchen. Auf diese Weise könnten auch andere Artefakte in den Artifact Store aufgenommen werden, die nicht in den eigenen Repositories der Konfigurationsmanagement-Tools liegen.
Die Stichworte, die für den Crawler ein Erkennen der Seite als Artefakt ausmachen, müssten dabei nicht zu spezifisch gewählt werden, da der Crawler bei Seiten, die keine Artefakte darstellen, beim Speichern einen Fehler feststellen wird und mit der nächsten Seite fortfährt.
Da die einzelnen Crawler, die API und die grafische Benutzeroberfläche modular implementiert wurden, also da jedes Modul unabhängig von den anderen ist, können dem Projekt neue Crawler hinzugefügt werden, ohne die schon vorhandenen anpassen zu müssen.
So könnten in Zukunft auch Crawler für weitere Konfigurationsmanagement-Tools oder für Artefakte der bisher gecrawlten Tools, aber aus verschiedenen Repositories hinzugefügt werden. Auch die Datenbank, die schließlich sowohl die Daten, als auch die Archivdateien der Artefakte beherbergt, kann ohne Weiteres ausgetauscht werden, falls es in Zukunft geeignetere Speichermöglichkeiten gibt.
Um die gespeicherten und ins TOSCA-CSAR-Format exportierten Artefakte schließlich mit TOSCA auch ausführen zu können, müssen in TOSCA noch die entsprechenden Artifact Types implementiert werden. Diese sollten über eine angepasste Ausführungsumgebung der jeweiligen Konfigurationsmanagement-Tools mit den originalen Dateien der Artefakte lauffähige Dienste erstellen können. Sobald dies der Fall ist, können über den hier vorgestellten Artifact Store Dienste aus den unterschiedlichen Konfigurationsmanagement-Tools nicht nur zusammengestellt, sondern auch ausgeführt werden. Diese Entwicklung kann schließlich den in Kapitel 2.2.2 beschriebenen „Vendor Lock-In" im Bereich des Konfigurationsmanagements beseitigen.

[2] https://github.com/

Literaturverzeichnis

[Apa] The Apache Software Foundation - What is nutch? URL http://nutch.apache.org/. (Zitiert auf Seite 30)

[BBB+01] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, D. Thomas. Manifesto for Agile Software Development, 2001. URL http://www.agilemanifesto.org/. (Zitiert auf Seite 9)

[BBH+13] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, S. Wagner. OpenTOSCA – A Runtime for TOSCA-based Cloud Applications. In *11th International Conference on Service-Oriented Computing*, LNCS. Springer, 2013. (Zitiert auf den Seiten 7, 27 und 28)

[BBLS12] T. Binz, G. Breiter, F. Leyman, T. Spatzier. Portable Cloud Services Using TOSCA. *IEEE Internet Computing*, 16(3), 2012. (Zitiert auf Seite 23)

[BLS11] T. Binz, F. Leymann, D. Schumm. CMotion: A Framework for Migration of Applications into and between Clouds. In *2011 IEEE International Conference on Service-OrientedComputing and Applications (SOCA)*. IEEE Computer Society, 2011. doi: 10.1109/SOCA.2011.6166250. (Zitiert auf Seite 23)

[chea] Chef - Nodes. URL http://docs.opscode.com/chef_overview_nodes.html. (Zitiert auf Seite 16)

[cheb] Chef with Erlang. URL http://www.getchef.com/blog/2013/02/15/the-making-of-erchef-the-chef-11-server/. (Zitiert auf Seite 16)

[Ewa14] J. Ewart. *Managing Windows Servers with Chef*. Packt Publishing Ltd, 2014. (Zitiert auf Seite 16)

[GHS10] S. Günther, M. Haupt, M. Splieth. Utilizing internal domain-specific languages for deployment and maintenance of it infrastructures. *Very Large Business Applications Lab Magdeburg, Fakultät für Informatik, Otto-von-Guericke-Universität Magdeburg, Tech. Rep*, 2010. (Zitiert auf Seite 13)

[HF10] J. Humble, D. Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010. (Zitiert auf Seite 10)

[inf12] *Infrastructure as Code*, 2012. URL http://sdarchitect.wordpress.com/2012/12/13/infrastructure-as-code/. (Zitiert auf Seite 10)

[juja] Canonical - Juju Overview. URL https://juju.ubuntu.com/resources/overview/. (Zitiert auf Seite 19)

[jujb] Juju Wordpress Scaling. URL http://www.jorgecastro.org/2012/08/31/the-way-to-run-wordpress-in-the-cloud/. (Zitiert auf Seite 19)

[KBBL13] O. Kopp, T. Binz, U. Breitenbücher, F. Leymann. Winery–a modeling tool for TOSCA-based cloud applications. In *Service-Oriented Computing*, S. 700–704. Springer, 2013. (Zitiert auf Seite 78)

[KVHK+13] S. Krum, W. Van Hevelingen, B. Kero, J. Turnbull, J. McCune. *Getting Started with Puppet*. Springer, 2013. (Zitiert auf Seite 13)

[Lab] P. Labs. Puppet Labs - Puppet Open Source. URL http://puppetlabs.com/puppet/puppet-open-source. (Zitiert auf Seite 13)

[Lip13] P. Lipton. *Escaping Vendor Lock-in with TOSCA, an Emerging Cloud Standard for Portability*. Industry Standards and Open Source, CA Technologies, 2013. (Zitiert auf Seite 23)

[Mic13] R. Michel. DevOps Enhancing collaboration of DevOps - Enhancing collaboration of Development and Operations. IBM SolutionsConnect2013, 2013. (Zitiert auf Seite 10)

[mul] HTTP Specification. URL http://www.w3.org/Protocols/rfc2616/rfc2616-sec3.html. (Zitiert auf Seite 50)

[MWGK] E. Mueller, J. Wickett, K. Gaekwad, P. Karayanev. The Agile Admin. URL http://theagileadmin.com/what-is-devops/. (Zitiert auf Seite 9)

[Net12] I. enStratus Networks. DevOps for the Cloud: Achieving agility throughout the application lifecycle, 2012. (Zitiert auf Seite 9)

[oha] Chef - Ohai Tool. URL http://docs.opscode.com/ohai.html. (Zitiert auf Seite 16)

[Pes12] P. Peschlow. Die DevOps-Bewegung. *Java Magazin*, 1, 2012. URL https://www.codecentric.de/files/2011/12/die-devops-bewegung.pdf. (Zitiert auf Seite 9)

[pup] Puppet Labs - The Puppet Language. URL http://docs.puppetlabs.com/puppet/latest/reference/lang_summary.html. (Zitiert auf Seite 15)

[Roy70] W. W. Royce. Managing the development of large software systems. In *proceedings of IEEE WESCON*, Band 26. Los Angeles, 1970. (Zitiert auf Seite 9)

[run] *Chef - Roles*. URL http://docs.opscode.com/essentials_roles.html. (Zitiert auf Seite 17)

[Rut] M. M. Rutkowski. *OASIS TOSCA Interop Demo Storyboard - SugarCRM Template*. OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) TC. URL https://www.oasis-open.org/committees/document.php?document_id=48698&wg_abbrev=tosca. (Zitiert auf Seite 25)

[SHI+13] B. Satzger, W. Hummer, C. Inzinger, P. Leitner, S. Dustdar. Winds of Change: From Vendor Lock-In to the Meta Cloud. *Internet Computing, IEEE*, 17(1):69–73, 2013. URL http://www.infosys.tuwien.ac.at/staff/bsatzger/publications/pdf/2013_ic_mc.pdf. (Zitiert auf Seite 22)

[TCAT] T. T. C. Topology, O. S. for Cloud Applications (TOSCA). *TOSCA Version 1.0*. OASIS. URL http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.pdf. (Zitiert auf den Seiten 7, 24, 26 und 27)

[WBB+13] J. Wettinger, M. Behrendt, T. Binz, U. Breitenbücher, G. Breiter, F. Leymann, S. Moser, I. Schwertle, T. Spatzier. Integrating Configuration Management with Model-Driven Cloud Management Based on TOSCA. In *Proceedings of the 3rd International Conference on Cloud Computing and Service Science, CLOSER 2013, 8-10 May 2013, Aachen, Germany*, S. 437–446. SciTePress, 2013. (Zitiert auf Seite 67)

Alle URLs wurden zuletzt am 10. 06. 2014 geprüft.

Erklärung

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

Ort, Datum, Unterschrift