Freie Universität Berlin
Institut für Informatik
AG Software Engineering

Bachelorarbeit

Patrick Hobusch
patrick.hobusch@gmail.com
Matrikelnummer: 4457604

Automatisierte Konfiguration des Build-Servers im Saros-Projekt mit Salt und Git

Gutachter:
Prof. Dr. Lutz Prechelt
Prof. Dr. Elfriede Fehr

Betreuer:
Franz Zieris
Christian Salzmann

Berlin, 17.04.2015

Zusammenfassung

Durch den Einsatz eines sogenannten Systemkonfigurationswerkzeugs erhoffte sich das Saros-Projekt, welches Plugins zur verteilten Paarprogrammierung für verschiedene IDEs entwickelt, den eigenen Build-Server wieder besser zu verstehen. Der Grund für die große Komplexität dieses Systems ist das starke Zusammenspiel der verschiedenen Anwendungen, in denen mit Code-Reviews, Builds sowie statischen und dynamischen Tests die Integration von Quellcode in das Versionsverwaltungssystem Git gesteuert wird. Der Wunsch war es, den Zustand des Servers mit dem Systemkonfigurationswerkzeug Salt deklarativ zu beschreiben, mit Git zu versionieren und auf diese Weise Änderungsängste zu verlieren. In dieser Arbeit beschreibe ich Ansätze, wie unter anderem die Webanwendungen Jenkins und Gerrit deklarativ bereitgestellt und konfiguriert werden können, zeige die Grenzen dieses Vorhabens auf und präsentiere eine Lösung, die auch die Anforderungen des IT-Dienstes erfüllt, welcher die Maschine zukünftig betreuen wird.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder Ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Berlin, 17.04.2015

Patrick Hobusch

Danksagung

Nachdem es bis ins vierte Semester dieses Studiums alles andere als danach aussah, dass ich jemals den letzten Meilenstein in Form einer Bachelorarbeit erreichen würde, bin ich einfach nur überglücklich darüber, meinen ersten Abschluss im absoluten Wunschfach, der Informatik, in wenigen Wochen in den Händen halten zu dürfen.

Ich möchte mich daher besonders bei meiner Familie, bei meinem Vater Bernhard, meiner Mutter Annette, meinem Bruder Andre und vor allem bei meiner wunderbaren Freundin Maria Sparenberg für die Unterstützung in allen Formen bedanken, ohne die dieses Studium niemals möglich gewesen wäre.

Großer Dank gilt außerdem allen an dieser Bachelorarbeit Beteiligten des Fachbereiches, meinen Betreuern Franz Zieris (AG Software Engineering und Saros-Projekt) und Christian Salzmann (IT-Dienst) sowie meinen Gutachtern Prof. Dr. Lutz Prechelt (AG Software Engineering) und Prof. Dr. Elfriede Fehr (AG Programmiersprachen).

Inhaltsverzeichnis

1 Einleitung

Der Einsatz von Systemkonfigurationswerkzeugen kann für den Benutzer zu einer Reihe nützlicher Vorteile führen. Mit einer deklarativen Zustandsbeschreibung eines Systems wird gleichzeitig dessen Dokumentation erreicht. Einmal entwickelt, entstehen homogene Konfigurationsmechanismen für verschiedenste Anwendungen oder gar Betriebssysteme, welche sich auf beliebig vielen Maschinen anwenden lassen und was zu enormer Zeiteinsparung führen kann. Außerdem lässt sich auf diese Weise der gesamte Zustand von Systemen in Versionsverwaltungssystemen festhalten und so im Fehlerfall jederzeit ein alter Zustand wiederherstellen.

Der in dieser Arbeit betrachtete Build-Server des Saros-Projektes, Saros-Build genannt, kann leider keinen dieser Punkte vollständig für sich vorweisen. Es existiert zwar eine Dokumentationsseite, aber auf dieser konnte natürlich nicht jedes kleine Detail erfasst werden. Dabei ist Dokumentation im Saros-Projekt, einem Softwareprojekt mit 200% Fluktuation jährlich, enorm wichtig. Doch bevor sich diesem Thema ausführlicher gewidmet wird, soll zunächst das Saros-Projekt kurz vorgestellt werden.

1.1 Saros-Projekt

Das Saros-Projekt[1] ist ein Open Source Projekt der AG Software Engineering des Informatik-Instituts der Freien Universität Berlin. Saros selbst ist ein Plugin für die Entwicklungsumgebungen Eclipse[2], NetBeans[3] und IntelliJ[4] zur verteilten Paarprogrammierung in Echtzeit [54].

Ein wesentlicher Bestandteil der Entwicklung von Saros, welche hauptsächlich von Studenten in Rahmen von Softwareprojekten und Abschlussarbeiten erfolgt, ist die eingesetzte Build-, Test- und Integrationsinfrastruktur, mit der eine Vielzahl hilfreicher Prinzipien und Praktiken der Softwaretechnik umgesetzt werden. Zu diesen gehören vor allem dynamische und statische Softwaretests sowie ein Code-Reviews zur Förderung einer inkrementellen Arbeitsweise und insgesamt zum Erreichen einer höheren Softwarequalität trotz der genannten hohen Fluktuation.

Dieses System, mit dem vor allem die Menge der Anwendungen auf dem

[1] `http://www.saros-project.org`
[2] `http://www.eclipse.org`
[3] `https://www.netbeans.org`
[4] `https://www.netbeans.org`

Saros-Build Server gemeint ist, ist mit der Zeit *organisch* gewachsen und befindet sich nun in einem Zustand, von dem man zwar weiß, *dass* er funktioniert, aber eigentlich nicht genau *wie*.

1.2 Zielstellung

Die Idee dieser Arbeit entstand durch die Kenntnis des Systemkonfigurationswerkzeugs Puppet von der Firma PuppetLabs[5], mit dem ein System deklarativ in ähnlicher Syntax der dynamischen Programmiersprache Ruby konfiguriert wird. Der Zustand einer Maschine kann so exakt beschrieben und die Änderungen mit Versionsverwaltungssystemen wie Git genau erfasst werden. Manche Teams treiben den Einsatz solcher Systemkonfigurationswerkzeuge sogar so weit, dass sie Teile ihres Systems absichtlich zerstören und diese zerstörten Teile anschließend automatisch wiederherstellen lassen, um die Güte und Abdeckung ihrer Konfiguration zu testen [10]. So weit wollte man im Saros-Projekt zwar gar nicht gehen, aber man hat festgestellt, dass mit der Zeit wegen wachsender Abhängigkeiten in und unter den Diensten eine Angst entstand, Änderungen vorzunehmen. Die Idee war es also, das System in einer Art zu erfassen, die es ermöglicht, nur „auf einen Knopf drücken“ zu müssen, um alle Anwendungen auf einem beliebigen System wiederherzustellen.

Ein weiterer Aspekt, der die Analyse und die Möglichkeit der Wiederherstellung der bisherigen Systeme notwendig macht, ist das Wachstum des Projektes und seiner Infrastruktur. Saros wurde seit Beginn der Öffnung für die Entwicklungsumgebungen Netbeans und IntelliJ in verschiedene Komponenten zergliedert, welche alle eigene Build-Prozesse mit eigenen Tests durchlaufen und deshalb auch mehr Systemressourcen verbrauchen. Gewachsen ist, nicht zuletzt auch wegen des *Google Sommer of Code 2015* [53], auch die Zahl der Entwickler und mit wachsender Popularität auch die Zahl der Nutzer des XMPP-Servers, dem verwendeten Messaging-Dienst. Bis auf die Tests mit echten Eclipse-Instanzen laufen alle Dienste auf dem Saros-Build Server, welcher den Anforderungen an Antwort- und Build-Zeiten schon länger nicht mehr genügt und dessen Betriebssystem-Support Ende April 2015 endet. Auch hier gilt also der Wunsch, das alte System schnell wieder herstellen zu können, um Saros-Build auf eine neue virtuelle Maschine umziehen zu können.

Mit der Inbetriebnahme der neuen virtuellen Maschine soll die Adminis-

[5] `https://www.puppetlabs.com`

tration dieser wieder vollständig in die Hand des fachbereichseigenen IT-Dienstes übergehen. Auch dort gibt es den Wunsch, auf der neuen Maschine ein Systemkonfigurationswerkzeug einzusetzen. Der IT-Dienst hat bereits in der Vergangenheit einige solcher Werkzeuge evaluiert und sich dabei für Salt von der Firma Saltstack[6] entschieden, welches auf der dynamischen Programmiersprache Python basiert. Dieses soll deshalb in der nachfolgenden Arbeit zur Systemkonfiguration eingesetzt werden. Die einzigen Anforderungen des Saros-Projektes waren die Erhaltung aller alten Prozesse und die Möglichkeit, leicht Dienste, Plugins oder ähnliche Ressourcen hinzuzufügen oder bearbeiten zu können.

1.3 Aufbau der Arbeit

Im Rahmen dieser Arbeit wird zunächst die zugrundeliegende Problematik der *DevOps* vorgestellt, welche oft im selben Atemzug mit Systemkonfigurationswerkzeugen genannt werden und eine zentrale Rolle in der Zusammenarbeit von Entwicklern und Betreibern spielen. Es folgt eine kritische Analyse des bisherigen Servers, mit der die einzelnen Bestandteile ermittelt und teilweise bessere Lösungsvorschläge aufgezeigt werden. Anschließend werden zwei Werkzeugkategorien - zur Systemkonfiguration und zur Container-Virtualisierung - vorgestellt, mit denen sich das geplante Vorhaben umsetzen ließe. Nachdem die Wahl des Werkzeuges auf Salt gefallen ist, beschreibe ich meine Erfahrungen aus dem Versuch, Webanwendungen auf möglichst flexible Art und Weise zu *salzen*, ehe es mir gelingt, alle bis dahin unklaren Anforderungen zusammenzubringen und daraus eine geeignete Lösung zu entwickeln.

[6] `http://www.saltstack.com`

2 DevOps

Bei diesem Kapitel handelt es sich bis einschließlich Abschnitt 2.3 um eine Paraphrase der Grundlagenkapitel des Buches „DevOps for Developers“ von Michael Hüttermann [28].

Während sich die Nutzung agiler Methoden in den verschiedensten Unternehmensformen bereits stark etabliert hat und in IT-Firmen im Zuge dessen die Rollen der Programmierer, Tester und Qualitätssicherer im Begriff Entwickler (*Developers*) zusammengefasst wurden, klaffen zwischen den Entwicklern und den Betreibern (*Operators*) häufig noch große Kluften. Vor allem in traditionell ausgerichteten Unternehmen geraten Entwickler und Betreiber bei der Auslieferung von Software oft miteinander in Konflikt. Das liegt daran, dass Entwickler ihre Änderungen, wie neue Features, möglichst schnell zum Kunden bringen möchten, Betreiber jedoch an Stabilität interessiert sind und das Produktionssystem so selten wie möglich ändern möchten. Dazu kommt, dass Entwickler und Betreiber häufig verschiedene Werkzeuge einsetzen. So betreiben Entwickler auf ihrem Windows-PC möglicherweise einen Nginx-Webserver und OpenEJB-Anwendungsserver, während auf den Linux-Produktionsmaschinen ein Apache-Webserver und JBoss-Anwendungsserver zum Einsatz kommen.

Da diese Konflikte in einer kleineren Dimension auch für die Reintegration der Server des Saros-Projektes in die Obhut des IT-Dienstes eine Rolle spielen, sollen sie nachfolgend gesondert betrachtet und Prinzipien und Praktiken aufgezeigt werden, wie Development und Operations unter dem Konzept der DevOps näher zusammengebracht werden können.

2.1 Definition

Die Bezeichnung DevOps ist eine Zusammensetzung von Development und Operations. Development beschreibt hierbei den gesamten Software-Erstellungsprozess einschließlich Programmierung, Testen und Qualitätssicherung, während Operations den Prozess beschreibt, diese Software in Produktion zu bringen, zu halten, und die Produktionsinfrastruktur zu betreuen. Dies schließt auch System-, Datenbank- und Netzwerkadministration ein. DevOps charakterisiert vor allem den Software-Auslieferungsprozess und ist bestrebt die Freigabezyklen zu verbessern. Durch den Versuch schnelleres und besseres Feedback zu erhalten, trägt DevOps ganzheitlich dazu bei, Software hoher Qualität zu entwickeln.

Der Begriff DevOps wurde erstmalig auf der Konferenz *Devopsdays Ghent 2009* in Belgien [15] genannt. Alternative Bezeichnungen sind *agile Operations* oder *agile Systemadministration*.

Abgrenzung

Der Begriff DevOps bezeichnet keine Tätigkeitsbeschreibung in Berufstiteln oder Stellenanzeigen. Auch reine Sammlungen von Softwarewerkzeugen haben zunächst nichts mit dem DevOps-Begriff zu tun. Gleichzeitig aber spielen geeignete Werkzeuge vor allem in Kombination mit agilen Prozessen für den Automatisierungsprozess eine zentrale Rolle.

2.2 Konflikte zwischen Entwicklern und Betreibern

In traditionellen Organisationen werden verschiedene Bereiche oftmals streng in eigene Teams getrennt und haben somit meist auch eigene Teamleiter, welche unterschiedliche Ziele für ihr eigenes Teams verfolgen. Während die Entwickler beispielsweise meist daran gemessen werden, wie schnell sie neue Features entwickeln, werden die Betreiber eher danach beurteilt, wie lange die Server online sind und wie hoch die Reaktionszeiten der Anwendungen sind. Oder anders ausgedrückt: Betreiber werden danach bemessen, dass sich möglichst wenig ändert (*Änderungsängste*), während Entwickler danach bemessen werden, dass sich möglichst viel ändert (*Änderungszwang*).

In agilen Projekten werden die Bereiche der Programmierung und des Testens durch eine gemeinsame Qualitäts- und Entwicklungsverantwortung, und durch weniger Tätigkeitsbindung an den Projektrollen deutlich näher zusammengebracht. Aber obwohl Programmierer, Tester und sogar Geschäftsleiter mit agilen Methoden stärker miteinander kooperieren, arbeiten Betreiber häufig weiterhin in isolierten Gruppen. Fällt in dieser Konstellation das Produktivsystem aus, kommt es in der Schuldfrage zu gegenseitigen Vorwürfen zwischen Development und Operations - dem *Blame Game*.

Operations als Flaschenhals

Während sich die Entwickler bei einem Fehler mit einem Verweis auf die letzte funktionierende Programmversion schnell aus der Misere retten können, bleibt die Arbeit, die nötig ist, um das System wieder online zu bringen, stets an den Betreibern hängen und so gelten meist die Administratoren als die Verantwortlichen des Ausfalls. Mit der Verwendung agiler Methoden haben sich die Konflikte zwischen Development und Operations sogar noch

weiter zugespitzt: Prozesse wie Scrum, mit dem iterative und inkrementelle Softwareentwicklung verfolgt wird, streben an, das Geschäft und die Entwicklung zusammenzubringen und kurze Freigabezyklen einzuführen. Doch während sich dieses Vorgehen stark durchgesetzt hat und in vielen Unternehmen tägliche Releases zum Tagesgeschäft gehören, wurde oft vergessen, ebenfalls mehr Personal auf Seiten der Betreiber für die Softwarebereitstellung zur Verfügung zu stellen.

2.3 Prinzipien und Praktiken

Unter dem Konzept der DevOps werden einige empfohlene Prinzipien und Praktiken zusammengefasst. Das wichtigste und naheliegendste Prinzip ist eine größtmögliche Kooperation der Teams. Dies setzt in einer guten Unternehmenskultur eigentlich selbstverständliches Verhalten wie gegenseitiger Respekt oder geteilte Werte voraus. Von den Team- und Projektleitern sind gemeinsame Ziele festzulegen, zu denen sich die Teams dann selbst verpflichten. Dafür müssen Development und Operations wie *ein* Team behandelt werden. Werden diese Ziele erreicht, werden alle Teams belohnt. Ein einfaches Beispiel für ein solches Ziel sind möglichst viele Freigaben, die stabil laufen und in Produktion gehen. Kollektiver Besitz ist empfehlenswert, um das *Ein-Team-Gefühl* noch weiter zu stärken. Dies schließt möglichst alle Prozesse und Werkzeuge ein, also natürlich auch alle Formen von Fremdsoftware. Für die eigene Unternehmenssoftware empfiehlt sich ein gemeinsames Versionsverwaltungssystem von Entwicklern und Betreibern.

Neben geteilten Vorzügen gelten aber auch geteilte Pflichten. Kommt es zum Ausfall eines Dienstes, muss jeder Bereich wissen, welche Schritte notwendig sind, um das Problem zu lösen und den Dienst wieder in Produktion zu bringen. Dies erfordert aber auch, dass stets alle notwendigen Rollen mit den entsprechenden Fähigkeiten zur Verfügung stehen und diese auch dazu in der Lage sind, miteinander zu arbeiten. Hierfür ist ein gutes Training unerlässlich.

Zu den wichtigsten Praktiken zählen die Festlegung fester Entwicklungs- und Freigabeprozesse, aber auch die Nutzung von Werkzeugen, die helfen, den Freigabeprozess zu automatisieren.

Prozesse

Grundsätzlich sind Prozesse in einer DevOps-orientierten Umgebung wichtiger als die eingesetzten Werkzeuge, da die passenden Werkzeuge immer noch ermittelt werden können, wenn die Prozesse feststehen, aber nicht immer

umgekehrt. Prozesse spielen eine entscheidende Rolle bei der Interaktion zwischen verschiedenen Abteilungen. DevOps anzuwenden bedeutet nicht Abteilungen zusammenzulegen, aber sehr wohl, dass es interdisziplinäre Experten an den Schnittstellen dieser Abteilungen geben muss. Jeder Abteilung werden feste Aufgaben in einem vollständigen Auslieferungsprozess zugeordnet und der Bereich Operations muss in agile Methoden wie Scrum eingebunden werden.

Werkzeuge

Ein wichtiger Bestandteil von DevOps ist die Automatisierung des Auslieferungsprozesses, um die kurzen Freigabezykluszeiten zu erreichen. Neben den üblichen Build-, Deploy- und Testwerkzeugen haben sich in den letzten Jahren eine Reihe von Systemkonfigurationswerkzeugen etabliert, mit denen mithilfe von domänenspezifischen Sprachen (*Domain specific language*, *DSL*) die Eigenschaften von Systemen konfiguriert und in Versionsverwaltungssystemen gespeichert werden können.

Diese Systemkonfigurationswerkzeuge bieten eine Reihe von Vorteilen gegenüber herkömmlichen Systemadministrationsverfahren, zu denen vor allem große Sammlungen von Shell-Skripten gehören. Mit dem Einsatz solcher Werkzeuge wird eine höhere Abstraktion und somit eine größere Plattformunabhängigkeit allein durch die zugrundeliegenden Programmiersprachen erreicht. Die Beschreibung des Verhaltens der zu konfigurierenden Systeme erfolgt deklarativ, sodass auch die mehrmalige Ausführung in einem deterministischen Ergebniszustand resultiert. Zur Synchronisierung der Umgebung und zur automatischen Bereitstellung der Änderungen können außerdem Versionsverwaltungssysteme wie Git und Continious Integration Server wie Jenkins verwendet werden. Durch den zusätzlichen Einsatz von Werkzeugen wie Vagrant können sogar vollständige Virtualisierungen angewandt werden. Werden die entstandenen Skripte allen Bereichen bereitgestellt, haben sowohl Entwickler als auch Betreiber die Möglichkeit, an der Entwicklung des Systems mitzuarbeiten oder zumindest Einblick in diese zu erhalten, um zum Beispiel Fehler besser nachvollziehen zu können.

2.4 Bedeutung für das Saros-Projekt

Zwar stehen das Saros-Projekt und der IT-Dienst in keinem solchen Verhältnis wie oben beschrieben. Dennoch gibt es einige Analogien, die zu beachten sind, wenn der IT-Dienst zukünftig wieder für die Wartung des Saros-Build Servers zuständig ist und keines der Saros-Mitglieder mehr vollen Admi-

nistrationszugriff auf den Server erhält. Schließlich soll nicht jedes noch so kleine Plugin vom IT-Dienst bereitgestellt werden müssen. Außerdem könnte es durchaus vorkommen, dass spätere neue Anwendungen von jemandem auf dem Server bereitgestellt werden muss, der diese Anwendungen auch wirklich versteht.

Dazu ist es notwendig, dass beide Teams überhaupt zu einer guten Zusammenarbeit bereit sind. Im weiteren Verlauf muss beiden Seiten klar sein, welche Anforderungen die jeweils andere Seite an die Lösung hat und wie neue Anwendungen in die Infrastruktur des IT-Dienstes eingegliedert werden. Außerdem sollte dem zugrunde liegen, dass beispielsweise für die Konfigurationsautomatisierung dieselben Werkzeuge eingesetzt werden.

Da Saros-Build auch bei der Betreuung durch den IT-Dienst noch gewissermaßen dem Saros-Projekt gehört, spielt kollektiver Besitz hier eine wichtige Rolle. Zwar wird das Produkt dieser Arbeit in das Versionsverwaltungssystem des IT-Dienstes übergehen, aber es sollte auch zukünftig die Möglichkeit geben, dass Studenten oder andere Projektmitglieder darauf zugreifen können, gerade auch aus akademischer Sicht.

Des Weiteren müssen die Schnittstellen zwischen den beiden Abteilungen mit möglichst interdisziplinären Experten besetzt werden. Während dieser Arbeit ist das mit Franz Zieris seitens des Saros-Projektes und Christian Salzmann seitens des IT-Dienstes, ohne dies bewusst zu tun, insgesamt gut gelungen.

Am schwierigsten erweist sich die Automatisierung des Auslieferungsprozesses von Änderungen an der Konfiguration. Hierfür müssen sich beide Parteien vermutlich noch erst etwas mehr aufeinander einstellen und Erfahrungen mit der neuen Situation sammeln.

Im weiteren Verlauf dieser Arbeit wird erneut auf die Problematik der DevOps eingegangen. Kapitel 5 behandelt ausführlich, welche Fehler bei der Entwicklung von Software *für* Betreiber passieren können und Kapitel 7 resümiert die Gesamterfahrung nochmals.

3 Kritische Analyse des Servers

Der Saros-Build Server ist eine von derzeit drei virtuellen Maschinen (VMs), die im Auftrag der AG Software Engineering des Informatik-Instituts vom fachbereichseigenen IT-Dienst an der Freien Universität betrieben werden. Die weiteren Server sind Saros-Eclipse1 und Saros-Eclipse2, deren Hauptaufgabe jeweils in der Ausführung zweier Saros-Instanzen für automatische Funktionstests in Eclipse besteht. Eine vierte virtuelle Maschine, Saros-Firewall genannt, wurde bereits im Verlauf dieser Bachelorarbeit vom IT-Dienst deaktiviert, da sie nicht mehr benötigt wurde.

Auf dem Saros-Build Server werden primär eine Reihe von Webanwendungen ausgeführt, deren Einsatzzwecke jedoch weit über das reine Bauen (Kompilieren) der Saros-Software hinausgehen. Das Herzstück bilden Jenkins, ein Dienst zur kontinuierlichen Integration (*Continous Integration*, *CI*) von Code-Änderungen in bestehenden Code, und Gerrit, ein Dienst zur Verwaltung von Codereviews.

Um die Automatisierung der Konfiguration des Servers zu ermöglichen, ist es zunächst notwendig, alle Ressourcen des aktuellen Servers, also Dateien, Benutzer, Dienste, etc. zu erfassen und zu entscheiden, ob und in welcher Form diese auf dem neuen System vorhanden sein sollen.

3.1 Betriebssystem

Zunächst wird das Betriebssystem des aktuellen Saros-Build Servers betrachtet. Auf diesem läuft Ubuntu 10.04 LTS, auch *Lucid Lynx* genannt [59]. Der Suffix LTS steht dabei für *Long Term Support*, also eine längere Supportperiode. LTS-Versionen werden bei Ubuntu fünf Jahre lang mit Aktualisierungen versorgt, von denen vor allem Sicherheitsupdates eine wichtige Rolle spielen [39]. Der Support des am 29. April 2010 erschienenen Lucid Lynx endet damit am 29. April 2015, was also sowieso dringend ein Betriebssystem-Update notwendig macht [59].

Aus eigener Erfahrung mit späteren Ubuntu-Versionen weiß ich, dass die Anwendung von Updates unter Ubuntu durchaus kritisch sein kann. So ist es mir beispielsweise einmal geschehen, dass während einer Aktualisierung die Systemsprache auf chinesisch umgestellt wurde, was sich nur noch über die Konsole beheben ließ, da es mir praktisch nicht mehr möglich war, die grafische Benutzeroberfläche zu verstehen. Ein anderes mal funktionierte nach einem Update der Bootloader *GRUB 2* nicht mehr, welcher benötigt wird,

um das Betriebssystem überhaupt starten zu können [24]. Wenn man, wie im Falle des Saros-Projektes, mit einer VM arbeitet, die durch einen externen Dienstleister bereitgestellt wird, ist es dem Projekt nicht immer selbst möglich, solche Fehler zu beheben.

```
$ curl -L http://saros-build.imp.fu-berlin.de/x
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>403 Forbidden</title>
</head><body>
<h1>Forbidden</h1>
<p>You don't have permission to access /x
on this server.</p>
<hr>
<address>Apache/2.2.14 (Ubuntu) Server at \
saros-build.imp.fu-berlin.de Port 80</address>
</body></html>
```

Abbildung 1: Versionsausgabe beim Apache httpd Webserver

Der Aufruf einer nicht existierenden Seite auf einem Apache2-Webserver genügt bei Standardeinstellungen, um die Version des Webservers zu erfahren. So lassen sich ohne großen Aufwand vollautomatisiert potentiell angreifbare Webserver ermitteln.

Derartige Bedenken waren es auch, die das Saros-Projekt dazu veranlassten, über einen längeren Zeitraum keine Updates mehr zu installieren bzw. die Maschine hinterher nicht neu zu starten, womit das System keine Sicherheitsaktualisierungen erhielt und angreifbar wurde: Während des Bearbeitungszeitraumes dieser Bachelorarbeit erhielt ein Angreifer vermutlich über eine Lücke im Webserver Zugriff auf den Saros-Build Server, wo er anschließend Code [23] für die Sicherheitslücke *CVE-2014-0196* (Linux-Kernel) [14] nachlud, um sich damit volle Administrationsrechte für das System zu verschaffen. Abbildung 1 legt dar, wie einfach es ist, verwundbare Webserver zu ermitteln. Außerdem macht dieser Vorfall deutlich, dass die Server des Saros-Projektes zukünftig besser gleich vom IT-Dienst gewartet werden sollten und für weniger Risiko bei Systemupdates eine stabilere Linux-Distribution wie Debian eingesetzt werden sollte. Mit Debian habe ich, im Gegensatz zu Ubuntu, in mehreren Jahren Nutzung noch nie Probleme nach Updates gehabt, was auf eine konservativere Aktualisierungsfreigabepolitik zurückzuführen ist. Da

Ubuntu auf Debian basiert, fällt der Wechsel bei Servern, die ohne grafische Oberfläche auskommen, besonders leicht [60].

3.2 Programme, Pakete und Dienste

Um den Benutzern die Installation neuer Software zu erleichtern, stellen die meisten Linux-Distributionen die populärsten und wichtigsten Anwendungen als sogenannte Pakete bereit, welche dann über ein Paketverwaltungssystem installiert werden können. Diese Pakete enthalten neben der eigentlichen Software zusätzlich Metadaten, zu denen Versionsnummern, Änderungshistorien oder Lizenzinformationen gehören. Außerdem werden viele Pakete mit Präoder Postinstallationsskripten ausgeliefert, mit denen die Software beispielsweise als Dienst registriert werden kann, welcher nach einem Systemstart automatisch gestartet wird. Debian und Ubuntu verwenden mit APT, dem *Advanced Packaging Tool*, dasselbe Paketverwaltungssystem, womit sichergestellt ist, dass alle Pakete für Ubuntu auch bei Debian erhältlich sind [7].

Es ist allerdings vor allem bei den Webanwendungen nicht damit zu rechnen, dass alle als Debian-Pakete erhältlich sind, weil es schwieriger ist, nicht eigenständige Software über Pakete in bestehende Infrastruktur zu integrieren. Damit alle Anwendungen später auch auf dem neuen System verfügbar sind, müssen sie zunächst erfasst und anschließend ermittelt werden, ob diese auch als Pakete zur Verfügung stehen.

Jenkins

Jenkins[7] ist ein quelloffenes Projekt, welches hauptsächlich in der Programmiersprache Java entwickelt wird [31]. Es wurde von seinem Mutterprojekt Hudson abgespalten, nachdem dieses von der Firma Oracle übernommen wurde, und wird seitdem eigenständig entwickelt [63]. Die Einstellung des Verhaltens der kontinuierlichen Integration erfolgt über sogenannte Jobs, welche über die Weboberfläche konfiguriert werden. Alle Jobs beziehen ihren Code in der Regel aus einem Versionsverwaltungssystem wie CVS, SVN oder Git. Mithilfe verschiedener Build-Tools oder ganzer Befehlssequenzen (i.d.R. Shell-Skripte) lässt sich der heruntergeladener Quellcode automatisiert kompilieren, testen oder freigeben. Außerdem können die Jobs untereinander Abhängigkeiten besitzen, zeit- oder aktionsgesteuert gestartet werden und verschiedene Plugins verwenden, mit denen sich der Funktionsumfang erweitert lässt. Im Saros-Projekt wird die kontinuierliche Integration sogar

[7] `https://www.jenkins-ci.org`

so weit getrieben, dass einige Jobs ganze Eclipse-Instanzen starten, um automatisiert zu überprüfen, ob das Saros-Plugin nach Hinzunahme von neuem Code noch die wichtigsten Aufgaben erfüllen kann.

Auf dem aktuellen System wurde Jenkins mit einer WAR-Datei (*Web Application Archive*) [12] installiert, welche auf der Jenkins-Homepage erhältlich ist und mithilfe des Apache Tomcat Webservers ausgeführt wird. Jenkins kommt ohne eine Datenbank aus, da sämtliche Konfigurationen und Ressourcen wie Benutzer oder Jobs in XML-Dateien gespeichert werden. Der Dienst kann aber auch eigenständig gestartet werden und nutzt dann den bereits enthaltenen Webserver. Außerdem ist Jenkins auch als Debian-Paket in einem Drittanbieter-Repository erhältlich und muss auf dem neuen System deshalb nicht mehr manuell installiert werden [32]. Auch ist es nicht notwendig, dass Jenkins im Tomcat Webserver ausgeführt wird, da bereits ein eigener Webserver enthalten ist [34]. Jenkins wird nicht auf dem Standardwebport ausgeführt, soll aber dennoch über diesen erreichbar sein. Diese Möglichkeit wird bei Betrachtung des Paketes Apache2 vorgestellt.

Gerrit

Gerrit[8] ist ebenfalls ein quelloffenes Projekt, geschrieben hauptsächlich in der Programmiersprache Java und entwickelt von der Firma Google zur Verwaltung von Git-Repositories und zur Durchführung von Softwaredurchsichten [21]. Dies bedeutet, dass neuer Quellcode zunächst von einer oder mehreren Personen einer Durchsicht unterzogen werden muss, bevor dieser in den Hauptzweig (*Master*) eines Git-Repositories aufgenommen wird. Außerdem kann auch der Erfolg von Builds oder automatischen Tests, welche in einer CI wie Jenkins ausgeführt werden, als Bedingung für die Hinzunahme von neuem Code zum Master ausgewertet werden. Dies reduziert gleichzeitig den Review-Aufwand, da Code, dessen Builds oder Tests fehlschlagen, keiner Durchsicht unterzogen werden muss.

Auch Gerrit wurde mit einer WAR-Datei installiert, welche auf der Gerrit-Homepage erhältlich ist und ebenfalls mithilfe des Apache Tomcat Webservers ausgeführt wird. Ich musste allerdings feststellen, dass die Konfiguration in diensteigenen Dateien und in Tomcat-Dateien teilweise redundant ist. Da das Gerrit-Projekt zur Ausführung den Jetty-Webserver empfiehlt, welcher in der WAR-Datei bereits enthalten ist, wird auch Gerrit zukünftig eigenständig, also ohne den Tomcat Webserver, ausgeführt werden [19]. Im

[8] `https://code.google.com/p/gerrit`

Gegensatz zu Jenkins ist Gerrit jedoch nicht als Debian-Paket erhältlich.

Damit überhaupt eigene Git-Repositories verwaltet werden können, in denen ohne manuelle Konfiguration von *pre-receive* Hooks direkte Commits auf den Master verhindert und über das Webinterface durchgeführt werden können, wird Gerrit mit einem eigenen SSH-Dienst ausgeliefert. Dafür kann zusätzlich unter anderem die Verschlüsselungsbibliothek Bouncy Castle installiert werden. Die Review-Daten selbst werden in einer MySQL-Datenbank gespeichert.

```
$ mysql

mysql> show databases;

+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| phpesp             |
| reviewboard        |
| reviewdb           |
| sonar              |
| testlink           |
+--------------------+
7 rows in set (0.14 sec)
```

Abbildung 2: Anzeige der MySQL-Datenbanken über das CLI

Das Command Line Interface wird mit dem Befehl *mysql* gestartet. In diesem erwirkt die Eingabe *show databases* die Ausgabe aller Datenbanknamen. Bei den Datenbanken *information_schema* und *mysql* handelt es sich um Datenbanken des Datenbanksystems. Die Datenbanken *phpesp*, *reviewboard* und *testlink* entsprechen dem Namen ihrer Anwendungen, *sonar* gehört zur Anwendung SonarQube und bei *reviewdb* handelt es sich um die Datenbank der Anwendung Gerrit.

MySQL

MySQL[9] gilt als das populärste quelloffene Datenbanksystem weltweit und wird auf dem Saros-Build Server für alle Anwendungen genutzt, die eine Datenbank benötigen [42]. Seit der Übernahme der Firma Sun Microsystems gehört MySQL ebenfalls zu Oracle [43]. Wegen Unzufriedenheiten mit der Entwicklung unter der Leitung von Oracle spalteten Michael Widenius und weitere Entwickler zunächst das Datenbanksystem SkySQL von MySQL ab, welches im Jahr 2014 in MariaDB[10] unbenannt wurde [64]. Wegen der Abspaltung ist MariaDB zu großen Teilen binärkompatibel zu MySQL. Dies ist nützlich, da das Datenbankverwaltungssysteme einfach ausgetauscht werden kann, ohne die Datenbank selbst zu verändern. Zumindest langfristig soll MariaDB nach Anforderung des IT-Dienstes auch auf dem Saros-Build Server zum Einsatz kommen. Zunächst jedoch wird MySQL betrachtet, welches im Debian-Repository als Paket erhältlich ist [41].

Durch die Anzeige der in MySQL vorhandenen Datenbanken lassen sich leicht weitere auf dem Saros-Build Server installierte Anwendungen finden, indem über die Kommandozeilenschnittstelle (*Command Line Interface*, *CLI*) der Befehl *show databases* ausgeführt wird. Dieser zeigte über die ausgegebenen Namen die Präsenz der Anwendungen phpESP, Review Board, Gerrit, SonarQube und TestLink auf (siehe Abbildung 2).

Apache2

Bei phpESP und TestLink handelt es sich um PHP-Anwendungen bzw. bei Review Board um eine Python-Anwendung, welche alle zur Laufzeit interpretiert werden. Um diese als Webanwendungen auszuführen, muss in den Webserver ein PHP- bzw. Python-Interpreter integriert sein. Der quelloffene HTTP Server von Apache[11], dessen korrekter Name eigentlich Apache httpd ist, aber oft mit Apache2 abgekürzt wird, enthält mit Modulen wie mod_fcgid (Fast CGI, u.a. für PHP) und mod_wsgi (für Python) genau solche Interpreter zur Ausführung von Skripten [2, 3].

Neben der Bereitstellung statischer und dynamischer Webseiten können mit dem Modul mod_proxy auch andere Webserver in die Pfadhierarchie des Apache2 eingebunden werden. Dies ist immer dann sinnvoll, wenn es mehrere Webserver gibt, da diese nicht alle an die Standardwebports 80 bzw. 443

[9] `http://www.mysql.com`
[10] `https://www.mariadb.org`
[11] `http://httpd.apache.org/`

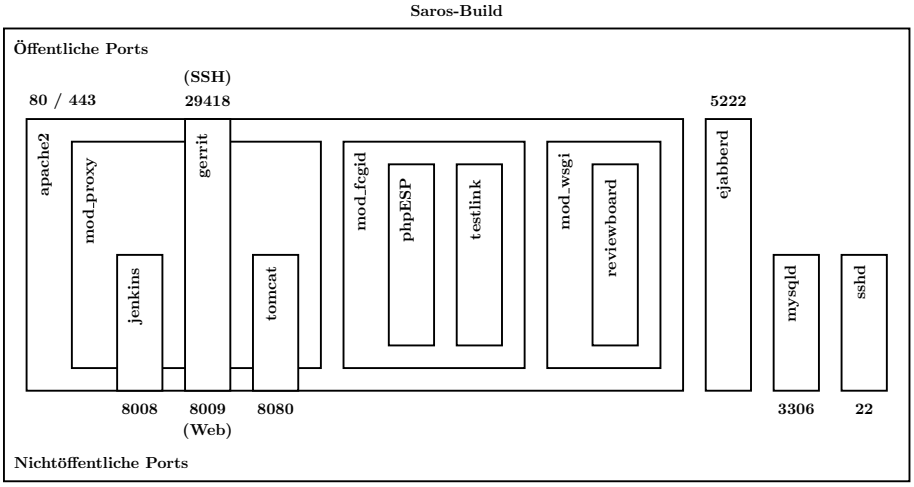

Abbildung 3: Schematische Darstellung der Dienste des neuen Saros-Build Servers

In dieser reduzierten Darstellung des neuen Servers entsprechen die inneren Rechtecke den Diensten oder Teilen eines Dienstes, welche entweder nichtöffentlich - dann haben sie nur die halbe Höhe - oder öffentlich sind. Gerrit, Jenkins und die Anwendungen im Tomcat-Webserver sind eigenständige, nichtöffentliche Webanwendungen, welche mithilfe des Moduls *mod_proxy* des Apache2-Webservers veröffentlicht werden. Der SSH-Dienst von Gerrit ist dagegen öffentlich. TestLink und phpESP können nur innerhalb des Moduls *mod_fcgid* ausgeführt werden. Review Board kann nur innerhalb des Moduls *mod_wsgi* ausgeführt werden. Ebenfalls öffentlich ist der XMPP-Dienst ejabberd. Dienste wie das Datenbanksystem MySQL oder Systemdienste wie SSH werden dagegen nichtöffentlich sein.

gebunden werden können, man aber keine (anderen) Ports in URLs nennen möchte. Die URL `http://localhost/dienst` könnte so als Proxy auf die Dienst-URL `http://localhost:1234` zeigen, wobei 1234 der Port ist, an den dieser Dienstwebserver gebunden ist, was beispielsweise für Jenkins, Gerrit und die Tomcat-Dienste hilfreich ist [4] (siehe auch Abbildung 3).

phpESP[12] ist eine Anwendung zur Erstellung, Verwaltung und Auswertung von Umfragen [44]. Saros-Benutzer werden nach einer Sitzung zu einem *Quick Survey* eingeladen, welches mit phpESP durchgeführt wird.

[12] `http://sourceforge.net/projects/phpesp`

Review Board[13] ist ebenfalls eine Code-Review System und der Vorgänger von Gerrit aus der Zeit, als im Saros-Projekt noch SVN genutzt wurde [48].

TestLink[14] ist eine Anwendung zur Verwaltung von Testfällen. Mit ihr können natürlichsprachlich Testfälle formuliert werden, welche dann manuell oder automatisch ausgeführt werden. Es ist sogar möglich, die Ergebnisse der Unit-Tests eines Jenkins-Jobs automatisch zu TestLink zu übertragen. Dafür wird im Quellcode lediglich vor den entsprechenden Test-Klassen eine *@TestLink*-Annotation benötigt [57]. Momentan wird TestLink nicht verwendet, jedoch soll die Möglichkeit, dies wieder zu tun, offen gehalten werden.

Apache2 ist im Debian-Repository als Paket erhältlich, die Anwendungen phpESP, Review Board und TestLink jedoch nicht [1].

Tomcat

Apache Tomcat[15], kurz Tomcat, ist ein quelloffener Webserver, der die Java Servlet und JavaServer Pages Technologien implementiert und damit die Ausführung von *Web Application Archives* ermöglicht, in denen auch Jenkins, Gerrit und der Saros-Statistikserver vorliegen [6]. Da Jenkins und Gerrit zukünftig eigenständig ausgeführt werden, wird Tomcat nur noch für den Saros-Statistikserver benötigt, welcher nicht eigenständig gestartet werden kann, da er nicht mit einem integrierten Webserver ausgeliefert wird.

Der Saros-Statistikserver ist einfacher Dienst, der die Daten entgegen nimmt, die übertragen werden, wenn ein Saros-Nutzer der anonymen Übermittlung von Nutzungsdaten zustimmt.

Tomcat ist im Debian-Repository als Paket erhältlich [5].

SonarQube

SonarQube[16] ist ein ein Werkzeug zur statischen Code-Analyse. Zu den getesteten Metriken gehören Kommentare, Coderegeln, mögliche Bugs, die Komplexität, Unit-Tests, Duplikationen und die Architektur und der Softwareentwurf [56]. Im Saros-Projekt wird SonarQube jedoch hauptsächlich für ein eigenes Auswertungsplugin genutzt, mit dem, zusammen mit der ebenfalls

[13] `https://www.reviewboard.org`
[14] `http://www.testlink.org`
[15] `http://tomcat.apache.org/`
[16] `http://www.sonarqube.org`

eigenen DSL Archnemesis[17], die Einhaltung der vorgegebenen Architektur geprüft wird [8].

Archnemesis wird über ein Shell-Skript installiert und stellt dann ein Webinterface und einen Port bereit, über den es von verschiedenen Build-Tool-Plugins angesprochen werden kann.

ejabberd

ejabberd[18] ist ein quelloffener XMPP Server, der in der Programmiersprache Erlang entwickelt wurde und auf dem Saros-Build Server von Saros-Nutzern verwendet werden kann, wenn diese bei keinen anderen Jabber-Dienst registriert sind [17]. Auch ejabberd ist im Debian-Repository als Paket erhältlich [16].

Systemdienste

Neben den oben genannten Anwendungen, die vor allem für das Saros-Projekt wichtig sind, gibt es natürlich noch eine Reihe von Systemdiensten, die für den IT-Dienst wichtig sind und welche nicht öffentlich sein müssen oder gar sollen. Auch der SSH-Dienst gehört dazu, denn das Saros-Projekt wird keinen SSH-Zugriff mehr benötigen. An dieser Stelle sei bereits vorweg genommen, dass diesen Systemdiensten im weiteren Verlauf der Arbeit meinerseits keine Aufmerksamkeit geschenkt werden muss und eine tiefere Analyse deshalb hinfällig ist.

3.3 Benutzer und Gruppen

Die Datei */etc/passwd* enthält eine Liste aller Benutzeraccounts, welche im Folgenden nur noch kurz Benutzer genannt werden. Diese Benutzer lassen sich in drei Kategorien unterteilen: der Benutzer des Saros-Projektes, Mitglieder des IT-Dienstes und Benutzer für installierte Dienste. Der Benutzer *administrator* des Saros-Projektes wird, wie eben beschrieben, nicht länger benötigt und den Benutzern des IT-Dienstes muss, wie den Systemdiensten, keine weitere Aufmerksamkeit geschenkt werden.

Es verbleiben also nur noch die Benutzer und Gruppen der Pakete. Die einzige Besonderheit bilden die oben genannten Webanwendungen, welche in einer

[17] `https://www.github.com/saros-project/archnemesis`
[18] `https://www.ejabberd.im/`

Webgruppe zusammengefasst werden, damit sie Dateien in das Webverzeichnis kopieren dürfen. Ansonsten gilt hier das übliche Muster, nach dem für jeden Dienst ein Benutzer mit gleichnamiger Gruppe angelegt wird.

3.4 Weitere Ressourcen

Welche Ressourcen spielen auf einem System noch eine zentrale Rolle? Einerseits wären dies sogenannte Cron-Jobs, bei denen es sich um zeit- oder aktionsgesteuerte Befehle handelt, die vom Betriebssystem mit den Rechten eines bestimmten Nutzers gestartet werden können. Mit einer einfachen Schleife über alle Benutzer und der Abfrage ihrer Cron-Jobs (siehe Abbildung 4) konnte ich jedoch schnell feststellen, dass gar keine definiert waren.

```
$ for user in $(cut -f1 -d: /etc/passwd); \
do echo $user; crontab -u $user -l; done
```

Abbildung 4: Einfache Schleife zur Auflistung aller Cron-Jobs

Es wird über alle Benutzer iteriert und anschließend für jeden Benutzer die Liste der definierten Cron-Jobs ausgegeben [25].

Andererseits wären da noch ganz allgemein alle relevanten Dateien, eine Auflistung dieser würde den Rahmen der Arbeit jedoch sprengen. Zu den wichtigsten zu sichernden Dateien zählen alle Dienstkonfigurationen sowie die Daten aus den Datenbanken, welche in einem Datenbank-Dump gesichert werden können, und die Anwendungsdaten von Jenkins und ejabberd.

Mit all diesen Ressourcen sind die relevanten Bestandteile des Systems erfasst und die wichtigsten Aspekte für den neuen Server genannt. Nun ist es an der Zeit, ein geeignetes Werkzeug für die Automatisierung zu bestimmen.

4 Evaluation möglicher Werkzeuge

Der Begriff des Systemkonfigurationswerkzeugs wurde nun bereits in der Zielstellung des Saros-Projektes und von Michael Hüttermann, dem Autor des in Kapitel 2 thematisierten Buches „DevOps for Developers“, als eine mögliche Lösung zur Konfigurationsautomatisierung genannt. Zu den bekanntesten Vertretern dieser Werkzeugkategorie gehören Puppet[19], Chef[20] und Ansible[21] [11]. Puppet ist vor allem wegen von der Community bereitgestellten Lösungen für die Konfiguration von Jenkins [13] und Gerrit [40] die interessanteste Wahl, mit der ich mich deshalb auch als erstes beschäftigte.

Der initiale Kontakt des Saros-Projektes zum IT-Dienst per E-Mail offenbarte dann sogar noch ein weiteres solches Werkzeug namens Salt[22], manchmal auch SaltStack genannt, obwohl dies eigentlich der Name der entwickelnden Firma ist. Während eines anschließenden, gemeinsamen Treffens stellte sich heraus, dass der IT-Dienst den Saros-Build Server mittelfristig übernehmen würde. Außerdem war zu erfahren, dass in der Vergangenheit bereits eine Evaluation möglicher Systemkonfigurationswerkzeuge durchgeführt und sich in Folge dessen für den Einsatz von Salt entschieden wurde. Zu den entscheidenden Gründen zählten die höhere Geschwindigkeit, die zugrundeliegende Programmiersprache und das verwendete Templating-System, welche nachfolgend näher vorgestellt werden. Ich müsste also schon triftige Gründe gehabt haben, mich der vorausgegangenen Wahl zu widersetzen.

Um überhaupt einschätzen zu können, ob es solch triftige Gründe gab, musste jedoch zunächst eine eigene Evaluation durchgeführt werden, auch, weil mit Virtualisierungscontainern noch wenigstens ein ganz anderes Konzept für das geplante Vorhaben existierte. Zunächst aber stelle ich im Folgenden die beiden Werkzeuge Puppet und Salt vor.

4.1 Systemkonfiguration mit Puppet

Der populärste und älteste Vertreter der Kategorie der Systemkonfigurationswerkzeuge ist Puppet. Puppet ist, wie alle oben genannten Systemkonfigurationswerkzeuge, ein quelloffenes Projekt. Entwickelt wird es seit 2005 und in der Programmiersprache Ruby.

[19] `https://www.puppetlabs.com`
[20] `https://www.chef.io`
[21] `http://www.ansible.com`
[22] `http://www.saltstack.com`

```
# COMMENT
RESOURCE { NAME:
  ATTRIBUTE => VALUE,
  ...
}
```

```
package { 'apache2':
  ensure => latest,
}

service { 'apache2':
  ensure  => running,
  enable  => true,
  require => Package['apache2'],
}
```

Abbildung 5: Deklarative Ressourcenkonfiguration mit Puppet

Dargestellt sind das Grundschema der Puppet Ruby DSL (oben) und die Bereitstellung des laufenden Apache2-Dienstes, wenn die Installation der spätesten Version des Webservers bereits erfolgt ist (unten) [9].

Wer beginnt mit Puppet zu arbeiten, stößt als erstes auf sogenannte Manifest-Dateien, welche wie alle reinen Puppet-Konfigurationsdateien die Endung *.pp* tragen. Mit diesen Manifests werden sämtliche Ressourcen eines Betriebssystems konfiguriert, also Benutzer, Gruppen, Pakete, Dienste, Dateien usw. Manifests lassen sich zu logischen Einheiten, den Modulen, zusammenfassen, um die Struktur übersichtlich zu halten. Das Hauptmanifest eines jeden Verzeichnisses trägt den Namen *site.pp*. Zur Konfiguration der Ressourcen in den Manifests wird eine eigene Puppet Ruby DSL verwendet. In dieser werden die Ressourcen deklarativ mithilfe ihres Namens konfiguriert, indem Ressourcen-spezifischen Attributen entsprechende Werte zugewiesen werden (siehe Abbildung 5) [61].

Bei der Konfiguration von Dateien spielt das Templating-System eine zentrale Rolle. Mit diesem lassen sich Dateien aus Templates generieren, indem Puppet die Daten an das Templating-System weiterreicht und diese dort den Regeln folgend eingesetzt werden. Ein Templating-System erweist sich so-

```
PUPPET_HOME
|-- manifests
|   |-- site.pp
|   |-- ...
|-- modules
|   |-- module_1
|   |   |-- site.pp
|   |   |-- ...
|   |-- module_2
|       |-- site.pp
|       |-- ...
|-- modules-forge
    |-- module-forge_1
    |-- ...
```

Abbildung 6: Einfache, mögliche Verzeichnisstruktur von Puppet

Obwohl Puppet mit beliebigen Modulverzeichnissen umgehen kann, hat sich in meiner Einarbeitungszeit folgende einfache Verzeichnisstruktur bewährt. Die Ordner *manifests* und *modules* werden in die eigene Versionsverwaltung aufgenommen, die Forge-Module werden von Puppet verwaltet, für die es deshalb einen eigenen Modulordner gibt.

mit als nützlicher Bestandteil für jedes Systemkonfigurationswerkzeug. Bei Puppet kommt das Templating-System ERB (*ERuby*) zum Einsatz, mit dem Ruby-Code in das Template eingebettet werden kann. Die Dateiendung von Templates lautet *.erb* [37].

Noch einmal zurück zu den Modulen: Sowohl von PuppetLabs, der entwickelnden Firma von Puppet, als auch von der Puppet-Community können Module über das sogenannte Forge-Repository bereitgestellt werden. So kann dem Nutzer ein großer Teil der üblichen Modul- und Templatinglogik abgenommen werden und es müssen nur noch wesentliche Parameter gesetzt werden. Module lassen sich über die Konsole mit Puppet-Befehlen aus dem Forge-Repository installieren und aktualisieren. Um diese Fremdmodule von der eigenen Versionsverwaltung und den eigenen Modulstrukturen auszuschließen, empfiehlt es sich, diese in ein eigenes Modulverzeichnis zu installieren (siehe Abbildung 6).

Puppet kann sowohl standalone (*masterless* oder *serverless*) als auch in einem Client-Server-Modell eingesetzt werden. In letzterem Fall wird der Server

Master und die Clients werden *Agents* genannt.

4.2 Systemkonfiguration mit Salt

Salt wird seit 2011 von der Firma SaltStack in der Programmiersprache Python entwickelt und ist somit von den am Anfang dieses Kapitels genannten vier Systemkonfigurationswerkzeugen das jüngste. Hauptentwickler von Salt und technischer Geschäftsführer von SaltStack ist Thomas Hatch [58]. Die Tatsache, dass er zuvor ein aktiver Nutzer von Puppet war [45], lässt darauf schließen, dass Salt aus der eigenen Unzufriedenheit mit Puppet heraus entwickelt wurde [51].

Salt hat viele grundlegende Konzepte von Puppet übernommen, ihnen jedoch neue Namen gegeben. Der Server heißt in Salt ebenfalls *Master*, Clients werden *Minions* genannt. Salts Pendant zu den Manifests sind die States. Sie tragen die Endung *.sls* im Dateinamen und werden voreingestellt in der Auszeichnungssprache YAML[23] geschrieben, für die Implementierungen in zahlreichen Programmiersprachen, so auch in Python, existieren. Auch States lassen sich zu Gruppen zusammenfassen und werden dann Formulas genannt [26, 62] (siehe Abbildung 7).

Formulas können, wie Module in Puppet, auch von SaltStack oder der Community bereitgestellt werden. Der Anlaufpunkt für Formulas ist die *saltstack-formulas* genannte Organisation auf GitHub[24]. Sie können also einfach mit Git herunterladen werden und benötigen somit im Gegensatz zu Puppet keinen proprietären Installationsmechanismus [50].

Als Templating-System kommt bei Salt Jinja[25] zum Einsatz. Der Funktionsumfang [35] von Jinja kann auch in den States genutzt werden, fällt aber deutlich geringer aus als der von ERB in Puppet. So ist es im Zusammenwirken von Jinja und YAML nicht möglich, mithilfe von Klassen eigene States mit eigenen Attributen zu entwickeln, wie dies in Puppet mit Ruby möglich ist (vgl. Abbildung 9 in Kapitel 5). Dies schränkt die Möglichkeiten der Generalisierung von Formulas leider stark ein.

Ein Mechanismus, den es in Puppet nicht gibt, sind die sogenannten Pillars. Dabei handelt es sich um eine Key-Value-Baumstruktur, die zum Setzen von Parametern genutzt werden kann. Pillars werden nur an die relevanten

[23] `http://www.yaml.org`
[24] `https://www.github.com/saltstack-formulas`
[25] `http://jinja.pocoo.org`

```
# COMMENT
NAME:
  RESOURCE.MODUL:
    - ATTRIBUTE: VALUE
  ...
}
```

```
apache2:
  pkg.latest: []
  service.running:
    - require:
      - pkg: apache2
```

Abbildung 7: Deklarative Ressourcenkonfiguration mit Salt

Die Salt DSL ähnelt stark der Puppet Ruby DSL, jedoch wird auf ein *ensure*-Attribut verzichtet. Stattdessen gibt es für die verschiedenen *ensure*-Werte aus Puppet in Salt jeweils ein Ressourcen-Modul, welches dann eine exakte Menge aller möglichen Attribute kennt. Außerdem können die States sogar zusammengefasst werden (siehe unten, vgl. Abbildung 5), was Wiederholungen vermeidet [26].

Minions übertragen und eignen sich so vor allem zum Speichern von Sicherheitsrelevanten Daten [52].

Im Umgang mit Verzeichnissen, in denen sich States, Formulas oder Pillars befinden, ist Salt wesentlich flexibler. In der Konfigurationsdatei des Masters können beliebig Verzeichnisse hinzugenommen und verschiedenen Umgebungen zugeordnet werden. States und Formulas müssen in die *file_roots* und Pillars in die *pillar_roots* aufgenommen werden [49]. Dies ermöglicht es, verschiedene Repositories für verschiedene Maschinen zusammenzuführen. Das würde eine Anwendungskonfiguration durch das Saros-Projekt und eine Betriebssystemkonfiguration durch den IT-Dienst in getrennten Repositories ermöglichen (siehe Abbildung 8). Tatsächlich jedoch wird die gesamte Konfiguration zum IT-Dienst übergehen, wo Mitglieder des Saros-Projektes jedoch die Möglichkeit hätten, sich durch Pull-Requests an der Entwicklung zu beteiligen.

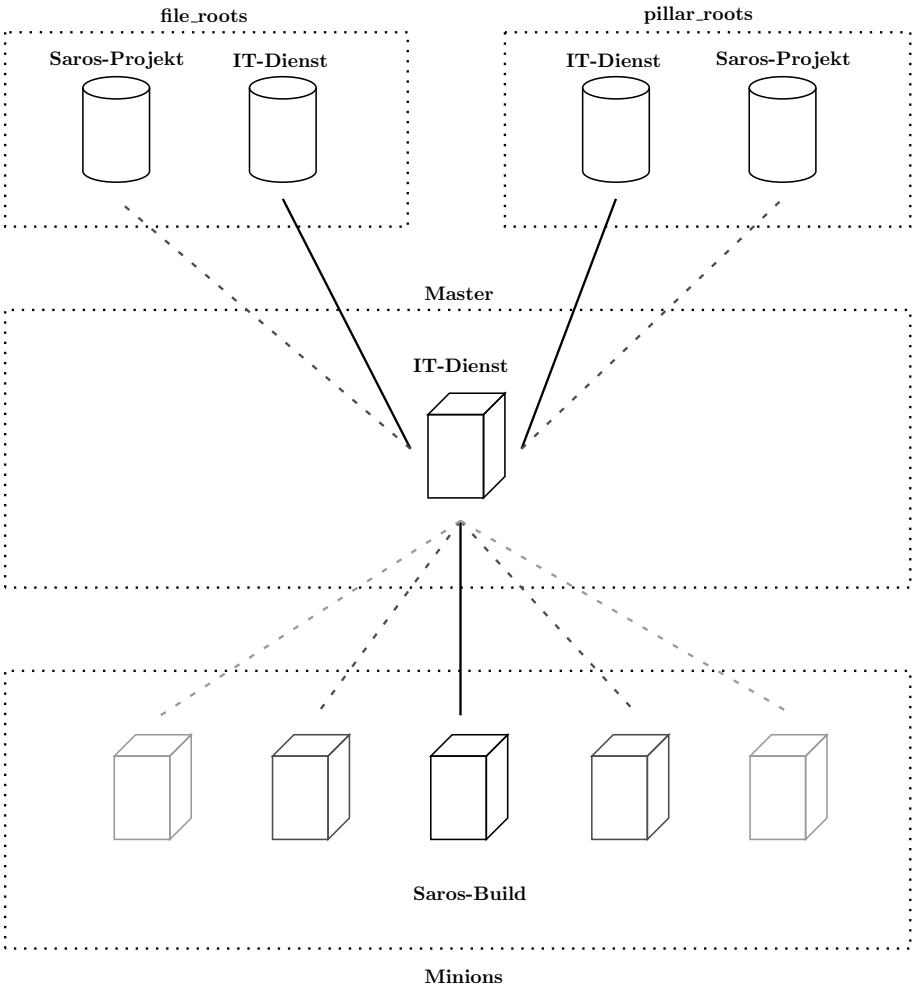

Abbildung 8: Mögliche Client-Server-Konfiguration von Salt

Salt bietet die Möglichkeit, verschiedene Repositories für die Konfiguration einzelner Maschinen oder ganzer Organisationen zusammenzuführen. So wäre es möglich, Saros-spezifische Anwendungen vom Saros-Projekt konfigurieren zu lassen, und Betriebssystem- und Netzwerkkonfiguration durch den IT-Dienst vorzunehmen. Dafür würden jeweils die verschiedenen Repositories (dargestellt als „Tonnen“) in die *file_roots* (für States und Formulas) bzw. *pillar_roots* (Pillars) eingehängt und über den Master an verschiedene Minions, wie den Saros-Build Server, (jeweils dargestellt als „Boxen“) ausgeliefert und angewandt werden.

4.3 Virtualisierung mit Docker

Ein Trend der letzten Jahre, dem ein anderer Ansatz zugrunde liegt, sind Virtualisierungswerkzeuge. Zwar gibt es schon seit langem das Konzept der virtuellen Maschine, welches der IT-Dienst ja unter anderem zur Bereitstellung von Servern wie dem Saros-Build einsetzt (siehe Kapitel 3), aber virtuelle Maschinen bieten kein Automatisierungspotential für einzelne Anwendungen. Mit der sogenannten Container-Virtualisierung wird genau dies möglich. Die Anwendungen werden für eine solche Virtualisierung aufbereitet und können anschließend auf jeder Maschine ausgeführt werden, die dieses Virtualisierungssystem unterstützt. Dazu kommt, dass die Anwendungen in einer Sandbox ausgeführt werden und eine Sicherheitslücke einer einzelnen virtualisierten Anwendungen kein Sicherheitsrisiko für das restliche System bedeutet [38].

Das populärste Virtualisierungswerkzeug ist Docker[26]. Auch der IT-Dienst beschäftigt sich derzeit mit der Evaluierung von Docker, was aber auch bedeutet, dass ein Einsatz in Produktivsystemen derzeit noch nicht in Frage kommt. Außerdem können ohnehin nicht alle Dienste am Fachbereich *dockerisiert* werden, sodass der bisherige Ansatz mit Salt nicht durch Docker ersetzt werden kann.

4.4 Die Entscheidung zugunsten von Salt

Obwohl Puppet mit vorhandenen Lösungen für die Konfiguration von Jenkins und Gerrit in Form von umfangreichen Modulen zumindest für eben diese beiden Webanwendungen ein erheblicher Gewinn gewesen wäre, reichte dies nicht aus, um den IT-Dienst von einem Einsatz von Puppet zu überzeugen. Schließlich wurden vom IT-Dienst in Salt bereits Formulas für zahlreiche andere Anwendungen, darunter Apache2, MySQL, SSH und zahlreiche andere Betriebssystem- und Netzwerkanwendungen, entwickelt. Außerdem konzentrierte sich meine Evaluierung vorrangig auf die Eignung für die Entwicklung einer komplexeren Konfigurationslogik, nicht jedoch auf eine Eignung für eine solch komplexe Infrastruktur, wie der IT-Dienst sie betreut. Des Weiteren gibt es zahlreiche Blog-Einträge, in denen die Autoren von ihrem Erfahrungen mit Puppet berichten und einen vorgesehen Wechsel zu Salt begründen [36, 46]. Und so schloss ich mich der Evaluierung des IT-Dienstes an, was sich wegen zusätzlicher Anforderungen des IT-Dienstes im weiteren Verlauf dieser Arbeit (siehe Kapitel 6.1) als richtige Entscheidung herausstellen sollte.

[26] `https://www.docker.com`

5 Der Versuch, Webanwendungen zu salzen

Webanwendungen - Entwickler diskutieren ihre Beliebtheit immer wieder kontrovers [47]. Sie werden vor allem in als Skriptsprachen verschrieenen Programmiersprachen mit meist dynamischem Typsystem entwickelt, die nicht kompiliert, sondern zur Laufzeit interpretiert werden [55] - oder in der Programmiersprache Java, deren Beliebtheit ebenso umstritten ist [18], wie die von Webanwendungen allgemein. Sie benötigen für ihre Ausführung stets zusätzliche Plugins, Frameworks oder Webserver, lassen sich deshalb aus Paketen heraus installiert schlecht in bereits bestehende Infrastruktur integrieren und verbrauchen deswegen oft viele Systemressourcen. Auch ihre Konfigurationsmöglichkeiten sind problematisch, denn mit Konfigurationen in Auszeichnungssprachen wie XML, direkt im Quellcode von Programmiersprachen wie PHP oder gar in Datenbanken lassen sich solche Webanwendungen deutlich schlechter administrieren als klassische Systemanwendungen mit zumeist Plaintext-Konfigurationen.

Wie also können solche Webanwendungen *gesalzen* werden, wie es die Salt-Nutzergemeinde nennt, wenn Anwendungen und ihre Konfiguration mit Salt automatisiert administriert werden? Der Blick zum im Auswahlverfahren in Kapitel 4 ausgeschiedenen Systemkonfigurationswerkzeug Puppet zeigt, dass dies grundsätzlich möglich ist.

Jenkins-Modul für Puppet

Das ursprünglich von R. Tyler Croy entwickelte Jenkins-Modul[27] für Puppet wurde mittlerweile auf GitHub in die Jenkins-Organisation aufgenommen und ermöglicht es den Betreibern einer Jenkins-CI, diese mit Puppet zu konfigurieren. Doch wie funktioniert das? Diese Frage soll nur kurz angeschnitten werden, um zu zeigen, dass das Vorhaben durchaus auch in Salt realistisch ist.

Das Jenkins-Modul muss in Puppet zunächst installiert und anschließend an einem Knoten, hinter dem sich mit dem Einsatz von Wildcards auch mehrere Hosts verbergen können, eingebunden werden. Durch Anwendung der Änderungen in Puppet wird Jenkins dann an Port 8080 gestartet (siehe Abbildung 9).

Auf die in Kapitel 4.1 genannte Art und Weise können Ressourcen wie Jobs, Benutzer, Plugins usw. eingerichtet werden. Sämtliche Ressourcen werden in

[27] `https://github.com/jenkinsci/puppet-jenkins`

```
node 'www.example.com' {
    include jenkins
}
```

```
jenkins::job { 'test-build-job':
    config => template(
        "${templates}/test-build-job.xml.erb"
    ),
}
```

Abbildung 9: Deklarative Jenkins-Konfiguration mit Puppet

Stelle Jenkins unter `http://www.example.com:8080` bereit (oben). Anschließend definiere die einfachste Form eines Jenkins-Jobs durch Einsatz eines Templates (unten) [13].

Jenkins mit XML konfiguriert und diese Dateien in entsprechenden Unterverzeichnissen des Jenkins-Hauptverzeichnisses gespeichert.

Mit Puppet können die XML-Dateien wie berichtet über Templates dynamisch aufgebaut und mit Inhalt gefüllt werden. Die enorme Komplexität jedoch, dies mit einem Systemkonfigurationswerkzeug umzusetzen, entsteht durch den Einsatz von Plugins, welche für den sinnvollen Einsatz von Jenkins unerlässlich sind und von denen es mehr als 1000 verschiedene gibt [63]. Beim Einsatz von Plugins wird an relevanten Stellen die Weboberfläche (*Front-End*) um weitere Bedienelemente und das Verhalten des Grundsystems (*Back-End*) um neue Funktionalität erweitert. Außerdem - und dies ist der relevante Punkt - müssen natürlich auch die XML-Dateien um Tags erweitert werden, um die Plugin-spezifischen Konfigurationen speichern zu können. Für eine sinnvolle Konfiguration der Ressourcen muss der Administrator das gesamte System inklusive (potentiell aller) Plugins enorm gut kennen und verstehen - oder es müsste für jedes einzelne Systemkonfigurationswerkzeug und für jedes Plugin einen ähnlich komplexen und zum Plugin redundanten Mechanismus in Form eines Templating-Systems entwickelt werden. Der sehr kurzen Dokumentation des Jenkins-Moduls lässt sich leider nicht entnehmen, wie dies genau funktioniert (vgl. Abbildung 9), allerdings ist das wegen des Einsatzes von Salt anstatt von Puppet auch nicht weiter

relevant.

Die Plugins selbst werden normalerweise über das Front-End aus einem Jenkins-Repository oder eigenen, selbst definierten Quellen heruntergeladen und nach einem Neustart automatisch installiert. Der Blick in den Code des Jenkins-Moduls offenbart, dass bei jeder Ausführung von Puppet in der Phase der Plugin-Installation zunächst sämtliche Plugins gelöscht und anschließend neu heruntergeladen werden, was aufgrund der Möglichkeit, Ruby in der Puppet DSL zu nutzen, nur schwer nachvollziehbar ist. Das Problem dabei ist, dass die zuletzt angegebenen Plugin-Versionen, definiert in den Manifests, in einer späteren Ausführung nicht mehr vorliegen und somit kein Vergleich möglich ist. Auch auf Dateiebene ist wegen fehlender Versionsnummern in den Dateinamen nicht ersichtlich, um welche Plugin-Version es sich bei den Dateien handelt.

5.1 Der Versuch, Jenkins zu salzen

Um mich nicht zu sehr von der Implementierung anderer, ähnlicher Module beeinflussen zu lassen, habe ich mir die eben genannten Details, wie die Installation der Plugins im Jenkins-Modul für Puppet anfangs nicht angesehen. Interessiert hat mich allerdings schon, *wie* diese Dienste dann später konfiguriert werden können und natürlich auch, ob bereits Lösungen für Salt existieren. Tatsächlich gibt es eine solche Jenkins-Formula[28], welche jedoch mit dem Jenkins-Modul für Puppet nicht zu vergleichen ist. Die einzigen Aufgaben dieser Formula sind zum aktuellen Zeitpunkt die Installation des Jenkins-Dienstes nach dem Standardmuster und die Kapselung der Anwendung durch Bereitstellung eines Proxies für den Nginx-Webserver.

Also versuchte ich eine eigene Jenkins Formula zu schaffen. Zuerst legte ich einfache States nach dem Standardmuster (siehe Abbildung 7 in Kapitel 4.2) zur Installation des Jenkins-Dienstes an. Dann untersuchte ich verschiedene Verfahren zur Jenkins-Konfiguration. Eine einfache Möglichkeit bestand darin, einfach alle Jenkins-XML-Dateien sowie Plugins usw. in einem Versionsverwaltungssystem aufzunehmen und diese mit Salt einfach an die entsprechende Stelle zu kopieren. Allerdings machte diese „Version 0“ nicht viel Sinn, denn jede Änderung, die über das Webinterface erfolgen würde, würde von Salt wieder mit den Dateien aus dem einmal angelegten Versionsverwaltungssystem überschrieben werden. Und so eignete sich diese Möglichkeit eher dazu, den Umgang mit Salt zu üben. Danach begann ich damit, Tem-

[28] `https://www.github.com/saltstack-formulas/jenkins-formula`

plates und States für Jobs zu entwickeln. Jedoch stellte ich dieses Vorhaben wegen der bereits genannten Bedenken hinsichtlich der Plugin-Kompatibilität schnell wieder ein, auch wenn das Vorhaben grundsätzlich schnell funktionierte und sich einfache Jobs ohne Plugin-Funktionalität erstellen ließen.

Ebenfalls mit Templates konnten so States für die Benutzer-Konfiguration geschaffen werden. Doch auch hier kamen schnell Zweifel auf, ob dies denn ein sinnvolles Vorgehen sei. Brauchte man überhaupt die Möglichkeit der Benutzer-Konfiguration, wenn die Konfiguration ganzheitlich mit Salt möglich wäre? Vermutlich nicht. Und besteht das Risiko der fehlenden Plugin-Kompatibilität nicht auch bei Benutzern? Ja, es gibt auch Plugins, die das Modell der Benutzer verändern (z.B. das Role Strategy Plugin [33]), auch wenn im Saros-Projekt kein solches in Betrieb ist.

Für den Download von Dateien hält Salt ein recht elegantes Verfahren bereit, von dem ich großer Hoffnung war, es für den Download der Plugins nutzen zu können. Aus dem Plugin-Namen und einer optionalen Versionsnummer ließe sich mithilfe des Jinja Templating-Systems eine URL auf das Plugin-Repository erstellen. Man benötigte dafür nur noch eine Prüfsumme, damit Salt die lokale Datei mit der herunterzuladenden vergleichen kann. Die Prüfsumme hätte sogar in Form einer weiteren URL angegeben werden können. Doch auch dieses Vorhaben scheiterte: Das Jenkins-Repository stellt keine Dateien bereit, die die Prüfsummen der einzelnen Plugins beinhalten. In der Deklaration der Plugins stets die händisch berechneten Prüfsummen mit anzugeben, konnte jedoch auch nicht die Lösung sein und so suchte ich nach anderen Möglichkeiten. In den States selbst gab es sonst keine weiteren, da eben immer eine Prüfsumme zum Vergleich mit den lokalen Dateien benötigt wurde und da der sehr begrenzte Funktionsumfang von Jinja auch nicht vorsah, Dateien herunterzuladen.

Eine andere Möglichkeit bestand in der Ausführung eines Skriptes, welches die Plugins herunterladen konnte, indem dem Skript von Salt der Name und die Versionsnummer eines jeden Plugins übergeben wurde. Wegen der größeren Betriebssystemunabhängigkeit und des höheren Abstraktionsniveaus entschied ich mich für die Verwendung von Skripten in der Programmiersprache Python. Außerdem konnte man sich wegen des Einsatzes von Salt auf das Vorhandensein einer Python-Installation verlassen. So ließen sich komfortabel alle Parameter auswerten, Plugins herunterladen und deren Versionsnummer in einer externen, versteckten Datei speichern, um nicht jedes Mal alle Plugins neu herunterladen zu müssen. Doch während diese Lösung für Plugins das erste Mal zufriedenstellend funktionierte, ahnte ich bereits,

dass der Gebrauch von *externen* Skripten nicht die richtige Herangehensweise war. Immerhin zeigte sich der IT-Dienst wegen der bereits investierten Arbeit *zunächst* damit einverstanden.

Noch größere Probleme bereiteten die von Jenkins für verschiedene Aufgaben verwendeten Softwaretools. Mit dem Ziel auch hier zunächst einfach den alten Zustand mit Salt wiederherzustellen, schenkte ich der Frage, wieso die Installation von Werkzeugen wie Maven innerhalb eines Tools-Ordners im Jenkins-Hauptverzeichnis erfolgte und welchen Sinn die verschiedenen Versionen hätten, in denen einige Werkzeuge vorlagen, zunächst nicht genug Aufmerksamkeit. Mangels einer geeigneten Lösung für dieses Problem und mangels eines gemeinsamen Repositories für all diese Werkzeuge, begann ich nun meine anfänglichen Prinzipien zu verletzten und sammelte von Webseiten wie SourceForge, dem Maven-Repository und verschiedensten Herstellerwebseiten die URLs aller Werkzeuge ein, lud diese herunter und trug die händisch ermittelten Prüfsummen der Dateien in die Salt Pillars ein, um zunächst überhaupt an dieses Ziel zu gelangen. Auch dies funktionierte zufriedenstellend, Salt entpackte sogar alle Archive in die gewünschten Ordner und lud die Dateien nur herunter, wenn die gewünschten Zielverzeichnisse noch nicht existierten. Doch nun war es fast schon offensichtlich, dass dies sowohl aus Nutzer- als auch aus Betreibersicht nicht die Anforderungen erfüllen würde. Mangels anderer Lösungen fand ich mich jedoch zunächst damit ab und widmete mich dem zweiten großen Dienst - Gerrit.

5.2 Der Versuch, Gerrit zu salzen

Ich war zuversichtlich mit Gerrit mehr Erfolg zu haben, da beispielsweise die Konfigurationsdateien deutlich einfacher aufgebaut sind. Aber es stellte sich schnell heraus, dass die Konfiguration von Gerrit ähnliche Probleme mit Plugins und anderen Softwarebibliotheken (*Libraries*) aufweist. Außerdem lässt sich Gerrit wegen fehlender Pakete für Debian leider nicht so leicht installieren wie Jenkins.

Die attraktivste Herangehensweise, die benötigte WAR-Datei einfach herunterzuladen, schied wieder wegen der fehlenden Prüfsummen, aber auch wegen fehlender Plugin-Pakete aus: Im Gegensatz zur Gerrit WAR-Datei konnte man die Plugins von der Webseite nur nach Anmeldung mit einem Google-Konto herunterladen. Dabei besitzt Gerrit ein paar Standardplugins, welche im Laufe der Zeit aus der Kernanwendung herausgelöst wurden, ohne die Gerrit für Open-Source-Projekte wie das Saros-Projekt jedoch wenig Sinn macht. Als Beispiel sei das *Replication*-Plugin genannt, welches es unter

anderem ermöglicht, das Git-Repository in andere Repositories zu kopieren (z.B. nach GitHub) [22].

Also entschloss ich mich, einfach den Quellcode von Gerrit mit Git herunterzuladen und die benötigten Java-Archive einfach selbst zu bauen. Da die Plugins und die Kernanwendung einen gemeinsamen Release-Zyklus hatten, musste dafür mit Salt immer nur eine Versionsnummer verwaltet werden. Zwar liegt auch der Gedanke nahe, die Gerrit WAR-Datei herunterzuladen und dann nur noch die Plugins selbst zu bauen, jedoch hängt der Build dieser Standardplugins vom Build der Gerrit-Kernanwendung ab, sodass dies nicht möglich war. Die eh schon komplizierte Bereitstellung des Gerrit-„Pakets“ wurde jedoch noch dadurch erschwert, dass das verwendete Build-Tool Buck ebenfalls nicht als Paket erhältlich ist und deshalb auch erst als Git-Repository bereitgestellt und dann noch mit dem Build-Tool Ant gebaut werden muss. Das GitHub-Plugin, welches das Saros-Projekt gerne einsetzen würde, jedoch mit den bisherigen Gerrit-Versionen nicht funktionierte, konnte dagegen eigenständig mit Maven gebaut werden.

Im weiteren Verlauf entstand eine riesige Menge von States für die Installation von Gerrit. Wegen zahlreicher und verschiedenster Arten von Abhängigkeiten und „Zustandsübergangsauslösern“ nahm die Salt-Formula schon die Form eines komplexen endlichen Automaten an. Schon wieder hatte ich den Eindruck, dass Salt für dieses Vorhaben nicht geschaffen worden war. Anstatt mit Salt das System zu konfigurieren, hantierte ich schon für die Bereitstellung eines einzigen Dienstes mit drei verschiedenen Build-Tools. Dass dieser Automat nie vollends funktionierte, lag leider daran, dass bedingte Befehlsausführungen mit verschiedenen Abhängigkeiten in Salt nicht das erwartete Verhalten zeigten und die entsprechenden Fehlerbehebungen noch nicht für Debian released wurden [30].

Aber auch sonst war dieser Ansatz nicht zufriedenstellend, das Problem mit den Libraries war hier ebenfalls ein ungelöstes und auch die Frage, wie denn eigentlich mit Daten umgegangen wird, die sich täglich ändern, war noch nie wirklich gestellt worden. Plötzlich brach das letzte Viertel der Bearbeitungszeit dieser Bachelorarbeit an. Ich hatte viele Arbeitsstunden in Code investiert, der die Anforderungen letztendlich nicht erfüllte, viele entstandene Produkte radikal wieder aussortiert und wusste eigentlich überhaupt nicht mehr, wie ich mit Salt eine geeignete Konfigurationsautomatisierung vornehmen sollte.

Und so fand ich mich selbst in Kapitel 2 über die DevOps-Problematik wie-

der. Ich stellte fest, dass ich ein Produkt für Developers entwickelte, eines, an dem schnell viele Änderungen durchgeführt werden konnten, aber keines, welches die Stabilität und Einfachheit mit sich brachten, die sich Operators wünschen. Genau genommen kannte ich die Anforderungen des IT-Dienstes nicht einmal. Und so wurde es dringend Zeit für ein längst überfälliges Gespräch, an dem *beide* Parteien teilnahmen.

6 Abschließende Implementierung

Das gemeinsame Treffen und die vorhergehende Präsentation meiner letzten Zwischenergebnisse schafften endlich Klarheit zu den Anforderungen beider Parteien und zeigten Lösungen auf, wie diese in Einklang gebracht werden konnten.

6.1 Vollständige Anforderungen

Das Saros-Projekt erwartet weiterhin, dass der aktuelle Prozess mit dem neuen System nicht verändert wird. Der große Gewinn durch den neuen Server soll darin liegen, dass Änderungen an den eigens konfigurierten Anwendungen im Fehlerfall rückgängig gemacht werden können, um die Änderungsängste zu überwinden. Dies vollständig deklarativ zu tun, schied jedoch bereits wegen der Komplexität eines möglichen Templating-Systems vor allem bei Jenkins aus (siehe Kapitel 5.1). Dennoch all die anderen Anwendungen vollständig deklarativ zu konfigurieren, erschien wegen ähnlicher Bedenken und dem Ziel einer möglichst homogenen Lösung jedoch wenig sinnvoll. Außerdem sollen Dienste und andere Dienstbestandteile wie Plugins leicht hinzuzufügen und wieder zu entfernen sein.

Der IT-Dienst legt großen Wert auf die Sicherheit des Systems, was sich für mich vor allem in einen so nicht erwarteten, stark reduzierten Internetzugriff äußert. Alle benötigten Programme und Bibliotheken dürfen nicht einfach aus dem Internet heruntergeladen werden. Vor allem Webseiten wie GitHub oder SourceForge sind zu meiden, da jeder dort beliebige Dateien hochladen kann. Dies bedeutete jedoch nicht, dass beispielsweise das Jenkins-Repository nicht mehr zugänglich gemacht werden kann, um Plugins zu installieren. Außerdem soll Salt nicht mehr als Build-Tool „missbraucht“ werden. Vor allem die Anforderungen des IT-Dienstes waren mir während meiner Versuche, möglichst *dynamische* Salt-Formulas zu schaffen, noch nicht bekannt.

6.2 Dienstinstallation mit Debian-Paketen

Statt die Dienste mit Salt herunterzuladen oder zu bauen, sollte für jeden Dienst, der noch nicht als Debian-Paket existierte, ein solches angelegt werden. Spätestens an dieser Stelle wären auch die in Kapitel 4 und 5 vorgestellten Puppet-Module nutzlos gewesen, da sie genau so arbeiteten, wie von mir mit Salt geplant. Für diese Saros-spezifischen Debian-Pakete wurde ein eigenes Git-Repository angelegt, in dem auch alle Binärdateien oder Dateiarchive gespeichert wurden. Durch das eigene Repository wird verhindert, dass

andere Saros-Repositories durch Binärdateien belastet werden, da Git mit solchen naturgemäß nicht gut umgehen kann. Das Anlegen der Dateien, die für die Paketierung benötigt werden, folgt einer strengen Anleitung, die in der Debian-Dokumentation beschrieben wird [27, 29] (siehe Abbildung 10). Wann immer nun ein Dienst geändert oder hinzugefügt werden soll, muss das entsprechende Paket bearbeitet oder angelegt werden. Der IT-Dienst hätte sowohl die Möglichkeit, das Repository automatisiert zu überwachen oder sich manuell benachrichtigen zu lassen und die Pakete dann zu bauen und in einem eigenen Debian-Repository bereitzustellen.

Für das komplizierteste Paket Gerrit existiert glücklicherweise eine Vorlage[29], die ich nutzen kann. Die anderen Pakete wie phpESP, TestLink und Review Board müssen nur im richtigen Verzeichnis entpackt werden und sind deshalb wesentlich leichter zu erstellen.

Über die Anwendungen selbst wird mit der Erstellung von Debian-Paketen bereits eine große Kontrolle erreicht, da sich alle Änderungen sehr gut nachvollziehen lassen. Wie können nun die Anwendungsdaten so erfasst werden, dass sich auch bei ihnen alle Änderungen nachvollziehen und rückgängig machen lassen?

6.3 Datei-Änderungsüberwachung mit Git

Der Vorschlag des IT-Dienstes zur Datei-Änderungsüberwachung war so einfach, dass ich selbst nie ernsthaft darüber nachgedacht hatte. Man erstellt einfach im Root-Verzeichnis des Servers ein Git-Repository und begrenzt die zu erfassenden Daten mit *.gitignore*-Dateien. In diese werden alle Dateien oder Verzeichnisse aufgenommen, die nicht im Versionsverwaltungssystem gespeichert werden sollen. Um Verzeichnisse später flexibel verschieben zu können, werden in den *.gitignore*-Dateien keine geschachtelten Pfade angegeben. Stattdessen erhält jedes Verzeichnis, wenn benötigt, eine eigene solche Datei und speichert in diesen immer nur die direkten, auszuschließenden Kinder des Verzeichnisses.

Auszuschließen sind - immerhin geht es ja nur um reine Anwendungsdaten, wie die Jobs in Jenkins - sämtliche Binärdateien bzw. Verzeichnisse, die nur solche enthalten. Außerdem müssen wirklich sicherheitsrelevante Daten ebenfalls von der Versionierung ausgeschlossen werden.

[29] `https://www.github.com/dnaeon/gerrit-debian`

```
GERRIT-DEBIAN
|-- debian
|    |-- changelog
|    |-- control
|    |-- copyright
|    |-- gerrit.default
|    |-- init.d
|    |-- postinst
|    |-- preinst
|    |-- rules
|    |-- ...
|-- src
     |-- gerrit.war
```

Abbildung 10: Die wichtigsten Bestandteile des Gerrit-Pakets

Das Debian Gerrit-Paket enthält ein *src*-Verzeichnis, in welchem die WAR-Datei liegt (vgl. *Gerrit* in Kapitel 3.2). Außerdem enthält es ein *debian*-Verzeichnis, in welchem sämtliche Metadaten definiert werden. Dazu gehören die Änderungshistorie mit Versionsnummern (changelog), Paketnamen, Paketeigentümer und Abhängigkeiten (control), Lizenzinformationen (copyright), Prä- und Post-Installationsskripte (preinst, postinst) sowie Installationsregeln (rules). Des Weiteren enthält das Verzeichnis in diesem Paket die Gerrit-Standardkonfigurationsdatei (gerrit.default) sowie das Skript zum Starten des Gerrit-Dienstes (init.d) [20].

Ein Skript überprüft die Daten dann auf Änderungen. Dafür eignet sich ein Cron-Job, welcher alle paar Minuten das Repository auf Änderungen überprüft, neue Dateien in die Versionsverwaltung aufnimmt und Änderungen an bestehenden Dateien übernimmt. Dieses Zeitintervall muss klein genug sein, um nicht zu viele kleinere Änderungen zu verpassen, andererseits aber auch groß genug, um bei der Wiederherstellung eines alten Zustands nicht ständig neue Commits zu erhalten. 15 Minuten scheinen für den Anfang ein vernünftiger Wert zu sein, welcher im Laufe der Zeit der neu gewonnen Erfahrung angepasst werden kann.

Doch wofür wird jetzt noch Salt benötigt?

```
debian_imp_repo:
  pkgrepo.managed:
    - name: >
        deb http://simpel.imp.fu-berlin.de/
        debian jessie main
    - file: /etc/apt/sources.list.d/debian-imp.list
    - key_url: >
        https://simpel.imp.fu-berlin.de/
        debian/imp-debian-repo.asc
    - gpgcheck: 1

gerrit:
  pkg.latest:
    - require:
      - pkgrepo: debian_imp_repo
  service.running:
    - enable: True
```

Abbildung 11: Gerrit-Installation mit Debian und Salt

Gerrit wird mit Salt installiert, indem zunächst das fachbereichseigene Paket-Repository angegeben wird. Dafür wird vor allem die URL des Repositories und der Schlüssel benötigt. Anschließend kann, abhängig von diesem Repository, das neueste Gerrit-Paket installiert und der Dienst gestartet werden. Aus Darstellungsgründen mussten die URLs umgebrochen werden.

6.4 Einfache Systemkonfiguration mit Salt

Salt kann nun auf die einfache Art eingesetzt werden, die der IT-Dienst sich wünscht. Für jeden Dienst werden nur noch Benutzer, Gruppen, Pakete (ggf. aus dem eigenen Debian-Repository), Dienste und wenige Dateien mit wirklich sicherheitsrelevanten Daten deklariert. Jedes Paket folgt so demselben Schema (siehe Abbildung 11). Und das wichtigste: Es werden keine Downloads mehr durchgeführt und es wird keine Software mehr gebaut.

7 Evaluation des Ergebnisses

Die enorme Bedeutung des 2. Kapitels wurde mir erst mit Abschluss dieser Arbeit und mit dem Scheitern des in Kapitel 5 beschrieben *Salzens* der Webanwendungen richtig bewusst. Klar, DevOps, das hat etwas mit Werkzeugen für die Systemautomatisierung zu tun. Aber wie dort beschrieben, sind es nicht die Werkzeuge, sondern die Qualität der Schnittstellen und der eingesetzten Prozesse zwischen zwei verschiedenen Abteilungen, hier den Entwicklern des Saros-Projektes und den Betreibern des IT-Dienstes, die über den Erfolg der Zusammenarbeit entscheiden.

Interessanterweise haben sich die Anforderungen an das Endprodukt eigentlich nie richtig geändert. Wieso also hat es so lange gedauert, bis ich alle Anforderungen zusammen hatte? Nun, sie wurden einfach nicht ausgesprochen. Dies ist zwar ein bekanntes Phänomen, aber es überrascht schon ein wenig, wenn eigentlich beiden Parteien, die ja sogar denselben fachlichen Hintergrund haben, die ganze Zeit über klar ist, was das Produkt am Ende leisten soll. Das Problem war also nicht, dass nicht verstanden wurde, *was* die Anforderungen sind, sondern die Tatsache, dass sich beide Seiten nicht vorstellen konnten, dass die Anforderungen, *wie* dies umzusetzen war, unklar sein könnten.

Wie sind meine Ergebnisse einzuordnen?

Mit den Ansätzen aus Kapitel 5 war ich auf einem guten Weg, sehr flexibel konfigurierbare Salt-Formulas zu schaffen. Mit einem einfachen Eintrag in die Salt-Pillars hätten beispielsweise Jenkins-Plugins bereitgestellt werden können, oder hätte sich Gerrit vollautomatisch aktualisieren lassen. In den beschriebenen Grenzen funktionierte dies sogar sehr gut und ähnliche Puppet-Module beweisen, dass es eine Nachfrage für solch flexible Lösungen gibt. Bei unserem IT-Dienst gibt es die aber eben nicht. Direkter Internetzugriff ist ebenso zu vermeiden wie eine zu hohe Automatisierung in Aktualisierungsvorgängen und so wurde in enger Zusammenarbeit eine Lösung auf Basis von bewährter Software gefunden, mit der in einfachen Einzelschritten dieselben Resultate erreicht werden.

Für alle Lösungen, die ich im Rahmen dieser Arbeit gefunden habe, bleibt aber insgesamt der Eindruck zurück, dass sich ein gesamtes System nie vollständig mit einem Systemkonfigurationswerkzeug automatisieren lässt. Und dies wird mit dem Einsatz komplexer Webanwendungen immer schwieriger. Die naive Idee, nach dem Aufsetzen einer neuen Maschine „auf einen Knopf

drücken“ zu können und danach alle Anwendungen, Daten und andere Ressourcen eines alten Systems komplett wiederhergestellt zu haben, ist unrealistisch.

Gleichzeitig aber bieten Werkzeuge wie Puppet oder Salt eine gute Möglichkeit, das Betriebssystem und einfachere Grunddienste zu konfigurieren und damit auch zu dokumentieren. Mit der Erstellung eigener Debian-Pakete kommen auch komplexere Anwendungen dazu, die sonst nicht als Paket erhältlich sind. Statt sich nun in die verstecktesten Winkel des Servers begeben zu müssen, liegen alle eingesetzten Anwendungen wohlgeordnet in Form von States oder kleinerer Formulas beieinander. Wenn die Umsetzung bis zur Präsentation abgeschlossen ist, gibt es exakt einen einheitlichen Weg, beispielsweise neue Plugins zu den Webanwendungen hinzuzufügen. Durch die Erfassung aller nicht mit Salt konfigurierbarer Daten mit Git erhält das Saros-Projekt endlich die Möglichkeit, fehlerhafte Änderungen rückgängig zu machen. Und mit dem Umzug auf die neue virtuelle Maschine übergibt das Saros-Projekt die eigene Verantwortung an das erfahrene Team des IT-Dienstes.

Literaturverzeichnis

[1] *Apache httpd für Debian Wheezy*. 13. Apr. 2015. URL: `https://packages.debian.org/de/source/wheezy/apache2`.

[2] *Apache httpd Homepage*. 12. Apr. 2015. URL: `http://httpd.apache.org`.

[3] *Apache httpd Modules*. 12. Apr. 2015. URL: `http://httpd.apache.org/modules/`.

[4] *Apache httpd Proxy Modul*. 12. Apr. 2015. URL: `http://httpd.apache.org/docs/2.2/mod/mod_proxy.html`.

[5] *Apache Tomcat für Debian Wheezy*. https://packages.debian.org/wheezy/tomcat7. 13. Apr. 2015.

[6] *Apache Tomcat Homepage*. 12. Apr. 2015. URL: `http://tomcat.apache.org`.

[7] *Aptitude bei Ubuntu*. 13. Apr. 2015. URL: `http://wiki.ubuntuusers.de/aptitude`.

[8] *Archnemesis auf GitHub*. 16. Apr. 2015. URL: `https://github.com/saros-project/archnemesis`.

[9] J. Arundel. *Puppet 3 Beginner's Guide*. Learn by doing : less theory, more results. Packt Publishing, 2013. ISBN: 9781782161257.

[10] *Chaos Monkey*. 16. Apr. 2015. URL: `https://github.com/Netflix/SimianArmy/wiki/Chaos-Monkey`.

[11] *Chef, Puppet or Ansible: easy server configuration management for infrastructure at any scale*. 15. Apr. 2015. URL: `http://kangaroot.net/easy-server-configuration-management`.

[12] D. Coward und Y. Yoshida. *Java Servlet Specification Version 2.4*. Techn. Ber.

[13] R. Tyler Croy. *Puppet-Modul: rtyler/jenkins*. 26. Jan. 2015. URL: `https://forge.puppetlabs.com/rtyler/jenkins`.

[14] *CVE-2014-0196 im Debian-Bug-Tracker*. 15. Apr. 2015. URL: `https://security-tracker.debian.org/tracker/CVE-2014-0196`.

[15] Patrick Debois. *Devopsdays Ghent 2009 - The conference that brings development and operations together*. 2009.

[16] *Ejabberd für Debian Wheezy*. 13. Apr. 2015. URL: `https://packages.debian.org/wheezy/ejabberd`.

[17] *Ejabberd Homepage*. 12. Apr. 2015. URL: `https://www.ejabberd.im`.

[18] *Forumsumfrage zur Beliebheit von Java*. 15. Apr. 2015. URL: http://www.quora.com/I-dont-care-for-Java-If-you-like-Java-why-do-you-like-it.

[19] *Gerrit - Standalone Daemon Installation Guide*. 15. Apr. 2015. URL: https://gerrit-review.googlesource.com/Documentation/install.html.

[20] *Gerrit for Debian*. 15. Apr. 2015. URL: https://github.com/dnaeon/gerrit-debian.

[21] *Gerrit Homepage*. 15. Apr. 2015. URL: https://code.google.com/p/gerrit.

[22] *Gerrit Replication Plugin*. 16. Apr. 2015. URL: https://gerrit.libreoffice.org/plugins/replication/Documentation/index.html.

[23] *GitHub: Slightly-less-than-POC privilege escalation exploit for kernels ¿= v3.14-rc1*. 15. Apr. 2015. URL: https://github.com/jocover/CVE-2014-0196/blob/master/cve-2014-0196-md.c.

[24] *Grundlagen für GRUB 2*. 13. Apr. 2015. URL: http://wiki.ubuntuusers.de/GRUB_2/Grundlagen.

[25] *How do I list all cron jobs for all users?* 13. Apr. 2015. URL: http://stackoverflow.com/questions/134906/how-do-i-list-all-cron-jobs-for-all-users.

[26] *How Do I Use Salt States?* 13. Apr. 2015. URL: http://docs.saltstack.com/en/latest/topics/tutorials/starting_states.html.

[27] *How to package for Debian*. 15. Apr. 2015. URL: https://wiki.debian.org/HowToPackageForDebian.

[28] Michael Hüttermann. *DevOps for Developers*. Apressus Series. ISBN: 9781430245698.

[29] *Introduction to Debian Packaging*. 15. Apr. 2015. URL: https://wiki.debian.org/IntroDebianPackaging.

[30] Thomas Jackson. *Salt-Fehlermeldung: unless + cmd.wait don't work*.

[31] *Jenkins auf GitHub*. 11. Apr. 2015. URL: https://github.com/jenkinsci/jenkins.

[32] *Jenkins für Debian*. 11. Apr. 2015. URL: http://pkg.jenkins-ci.org/debian.

[33] *Jenkins Role Strategy Plugin*. 16. Apr. 2015. URL: https://wiki.jenkins-ci.org/display/JENKINS/Role+Strategy+Plugin.

[34] *Jenkins standalone*. 11. Apr. 2015. URL: https://wiki.jenkins-ci.org/display/JENKINS/Starting+and+Accessing+Jenkins.

[35] *Jinja Template Designer Documentation*. 16. Apr. 2015. URL: http://jinja.pocoo.org/docs/dev/templates/.

[36] Ryan D. Lane. *Moving away from Puppet: SaltStack or Ansible?* 16. Apr. 2015. URL: http://ryandlane.com/blog/2014/08/04/moving-away-from-puppet-saltstack-or-ansible.

[37] *Learning Puppet - Templates*. 15. Apr. 2015. URL: https://docs.puppetlabs.com/learning/templates.html.

[38] Thorsten Leemhuis. "Linux: Container-Virtualisierung mit Docker". In: *c't Magazin - Heise Zeitschriften Verlag* (2014).

[39] *Long term support bei Ubuntu*. 13. Apr. 2015. URL: https://wiki.ubuntu.com/LTS.

[40] DataCentred Ltd. *Puppet-Modul: datacentred/gerrit*. 26. Jan. 2015. URL: https://forge.puppetlabs.com/datacentred/gerrit.

[41] *MySQL für Debian Wheezy*. 13. Apr. 2015. URL: https://packages.debian.org/de/source/wheezy/mysql-5.5.

[42] *MySQL Homepage*. 12. Apr. 2015. URL: http://www.mysql.de.

[43] *Oracle und Sun Microsystems*. 12. Apr. 2015. URL: http://www.oracle.com/us/sun/index.html.

[44] *phpESP Homepage*. 16. Apr. 2015. URL: http://sourceforge.net/projects/phpesp/.

[45] *Puppet-Benutzerkonto von Thomas Hatch*. 9. Apr. 2015. URL: https://projects.puppetlabs.com/users/1301.

[46] Corey Quinn. *A Taste of Salt: Like Puppet, But Less Frustrating*. 16. Apr. 2015. URL: http://blog.smartbear.com/devops/a-taste-of-salt-like-puppet-except-it-doesnt-suck.

[47] *Reddit-Diskussion: Does anyone actually *like* web development?* 15. Apr. 2015. URL: http://www.reddit.com/comments/arrva/does_anyone_actually_like_web_development.

[48] *Review Board Homepage*. 16. Apr. 2015. URL: https://www.reviewboard.org.

[49] *Salt - Configuring the Salt Master*. 16. Apr. 2015. URL: http://docs.saltstack.com/en/latest/ref/configuration/master.html.

[50] *Salt Formulas*. 15. Apr. 2015. URL: http://docs.saltstack.com/en/latest/topics/development/conventions/formulas.html.

[51] *Salt Homepage*. 15. Apr. 2015. URL: http://www.saltstack.com.

[52] *Salt Pillar Walkthrough*. 15. Apr. 2015. URL: http://docs.saltstack.com/en/latest/topics/tutorials/pillar.html.

[53] *Saros beim Google Summer of Code*. 15. Apr. 2015. URL: http://www.google-melange.com/gsoc/org2/google/gsoc2015/saros.

[54] *Saros Homepage*. 16. Apr. 2015. URL: http://www.saros-project.org.

[55] *Skriptsprachen verwenden dynamisches Typsystem*. 15. Apr. 2015. URL: http://www.itwissen.info/definition/lexikon/Scriptsprache-script-language.html.

[56] *SonarQube Homepage*. 16. Apr. 2015. URL: http://www.sonarqube.org.

[57] *TestLink-Java-Annotation vor einer Test-Klasse in Saros*. 16. Apr. 2015. URL: https://www.reviewboard.org.

[58] *Thomas Hatch genannt als CTO von SaltStack*. 9. Apr. 2015. URL: http://saltstack.com/leadership.

[59] *Ubuntu Lucid Lynx*. 13. Apr. 2015. URL: http://wiki.ubuntuusers.de/Lucid_Lynx.

[60] *Ubuntu und Debian*. 13. Apr. 2015. URL: http://www.ubuntu.com/about/about-ubuntu/ubuntu-and-debian.

[61] *Using Ruby in the Puppet Ruby DSL*. 15. Apr. 2015. URL: https://puppetlabs.com/blog/using-ruby-in-the-puppet-ruby-dsl.

[62] *YAML Homepage*. 15. März 2015. URL: http://www.yaml.org/.

[63] *Über Jenkins*. 11. Apr. 2015. URL: http://jenkins-ci.org/content/about-jenkins-ci.

[64] *Über MariaDB*. 12. Apr. 2015. URL: https://mariadb.com/about.

Abbildungsverzeichnis