Hochschule für Angewandte Wissenschaften Hamburg
*Hamburg University of Applied Sciences*

# Bachelorarbeit

**Patrick Steinhauer**

## Realisierung einer Infrastructure-as-Code-Anwendung Zum automatisierten Aufsetzen eines CI-Service

*Fakultät Technik und Informatik*
*Department Informatik*

*Faculty of Engineering and Computer Science*
*Department of Computer Science*

**Patrick Steinhauer**

# Realisierung einer Infrastructure-as-Code-Anwendung zum automatisierten Aufsetzen eines CI-Service

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stefan Sarstedt
Zweitgutachter: Diplom Informatiker Guntmar Kirck

Abgegeben am 20 Februar 2017

**Patrick Steinhauer**

**Thema der Arbeit**

Realisierung einer Infrastructure-as-Code-Anwendung zum automatisierten Aufsetzen eines CI-Service

**Stichworte**

Infrastructure-as-Code, Docker, Ansible, SaltStack, Continuous Integration, Continuous Delivery, Jenkins

**Kurzzusammenfassung**

Die stark ansteigende Anzahl von Cloudsystemen, Servern oder anderer IT-Systeme gelangt an einen Punkt, bei dem eine manuelle Verwaltung dieser Systeme immer schwieriger wird. Infrastructure-as-Code setzt genau hier an und dient als Unterstützung bei der Einrichtung oder Konfiguration solcher Systeme. Diese Arbeit zielt darauf ab verschiedene Vertreter der Technologie zu analysieren. Hierbei werden Vor- und Nachteile aufgezeigt, ein Vergleich durchgeführt und spezielle Merkmale der Produkte betrachtet, um eine Auswahl für die Verwendung vereinfachen zu können. Abschließend wird für jede der analysierten Technologien eine CI-Pipeline exemplarisch umgesetzt.

**Patrick Steinhauer**

**Title of the paper**

Realization of an infrastructure as code application for the automated creation of a ci-service

**Keywords**

Infrastructure-as-Code, Docker, Ansible, SaltStack, Continuous Integration, Continuous Delivery, Jenkins

**Abstract**

The number of cloud systems, servers and other IT systems has reached the point, where the manually administration will become more difficult. Infrastructure-as-Code starts here as a support for configuring such systems. This Thesis has the goal to analyse various representatives of this technologie. Therefore, the advantages and disadvantages will be shown. A comparision of them will be done and special characteristics will be outlined. The last part shows an example implementation for every analyzed technology, which should help to choose one of the tools.

# Inhaltsverzeichnis

# 1 Einleitung

## 1.1 Problemstellung und Motivation

In der heutigen Zeit ist in Softwareunternehmen die Grundlage für erfolgreiches Entwickeln von Software ein Continuous Integration Service (im Folgenden auch CI-Service genannt). Dieser Service unterstützt die Entwicklungs-Teams des Unternehmens in ihren Entwicklungsprozessen.
Das Unternehmen Werum IT Solutions GmbH entwickelt ein Softwareprodukt auf Basis einer Microservice Architektur. Dieses Produkt wird durch Anpassungen innerhalb verschiedener Projektteams bei Werum für konkrete Kundeninstallationen individualisiert. Die Struktur für einen CI-Service ist für jedes Team aufgrund der weiter zu entwickelnden Software vorgegeben. Weiterhin ist es wichtig, dass Projekte für Wartungsarbeiten in der Lage sein müssen historische Versionen des CI-Service auch nach zehn Jahren erneut in Betrieb nehmen zu können. Damit ist die besondere Herausforderung gegeben bis zum Start des CI-Service, die komplette Definition des Aufbaus des CI-Service, mithilfe von Infrastructure-as-Code (wird im weiteren Verlauf auch IaC genannt) zu entwickeln und innerhalb eines SCM[1] zu verwalten. Zurzeit besteht die vorhandene CI-Service Lösung aus einer SaltStack Bereitstellung, welche eine CI-Service Struktur auf einem Microsoft Windows 2012 Server aufsetzt. Als CI-Server Lösung wird Jenkins[2]

[1] Die Abkürzung SCM steht für „Software Configuration Management“ und erlaubt innerhalb der Softwareentwicklung die Versionierung von Programmcode. Ein Beispiel für ein Tool wäre GIT.
[2] Jenkins ist eine Software, die den Prozess von Continuous Integration unterstützt.

eingesetzt. Dieser Ansatz für eine CI-Lösung ist jedoch lückenhaft in seiner Definition, da gewisse Teilschritte innerhalb der Bereitstellung manuell erfolgen, wie z.B. die Jenkins Job Konfiguration. Außerdem ist die Lösung unflexibel, da nicht triviale Aktualisierungen aufgrund von manuellen Schritten erfolgen müssen.
Hierzu ist nun ein neuer prototypischer Architekturentwurf zu realisieren. Dieser neue Entwurf basiert auf den Technologien Linux, Ansible, SaltStack, Docker und Jenkins. Linux ist hierbei das neu zu verwendende Betriebssystem, welches anstatt des Windows Betriebssystems genutzt werden soll. Ansible, SaltStack und Docker werden hier für die Provisionierung genutzt. Die entstehenden Lösungen können dann für eine Portierung von Jenkins auf GoCD genutzt werden.

## 1.2 Zielsetzung

Ziel dieser Arbeit ist es, im Zuge der voranschreitenden Technologie, eine bestehende Continuous Integration Lösung mittels ausgewählter und neuerer Technologien zu optimieren bzw. zu verbessern. Um dieses Ziel zu erreichen, werden ausgewählte Technologien verwendet und untersucht. Die durch den Betrieb gewählten Technologien werden hierfür auch mit anderen Technologien verglichen, um aufzeigen zu können, was für Unterschiede die Auswahl eines entsprechenden Tools mit sich bringen kann.
Zuerst wird dazu eine grundlegende Einführung in das Themengebiet IaC gegeben, um Grundzüge sowie Ideen des Themas verständlich zu machen. Dabei werden zusätzlich die gewählten Tools für diese Arbeit einführend beschrieben, damit ein Grundverständnis aufgebaut wird. Danach werden Herausforderungen des Themas genau analysiert, damit verschiedene Schwierigkeiten der Technologien deutlich werden. Unterstützend dient dies dazu, sich ein genaues Bild davon machen zu können, welche Einsatzgebiete hierbei möglich sind. Abschließend folgt dann der Entwurf der unterschiedlichen Lösungen durch die Vertreter Ansible, SaltStack und Docker. Dort wird ein klares Konzept entwickelt, welches eine möglichst gute neue CI-Service Lösung liefert. Abschließend werden diese Lösungen miteinander verglichen um gezielt die Vor- und Nachteile, die bei der Entwicklung ermittelt wurden, aufzuzeigen.

# 2 Einführung Infrastructure-as-Code

In diesem Kapitel soll ein grundlegendes Verständnis für das Thema Infrastructure-as-Code aufgebaut werden. Hierzu gibt es einen kleinen Einblick in die Historie sowie in das Thema selbst. Ferner werden Technologien aufgezählt und beschrieben, die im Bereich von IaC aktuell in Verwendung sind. Schlussendlich werden diese Technologien miteinander verglichen und bewertet.

## 2.1 Grundlagen

### 2.1.1 Historie

Bei einer historischen Betrachtung dieses Themas lässt sich feststellen, dass die Menge an Technologien und Tools u.a. weiterhin stark zunehmen wird. Hierbei ergibt sich für IT-Administratoren zunehmend der Aufwand, sich mit dem kompletten Paket an Technologien auseinanderzusetzen. Eine derartige Auseinandersetzung ist deswegen grundsätzlich sinnvoll, da Arbeitsabläufe von Administratoren somit erleichtert werden. Ein solcher administrativer Aufwand lässt sich nur mit einer gewissen Anzahl an Ressourcen bewerkstelligen, weswegen die Entwicklung von Automatisierungstechnologien immer gefragter ist. Ebenfalls ein maßgeblicher Antreiber der Automatisierung von Infrastrukturen ist die Cloud. Dies lässt sich damit begründen, dass es vor dem Zeitalter der Cloud wesentlich weniger virtuelle Maschinen und Server gab. Für Administratoren war es so deutlich leichter die Anzahl an Hardware und deren Software zu administrieren und zu verwalten. Aufgrund der stark ansteigenden Zahlen von Cloud, Servern und virtuellen Maschinen wird diese Aufgabe zu einem immer schwierigeren

Unterfangen. Darüber hinaus ist die Elastizität bezogen auf das Vorher sowie das Nachher zu beachten. Betrachtet man Infrastrukturen, wie sie vor dem Zeitalter der Cloud vorhanden waren, so waren diese sehr starr in dessen Aufbau. Diese Starrheit ist in dem späteren Modell der Cloud weniger gegeben, da dort eine dynamischere Anpassung möglich ist. Um genau diese Schwierigkeiten erleichtern zu können, ist die Bewegung um das Thema Infrastructure-as-Code entstanden. IaC setzt genau dort an, indem administrative Prozesse automatisiert werden, sodass eine manuelle Verwaltung weitestgehend ersetzt wird.

### 2.1.2 Infrastructure-as-Code

Bei Infrastructure-as-Code handelt es sich um eine Automatisierungs-Technik zum Einrichten und Verwalten von IT-Infrastrukturen. Diese Technik verwendet zum Aufsetzen der Infrastrukturen vorhandene Praktiken aus dem Bereich der Softwareentwicklung. [vgl. Morris 2016, Kapitel 1, S. 5].

*„The enabling idea of Infrastructure as Code is that the systems and devices used to run software can be treated as if they, themselves, are software."*
[Morris 2016, Kapitel 10, S. 179]

Diese Aussage beschreibt, dass Systeme, auf denen Software läuft bzw. laufen soll, schon als eigenständige Software behandelt werden. Begründet ist dies damit, dass diese Art von System auch mithilfe von Software eingerichtet wird. Des Weiteren können diese Systeme einfach und ohne viel Aufwand entfernt und wieder eingerichtet werden. Ermöglicht wird dies dadurch, dass die Systeme nicht aus physikalischen Maschinen bestehen, denn sie werden dort nur konfiguriert. Außerdem entsteht die Möglichkeit, dass spezielle Tools aus dem Software-Development Umfeld auch bei diesen Systemen und Devices verwendet werden können. Dies eröffnet wiederum verschiedene neue Möglichkeiten. [vgl. Morris 2016, Kapitel 10].

Ein weiterer Punkt für den Nutzen von IaC ist die Veränderung von Infrastrukturen innerhalb eines Entwicklungszyklus. Hierbei wird, um technologisch auf dem

neusten Stand zu bleiben, versucht, den aktuellen Bewegungen und Trends zu folgen, sofern sich diese als sinnvoll und wertvoll erweisen. Genau diese Veränderungen können sowohl manuell als auch mittels IaC realisiert werden. Vorteilhaft an der Automatisierung ist, dass bestehende Komponenten durch die codierten Skripte leicht einfließen können, indem benötigte Ressourcen und Codeblöcke ebenfalls mit versioniert werden.

### 2.1.2.1. Provisionierung

Als nächstes folgt eine Erläuterung darüber, wie Infrastructure-as-Code überhaupt funktioniert. Diese Art der Automatisierung wird auf verschiedene Weisen realisiert.

**Push-Provisionierung**

Eine Variante wäre die sogenannte Push-Provisionierung. Wie diese Provisionierung funktioniert, wird in der unten aufgelisteten Abbildung dargestellt.

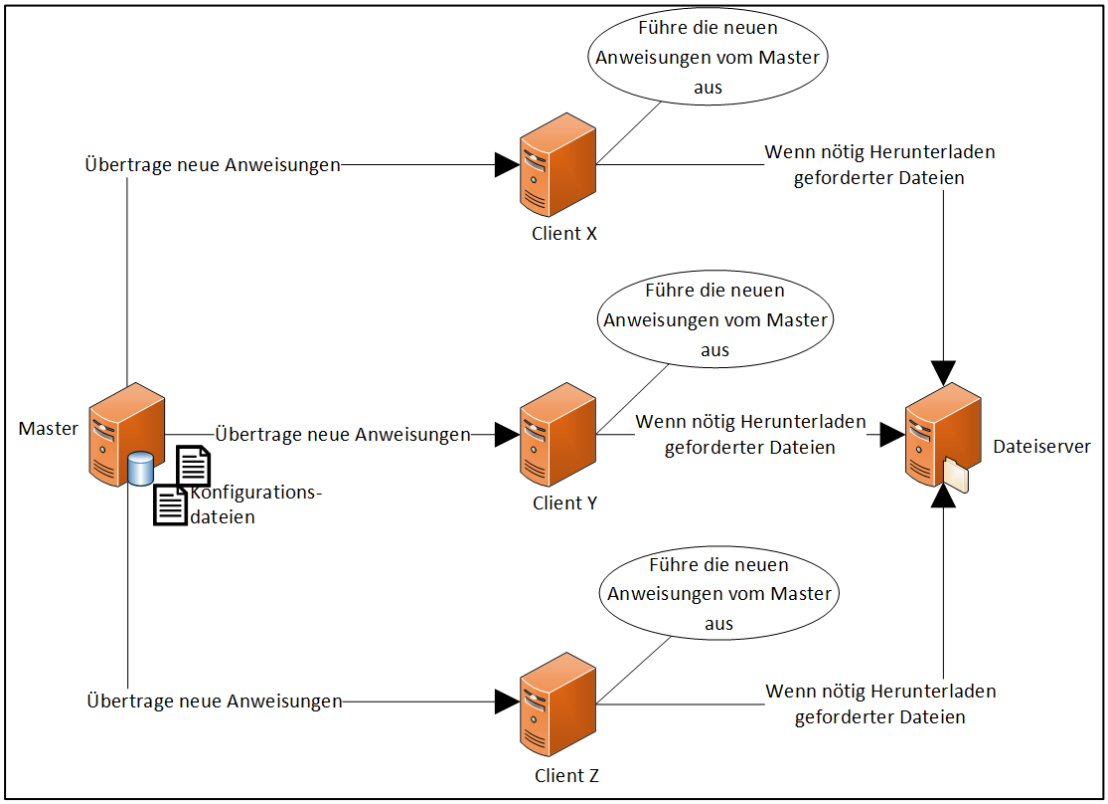


Abbildung 1 Push-Provisionierung

Die Push-Provisionierung funktioniert, wie in *Abbildung 1 Push-Provisionierung* gezeigt, mit einem Server, welcher zum Master bestimmt wird. Dieser Master koordiniert mittels seiner Funktion alle zu konfigurierenden Slaves (verschiedene Namen sind hier bekannt) und gibt ihnen die Anweisungen, etwas Bestimmtes zu tun, wie z.B. das Installieren neuer Software oder die Änderung einer Systemvariable. Im Fall einer Installation gehen die Slaves über ein vorhandenes Protokoll auf einen Dateiserver, um die gewünschte Installationsdatei dann dort herunterladen zu können.

**Pull-Provisionierung**

Als zweite Variante der Provisionierung gibt es das Gegenstück, auch genannt Pull-Provisionierung. Dieses Prinzip wird in *Abbildung 2 Pull-Provisionierung* genauer dargestellt und erläutert.

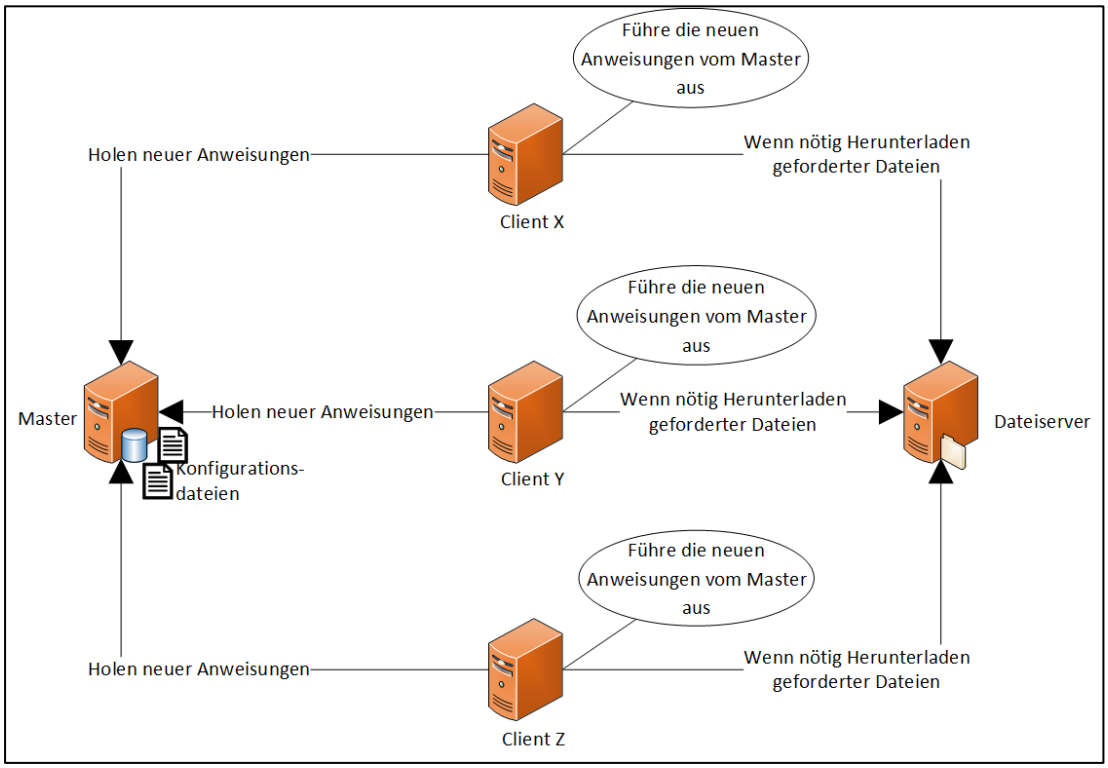


Abbildung 2 Pull-Provisionierung

Wie in der *Abbildung 2 Pull-Provisionierung* zu sehen ist, arbeitet die Pull-Provisionierung im Gegenteil zur Push-Provisionierung auf eine andere Art und Weise. Jeder Slave fragt beim Master nach, ob irgendwelche Änderungen vorgenommen werden sollen. Wenn dies der Fall ist, bekommt der anfragende Client vom Master die nötigen Informationen und kann dann wie bei der Push-Provisionierung ebenfalls notwendige Installationen durchführen bzw. etwas verändern.

Beide Möglichkeiten zur Provisionierung werden mittlerweile durch die unterstützenden Tools verwendet. Je nach Anwendungsfall kann man sich das Tool seiner Wahl verwenden. Beispiele für diese Provisionierungen sind in der folgenden Tabelle zu sehen.

| Tool | Push-Provisionierung | Pull-Provisionierung |
|---|---|---|
| **Ansible** | √ | √ |
| **Puppet** | X | √ |
| **SaltStack** | √ | √ |
| **Chef** | X | √ |

Tabelle 1 Provisionierungsunterstützungen verschiedener Tools

Obwohl manuelle Konfigurationen seit langer Zeit vollzogen werden, gibt es hierbei Probleme, die mittels Infrastructure-as-Code behoben werden können. Dies bedeutet jedoch nicht, dass alles an einer manuellen Konfiguration schlecht ist und IaC die Lösung für jedes Problem ist. Ein Beispiel innerhalb der Softwareentwicklung ist vielen Softwareunternehmen bekannt, das Problem der Inbetriebnahme von historischen Systemen. Wenn beispielsweise nach zehn Jahren Betrieb einer Software wieder etwas verändert werden muss, weil Fehler auftreten, muss das komplette System in der Firma wieder ausgerollt und konfiguriert werden. Dies bedeutet, dass man Continuous Integration Services, Entwicklungsumgebungen zusammen mit der IDE u.a. wieder zum Leben erwecken muss, was wiederum sehr zeitintensiv sein kann. Mithilfe von Infrastructure-as-Code wird dieser Prozess unterstützt, da ein erneutes Aufsetzen nun durch ein codiertes Skript passiert, das über ein Versionskontrollsystem bereitgestellt wird. Dieses Skript sorgt dafür, dass

die entsprechende Umgebung vollständig aufgebaut wird. Voraussetzung dafür ist jedoch, dass alle Installations- und Konfigurationsdateien zur Verfügung stehen. Diese Dateien müssen für dessen erneute Benutzung auf einem langzeitarchivierten Server bzw. Repository bereitliegen. Dieser Ausführungsprozess kann im Zweifelsfall eine große Anzahl an Stunden der Konfiguration ersparen.

Ebenso wichtig ist die heutige Betrachtung der Cloud. Die Cloud-Provider sind stark darauf fixiert, schnell unterschiedliche Cloud basierte Lösungen anzubieten. Als Beispiel kann man Amazon nennen, welche einen großen Pool an Cloud-Lösungen anzubieten hat. Wenn ein Kunde eine solche Lösung nun bestellen bzw. einkaufen möchte, so sollte ein schnelles Einrichten dieses Services möglich sein. Manuell kann dies jedoch eine gewisse Zeit dauern, da möglicherweise für eine große Anzahl von Mitarbeitern dieser Service bereitgestellt werden muss. Mithilfe von IaC werden diese vielen Cloud-Lösungen in kürzester Zeit per Knopfdruck eingerichtet. Während dieser Einrichtung dauert der Prozess je nach Anforderung kürzer oder länger, da die Installation von viel Software einen zusätzlichen Zeitaufwand bedeutet.

## 2.2 Aktuelle Technologien

Dieser Bereich beschreibt aktuelle Technologien, mit denen sich Infrastructure-as-Code realisieren lässt. Die aufgelisteten Tools, wie auch schon in Bereich *2.1.2 Infrastructure-as-Code* in der Tabelle mit den Provisionierungs-Technologien beschrieben, sind lediglich die Repräsentanten, die für diese Arbeit gewählt wurden. Weitere zahlreiche Tools bzw. Technologien aus diesem Umfeld sind natürlich vorhanden.

### 2.2.1 Ansible

Ansible ist ein Tool für den Bereich IaC, da es zum Bewältigen der nachfolgenden Aufgaben vorhanden ist.

- Deployment
- Konfigurationsmanagement
- Orchestrierung

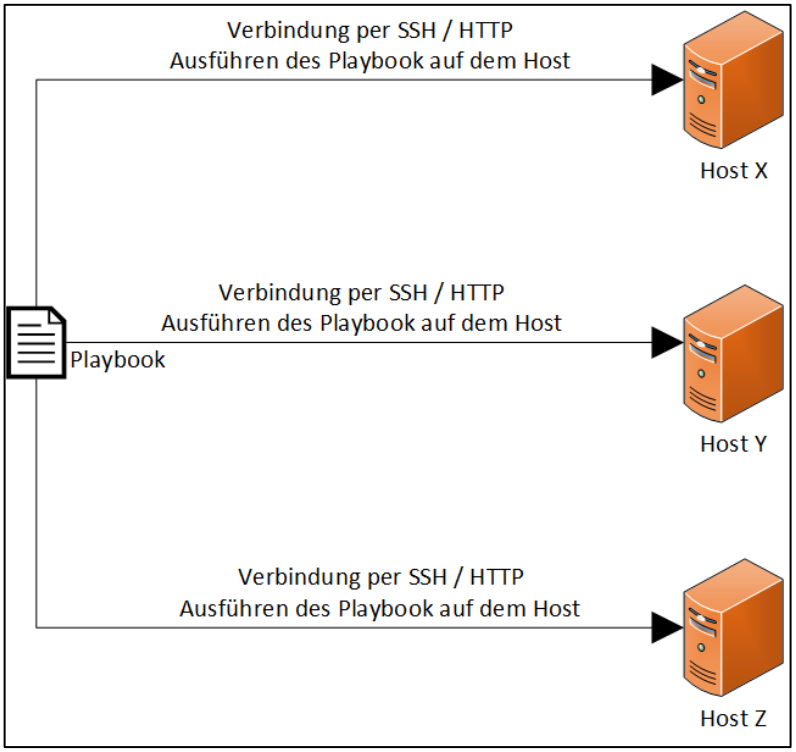


Abbildung 3 Ansible Push-Provisionierung

Wie bereits im *Kapitel 2.1.2 Infrastructure-as-Code* dargestellt, wird bei Ansible die Push-Provisionierung verwendet. Diese Provisionierung wird in der nachfolgenden Grafik speziell für Ansible als Veranschaulichung gezeigt. Ansible arbeitet zum Konfigurieren mit den von Ansible definierten Playbooks. Mittels einer Remoteverbindung über SSH, HTTP oder WinRM (Windows Remote Management), wird auf den spezifizierten Host Systemen der Zielzustand des Playbooks umgesetzt. Hinzuzufügen ist jedoch, dass man die Push-Provisionierung bei Ansible auch um konfiguriert werden kann, sodass eine Pull-Provisionierung entsteht.

#### 2.2.1.1. Playbooks

*„Reading an Ansible playbook is easy. This is really important because it becomes self-documenting. Knowing Ansible, I can go into an organization, read their playbooks, and understand more or less how things work.“*
[Matt Coddington: https://www.ansible.com/configuration-management]

Mit dem obigen Zitat von Matt Coddington wird speziell für Ansible und dessen Playbooks darauf hingewiesen, dass eine Verwendung leicht ist, da der Code eines solchen Skripts selbst erklärend ist. Dadurch kann ein Verständnis über die Infrastruktur schnell nachgelesen werden und muss nicht aufwendig erarbeitet werden. Die Ansible Playbooks werden genutzt, um sämtliche Aufgaben des Ansible Umfeldes zu lösen. Hierzu gehören die in *Kapitel 2.2.1 Ansible* aufgeführten Aufgabenbereiche. Folgend soll nun vermittelt werden, wie ein solches Playbook aussieht, welche Besonderheiten es gibt und was bei einer Verwendung zu beachten ist.

Um ein Playbook schreiben zu können, werden Kenntnisse über die *markup language YAML*[3] benötigt. In der YAML Notation werden dann die entsprechenden Konfigurationsbefehle geschrieben, mit denen dann das gewünschte Ergebnis modelliert wird. Die Syntax der Befehle sowie die Befehle selbst in einem Playbook

[3] YAML ist eine Ausdruckssprache, welche für Menschen leicht lesbar ist und wie XML für Konfigurationsdateien genutzt wird.

werden in der Ansible Dokumentation gut und verständlich erklärt, was aber sicherlich von Person zu Person anders empfunden wird.

Um nun auf den Aufbau eines Playbooks zu kommen, werden die Kernelemente dazu kurz erläutert und beschrieben. Als erstes werden in einem Playbook die zu konfigurierenden Hosts (Zielsysteme) aufgelistet. Darunter befinden sich die systembedingten Anmeldeinformationen. Hat man die Systeme vollständig aufgelistet und beschrieben, beginnt der Teil der Konfiguration der Maschinen. Hierzu werden sogenannte tasks definiert. Mithilfe der tasks beschreibt man durch bestimmte Kommandos wie der Zustand jedes einzelnen Hosts aussehen soll. Teilt man viele unterschiedliche Tasks in mehrere verschiedene Playbooks auf, so nennt man diese einzelnen Playbooks role. Mit diesen roles ist eine Abstraktion vorhanden, mit der Playbooks gut strukturiert werden können. Beim Definieren der unterschiedlichen Hosts nennt man jeden einzelnen Host, den man beschreiben möchte, play. Weiterhin lassen sich innerhalb eines Playbooks mehrere plays definieren. Diese unterschiedlichen plays werden dann hierarchisch nacheinander ausgeführt. Ein Vorteil hierbei ist, dass mehrere plays völlig unterschiedlich definiert werden können, sodass ein Playbook nicht nur komplett gleiche Systeme konfiguriert. Wenn der Code für die Konfiguration fertig ist lässt sich das Playbook leicht über die Kommandozeile ausführen, woraufhin dann Ansible die Systeme einrichtet.[4]

#### 2.2.1.2. Ansible Module & Ansible Galaxy

Nachdem nun die Ansible Playbooks beschrieben wurden, folgen zusätzlich ein paar weitere Informationen über Ansible Modules und dem Ansible Galaxy. Diese beiden Punkte haben in gewisser Weise einen Bezug zu den Ansible Playbooks, was im folgenden Teil dargestellt wird.

Bei den Ansible Modulen handelt es sich um einen zentralen Bestandteil von Ansible. Module sind für die hauptsächliche Arbeit bei Ansible verantwortlich, da sie die gesamten Funktionalitäten der Playbook tasks verrichten. Das bedeutet, dass

[4] http://docs.ansible.com/ansible/playbooks_intro.html#

bei jedem task, der ausgeführt wird, ein Modul angesprochen wird. Dieses Modul ist dann für die eigentliche Arbeit zuständig. Des Weiteren hat Ansible eine Unterstützung, welche mit Ansible Galaxy vorhanden ist. Ansible Galaxy ist eine Plattform, die man mit einem Repository vergleichen kann. In diesem Repository sind offizielle Playbooks von Ansible enthalten sowie verschiedene Eigenkreationen der Community. Somit ist eine komplette Neuentwicklung im Optimalfall nicht erforderlich.[5]

#### 2.2.1.3. Vorteile und Nachteile

Zum Schluss werden einige Vor- und Nachteile gewählt, die bisher erkennbar sind. Folgende Defizite sind erkennbar:

- Windows Unterstützung ist nur bedingt gewährleistet
- Single Point of Failure
- Sehr komplex
- Administrative Aufgaben erfordern Programmierkenntnisse

Einige dieser Nachteile können aufgrund der Architektur von Ansible nicht verhindert werden. Andererseits lassen sich möglicherweise spezielle Dinge auch umsetzen, was jedoch absichtlich nicht gewollt ist (Single Point of Failure). Damit diese Nachteile das Gesamtbild von Ansible nicht zunichtemachen, kommen nun die Vorteile, die eine Verwendung der Technologie befürworten.

- Erleichterung administrativer Aufgaben
- Unterschiedliche Anwendungsgebiete sind möglich
- Verwendung einer einfachen Beschreibungssprache
- Viele Funktionalitäten vorhanden
- Flache Lernkurve

Diese Vorteile bieten in dem Bereich von IaC eine gute Grundlage zum Automatisieren einiger Infrastruktur Themen. Je nach Anwendungsfall ist jedoch abzuwägen, ob die Vor- und Nachteile in die eigene Struktur hinein passen. Bei genauerer Betrachtung des Ganzen lassen sich sicher weitere Vor- und Nachteile

[5] http://docs.ansible.com/ansible/modules_intro.html

aufzeigen. Diese Untersuchung sollte aber nicht im Fokus der Arbeit stehen. Schlussendlich ist festzuhalten, dass die oberen genannten Punkte auch als persönliche Vor- und Nachteile betrachtet werden können, da die Sichtweise von Person zu Person abweichen kann.

### 2.2.2 SaltStack

SaltStack ist wie Ansible eine Software, die eine Automatisierung von Infrastruktur ermöglicht. Die Aufgabenbereiche von SaltStack liegen im Bereich der Infrastruktur Automatisierung, event-driven data center orchestration[6] sowie des configuration management. Mit SaltStack ist im Gegensatz zum gedachten Ansible Prinzip der Push-Provisionierung auch eine Lösung der Pull-Provisionierung vorhanden (siehe *Kapitel 2.1.2 Infrastructure-as-Code*). Um das Prinzip der Pull-Provisionierung im Fall von SaltStack zu erläutern, folgt hierzu eine Grafik zur Veranschaulichung.

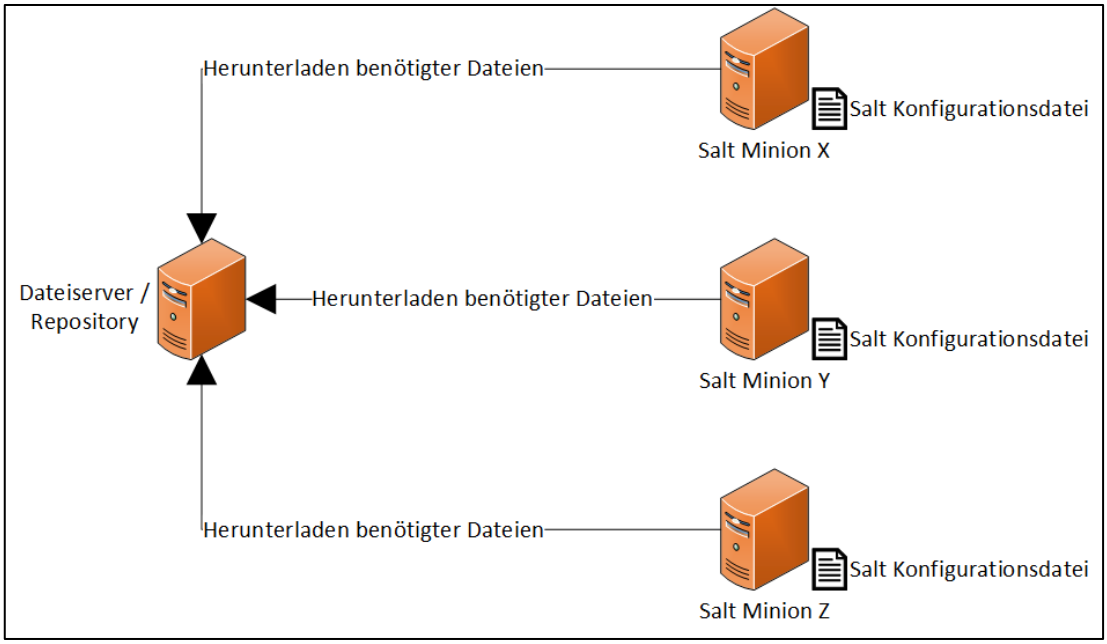


Abbildung 4 SaltStack Pull-Provisionierung

[6] Event-driven data center orchestration beschreibt eine Event gesteuerte und automatisierte Einrichtung, Koordination oder das Management von Computer Systemen und deren Infrastruktur.

Bei der Software von SaltStack spricht man nun von einem Modell, welches Master und Minions besitzt. Der Master ist hierbei als Koordinator zu sehen und ein Minion wird mit einem Agenten gleichgesetzt. Dieses Modell agiert im Salt-Umfeld als Push-Provisionierung. Die Pull-Provisionierung hingegen arbeitet masterless. Dabei besitzen alle notwendigen Minions die Salt Konfigurationsdatei, welche auf unterschiedlichstem Wege auf diese Umgebungen abgelegt wird. Durch diese Datei ist der Minion in der Lage alles Benötigte zu installieren und zu konfigurieren. Hierbei können die Salt Befehle ohne den Master autark auf dem Minion ausgeführt werden. Eine solche Verwendung des Salt Minion wird auch Standalone Minion genannt. Vorteilhaft an einer solchen Verwendung von SaltStack, ist das keine Abhängigkeit zu einem Salt-Master mehr besteht.

#### 2.2.2.1. Salt Begrifflichkeiten

Damit die Basis für SaltStack verstanden werden kann, folgen in diesem Kapitel einige der Salt spezifischen Begrifflichkeiten. Diese gehören im Wesentlichen zu den Grundlagen von Salt. Hierzu gehören die Salt state files, Salt pillars und die Salt formulas.

**Salt State file**

Zu einer SaltStack Konfiguration werden die Salt State files verwendet. Salt State files sind Konfigurationsdateien, die auf die Endung *.sls*[7] enden. Zu dieser Endung ist zu sagen, dass es hierzu eine wichtige top.sls gibt und jegliche andere .sls Datei nur eine state Datei ist. Innerhalb einer solchen Konfigurationsdatei sind die Salt States enthalten, welche dafür sorgen, dass das Zielsystem dem gewünschten Zustand der Konfigurationsdatei entspricht. Ein solches state besteht aus einer state declaration und den state functions. Die declaration eines States besteht aus einem eindeutigen Namen, der unique ID. Bei der State function hingegen handelt es sich um die direkten Kommandos, mit deren Hilfe die Konfigurationen durchgeführt werden. Um Konfigurationen nun durchzuführen ist es möglich, mehrere state Dateien anzulegen.

[7] Die Endung SLS steht für **S**a**l**t **S**tate [https://docs.saltstack.com/en/latest/ref/states/, abgerufen am 5.10.2016]

**Salt Pillar Datei**

Salt Pillar Dateien sind, wie auch die State Dateien aus Salt, mit der .sls Endung versehen. Innerhalb einer solchen Datei werden für SaltStack systemspezifische Variablen und Daten definiert. Dadurch ist es möglich, eine große Anzahl an Rechnern mit den gleichen Konfigurationen zu versorgen. Diese Dateien werden auf einem Salt Master definiert und werden anschließend, wenn notwendig, auf diverse Minions verteilt. Diese Minions können dafür zusätzlich separat selektiert werden, sodass nicht jeder Minion diese Konfigurationen bekommen muss.

**Salt Formula**

Damit das bisherige Bild der Salt Dateien abgerundet wird, gibt es die noch zu erwähnenden SaltStack Formulas. Wenn man von einem Formula spricht, handelt es sich um eine größere Sammlung an Salt State und Salt Pillar Dateien. Mithilfe eines solchen Formula ist es möglich spezielle Systeme gezielt zu konfigurieren. Dies ist auch mit einer Datei möglich, jedoch lässt sich eine Konfiguration durch diese Technik deutlich besser strukturieren.

#### 2.2.2.2. Salt State Repository

Das Salt State Repository ist, wie auch bei Ansible in *Kapitel 2.2.1.2 Ansible Modules & Ansible Galaxy* beschrieben, eine eigene Plattform von SaltStack, in dem sich verschiedenste Salt State Dateien befinden. Zur Verwendung dieser state Dateien müssen die state functions lediglich richtig benutzt werden. Das passiert durch das Auflisten der package Namen, welche installiert werden sollen. Weiterhin ist zu beachten, dass es für die verschiedenen Betriebssysteme Windows, Linux und Mac OS X[8] jeweils ein eigenes Repository für die zu konfigurierenden Installationen gibt.

[8] Das Repository für das Betriebssystem OS X ist bisher noch nicht mit Applikationen zum Installieren gefüllt.

#### 2.2.2.3. Vorteile und Nachteile

Die bisher dargestellten Informationen zeigen, wie die Technologie in ihren Grundzügen funktioniert. Während ein Grundverständnis aufgebaut wurde, ist aufgefallen, dass auch bei dieser Technologie gewisse Nachteile entstehen:

- Sehr komplexe Dokumentation
- Ausführung von Jinja Code direkt zu Beginn der Ausführung
- Hauptunterstützung von Linux Systemen
- Viele verschiedene Namen für Features / Bestandteile
- Minion wird bei Verwendung des Masters benötigt

Abschließend folgen die Vorteile, die sich aus den bisherigen Informationen und weiteren Recherchen ergeben haben:

- Extrem große Anzahl an Rechnern automatisiert versorgen
- Leichter Einstieg in die Grundlagen
- Kompakte und verständliche Beschreibungssprache
- Mehrere Paketmanager durch einen einzigen Befehl
- Sehr viele Funktionalitäten sind vorhanden

### 2.2.3 Docker

Docker ist zurzeit eine sehr rasant wachsende Technologie. Der Grundgedanke von Docker ist, dass man Software in kleinen leichtgewichtigen Container verpackt. Dieser Container beinhaltet ein vollständiges Dateisystem mit allen benötigten Informationen, die zum erfolgreichen Starten eines Containers mit dessen Applikationen benötigt werden. Des Weiteren ist es mit dieser Technologie möglich die Docker-Container[9] zu erstellen, zu starten und zu entfernen. Dadurch entstehen Möglichkeiten, Ressourcen effizienter zu nutzen. Eine Applikation, die auf einem Server betrieben wird, sollte, wenn sie nicht verwendet wird, leicht entfernt werden können. Ein Beispiel ist das Testen von Software, da nach der Ausführung der Tests dieses System vorrübergehend nicht mehr benötigt wird, bis die Tests erneut ausgeführt werden sollen.

#### 2.2.3.1. Grundbegriffe

#### Docker-Container

Wie bereits in der kurzen Einführung zum Thema Docker beschrieben, handelt es sich bei Docker um leichtgewichtige Linux-Container, die die Grundfunktionalitäten zum Betreiben von Applikationen innerhalb des Containers beinhalten. Dabei beschränken sich diese Funktionalitäten auch nur auf die wesentlichen, die hierzu benötigt werden. Außerdem ist ein Docker-Container auf einem Host-OS ein eigenständiger Prozess. Erstellt wird ein solcher Docker-Container, indem man ein sogenanntes Docker-Image verwendet.[10] Im Aufbau gibt es ein Basis Image, welches ein Betriebssystem beinhaltet. Das OS-Image kann in diesem Fall zusätzlich extrem klein sein, da viele Funktionalitäten gar nicht benötigt werden. Zum Beispiel gibt es ein Linux Alpine Image, welches nur eine Größe von ca. 6 MB aufweist. Innerhalb dieses Containers können nun weitere Images benutzt werden, um sie dort in

[9] Docker-Container sind kleine leichtgewichtige Linux-Container die auf verschiedenen Betriebssystemen verwendet werden können. Unter Windows und Mac wird zurzeit eine VM benötigt.

[10] https://www.docker.com/what-docker#/VM

verschiedene Layer zu schichten. Die Begrifflichkeiten Docker-Image und Docker-Layer werden zur Erläuterung in den nachfolgenden Kapiteln beschrieben. Von großer Bedeutung ist hier, dass mehrere Docker-Container zusammen die gleichen Images verwenden können. Ausschlaggebend ist dafür der oberste Layer eines jeden Docker-Containers. Jegliche Schreiboperationen auf einen Container werden nur auf diesen Layer vollzogen (oberster Layer), was wiederum bedeutet, dass alle darunter liegenden Layer unberührt und unverändert bleiben.[11]

## Docker-Image

Docker-Images sind eine zentrale Komponente der Docker Technologie. Ein Docker-Image enthält im Speziellen gewisse Funktionalitäten wie z.B. Frameworks, Software oder ein Betriebssystem. Wie im Bereich der Docker-Container bereits beschrieben entsteht aus den Docker-Images ein Docker-Container. Innerhalb dieses Containers können nun verschiedenste Applikationen, wie beispielsweise ein Web-Server verwendet sowie verwaltet werden. Verschiedene Docker-Images lassen sich in dem von Docker eigenem Docker-Hub finden. Dort kann man nach unterschiedlichsten Images suchen und diese herunterladen. Auch die Erstellung eigener Docker-Images ist möglich. Diese können bei Bedarf ebenfalls in den Docker-Hub geladen werden.[12] Erstellt wird ein Image durch ein Dockerfile. Diese Datei enthält Kommandos und Befehle, durch die festgelegt wird, wie das Docker-Image aussehen wird. Genauer betrachtet werden können diese Dateien im Anhang oder auf den von Docker vorgegebenen Internetseiten.

## Layer

Als nächstes werden die bisherigen Begrifflichkeiten Container und Images zusammen verwendet und mit den Layern in Verbindung gesetzt. Wie aus dem vorherigen Teil entnommen werden kann, besteht ein Container aus einen bzw. mehreren Docker-Images. Diese Images bilden nun viele verschiedene Layer, wobei der oberste Layer auch Container Layer genannt wird. Dieser Layer wird zum

[11] https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/
[12] https://docs.docker.com/docker-hub/

Schreiben, Löschen und Modifizieren von Dateien verwendet. Jeder Layer, der durch das Hinzufügen eines neuen Images dazu kommt wird dabei immer auf den vorherigen, oberen Layer gestapelt. Dies betrifft jedoch nicht den Container Layer. Um diese Schichtung zu gewährleisten, verwendet man die bestehende Dateisystemtechnologie des Union-Filesystems.

## Docker-Daemon

Eine weitere Hauptkomponente von Docker ist der Docker-Daemon, der einige Aufgaben übernimmt, die zum erfolgreichen Konzept der Docker-Plattform beitragen. Der Aufgabenbereich des Docker-Daemon liegt darin, die Container zu erstellen, starten und zu überwachen (Monitoring). Außerdem werden durch den Docker-Daemon die Images gebaut und gelagert [vgl. Mouat 2015, Chapter 4.].

## Docker-Client

Mithilfe von Docker-Client ist eine Kommunikation zum Docker-Daemon vorhanden. Der Docker-Client ist zudem die definierte Benutzerschnittstelle zur Kommunikation mit Docker. Diese Kommunikation findet über das http-Protokoll statt und wird beispielsweise in Form der Kommandozeile genutzt. Dort können dann Befehle ausgeführt werden, mit denen Container gebaut, Images heruntergeladen und Container gestartet werden können. Des Weiteren ist es möglich, mit dem Docker-Client mehrere Docker-Daemons anzusprechen, die wiederum keine Beziehungen zueinander haben.[13]

## Registry

Abschließend in der Reihe der Grundbegriffe ist noch die Registry zu nennen. Innerhalb einer Registry befinden sich bei Docker die Docker-Images. Die von Docker erstellte Registry Docker-Hub beinhaltet die offiziellen Images, welche von Docker bereitgestellt werden und sicher sind. Diese können auch von privaten Nutzern in den Docker-Hub geladen werden, was dazu führt, dass kein Schutz vor

[13] https://docs.docker.com/engine/understanding-docker/

Schadcode mehr vorhanden ist. Dadurch lässt sich im Docker-Image Bereich eine große Sammlung an Images finden. Ebenfalls möglich ist es, sich eine eigene Registry einzurichten um ein privates Lagern der Images zu ermöglichen.

### 2.2.3.2. Docker-Architektur

Mithilfe der vorangegangenen Grundbegrifflichkeiten kann die Architektur von Docker im weiteren Detail betrachtet werden. Unterstützend hierzu gibt es folgende Abbildung, welche zeigt, wie die zuvor genannten Komponenten miteinander interagieren.

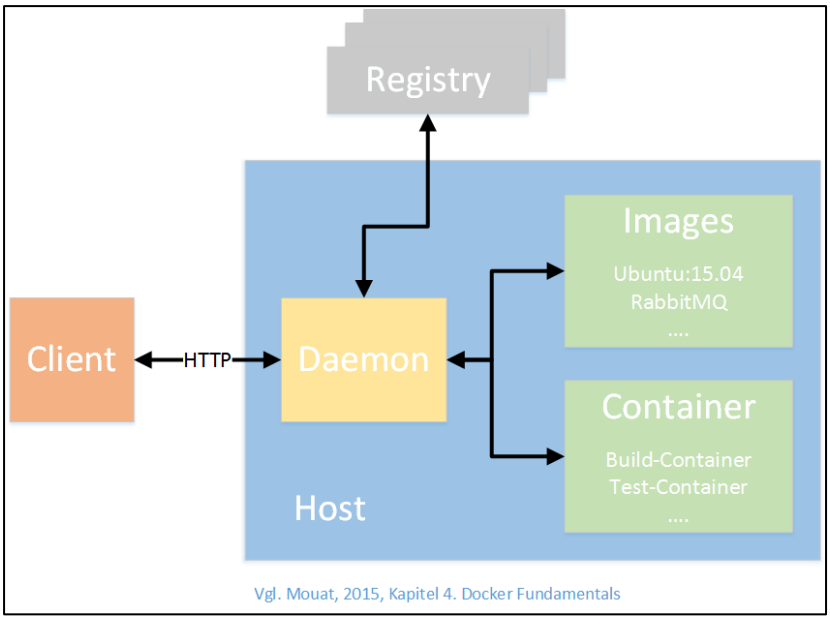


**Abbildung 5 Docker-Architektur**

Wie hier zu sehen ist, ist der Docker-Daemon, wie bereits erwähnt, eine zentrale Komponente und steht in Verbindung zu den weiteren genannten Komponenten Client, Registry, Container und den Images.

### 2.2.3.3. Provisionierung mit Docker

Nachdem nun die Grundlagen von Docker aufgezeigt wurden, folgt der Abschnitt, mit dem erläutert werden soll, wie man mithilfe von Docker eine Provisionierung realisieren kann. Hierbei gibt es zwei verschiedene Möglichkeiten, die in Betracht gezogen werden können, um dieses Ziel zu erreichen. Die erste Möglichkeit ist, dass man je nach Infrastruktur eine Realisierung mit Docker selbst macht. Dafür wird im wesentlichen Docker zusammen mit einer Registry benötigt. Innerhalb der Registry werden dann die benötigten Infrastrukturkomponenten als Images abgelegt und können von Docker verwendet werden. Dabei wird im Einzelfall ein Container mit der zu konfigurierenden Infrastruktur bereitgestellt. Die zweite Möglichkeit umfasst eine indirekte Provisionierung durch Docker. Hierbei kann ein Tool zur Infrastruktur Automatisierung genutzt werden, die mit Docker zusammen die vollständige Einrichtung realisiert. Hierzu wird dieses Tool wie z.B. SaltStack genutzt, um Docker-Container aufzusetzen, welche im Anschluss den Rest der Infrastruktur bereitstellen. Im Fall von vielen Systemen mit der gleichen Infrastruktur kann das von Docker eigene Docker-Swarm verwendet werden. Docker-Swarm ist im Prinzip ein eigenständiger Container, der weitere Container beinhaltet und diese verwaltet. Das bedeutet, dass die Konfiguration mittels Docker keine Abhängigkeiten zu einer anderen Technologie benötigt, wie in der ersten Variante beschrieben wurde. Diese können dann bei Bedarf als weitere Unterstützung innerhalb des Docker-Swarm verwendet werden.

### 2.2.3.4. Vorteile und Nachteile

Abschließend zum Thema Docker werden Vor- und Nachteile aufgezeigt, die diese Technologie mit sich bringt. Angefangen bei den Nachteilen lassen sich folgende Schwachpunkte hier finden:

- Stark in der Entwicklung[14]
- Dokumentation veraltet sehr schnell
- Abhängigkeiten zu einer Registry
- Verwaltung von vielen Containern (Kubernetes, Apache Mesos)

[14] Kann als Nachteil gesehen werden, da viele und schnelle Änderungen stattfinden

Im Wesentlichen sind diese Nachteile weniger schlimm für diese Technologie, da die Nachfrage extrem hoch ist. Aus diesem Grund ist auch der erste Punkt der Nachteile vollkommen verständlich, da viel Bewegung auch positive Aspekte besitzt. Damit nun aber die Vorteile von Docker nicht verloren gehen, werden hier nachfolgend die größten Vorteile von Docker erwähnt.

- Die Containertechnologie ist leichtgewichtig
- Die Containertechnologie ist schnell
- Mehrere Betriebssysteme werden unterstützt
- Benötigter Ressourcenbedarf ist gering
- Wartezeiten entgegen virtueller Maschinen ist geringer

Diese kleine Anzahl an Vorteilen ist nur der Anfang von dem, was mit der Docker-Technologie noch möglich ist. Unterschiedliche Punkte können je nach Nutzer anders ausfallen bzw. empfunden werden. Zum Schluss kann man sagen, dass man mit Docker eine neue und erfolgreiche Technologie bekommt, die gewisse Stärken und Schwächen besitzt und besonders an Wichtigkeit gewinnt.

### 2.2.4 Technologie Vergleich

In diesem Abschnitt werden die drei aufgeführten Technologien SaltStack, Ansible und Docker miteinander verglichen. Hierbei sollen Unterschiede der einzelnen Technologien gezeigt werden, damit man eine gezieltere Auswahl für dessen Verwendung in verschiedenen Bereichen treffen kann.

#### 2.2.4.1. SaltStack vs. Ansible

Als erstes betrachtet werden die beiden Technologien Ansible und SaltStack. Auffallend bei diesen beiden Vertretern ist, dass unterschiedliche Ansätze bezüglich der Provisionierung bevorzugt werden. Auf die bevorzugte Variante wird hier das Hauptaugenmerk gelegt, aber auch die jeweils andere Lösung ist durch gezielte Änderungen möglich.

Weiterhin ist es den Entwicklern von SaltStack wichtig, dass extrem viele Maschinen gleichzeitig verwaltet werden können. Dieses Ziel hat sich Ansible nicht gesetzt, somit wird es dort zwar ab einer bestimmten Anzahl an gleichzeitig verwalteten Maschinen nicht mehr funktionieren oder einfach schlechter, jedoch wird dieses Ziel auch nicht verfolgt, da man selten eine solch große Anzahl gleichzeitig ansprechen möchte. Da dieses aber ein Ziel von SaltStack ist, ist die Technologie SaltStack auch sehr schnell bei der parallelen Konfiguration einer großen Anzahl von Maschinen.

Problematisch ist jedoch hierbei das Thema Sicherheit bei der Übertragung und Erfolgsrate. Salt hat im Gegensatz zu Ansible keine grundlegende Sicherheit anzubieten, da dort das SSH Protokoll, wie es in Ansible verwendet wird, nicht genutzt wird. Ein Grund dafür ist der Faktor der Verlangsamung bei der Provisionierung, denn ein gewisser Sicherheitsstandard beinhaltet mehrere Schritte, die durchgeführt werden müssen, damit diese Sicherheit auch gewährleistet werden kann. Auch hier ist es möglich, das Salt-SSH zu nutzen, hierbei wird direkt darauf verwiesen, dass dieses Vorgehen den Salt Konfigurierungs-Prozess verlangsamt[15]. Ein weiterer Unterschied ist in Bezug zur Architektur vorhanden und zwar nutzt die Salt Implementation die Master & Minion Architektur, währenddessen Ansible eine Agentenlose Architektur besitzt. Diese Agentenlose Architektur ist im Gegensatz zu der Salt Master & Minion Architektur auf kleinere Systeme effizienter, um beispielsweise kleine Tests durchführen zu können, da kein separater Agent installiert werden muss[16].

#### 2.2.4.2. SaltStack vs. Docker

Als nächstes folgt eine Betrachtung von SaltStack und Docker. Eine wichtige Information vorab ist hier, dass man mit Docker keine richtige Provisionierungs-Software besitzt. Dies ist damit zu begründen, da Docker eine Containertechnologie ist, die zur Bereitstellung von Software, Services etc. entwickelt wurde. Eine Provisionierung wird in diesem Kontext grundsätzlich nur durch spezielle Methoden

[15] https://docs.saltstack.com/en/latest/topics/ssh/
[16] https://www.upguard.com/articles/ansible-vs-salt

realisiert. Diese Information ist für den nachfolgenden Teil in *2.2.4.3 Ansible vs. Docker* ebenfalls zu beachten.

Aufgrund der Tatsache, dass Docker keine Provisionierungs-Technologie ist fällt direkt auf, dass es in Docker das Konzept der Beschreibung eines Zustandes, wie es in SaltStack der Fall ist, nicht gibt. Hier verwendet man die Images, welche nicht wie eine Salt State Datei den Zustand des gewünschten Systems beschreiben. In den Docker-Images können verschiedenste Befehle verwendet werden, wobei es sich hauptsächlich um CMD Befehle handelt. Diese sind oftmals jedoch sehr komplex und setzen ein gutes Grundverständnis im Umgang mit der Kommandozeile voraus.

Des Weiteren muss eine Docker-Umgebung erst einmal auf einem Host-System aufgesetzt werden, was entweder manuell oder durch die Verwendung eines Provisionierungs-Tools geschehen kann. Für die Verwendung von SaltStack ist bei der Master und Minion Variante dies ebenfalls notwendig. Damit man mit Docker zusätzlich eine Multi-Provisionierung unterstützen kann, wird entweder die Docker-Compose oder die Docker-Swarm Technik benötigt. Mithilfe dieser Techniken lassen sich dann ebenfalls mehrere Docker-Systeme verwalten. Die Umsetzung von SaltStack ist in diesem Bereich etwas verständlicher und einfacher, da zusätzliche Technologien nicht benötigt werden. Im Allgemeinen ähneln sich diese beiden Technologien in Bezug auf die Provisionierung.

#### 2.2.4.3. Ansible vs. Docker

Zum Schluss folgt für das Kapitel des Technologievergleichs ein Vergleich von Ansible und Docker. Für diesen Vergleich lässt sich die Unterscheidung von Docker und SaltStack besonders gut verwenden. Blickt man nun auf die verschiedenen Architekturen von Docker und SaltStack ist im Wesentlichen schnell erkennbar, dass die gleiche Art verwendet wurde und zwar das Master & Slave Prinzip (siehe Docker-Swarm). Untersucht man nun jedoch die Technologien Ansible und Docker fällt das genaue Gegenteil dieser Gemeinsamkeiten von Salt und Docker auf. Ansible besitzt nicht wie Salt oder Docker das Prinzip von Master & Slave, sondern verwendet eine Masterlose Architektur.

Als Gemeinsamkeit der zwei Technologien ist die Unterstützung von Sicherheitsmaßnahmen zu nennen, da diese sowohl in Ansible als auch in Docker umgesetzt sind. In Docker ist die Sicherheit über die Protokolle HTTPS, SSH und SSL[17] gegeben. Währenddessen sind in Ansible die Sicherheitsprotokolle über HTTPS und SSH vorhanden.

Neben der Umsetzung von Sicherheit ist ein weiterer großer Unterschied zwischen Ansible und Docker, deren unterschiedliche Art der Funktionsweise. Während Docker durch ein Skript mit CMD-Befehlen den gewünschten Stand erreicht, so nutzt die Ansible Implementierung eine Methode, welche den Zielzustand des Systems beschreibt. Dieser Unterschied geht auf die verschiedenen Konzepte von funktionalen und imperativen Programmiersprachen zurück[18]. Zudem ist die Verwendung der Ansible Playbooks erheblich leichter als die der Dockerfiles, weil der Aufbau verständlicher ist, als ein komplexes und verschachteltes CMD-Skript. Weitere Unterscheidungen können innerhalb eines intensiven Vergleiches gemacht werden, wobei man trotzdem darauf achten sollte, dass die konzeptionellen Visionen der Technologien grundlegend verschieden sind.

[17] SSL ist wie auch HTTPS und SSH ein Protokoll zur Kommunikation. SSL ist jedoch im Bereich der Protokolle ein Sicherheitsprotokoll, welches spezielle Zertifikate zur Sicherung verwendet.

[18] Die funktionale Programmierung beschreibt man, was das Ergebnis sein soll und nicht wie man dorthin gelangt. Bei der imperativen Programmierung hingegen beschreibt man den Weg zum Ergebnis.

# 3 Herausforderungen von Infrastructure-as-Code

In diesem Kapitel wird spezieller auf das Thema Infrastructure-as-Code eingegangen. Im Speziellen soll hierbei darauf geschaut werden, wie gut diese Technologie für den Einsatz zum Einrichten von Infrastrukturen geeignet ist. Des Weiteren zeigt dieser Abschnitt einen Überblick über die zu bewältigenden Herausforderungen, die Infrastructure-as-Code bewältigen muss und welche Anforderungen eine solche Automatisierungs-Technik mitbringen muss, um die Wünsche der Stakeholder zu erfüllen.

## 3.1 Legacy Anwendungen

Historische Softwareprodukte werden oftmals auch noch nach vielen Jahren verwendet. Deswegen muss es auch hier möglich sein, weitere Wartungen durchzuführen, für den Fall, dass Fehler auftreten sollten. Dazu wird vorausgesetzt, dass die alten Umgebungen zur erfolgreichen Weiterentwicklung wieder so eingerichtet werden, wie diese zum Stand des Release waren. Genau diese Realisierung gehört zu den Herausforderungen von Infrastructure-as-Code.

### 3.1.1 Reproduzierbarkeit

Das Aufsetzen von Softwaresystemen und deren Infrastruktur ist ein essentieller Bestandteil der Softwareentwicklung. Durch diese Technik ist es möglich, solche Systeme bzw. Infrastrukturen ohne weitere Schwierigkeiten zu reproduzieren, da z.B. Entscheidungen über Softwareversionen etc. nicht getroffen werden müssen. Dementsprechend können die gesamten Informationen, die zu einer erneuten

Konfiguration benötigt werden, aus den Skripten der gewählten Provisionierungs-Tools entnommen werden. Dadurch entfallen auch folglich die bisherigen Anleitungen, welche als Textdokumente archiviert wurden [vgl. Morris 2016, S. 10]. Ebenso ist der Prozess der Provisionierung sehr vorteilhaft, weil benötigte Software mit deren richtigen Versionen für die Verwendung innerhalb einer Versionsverwaltung, Registry oder auf einem Server archiviert werden und deshalb auf Wunsch abrufbar sind.

Zugleich hat der Prozess jedoch den Nachteil, dass die Infrastruktur bei Änderungen des Betriebssystems im Zweifelsfall schlecht bis gar nicht mehr funktionsfähig ist. Dies hat dann zur Folge, dass man eine erneute Planung bezüglich der Infrastruktur treffen muss, was wiederum einen weiteren Aufwand bedeutet. Die Wahrscheinlichkeit, dass ein derartiger Fall eintreten könnte, ist mittlerweile jedoch relativ gering, da ein solches Szenario im schlechtesten Fall nur eintritt, wenn Software nicht mehr vom Hersteller unterstützt wird, verwendete Software plötzlich andere Betriebssysteme oder andere Software benötigt oder auf Wunsch zwingend etwas geändert werden soll. Angesichts dieser Aspekte ist erkenntlich, inwiefern diese Automatisierungstechnologie innerhalb der Informatik von Nutzen ist und was in speziellen Fällen zu schwerwiegenden Problemen führen könnte.

### 3.1.2 Verknüpfung mit Infrastructure-as-Code

Die Altsysteme aus Softwareunternehmen wurden bis zu dem Zeitpunkt der Entstehung von Infrastructure-as-Code meistens lange Zeit am Leben gehalten, damit eine weitere Unterstützung von Kunden machbar war. Mit dem Beginn von IaC stand man jedoch vor dem Problem eine Integration der Technologie mit dem Legacy System zusammen zu schaffen. Um Verknüpfungen dieser Art zu realisieren, sind gründliche Analysen notwendig, die potentielle Risiken schon am Anfang aufzeigen, damit darauf entsprechend reagiert werden kann. Für die Umsetzung selbst muss dann nur noch das Automatisierungs-Tool verwendet werden, indem dort alle nötigen Informationen wie Repository, Software-Code etc. richtig miteinander verbunden werden, sodass am Ende das vorherige System wieder einsatzbereit ist. Daraus entnehmend ist nun zu sagen, dass der Prozess der Analyse

für eine Verwendung der ganzen Technologie weitaus kostenintensiver ist als die Umsetzung selbst. Das vorausgesetzte Wissen für die Verwendung dieser Technik ist das Hauptproblem. Durch das Erlernen der benötigten Wissensbasis, um ein solches Projekt zu realisieren, wird der gesamte Prozess natürlich wiederum langwieriger und verursacht eine Menge an Mehrkosten.

### 3.1.3 Trennung von Anwendungscode und Infrastrukturcode

Ein weiterer wichtiger Aspekt bei der Betrachtung der Legacy Anwendungen bezüglich deren Infrastruktur ist die Organisierung der verschiedenen Bereiche der Implementierung. Strukturelle Trennungen unterschiedlicher Code Gebiete ist für die Infrastrukturautomatisierung durch IaC gleichermaßen von hoher Wichtigkeit. Hierbei zielt dies darauf ab, dass durch den Code der Infrastructure-as-Code Technologie die Konfiguration des Systems übernommen wird. Daraufhin muss man daran denken, innerhalb der konfigurierten Systeme nichts manuell hinzuzufügen, da sonst die verschiedenen Code Basen miteinander kollidieren können und gewünschte Ziele nicht mehr erreicht werden. Manuelle Änderungen können hier im Zuge einer regelmäßigen Neuinstallation entfernt werden, weil ein fest definierter Zustand wiederhergestellt werden soll (Basis Installation). Hinsichtlich dieser Tatsache ist daher die Trennung von Code in diesem Bereich von großer Bedeutung.

Diese Technologie führt die gewünschten Veränderungen für ein System durch, indem die fest definierten Installationsdateien hierzu verwendet werden. Von außen einwirkende Codierungen sollten bei der Verwendung von Infrastructure-as-Code möglichst verhindert werden, um potentielle Konfigurationsfehler auf den vorhandenen Maschinen zu verhindern.

Im Allgemeinen sollen Basis Einstellungen der Automatisierung nicht geändert werden. Bei spezielleren, unterschiedlichen Konfigurationen sind verschiedene Systeme natürlich nicht schlecht und somit auch möglich [vgl. Morris 2016, S. 7].

## 3.2 Abzuwägende Faktoren

Damit eine Technologie wie Infrastructure-as-Code von Anwendern genutzt wird, bedarf es hier an ordentlichen Analysen sowie Überprüfungen auf dessen Nutzungsfähigkeit für den Gebrauch (siehe 3.1.2 Verknüpfung mit Infrastructure-as-Code). Faktoren die dabei wichtig sind, sind je nach Anwender möglicherweise unterschiedlicher Art. Grundsätzliche Faktoren wie Qualität, Geschwindigkeit und Komplexität werden hier oftmals als Richtlinie genommen und überprüft. Speziell für dieses Thema können aber auch die Faktoren Entwicklungsfähigkeit, Qualität oder Zuverlässigkeit als Maßstab verwendet werden. Genau diese Punkte folgen nun in dem nächsten Abschnitt und werden erläutert.

### 3.2.1 Qualität

Beschreibt man die Qualität von Infrastructure-as-Code, kann man auf mehrere, verschiedene Merkmale zurückgreifen und den Begriff für dieses Thema definieren. Aus diesem Grund wird in diesem Abschnitt nur eine geringe Menge an ausgewählten Kennzeichen für die Definition der Qualität beschrieben.

Einleitend wird mit dem allgemeineren Kennzeichen des Software Engineering (SE) begonnen. Beginnt man damit eine Automatisierungstechnologie zu verwenden, kann das Einhalten von Software Engineering Regeln zu einer Steigerung der Qualität beitragen. Dazu gehören wie auch in der normalen Softwareentwicklung Clean Code, Dokumentation des Programmcodes oder die Verwendung von vorhandenen Design-Patterns. Zusätzlich wird durch die Verwendung dieser Konzepte die Qualität des Softwarecodes gesteigert, was für den Lebenszyklus von Software enorm wichtig ist. Außerdem dienen die SE-Richtlinien im Allgemeinen noch dazu, die Entwicklung einer Software zu unterstützen.
Als nächstes folgt ein Qualitätsmerkmal, das gern in der Softwareentwicklung gesehen ist. Es handelt sich um das Änderungsfeedback der Technologie. Damit ein Softwareentwickler seinen entwickelten Programmcode auf Korrektheit überprüfen kann, benötigt er Ausführungsresultate. Um ein solches Feedback ermöglichen zu können, kann dabei auf die Durchführung von Tests, das Logging der Software oder

die Ausgabe von Resultaten der Provisionierung gesetzt werden. Besonders hilfreich ist ein schnelles Feedback durch Tests, für Fehler, die durch kleine Änderungen entstanden sind. Nachdem die Fehlerursache lokalisiert wurde kann zeitig eine Korrektur eingepflegt werden, was zur Folge hat, dass die Qualität wieder zunimmt. Genauso hilfreich für das Erreichen eines qualitativen Prozesses zur Einrichtung einer Infrastruktur ist eine gut verständliche Beschreibung für die Verwendung der Tools. Dokumentationen, die schlecht strukturiert, unvollständig oder missverständlich geschrieben sind, können Verschlechterungen hervorrufen, da Umsetzungen vielleicht anders verstanden werden, als in der Dokumentation beschrieben. Bei diesem Qualitätspunkt ist es vor allem wichtig, die Dokumentation kontinuierlich zu pflegen und aktuell zu halten. Ansonsten ist es schwierig, neue Features sowie Änderungen alter Versionen zu verstehen und anzuwenden. Zusätzlich kann man in dem Bereich der Qualität auch das Auftreten der Entwickler oder des Unternehmens eingliedern. Wenn abzusehen ist, dass ein solches Produkt von einer oder mehreren finanzstarken Unternehmen unterstützt wird und die Entwickler und Mitarbeiter engagiert und kontinuierlich an dem Produkt arbeiten, so zählt auch dies zur qualitativen Wertung der Technologie. Damit der Punkt jedoch nicht missverstanden wird, ist zu sagen, dass auch Entwickler ohne eine Firma als Unterstützung erfolgreich und gut am Markt bestehen können.

### 3.2.2 Entwicklungsfähigkeit

Die Entwicklungsfähigkeit von Unternehmen beschreibt im Wesentlichen, wie gut die Entwicklung der Software voran geht. Hierbei ist es wichtig, mehrere Aspekte zu untersuchen, die für diese Eigenschaft bedeutend oder nachteilig sind. Dieses Unterkapitel stellt außerdem ausgiebig den Bezug zu Infrastructure-as-Code her. Das geschieht durch die Verknüpfung von bestehenden Merkmalen zur Entwicklungsfähigkeit und der IaC Technologie.

Zu Beginn werden die Entwickler für die Eigenschaft betrachtet. Für eine positive Entwicklungsfähigkeit sorgt zu großen Teilen jeder Mitarbeiter eines Softwareunternehmens. Wenn die Masse an Mitarbeitern motiviert und engagiert die Arbeiten aufnimmt, so steigert dies die Zusammenarbeit und die Entwicklung

wird vorangetrieben. Als nächstes folgt die Verwendung von IaC durch die Software-Entwickler oder Administratoren. Dessen Verwendung dient den bereits genannten Stakeholdern als Unterstützung, was wiederum dazu führt, dass die Entwicklungsfähigkeit verbessert wird. Resultierend werden so die Grundlagen für eine gut geplante Infrastruktur gelegt. Daraufhin folgt dann der nächste Schritt, der weitestgehend nur die konfigurierte Infrastruktur darstellt. Hier wird sichergestellt, dass die gewünschte Infrastruktur den Vorgaben entspricht und wird bei Unstimmigkeiten diesbezüglich angepasst bis der Zielzustand erreicht ist. Existiert die Infrastruktur im vollen Umfang, beginnt eine Phase, in der die Arbeit aufgenommen und umgesetzt werden kann. Dabei muss dennoch darauf geachtet werden, dass aus den bisherigen Schritten koordinierte und strukturierte Arbeitsprozesse entstehen, durch die klar zu erkennen ist, wie etwas ablaufen soll. Ist hierbei eine ordentliche Struktur vorhanden, so folgt der erste Punkt der motivierten Mitarbeiter wieder. Werden diese Schritte nach und nach realisiert, so steigert das die Entwicklungsfähigkeit durch die Verwendung der Technologie. Aufgrund dieser Betrachtung kann aus den oben genannten Schritten ein Kreislauf konstruiert werden, der in der folgenden Grafik noch einmal dargestellt wird.

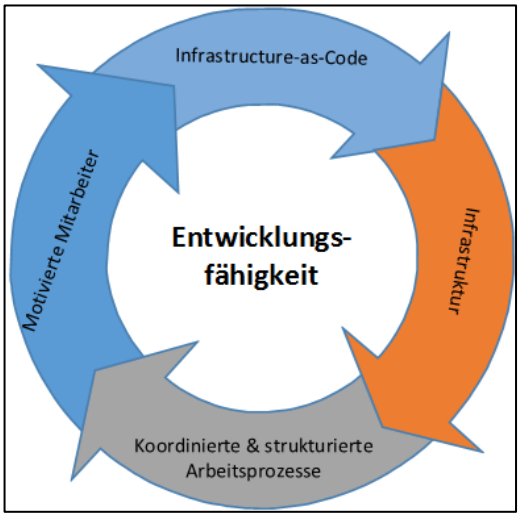


Abbildung 6 Entwicklungsfähigkeit mit Infrastructure-as-Code

Ein Nachteil, der in diesem Kreislauf allerdings erkennbar ist, ist die Abhängigkeit zu Infrastructure-as-Code. Um mit dem Schritt der Infrastruktur beginnen zu können,

müssen zuerst die Konfigurationsdateien des zugrundeliegenden IaC Tools fertiggestellt werden. Sobald diese Dateien den gewünschten Zielzustand repräsentieren, besteht der Vorteil, dass keine manuellen Konfigurationen mehr stattfinden müssen. Dementsprechend bedeutet dies, dass keine unterschiedlichen Ergebnisse mehr entstehen sollten, was zudem die Motivation der Mitarbeiter wieder steigert.

### 3.2.3 Zuverlässigkeit

Die Zuverlässigkeit für das Anwendungsgebiet von Infrastructure-as-Code ist besonders interessant, weil Konfigurationsprozesse mit einer unzuverlässigen Erfolgsrate ein Hindernis für die Prinzipien von IaC sind. Daher müssen die Tools zur Unterstützung der Technologie darauf ausgelegt sein, zuverlässige und konsistente Installationen zu generieren. Untersucht man nun diesen Faktor, können unterschiedliche Anforderungen gefunden und festgelegt werden. Bekanntlich ist aber darauf zu achten, dass diese Anforderungen von jeder Person anders wahrgenommen werden können. Wenn man sich überlegt, was man mit zuverlässig meint, kann dies extrem viele unterschiedliche Meinungen hervorbringen. Um eine Begründung zu finden, was zuverlässig in diesem Kontext bedeutet, kann der manuelle Konfigurationsprozess mit dem Prozess der automatisierten Variante verglichen werden.

Bei der manuellen Methode fällt schnell auf, dass eine einzelne Konfiguration leicht durchzuführen und weniger kritisch ist. Wenn etwas geändert werden muss ist auch dies zuverlässig und schnell gemacht. Wächst die Anzahl der einzurichtenden Systeme jedoch an, so wird die Übersicht eines manuellen Prozesses schwieriger. Daraufhin wird die Wahrscheinlichkeit einer erhöhten Fehlerquote steigen und eine Eigenschaft wie die Zuverlässigkeit nimmt rapide ab. Zugleich sieht man bei der automatisierten Methode, dass einzelne Konfigurationen ebenfalls problemlos durchgeführt werden können, wobei hier auf die Verlässlichkeit des verwendeten Tools vertraut werden muss. Während einzelne Installationen bei beiden Methoden recht ähnlich sind, ist der Unterschied mit mehreren Konfigurationen umso größer. Das manuelle Setup ist wie oben bereits beschrieben beschränkt, weil ein Mensch

ggf. Einstellungen eines Systems auf einem anderen vergisst, wenn die Menge der zu verwaltenden Systeme zu groß wird. Beschreibt man nun die Zielsysteme mit einer Konfigurationsdatei und der Hilfe von IaC, werden alle Systeme automatisiert aufgesetzt und sollten nach Ausführung alle den gleichen Zielzustand aufweisen. Dabei ist es von Vorteil, dass keine Einstellungen vergessen werden können, da der beschriebene Zustand einheitlich durch dessen Definition ist. Deshalb zeigt das Resultat der beiden Vorgehensweisen, dass die Automatisierung im Bereich des zuverlässigen Arbeitens stärker überzeugen kann. Tatsächlich beinhaltet aber auch die Automatisierung Nachteile betreffend der hier aufgeführten Eigenschaft. Passiert etwas automatisch durch ein Programm, Tool o.ä., kann es angesichts der Technik zu Verbindungsabbrüchen kommen. Genau das schränkt auch IaC in diesem Fall ein, da bei einer manuellen Konfiguration ein solches Problem im Normalfall nicht existiert. Ausgeschlossen sind diese jedoch auch bei einer manuellen Konfiguration nicht.

# 3.3 Komplexität

In diesem Unterkapitel werden Themen beschrieben, die sich umfassend mit Merkmalen von Infrastructure-as-Code bezüglich der Komplexität auseinandersetzen. Fokussiert werden hier besonders Gebiete, die bei der Umsetzung der Technologie Schwierigkeiten, Risiken oder Probleme darstellen können. Infolge dessen wird es möglich sein, die Thematik aus anderen Perspektiven zu betrachten.

## 3.3.1 Verlust von Flexibilität

Wie bereits in einigen Abschnitten dieser Arbeit beschrieben wurde, verliert man durch die automatisierte Konfiguration von Systemen unter anderem an Flexibilität. Dieser Verlust ist dadurch zu begründen, dass das Einrichten von Infrastrukturen mittels der Tools für das automatisierte Konfigurationsmanagement geschehen soll. Was innerhalb der Einleitung deutlich gemacht wurde, ist, dass man dazu die Konfigurationsdateien verwenden soll, um einzelne Systeme zu provisionieren. Das bedeutet, dass man für kleine Änderungen, auch wenn es sich beispielsweise nur um ein einziges System handelt, die Konfigurationsdatei dafür ändern muss. So wird schnell deutlich, wieso die Flexibilität darunter leidet, denn nachdem die Änderung eingepflegt wurde, muss sie sehr wahrscheinlich durch ein Sourcecode-Review angeschaut, getestet und schließlich ausgeführt werden. Tatsächlich kann die manuelle Änderung in diesem Fall höchstwahrscheinlich schneller und mit weniger Aufwand umgesetzt werden.

Ein weiterer Flexibilitätsverlust entsteht wegen der Abhängigkeit zu bestimmten Tools aus dem Konfigurationsmanagement. Die Einschränkung entsteht hier, durch die Auswahl eines bestimmten Tools, weil nicht jedes Tool alle gewünschten Funktionen anbieten kann. Wenn dies der Fall ist muss man für die Wahl dementsprechend Kompromisse eingehen und sorgfältig entscheiden. Des Weiteren kann es passieren, dass Verantwortlichkeiten für Infrastrukturen auf andere Mitarbeiter übergehen. Wenn die Infrastruktur dann auf einer IaC Technologie basiert und der neue Mitarbeiter keine Erfahrung mit dem Tool hat,

können hier die Änderungen sogar erst nach Einarbeitung in die Technik vollzogen werden. Dieser Problemfall sollte jedoch durch Schulungen o.ä. vermieden werden können. Angesichts der genannten Verluste muss trotzdem gesagt werden, dass Infrastructure-as-Code auch positive Aspekte zur Flexibilität mitbringen wird.

### 3.3.2 Plattformabhängigkeiten

Bei diesem Teil der Arbeit soll die Plattformkompatibilität der verschiedenen Betriebssysteme zu den Provisionierungs-Tools dargestellt werden. Um zeigen zu können, auf welche Plattformen die verschiedenen Vertreter der IaC Tools hier fokussiert sind, werden hier die in der Einführung genannten Tools referenziert. Zum Einstieg soll die folgende Tabelle einen Überblick über die jeweiligen Tools geben, die eine Unterstützung der drei Plattformen gewährleisten.

| Betriebssystem | Ansible[19] | SaltStack[20] | Docker[21] |
|---|---|---|---|
| **Windows** | **(√)**[22] | **(√)** | **(√)** |
| **Linux** | **√** | **√** | **√** |
| **Mac OS** | **√** | **(√)** | **(√)** |

Tabelle 2 Plattformabhängigkeiten der Provisionierungstechnologien

[19] http://docs.ansible.com/ansible/intro_installation.html#latest-releases-via-pip

[20] http://saltstack.com/wp-content/uploads/2016/08/SaltStack-Supported-Operating-Systems.pdf

[21] https://docs.docker.com/engine/installation/

[22] Bedeutet, dass dieses Tool die Umsetzung für das OS nicht unterstützt, die Lösung virtualisiert ist, nur spezielle Versionen funktionieren oder es wurde im Zeitraum der Arbeit umgesetzt.

#### 3.3.2.2. Windows

Das Betriebssystem Windows von Microsoft ist als Marktführer der Betriebssystembranche für die Desktop-PCs bekannt. Da Microsoft aber auch in der Serversparte aktiv ist, ist eine Automatisierung in diesem Gebiet grundsätzlich erst einmal denkbar. Auf Grund der hohen Anzahl an Windows Systemen kann man behaupten, dass eine Plattformunterstützung von Windows Systemen unbedingt vorhanden sein muss. Dafür muss jedoch in Erwägung gezogen werden, unter welchen Umständen eine Automatisierung hier sinnvoll ist und was möglicherweise gegen eine Umsetzung für Windows sprechen könnte. Wenn man sich zu Beginn nun die drei vorgestellten Technologien anschaut, sieht man, dass die Tools eine Unterstützung von Windows anbieten. Dieses Angebot ist bzw. war bisher eingeschränkt, wird aber nach und nach verbessert (siehe Tabelle *Kapitel 3.3.2 Plattformabhängigkeiten*).

In Ansible wurde inzwischen die Kompatibilität unter Windows realisiert, es ist jedoch recht neu. Dieses Feature beinhaltet aber nur die Chance, Windows Maschinen einzurichten. Als Kontrollmaschine[23] wird weiterhin ein Linux System benötigt, da Windows von Seiten der Entwickler hierzu nicht gedacht oder geplant ist[24].

Ein einigermaßen ähnliches Szenario ist auch durch die SaltStack Lösung gegeben. Ausgehend von der SaltStack Architektur, kann ein Windows System als Minion agieren, nicht aber als Master. Tatsächlich unterstützt wird es trotzdem für diverse Windows Plattformen, worunter sich auch die Windows Server befinden. Bevor SaltStack verwendet werden soll muss man darauf achten, die richtige Version zu verwenden, damit die gewünschten Versionen der Betriebssysteme auch unterstützt werden.

Abschließend kommen Aspekte zur Plattformabhängigkeit, die mit der Technologie Docker entstanden sind. Auch Docker beinhaltet einige Punkte zum Thema

[23] Maschine, von der aus die Provisionierung stattfindet (ggf. auch Administratormaschine genannt).
[24] http://docs.ansible.com/ansible/intro_windows.html

Betriebssystemabhängigkeit. Da die Entwicklung von Docker stark voran geht kommen schnell neue Features hinzu, was im folgenden Text skizziert wird.

Bisher hatte man unter Windows nur die Möglichkeit, Docker mithilfe einer virtuellen Maschine zu nutzen. Bereits in der frühen Phase von Docker wurde für Microsoft Nutzer das Komplettpaket *Docker-Toolbox* bereitgestellt, welche alle nötigen Elemente zur Verwendung von Docker beinhaltet. Nach der Einrichtung kann Docker dann problemlos genutzt werden. Mittlerweile ist im Rahmen von Zielen und Wünschen eine Lösung entwickelt worden, die ohne die Hilfe einer Virtualisierungs-Software wie z.B. *Virtualbox* auskommt. Dadurch kann man in Windows jetzt ebenfalls eine native Docker-Implementierung verwenden. Möglich ist dies jedoch erst seit kurzem unter Windows 10. Für die native Lösung in Windows wird zwar weiterhin eine Virtualisierung genutzt, da sie vorausgesetzt wird. Diese VM ist nun jedoch wesentlich schlanker, leichtgewichtiger und läuft im Hintergrund des Host-Systems.

#### 3.3.2.3. Linux

Sieht man sich nun die Spezifikationen der drei aufgeführten Tools genauer an fällt auf, dass alle drei erweiterte Schnittstellen für Linux offerieren (siehe Tabelle *Kapitel 3.3.2 Plattformabhängigkeiten*). Eine genauere Betrachtung jedes einzelnen Tools zeigt, dass der Linux Support stark fokussiert wird. Zu sehen ist dies bei Docker an dem Docker-Daemon, welcher in Linux direkt eingebettet ist, dass bei Salt der Master nur ein Linux System sein kann oder bei Ansible die Linux Versionen zum Installieren der Kontrollmaschine direkt aufgeführt sind. Auf Microsoft und Apple wird hingegen wenig bis gar nicht eingegangen.

Benutzt man also ein Linux OS stehen die Chancen auf die Unterstützung der spezialisierten Ausprägung des Linux sowie der Support sehr gut. Hierbei muss der Anwender ausschließlich darauf achten, dass das verwendete Linux System aufgelistet ist und welche Features für dieses bei der Verwendung möglicherweise nicht umgesetzt sind. Die Versionen Ubuntu, Red Hat und Debian sind allerdings bei jeder der Technologien schnell zu finden. Diese Varianten sind beispielsweise auch

im SaltStack Umfeld vollständig, unter der Voraussetzung, dass die Salt Version aktuell oder passend ist, unterstützt[25].

Des Weiteren ist die Umsetzung für Docker mit Abstand am besten, da ein Docker-Container ein Linux Container ist. Inmitten eines Linux Systems wird zudem keine Virtualisierung benötigt um die Docker-Umgebung starten zu können, was ein weiterer Pluspunkt für das Betriebssystem ist.

Die bisherigen Aussagen sind auf die Desktop PCs bezogen, jedoch sind auch die Server Varianten von Linux für die Automatisierungs-Software verwendbar. Deswegen kann der Schluss gefasst werden, dass die Betriebssysteme von Linux in diesem Bereich bisher einen ziemlich guten Standpunkt haben, was in erster Linie aber nur auf die drei Technologien dieser Arbeit zutrifft. Andere Software-Anbieter haben höchstwahrscheinlich andere Ziele und werden hier nicht weiter analysiert.

#### 3.3.2.4. Mac OS

Damit das Kapitel der Plattformabhängigkeiten übersichtlich vervollständigt werden kann, wird auch das von Apple entwickelte Betriebssystem Mac OS betrachtet. Die Betrachtungsweise bei diesem Anbieter ist ebenfalls interessant, da man bei Apple größere Schwierigkeiten bezüglich externer Software und anderer Kompatibilitäten hat. Deshalb ist die Sicht auf das Mac OS genauso wichtig für die Plattformabhängigkeiten wie die der zwei beschriebenen Vorgänger. Wie auch schon bei Windows zu sehen war, ist die Bereitstellung der Funktionalitäten für Apple durch diese Tools im Gegensatz zu Linux deutlich schlechter ausgebaut.

Derartig gut lässt sich Ansible zusammen mit dem Mac OS verwenden. Wenn man Mac Maschinen mit Ansible einrichten möchte, ist dies machbar. Außerdem ist es möglich, einen Mac als Kontrollmaschine zu verwenden, was bei Windows nicht unterstützt wird. Damit ist zu erkennen, dass die Entwickler von Ansible den Fokus

[25] Vollständig unterstützt bedeutet hier: Master Support, Minion Support und der Full Support. Diese drei Punkte bieten die bestmögliche Unterstützung von SaltStack aus (siehe Fußzeile 15).

eher auf die Systeme Linux und Mac OS gerichtet haben. Ferner ist es komfortabel, dass keine VM zur Verwendung benötigt wird.

Nachdem gezeigt wurde, dass die Ansible Realisierung hier als bedeutender angesehen wird, folgt die Sichtweise in Bezug auf SaltStack. Mithilfe der Technologie von SaltStack ist es möglich, Mac Systeme als Minion zu konfigurieren. Aber wie auch bei Windows besteht hier bisher keine Möglichkeit einer Masterkonfiguration der Systeme. Daher ist auch hier eine Einschränkung zur Plattform vorhanden, wie es auch schon bei Windows der Fall war. Ein vollständiger Support ist jedoch auch für Apple vorhanden.[26]

Als letztes soll dargestellt werden, wie sich das Betriebssystem von Apple mit der Technologie von Docker nutzen lässt. Dazu muss wie bei Linux und neuerdings auch Windows keine virtuelle Maschine mehr eingerichtet werden. Diese virtuelle Maschine wird ebenfalls direkt von Docker mitgeliefert. Die in *Kapitel 3.3.2.2 Windows* bereits dargestellte native Docker Lösung für Windows, ist unter Mac OS nun ebenfalls eine native Implementierung. Auch hier ist die benötigte Virtualisierung im Hintergrund aktiv, ohne dass ein Anwender aufwendige Software starten muss[27].

### 3.3.2.5. Die Cloud

Als Abschluss soll für das weitere Interesse ein kurzer Abschnitt folgen, der die Relation der Tools zu der Cloud darstellen soll. Vielen ist wahrscheinlich mittlerweile bewusst, dass die Cloud heutzutage immer mehr Aufsehen erlangt.

Ansible hat für Cloud-Provider bereits Lösungen, um deren Produkt dort verwenden zu können. Dazu gibt es innerhalb der Ansible Dokumentation einen eigenen Abschnitt, in dem die jeweiligen Provider, die unterstützt werden, aufgelistet sind.

[26] http://saltstack.com/wp-content/uploads/2016/08/SaltStack-Supported-Operating-Systems.pdf

[27] Für detailliertere Informationen kann unter der folgenden Quelle genauer nachgelesen werden. https://blog.docker.com/2016/05/docker-unikernels-open-source/ , Stand 15.11.2016

Für den Bereich der Cloud gibt es zusätzlich die Möglichkeit, die sogenannten Cloud Module zu verwenden[28].

Mit SaltStack ist ebenfalls ein großes Portfolio für die Cloud Verwendung vorhanden. Möchte man also SaltStack zur Provisionierung einer oder mehrerer Clouds nutzen, ist dies über Salt Cloud durchführbar.

Als letztes ist noch Docker als Vertreter zur Anwendung in der Cloud übrig. Anders als bei Ansible und SaltStack wird bei Docker im Abschnitt der Cloud nicht explizit aufgeführt, welche Cloud-Anbieter bisher mit deren Technologie angesprochen werden können. Beschrieben wird nur, dass für die Cloud alle Betriebssysteme unterstützt werden, die auch einen Support bei Docker haben. Damit sind Linux, Windows und Mac Systeme einbezogen auf denen auch standardmäßig Docker verwendet werden kann[29].

Während die Software-Anbieter bisher einige Systeme verschiedener Cloud-Provider in ihrem Produkt aufgenommen haben, wird die Ausweitung in dem Cloud Areal weiter ansteigen. Mittlerweile ist der Aufschwung dieses Gebietes auf einem Level, bei dem das Ringen um möglichst viele Kunden für große Unternehmen teilweise ausschlaggebend ist.

[28] http://docs.ansible.com/ansible/list_of_cloud_modules.html
[29] https://docs.docker.com/engine/installation/

### 3.3.3 Nichtfunktionale Anforderungen

Nichtfunktionale Anforderungen tauchen in der Informatik immer wieder auf. Wenn eine Software entwickelt werden soll heißt es oftmals von den Kunden, dass die Software schnell, hoch verfügbar, sehr sicher oder gut skalierbar sein muss. Damit man diese Anforderungen erfüllen kann, muss man genauer spezifizieren, was speziell mit diesen Anforderungen gemeint ist und wie sie erreicht werden können. Nachdem bereits die abzuwägenden Faktoren aus *Kapitel 3.2 Abzuwägende Faktoren* aufgezählt und vertieft wurden werden anschließend die nichtfunktionalen Anforderungen an Infrastructure-as-Code analysiert.

#### 3.3.3.1. Sicherheit

Den Anfang der nichtfunktionalen Anforderungen macht der Punkt Sicherheit. Aufgrund der Tatsache, dass die Technik ziemlich schnell angetrieben wird, benötigt man für den Schutz vertraulicher Daten verschiedenste Sicherheitsmechanismen. Als erstes sollte festgestellt werden, was bei Informationssystemen gesichert werden soll, weswegen nun denkbare Punkte zum Sichern bei Infrastructure-as-Code genannt werden.

Zu einer allgemeinen Sicherheit gehört es natürlich auch hier, die Daten der Nutzer zu schützen. Damit personenbezogene Daten wie Passwörter, Nutzernamen, E-Mail-Adressen o.ä. geschützt werden können, muss man mehrere Stellen der Provisionierungs-Technologien absichern. Anschaulich denken viele Anwender wahrscheinlich schnell an das System, welches durch IaC entstehen soll. Dieses muss natürlich auch gegen Angriffe von außen verteidigt werden. Dabei sollten die Daten auf den Systemen vor unerlaubten Zugriffen geschützt sein. Weiterhin muss bestmöglich verhindert werden, dass Viren die Systeme befallen können, denn ansonsten könnten Angreifer diese für kriminelle Zwecke missbrauchen, was wiederum verhindert werden soll. Ein weiterer Punkt wäre auch der Schutz vor Spionage bei Unternehmen oder Ländern. Diese Fälle sind allein leider etwas allgemein gehalten. Eine detailliertere Sichtweise liefert hier womöglich der Blick auf den Weg, der zwischen der Kontrollmaschine und der zu konfigurierenden

Maschine liegt. Während eines Provisionierungs-Prozesses muss explizit verhindert werden, dass ein Angreifer in den Kommunikationsweg eindringt und somit schon Daten abgreifen kann, bevor das System fertig konfiguriert wurde[30]. Über verschiedene weitere Sicherheitsaspekte kann definitiv fortführend geschrieben werden, denn sie sind ein besonders wichtiger Bestandteil der Informatik. Ziel dieser Beispiele ist es aber, einen Einblick in die nichtfunktionalen Anforderungen bei IaC zu bekommen und nicht die Einsicht in die IT-Sicherheit.

Anknüpfend an die obigen Sicherheits-Voraussetzungen ist nun trotzdem ein sehr positiver Faktor der Infrastructure-as-Code Technologien zu nennen. Und zwar erfüllt ein System, das durch einen Infrastruktur-Code erstellt wurde, die aus dem Skript definierten Sicherheitskriterien und ist daher generell erst einmal sicher. Ausschlaggebend ist hierbei jetzt, dass jedes weitere System, das mit diesem Skript konfiguriert wird, die gleichen Sicherheitskriterien erfüllt [vgl. Morris 2016, S. 298 Absatz Security]. Deshalb kann mit Infrastructure-as-Code binnen kurzer Zeit ein gut gesichertes Netzwerk zusammengestellt werden, welches auch noch die folgenden Eigenschaften besitzt:

- Reproduzierbar
- Änderungen sind auf allen Systemen gleich
- Schnelle Konfiguration abgesicherter Systeme

#### 3.3.3.2. Geschwindigkeit

Mit der Eigenschaft der Geschwindigkeit wird häufig ein Kriterium aufgestellt, welches enorm wichtig für die heutige Gesellschaft ist. Heutzutage verlangen Anwender von Software und vielen anderen Technologien, dass diese eine sehr schnell Antwortzeit besitzen. Zu sehen ist dies an dem Beispiel des Internets. Wenn bestimmte Internetseiten wie Amazon oder Google nicht im Verlauf kürzester Zeit ein Resultat liefern, so wird dies schnell und immer häufiger als störend empfunden.

[30] **Man-in-the-Middle-Angriff:**
https://www.bsi.bund.de/DE/Themen/ITGrundschutz/ITGrundschutzKataloge/Inhalt/_content/g/g05/g05143.html

Je nach Anwendungsgebiet werden abweichende Performanceanforderungen branchenabhängig definiert. Zum einen gibt es Startup-Unternehmen, die zu Beginn der Softwareentwicklung ihr Hauptaugenmerk auf das Herausbringen neuer Produkte richten. Sie benötigen für diesen Einstieg ein Grundgerüst, damit die Erstellung einer Infrastruktur zügig ablaufen kann. Infrastructure-as-Code bietet für genau diesen Bereich den richtigen Faktor an Schnelligkeit, indem durch die Ausführung des Programmes entwicklungsfähige Umgebungen auf Knopfdruck bereitgestellt werden. Mit den Startups gleichzusetzen sind auch neue Projekte, die in einem Unternehmen zustande kommen. Sie setzen mit großer Wahrscheinlichkeit auf ein Basis Produkt, bei dem der bereits eine Infrastruktur vorhanden ist. Damit eine unabhängige Entwicklung mit deren Tests etc. stattfinden kann muss auch hier eine neu aufzusetzende Infrastruktur konfiguriert werden. Zum anderen gibt es Vorgänge in Unternehmen, bei denen es einen geringeren Geschwindigkeitsschub bedarf. Dazu zählen beispielsweise Firmen, die in ihrer Infrastruktur einen großen Umbruch wagen wollen. Dort ist der Prozess der sorgfältigen Analyse von größerer Bedeutung, da entstehende Risiken hierbei möglichst vermieden bzw. minimiert werden sollen. Die eigentliche Umsetzung erfolgt aus diesem Grund je nach Größe der Umstrukturierung langsamer, weswegen eine Bereitstellung der gebrauchten Systeme abhängig zum Fortschritt der Entwicklung ist. Weiterhin ist auch die Bereitstellung von geforderten Systemen während des laufenden Betriebs gegeben und darf nicht außer Acht gelassen werden. Diese Art der Bereitstellung erhöht die Performance in diesem Fall dadurch, indem Entwickler zum Entwickeln und Testen weniger Zeit beanspruchen müssen und somit der gesamte Entwicklungszyklus beschleunigt werden kann. Abschließend ist hierzu jedoch zu sagen, dass der zeitliche Faktor innerhalb der laufenden Entwicklung nicht so kritisch betrachtet werden muss. Dies lässt sich damit begründen, dass bei der Einrichtung reichlicher Systeme, die bei der Einrichtung einen erhöhten Zeitbedarf besitzen, auch nicht reguläre Arbeitszeiten genutzt werden können.

#### 3.3.3.3. Skalierbarkeit

Eine andere Anforderung stellt die Skalierbarkeit dar. Viele Systeme behandeln heutzutage diverse komplexe Anwendungsfälle, sodass die Fokussierung der Performance problematisch wird. Entgegenwirkend kann man mithilfe der Skalierung Anwendungsfälle besser abarbeiten bzw. vorausschauend Vorbereitungen treffen, was für Ressourcen man benötigen wird. Realisiert werden kann die Skalierung durch die horizontale Skalierung (scale out) oder die vertikale Skalierung (scale up). Interessant für die zwei Techniken ist, welche Lösungen die Vertreter der Provisionierungs-Technologien anbieten und welches Tool sich für welche Art der Skalierung besser eignet. Dafür wird in den nächsten zwei Abschnitten wieder auf die bekannten Tools zurückgegriffen.

##### Horizontale Skalierung

Bei der horizontalen Skalierung (scale out) hat man das Ziel, eine möglichst große Anzahl an Hardware Maschinen parallel zu betreiben. Aufgrund der noch kaum vorhandenen Kapazitätsbegrenzung von Hardware ist diese Art und Weise der Skalierung in der heutigen Zeit selten noch ein Problem. Begründet ist das durch die vergleichsweise geringen Kosten der Hardware von heute im Vergleich zu früher.

*SaltStack*

Beginnend wird das Tool SaltStack betrachtet. SaltStack verspricht, dass bis zu mehrere tausende Maschinen innerhalb von Sekunden konfiguriert werden können. Ein derartiges Versprechen klingt generell für eine Skalierung in die Horizontale ziemlich gut, da hierbei viele Maschinen existieren können. Blickt man aber darauf, dass zur Provisionierung auf jeder Maschine bei SaltStack ein Minion installiert werden muss, wird aus dem vorher guten Szenario ein aufwändiges Installieren der vorausgesetzten Salt-Minions. Nimmt man daraufhin jedoch an, dass die Maschinen bereits als Minion konfiguriert sind, ist eine horizontale Skalierung wegen der SaltStack Architektur gut umzusetzen. Bei Bedarf kann hierzu ergänzend die Grafik der Salt-Provisionierung in *Kapitel 2.2.2 SaltStack* angesehen werden, da es sich dort um eine horizontale Darstellung handelt.

### *Ansible*

Das zweite Tool innerhalb der Betrachtung einer horizontalen Skalierung ist auch hier Ansible. Blickt man nun auf den bisherigen Inhalt dieser Arbeit zurück, lässt sich erkennen, dass die Skalierung durch diese Software sehr gut umgesetzt werden kann. Zu sehen ist dies daran, dass für eine Konfiguration beispielsweise alle anzusprechenden Host Systeme in einem Playbook definiert werden, woraufhin dann die Konfiguration von einem einzigen Rechner gestartet werden kann. Vorteilhaft ist hier natürlich der agentenlose Architekturstil, wodurch eine Installation vorausgesetzter Systeme wie bei SaltStack vermieden wird. Vergessen werden darf aber nicht, dass hier eine Einrichtung von SSH benötigt wird. Ebenso wie bei SaltStack kann zur Visualisierung des Ganzen die Grafik aus dem *Kapitel 2.2.1 Ansible* verwendet werden, da dort ebenfalls eine horizontale Darstellung zu sehen ist.

### *Docker*

Als abschließende Technologie für das Thema der horizontalen Skalierung bleibt nun noch das dritte der drei Tools, Docker. Bei Docker ist die Skalierung durch verschiedene Vorgehensweisen umgesetzt worden. Damit das Grundgerüst für die Verwendung aber erst vorhanden ist, muss hier auf jedem der physikalischen Systeme das für Docker benötigte Environment eingerichtet werden. Diese Einrichtung kann, wie bei den Minions von SaltStack, auch durch ein anderes Automatisierungs-Tool geschehen. Danach können durch Docker zwei verschiedene Szenarien angewendet werden. Das erste wäre hierbei eine Einzelinstallation des Docker-Daemons etc. auf jeder Maschine, die dann bestimmte Services dort bereitstellt. Die zweite Möglichkeit ist die Verwendung von Docker-Swarm übergreifend auf mehrere Maschinen. Nachdem das Grundgerüst für Docker auf den einzelnen Maschinen vorhanden ist muss es aus dem Docker-Swarm Kontext gesehen eine Swarm-Master Maschine geben, die für die Verwaltung der ihm angehörenden Swarm-Agents Maschinen bestimmt ist. Die dadurch entstehende Skalierung ist aufgrund der speziellen Architektur und Umsetzung jedoch stets noch anders zu betrachten als die der eigentlichen Provisionierungs-Technologien.

### Vertikale Skalierung

Mit der vertikalen Skalierung (scale up) ist der genaue Gegensatz zu dem vorherigen Beispiel der horizontalen Skalierung gegeben. Wenn man nach diesem Prinzip vorgeht, hat man anstatt vieler kleiner Maschinen ein großes Supersystem, welches immer mehr Leistung bekommt, indem die dafür benötigten Komponenten darin verbaut werden.

Innerhalb der vertikalen Skalierung sind die Unterschiede zwischen den einzelnen Technologien eher gering. Auch hier wird es sicherlich die einen oder anderen Schwierigkeiten und Probleme geben, die man bei einer Provisionierung von horizontalen Systemen gleichermaßen beachten muss. Die Unterscheidung ist jedoch dadurch erkennbar, dass bei einem großen Netzwerk mehrere Faktoren wie z.B. Netzwerkkommunikation oder Tool spezifische Abhängigkeiten beachtet werden müssen. Diese Faktoren fallen bei einem einzigen großen Gerät währenddessen anders aus oder sind ganz und gar anders. Das Prinzip bei diesem Vorgehen ist im Wesentlichen Aufbau bei den Tools das gleiche. Es soll jeweils mehr Funktionalität in einer Maschine untergebracht werden, was bei den vorhandenen Tools für Infrastructure-as-Code entweder leicht oder schwierig umzusetzen ist. Um die Einrichtung von mehr Funktionalität in einem einzigen System zu realisieren, benötigt man viele zusätzliche Konfigurationsschritte, die bei Konfigurationen von mehreren alleinstehenden Systemen nicht umgesetzt werden müssen (Vergabe von Ports für verschiedene Services).

Bei SaltStack und Ansible ist nur deren Umsetzung der Provisionierung von Systemen zu beachten, da wie vorher beschrieben, hier die Komplexität unterschiedlich stark ausfallen kann. Bei Docker hingegen ist die Verwirklichung durch den Docker-Swarm eine elegante Lösung für die vertikale Skalierung, da der Master hier mit integriert ist und somit die Verwaltung von Master & Agent in einem System geschieht.

#### 3.3.3.4. Verfügbarkeit

Sieht man sich den aktuellen Stand der Technik an, wird sehr schnell deutlich, dass die Verfügbarkeit heutiger Systeme enorm wichtig ist. Beispielhaft kann hier erneut Google genannt werden, denn dessen Geschäft lebt von der Verfügbarkeit. Was das nun aber mit Infrastructure-as-Code zu tun hat, soll dieses Kapitel über die Verfügbarkeit bezogen auf die Infrastruktur zeigen.

Betrachtet man eine Infrastruktur in einem Software Unternehmen, gibt es dort in der Regel Continuous Integration bzw. Continuous Delivery. Das Continuous Integration System beinhaltet Build Systeme, Test Systeme und auch die Software Code Repositories. Diese Services sollten für die Entwickler während der Kernarbeitszeiten auf jeden Fall zur Verfügung stehen. Da hierbei ebenfalls Daten auf den genannten Systemen liegen, muss sichergestellt werden, dass die Verfügbarkeit der Daten auch gesichert ist. Das gesamte CI-System muss daher ausfallsicher sein und mit Störungen im laufenden Betrieb zurechtkommen. Die Daten gehören zwar auch in diesen Bereich, jedoch wird dies durch Replikation[31] o.ä. gewährleistet, was wiederum hier nicht weiter behandelt wird. Speziell ist im Bereich der Infrastrukturen, dass sie häufig geändert werden, wobei die Änderungen oftmals Updates sind, oder Daten bzw. Ressourcen die hinzugefügt, geändert oder entfernt werden [vgl. Morris 2016, S. 275]. Gezielte Wartungen sollten jedoch unbedingt separat bedacht werden, da diese Auszeiten keine richtigen Ausfälle des Systems sind. Dabei werden immerhin lediglich Updates, Verbesserungen oder Neuerungen eingespielt. Systemausfälle wie diese werden außerdem nicht in die Betrachtung mit einbezogen, da es sich hierbei um geplante Vorgänge handelt und somit betroffene Nutzer rechtzeitig informiert werden können. Des Weiteren wird die Verfügbarkeit durch IaC ein bisschen aufgeweicht, da betroffene Systeme schnell neu erzeugt werden können und somit die Wartung und das weitere Arbeiten der Mitarbeiter parallel geschehen kann [vgl. Morris 2016, S. 276-277].

[31] Replikation beschreibt einen Verfahren, bei denen Daten mehrfach auf verschiedenen Systemen abgespeichert werden, um für deren Verfügbarkeit zu garantieren.

### 3.3.4 Anwendungsgebiete

Abgeschlossen wird das Kapitel Herausforderungen von Infrastructure-as-Code mit einigen Anwendungsgebieten, die für diese Art Technologie möglich sind und bereits unterstützt oder noch einbezogen werden können.

#### 3.3.4.1. Continuous Integration

Das Anwendungsgebiet von Continuous-Integration wurde innerhalb dieser Arbeit bereits des Öfteren angesprochen und ist, wie im Titel zu sehen, auch ein Bestandteil davon. Der Beitrag von Continuous Integration gehört mittlerweile zum Grund Repertoire einer erfolgreichen Softwareentwicklung. Deswegen ist es besonders attraktiv für CI Installationen, wenn diese schnell und einfach eingerichtet, sowie deren Services fix genutzt werden können. Für bisherige Zwecke wurden die benötigten Continuous Integration Tools mit IaC eingerichtet, woraufhin dann die definierten Prozesse für das CI umgesetzt wurden.

Aus der Sichtweise von Infrastructure-as-Code wird die Unterstützung durch das Continuous Integration nun noch deutlich verbessert. Dort kann an der bisherigen Funktionalität angeknüpft werden, indem durch die Definition Files automatisiert CI Umgebungen entstehen. Dadurch können Systeme, die dafür bisher Ressourcen dauerhaft belastet haben flexibel eingerichtet werden. Deswegen ist es möglich, das Testen oder Bauen von Software durchzuführen, wenn dies nötig ist, wobei die für diesen Zeitraum bereitgestellten Systeme nach Beendigung ihrer Aufgaben gezielt entfernt werden können, um Ressourcen nicht durchgehend belasten zu müssen. Beachtet werden muss aber unbedingt, dass eine Integration der Infrastructure-as-Code Tools in der verwendeten Continuous Integration Software auch umgesetzt ist. Das bedeutet, dass es möglich sein sollte, eine Konfiguration aus dem IaC Tool selbst heraus durchzuführen um beispielsweise einen Jenkins einzurichten.

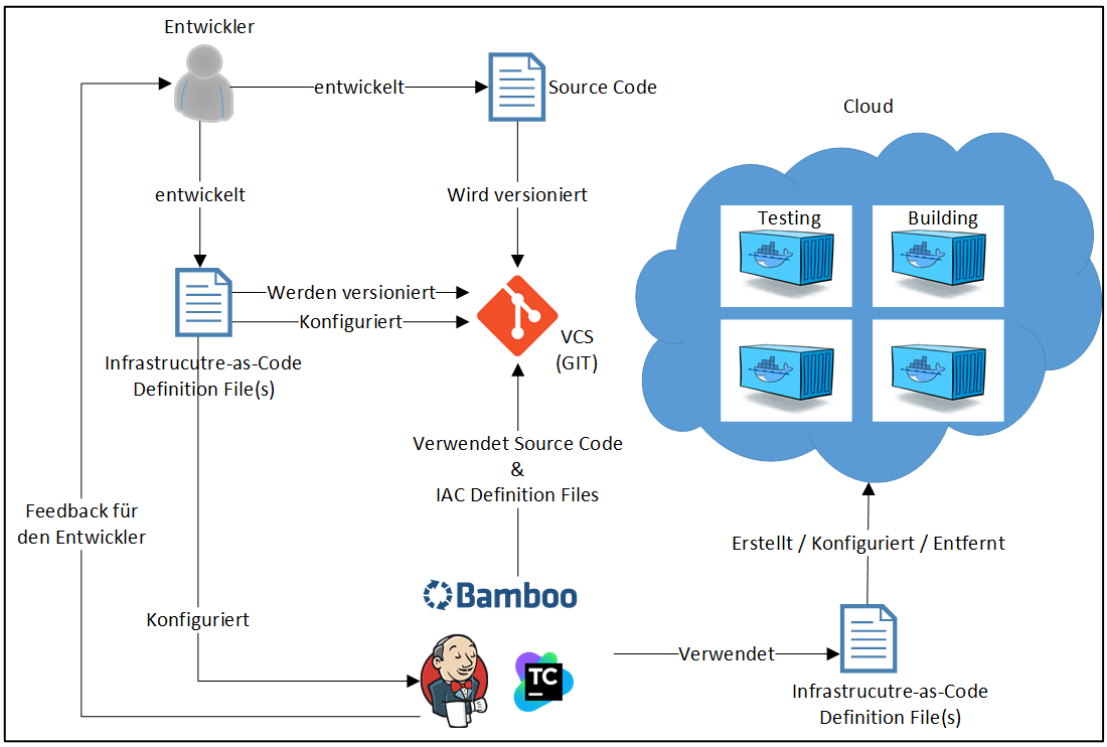


Abbildung 72 Continuous Integration & Infrastructure-as-Code[32]
Vgl. http://www.oracle.com/technetwork/articles/java/deployment-illo-2228199.jpg

### 3.3.4.2. Continuous Delivery

Anknüpfend an das Thema Continuous Integration ist die weiterentwickelte Form des Continuous Delivery (auch CD genannt) zu nennen, welche ebenfalls gut durch die Verwendung von Infrastructure-as-Code zu realisieren ist. Da es sich hierbei um eine Erweiterung des CI handelt, werden lediglich weitere Schritte in den Zyklus eingefügt.

Ergänzend zu dem bereits bestehenden Zyklus werden hier die Schritte Deployment und Release eingeführt. Die Integration dieser Zusätze in den Continuous Integration Zyklus bewirkt, dass der Continuous Delivery Kreislauf entsteht. Mithilfe des neu entstandenen Kreislaufes verspricht man sich das bislang gute Feedback durch die CI Unterstützung noch deutlich zu verbessern. Diese Verbesserung wird

[32] Quellennachweise zu den Bildern befinden sich aus Gründen der Übersicht im Quellenverzeichnis.

erreicht, indem nach jedem Schritt des CD Kreislaufes der Entwickler Feedback über den aktuellen Status bekommt. Zur Veranschaulichung kann der Zyklus in der folgenden Grafik betrachtet werden.

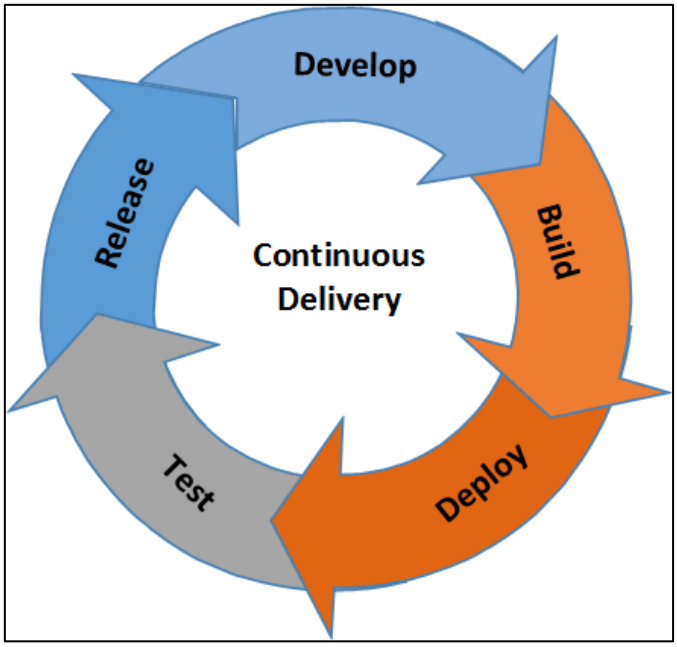


Abbildung 8 Continuous Delivery Zyklus

Nimmt man bei CD die Abbildung aus dem vorherigen Kapitel zur Hilfe, wird bei der Kombination von IaC und CD der Prozess von Deployment beispielsweise zu den Test- und Build Servern gruppiert. Damit hätte man zusätzlich Deployment Server, die einen weiteren Teil des Continuous Delivery erledigen. Auch hier gilt, dass nach Beendigung des Service dieser entfernt werden kann, wenn das aus dem Schritt resultierende Feedback an die Entwickler übermittelt wurde. Nachdem das Deployment abgeschlossen ist, müssen noch einmal die komplexeren Tests der Software ausgeführt werden. Wenn nach der Ausführung keine Fehler auftreten, ist das Produkt bereit zum Release.

#### 3.3.4.3. Entwicklungsumgebungen

Zum Schluss folgt ein Anwendungsgebiet, welches in erster Linie nicht in die bisher beschriebene Art der Infrastruktur passt. Mit Entwicklungsumgebungen ist nicht nur ein Entwicklungstool IDE gemeint, sondern ein komplettes Development-Paket für Entwickler.

Oftmals ist man in einem Unternehmen als neuer Mitarbeiter dem allgemeinen Installationsprozedere ausgesetzt, welches Voraussetzung für den eigentlichen Beginn der Arbeit ist. Je nach Anzahl an benötigten Tools etc. kann dies schon mal von wenigen Stunden bis zu mehreren Tagen dauern, damit auch alles funktionsfähig ist. Darum ist für ein solches Anwendungsszenario eine Automatisierung ebenso hilfreich und denkbar wie auch für andere Infrastrukturelemente. Anfänglich muss nachgesehen werden, welches Betriebssystem innerhalb des Unternehmens verwendet wird, woraufhin man die Wahl des Automatisierungs-Tools treffen kann (siehe *Kapitel 3.3.2 Plattformabhängigkeiten* für weiteres), um Kompatibilitätsproblemen zu entgehen. Danach kann die gesamte Entwicklungsumgebung entweder auf dem Host OS (wenn möglich) oder mithilfe einer virtuellen Maschine eingerichtet werden. Hierbei enthalten sind dann alle notwendigen Tools, Frameworks und Konfigurationen, die erforderlich sind, um mit der Entwicklung zu beginnen. Abzuwägen sind natürlich auch die Möglichkeiten des Tools zur Provisionierung in diesem Bereich bezüglich der erreichbaren Tiefe der Konfiguration. Wenn diese Basis Installation eingerichtet wurde, erspart man sich vielleicht die ein oder andere Installation. Nicht auszuschließen ist aber, dass man selbstständig ein paar Änderungen an speziellen Tools der Entwicklungsumgebung durchführen muss, welche durch die Provisionierungs-Technologie nicht ermöglicht sind. Dieser Prozess der Installation ist für so manche Basis Installationen ein guter Kompromiss.

# 4 Umsetzung der Continuous Integration Pipeline

In diesem Kapitel wird die Umsetzung der Continuous Integration Pipeline auf Basis des bisher gesammelten Wissens beschrieben. Hierbei wird als erstes darauf eingegangen, welche Voraussetzungen vorhanden sind und wie das Grundgerüst für die Entwicklung aussieht. Danach folgt eine detaillierte Beschreibung der Entwicklung der ersten drei Prototypen, woraufhin eine Erläuterung der entwickelten Pipeline durch jedes Tool folgt. Abgeschlossen wird dieses Kapitel dann damit, dass die Automatisierungen der Continuous Integration Pipeline durch Ansible, SaltStack und Docker miteinander verglichen werden und somit ein Fazit gezogen werden kann. Dieses Kapitel beschreibt die Umsetzung des praktischen Anteils aus einer technisch distanzierten Sichtweise, um zu vermeiden, dass der Anteil der technischen Befehle den Lesefluss einschränkt. Besteht hier aber das Interesse eine technisch detaillierte Anleitung zu lesen, kann dies mithilfe der im Anhang beigefügten Anleitung gemacht werden.

## 4.1 Das Grundgerüst

Damit die Entwicklung der Pipeline durch die verschiedenen Tools gestartet werden kann, muss eine homogene Umgebung eingerichtet werden, damit die Voraussetzung gegeben ist, dass alle Tools den gleichen Bedingungen ausgesetzt sind. Dazu werden kurz die technischen Angaben angegeben, die zur Bearbeitung der Aufgabe vorhanden waren.

### 4.1.1 Technische Details (Systemvoraussetzungen)

Vorab folgen hier kurz die benötigten Systemvoraussetzungen, welche zur Realisierung der Prototypen sowie der Pipeline verwendet wurden. Das Entwicklungssystem, das zur Umsetzung verwendet wurde, ist ein Windows 7 64 Bit-Betriebssystem System mit 32 GB RAM, einem Intel Core i7 mit 3.60GHz und einer SSD. Die Angabe dieser Rahmenbedingungen dient allein dem Zweck einer Reproduzierung der CI-Pipeline unter ähnlichen oder identischen Bedingungen. Für diese Umsetzung reichen standardmäßig auch Komponenten, die weniger Leistung besitzen. Bezüglich der RAM Größe ist aus eigenen Erfahrungen jedoch mindestens eine Kapazität von 8 GB empfehlenswert.

### 4.1.2 Aufbau der Umgebung

#### 4.1.2.1. Vorbereitung

Nachdem in *Kapitel 4.1.1 Technische Details (Systemvoraussetzungen)* technische Angaben zum verwendeten System gemacht wurden, kommt der Aufbau der verwendeten Umgebung zum Entwickeln der der jeweiligen Pipelines. Damit die drei gewählten Tools unter den gleichen Bedingungen verwendet werden können, muss hierzu eine Umgebung eingerichtet werden, die die jeweiligen Technologien auch unterstützen. Aufgrund der in *Kapitel 3.3.2 Plattformabhängigkeiten* dargestellten Plattformabhängigkeiten ist es am sinnvollsten, das Betriebssystem Linux zu verwenden. Gründe dafür werden ebenfalls in dem *Kapitel 3.3.2 Plattformabhängigkeiten* genannt, weswegen die Kontrollmaschine (Administrator-Maschine, Master) sowie die zu konfigurierende Maschine aus Linux bestehen sollten. Da innerhalb der Firma Werum das Linux Betriebssystem CentOS verwendet wird, wurde aus dem Pool der verschiedenen Linux Systeme das bei Werum eingesetzte gewählt.

Weiterhin wird für die Umsetzung der Pipeline innerhalb des Host Systems auf virtuelle Maschinen zurückgegriffen, um die Verwendung kostenverursachender Ressourcen zu vermeiden. Diese VMs werden mithilfe der Software von Vagrant

eingerichtet und in Kombination mit Virtualbox in der Version 5.0.24 verwendet. Vagrant wurde für die Einrichtung der virtuellen Maschinen ausgewählt, weil die Handhabung durch diese Toolunterstützung beachtlich vereinfacht wird, was aber nicht bedeuten soll, dass es nicht sehr komplex werden kann.

#### 4.1.2.2. Konfiguration von Vagrant

Angesichts der Tatsache, dass innerhalb der Einleitung kein separates Kapitel zu Vagrant vorhanden ist wird es hier eine grundlegende Einführung für die Konfiguration der Vagrant Boxen geben. Wenn unbekannte Begrifflichkeiten aus dem Vagrant Umfeld dabei auftauchen sollten, wird hierzu auf weiterführende Informationen in der Fußzeile verwiesen.

Als erstes muss die Software von Vagrant mit der Version 1.8.7 installiert werden, da diese Version von Vagrant für die Umsetzung in dieser Arbeit verwendet wurde. Während der Installation kann bereits der Download und die Installation von Virtualbox in der Version 5.0.24 gestartet werden. Virtualbox muss extra installiert werden, weil es leider nicht in der Installation von Vagrant mit vorhanden ist. Nachdem die beiden Installationen beendet wurden, kann der nächste Schritt durchgeführt werden, bei dem zwei Verzeichnisse für die benötigten Vagrant Boxen erstellt werden müssen. Diese Verzeichnisse haben keinen vorgegebenen Pfad und können deswegen an beliebiger Stelle erstellt werden (bestimmte Verzeichnisse wie z.B. Systemverzeichnisse sollten trotzdem nicht genutzt werden). Erleichternd für die Verwaltung der Boxen sollten bereits bei der Erstellung der Verzeichnisse sprechende Namen verwendet werden, da zwischen der administrativen VM und der zu konfigurierenden VM unterschieden wird. Wenn die zwei oberen Schritte beendet wurden, kann damit begonnen werden, die Vagrant Umgebung zu konfigurieren.

Zuerst wird sich um die zu konfigurierende Vagrant Box gekümmert, indem über die Kommandozeile in das entsprechende Verzeichnis gewechselt wird. Befindet man sich nun in dem Verzeichnis, muss Vagrant mit dem entsprechenden Kommando initialisiert werden. Hat man diese Initialisierung durchgeführt, sollte sich

anschließend in dem Verzeichnis ein sogenanntes Vagrantfile befinden. Danach muss eine Konfiguration des Vagrantfile vorgenommen werden, bei der man das gewünschte Betriebssystem der VM angibt, was für das Starten der Box erst einmal auch ausreichend ist. Infolgedessen, dass sich die gesamte Entwicklungsumgebung[33] auf einem Host System befindet, muss zudem noch konfiguriert werden, dass es sich bei dem Projekt um ein Host-only Netzwerk handelt. Die Konfiguration dieser Eigenschaft ist für diese Arbeit enorm wichtig, da eine Kommunikation der virtuellen Maschinen auf einem Host System nur funktioniert, wenn jede ihre eigene IP-Adresse innerhalb des Host-only Netzwerks zugewiesen bekommt. Sind die eben genannten Schritte ausgeführt, kann die Vagrant Box, welche provisioniert werden soll, hochgefahren werden. Möchte man nun in die VM hereinschauen, kann dies über die Vagrant SSH Funktionalität geschehen und man befindet sich (zumindest in diesem Beispiel) auf einer Box mit einem installierten Linux CentOS.

Wenn die obere Anleitung korrekt befolgt wurde und ein Login in die Box möglich ist, so kann die Vagrant Box, welche den administrativen Teil übernehmen soll auf die gleiche Art und Weise eingerichtet werden. Zusätzlich sollte in dieser Maschine jedoch zu Beginn das Updaten der Packages des Linux Systems durchgeführt werden, um sicher zu gehen, dass alles auf dem neusten Stand ist. Warum die Updates auf der vorher genannten Vagrant Box nicht gemacht wurden, wird im weiteren Verlauf erläutert.

[33] Mit dieser Entwicklungsumgebung ist keine IDE gemeint, sondern das komplette Entwicklungsareal für die Entwicklung (Host Maschine + virtuelle Maschinen).

# 4.2 Die Prototypen

Jetzt wird an die Einrichtung der Entwicklungsumgebung aus *Kapitel 4.1.2.2 Konfiguration von Vagrant* mit der prototypischen Entwicklung durch jedes der drei Tools angeknüpft. Das Ziel jedes Prototyps ist die erfolgreiche Installation des Jenkins auf der zu konfigurierenden VM sowie das Starten des Service.

## 4.2.1 Prototypentwicklung mit Ansible

Die in diesem Abschnitt beschriebene prototypische Entwicklung mit Ansible setzt voraus, dass man die zwei Vagrant Boxen aus dem vorherigen *Kapitel 4.1.2.2 Konfiguration von Vagrant* erfolgreich eingerichtet wurden. Wenn das der Fall ist, kann zu Beginn die administrative VM per Kommandozeile hochgefahren werden, woraufhin man sich dann auch in die Maschine einwählen kann. Hat man nun die vorhandene VM noch nicht bezüglich ihrer Packages aktualisiert, so sollte man diesen Schritt als erstes durchführen, sobald man den Zugriff auf das System hat. Anschließend kann die Software von Ansible installiert werden. Für die gewählte Installation dieser Arbeit müssen vorher jedoch die Extra Packages for Enterprise Linux (EPEL)[34] installiert werden, da diese als Grundlage der Installation sind.

Bevor der eigentliche Prototyp weiter beschrieben wird, folgt eine vorgezogene Information zur weiteren Entwicklung. Versucht man nun innerhalb dieser speziellen Umgebung Befehle von Ansible auszuführen, um einen Test Befehl wie Ping auszuführen, werden SSH Fehler auftreten, die behoben werden müssen. Deswegen muss für dieses Szenario eine SSH Konfiguration durchgeführt werden. Angewendet wurde hier die manuelle Konfiguration von SSH aufgrund dessen, da nur eine Ziel VM existiert. Bei der Konfiguration von SSH wird ein Schlüsselpaar generiert, dass aus einem öffentlichen Schlüssel und einem privaten Schlüssel besteht. Sobald die Generierung des Schlüsselpaares abgeschlossen ist muss der öffentliche Schlüssel auf das System kopiert werden, auf welches man per SSH zugreifen möchte. Erst nach diesem Kopiervorgang ist es möglich sich von einem

[34] https://fedoraproject.org/wiki/EPEL

System aus mit dem Zielsystem zu verbinden. Besonders wichtig ist bei diesem Kopiervorgang, dass man darauf achten muss, mit welchen Benutzerrechten man die Generierung der Schlüssel durchführt. Hierzu folgen bei der Beschreibung der durch Ansible eingerichteten Pipeline genauere Informationen (siehe Skript im Anhang). Der hier gewählte manuelle Vorgang kann durch gewisse Kommandos auch auf eine andere Art und Weise realisiert werden. Spezifischere Details zu SSH werden nicht aufgeführt, da Sicherheitskonzepte im Allgemeinen kein Teil der Arbeit sind.

Jetzt ist es möglich die ersten Ansible Befehle über die Kommandozeile auszuführen, die auch schon Provisionierungen übernehmen können. Damit dies aber etwas eleganter umgesetzt ist, schreibt man hierfür nun ein Ansible Playbook. In diesem Playbook werden nun die benötigten Ressourcen für die zu provisionierende VM aufgelistet. Die für die Provisionierung des Jenkins vorausgesetzten Ressourcen für das Playbook werden im folgenden Teil aufgelistet und beschrieben.

Damit der Continuous Integration Server Jenkins verwendet werden kann muss zuerst Java installiert werden, da dies eine explizite Voraussetzung ist. Danach müssen auf dem Zielsystem die EPEL installiert werden um dann die Installation des Jenkins durchführen zu können. Dazu wird das Jenkins Repository dem System hinzugefügt, woraufhin dann die Jenkins Installation und das Starten des Service fertig gestellt werden können. Zum Schluss sollte es möglich sein mithilfe des Playbooks die zu konfigurierende Maschine zu provisionieren, was danach durch den Status des Jenkins Service überprüft werden kann.

### 4.2.2 Prototypenentwicklung mit SaltStack

Der zweite Prototyp, der entwickelt wurde ist mit SaltStack erstellt worden. Um hier, wie auch bei dem Ansible Prototypen, ein gleiches Umfeld zu schaffen wird wieder die Vagrant Umgebung genutzt. Die administrative Maschine muss hierbei nicht unbedingt entfernt und neu eingerichtet werden, da lediglich eine andere Software verwendet wird. Zugleich ist es aber auch ohne weiteres möglich, auch

diese Vagrant Box erneut von Grund auf bereitzustellen. Die zu konfigurierende Vagrant Box hingegen muss auf jeden Fall entfernt und komplett neu bereitgestellt werden, damit dort die gleichen Voraussetzungen wie auch in *Kapitel 4.2.1 Prototypentwicklung mit Ansible* bei Ansible vorhanden sind.

Zu Beginn benötigt man auch hier den Zugriff auf die jeweiligen Maschinen. Bei SaltStack muss beachtet werden, dass beide Vagrant Boxen eingerichtet werden müssen. Dabei wird die administrative Vagrant Box zuerst betrachtet. Damit man SaltStack auf dieser virtuellen Maschine nutzen kann, muss hier als erstes das offizielle SaltStack Repository hinzugefügt werden. Folgend kann dann auf diesem System der Salt-Master installiert werden[35]. Ist der Salt-Master ordnungsgemäß installiert, kann die richtige Konfiguration beginnen. Dazu muss beim Master erst einmal nur ein Pfad für den Salt Workspace innerhalb der Konfigurationsdatei eingefügt werden. Anschließend startet man den Salt-Master neu und beginnt parallel nun die Salt-Minion Konfiguration, weil diese für das weitere Vorgehen wichtig ist. Wie gerade eben schon beschrieben, muss man die Schritte der Installation von Salt mittels des Repositories auch auf der Minion VM durchgeführt werden. Der Unterschied hierbei ist, dass nicht der Salt-Master installiert wird, sondern der Salt-Minion. Die fortzusetzende Konfiguration ist beim Minion jedoch sehr kurz gehalten. Dort muss nur die Konfigurationsdatei des Minion so angepasst werden, dass bei Master die IP-Adresse des Salt-Masters eingetragen wird. Anschließend muss der Minion nur noch neu gestartet werden. Jetzt versucht der Salt-Minion sich mit seinem Salt-Master zu verbinden. Ferner kann man sich wieder um den Master kümmern.

Hat man den Minion vollständig konfiguriert, wird dieser mit sehr großer Wahrscheinlichkeit seinen Master schon versuchen zu kontaktieren. Dies kann dann mithilfe des richtigen Befehls leicht überprüft werden. Wenn hier tatsächlich eine Verbindungsanfrage eingegangen ist, wird diese bei Salt als „Unaccepted Key“ dargestellt. Dieser muss für den restlichen Ablauf angenommen werden, woraufhin die eine Kommunikation zwischen Master und Minion durchführbar ist. Im Anschluss an die gelungene Verbindung der zwei Vagrant Boxen Master und

[35] https://repo.saltstack.com/#rhel

Minion, kann damit angefangen werden mit SaltStack zu provisionieren. Hierzu erstellt man ähnlich wie bei Ansible eine Konfigurationsdatei, die im SaltStack Umfeld SLS File genannt wird. In genau dieser Datei wird analog zu Ansible alles aufgelistet, was für den Jenkins Prototypen gebraucht wird. Daher muss auch hier als erstes Java installiert werden (siehe *Kapitel 4.2.1 Prototypentwicklung mit Ansible*). Dementsprechend muss auch hier das Jenkins Repository hinzugefügt werden, um hinterher die Definition zur Installation von Jenkins hinzufügen zu können, sodass diese auch erfolgreich sein wird. Nun ist noch der Zustand zu definieren, dass der Jenkins Service gestartet werden soll. Abschließend kann mit dem richtigen Befehl vom Salt-Master aus der Minion mittels SLS File provisioniert werden.

### 4.2.3 Prototypenentwicklung mit Docker

Der letzte Prototyp entsteht durch die Technologie Docker. Der Entwurf dieses Prototyps grenzt sich von den bisherigen zwei Prototypen etwas mehr ab, da die Umsetzungen hier ein wenig anders sind. Der erste Unterschied, der hier zu nennen ist, ist der, dass die zuvor verwendete administrative Vagrant Box nicht gebraucht wird. Deswegen muss hier wieder die zu konfigurierende Vagrant Box neu erstellt werden, um wieder den gleichen Workaround zu schaffen.

Hat man hier nun Zugriff auf die Box, beginnt man damit, das Docker-Repository hinzuzufügen. Genaue Beschreibungen hierzu befinden sich in den im Anhang beigefügten Inhalten. Nachdem man das Repository hinzugefügt hat, muss Docker installiert, aktiviert und schließlich gestartet werden. Zum Testen empfiehlt es sich hier das von Docker vorgegebene Beispiel von „Hello World[36]“ auszuführen, damit die Gewissheit besteht, dass der Docker-Service voll funktionsfähig ist.

Wurde die Einrichtung auf dem Zielsystem korrekt durchgeführt, folgt die Realisierung des Prototyps. Zu Beginn wird hier ein Verzeichnis benötigt, welches an einem frei wählbaren Ort erstellt werden kann. Folgend erstellt man dort das

[36] Schritt sieben zeigt das Kommando: https://docs.docker.com/engine/installation/linux/centos/

Dockerfile, das benötigt wird, um das später benötigte Docker-Image zu generieren. Daraufhin wird dem Dockerfile alles Relevante hinzugefügt, damit das Starten eines Docker-Containers innerhalb der CentOS VM möglich ist. Hierzu gehören wie bei den zwei Vorgängern das Aktualisieren aller bisherigen Linux Packages, die Installation von Java und dem Jenkins selbst. Die Installationen, die zur erfolgreichen Einrichtung des Jenkins benötigt werden, orientieren sich zusätzlich auch hier stark an den Vorgänger Prototypen. Der Unterschied ist, dass die Ausführung der Befehle innerhalb des Dockerfile lediglich anders realisiert ist. Anstatt die üblichen Kommandozeilen Befehle zu verwenden, wird hier ein Schlüsselwort vor jedem Befehl ergänzt. Außerdem verändert hat sich das Starten des Jenkins, indem hier nicht die üblichen Befehle verwendet werden, weil das Hochfahren des Service nun als Java Prozess vollzogen wird. Diese Variante zum Starten zielt auf das Konzept der leichtgewichtigen Container von Docker ab und wird daher bevorzugt. Anderseits ist es auch möglich, den Jenkins als einen richtigen Service zu starten. Damit dieser Start mit Docker durchgeführt werden kann, braucht man für die CentOS Umgebung allerdings einige zusätzliche Packages zur Ausführung. Installiert man diese Extra-Packages ebenfalls, so wächst die Größe des zuvor erzeugten Containers wieder. Deshalb sollte darauf verzichtet werden, diesen Weg der Erzeugung eines Jenkins Service zu wählen, um die Prinzipien der Technologie Docker einzuhalten. Die Information dieser Möglichkeit soll jedoch zeigen, dass auch hier andere Wege für eine Realisierung denkbar sind. Ein wichtiger Hinweis für das letztere Vorgehen ist, dass es unter CentOS ein bekanntes Problem ist, dass benötigte Packages nicht vorhanden sind. Die korrekte Einrichtung dieser Vorgehensweise wird durch einen von Docker vorhandenen Workaround bereitgestellt[37].

Nachdem man das Dockerfile mit allen Informationen befüllt hat, kann man das aus dem Dockerfile entstehende Docker-Image generieren lassen. Dazu verwendet man das dafür vorgesehene Kommando, welches dann die Generierung durchführt. Abschließend muss nun der Docker-Container aus dem zuvor generierten Docker-Image gestartet werden, wonach eine Verwendung des in ihm enthaltenden Jenkins möglich ist.

[37] https://hub.docker.com/_/centos/

# 4.3 Entwicklung der Continuous Integration Pipeline

Das Kapitel 4.3 ist der Einstieg in die konkrete Umsetzung der Continuous Integration Pipeline auf Basis der zuvor entwickelten Prototypen. Die zu entwickelnde Pipeline für den Jenkins CI Server soll die Pipeline für ein gegebenes Projekt beispielhaft darstellen. Dieser Entwurf einer CI Pipeline zeigt mithilfe des Springboot Beispielprojekts[38] wie man unterschiedliche Jobs für ein Softwareprojekt automatisiert einrichten kann. Innerhalb von Jenkins können diese Jobs durch die Verwendung von bereits entwickelten Plugins umgesetzt werden, dazu jedoch später mehr. Übertragungen und Weiterentwicklungen einer solchen Pipeline sind im Nachhinein trivial in deren Umsetzung.

## 4.3.1 Entwicklung mit Ansible

Im folgenden Abschnitt wird ausführlich beschrieben, wie man mit dem Tool Ansible eine Continuous Integration Pipeline für den CI Server Jenkins einrichtet. Als erstes kann man für die Entwicklung den vorher entwickelten Prototypen nutzen, sofern dieser gesichert wurde. Dadurch ergibt sich der Vorteil, dass die erneute Einrichtung des Prototyps entfällt. Falls eine Sicherung hierbei nicht vorgenommen wurde, muss vorher die Einrichtung, wie in *Kapitel 4.2.1 Prototypenentwicklung mit Ansible* beschrieben, stattfinden. Hat man nun den funktionsfähigen Prototypen, so kann damit begonnen werden, sich mit Ansible im Spezielleren auseinanderzusetzen. Für diesen Teil der Entwicklung kann die gesamte Ausführung durch ein einzelnes Playbook realisiert werden oder man versucht sich an einer von Ansible gegebenen Struktur der Organisierung. Aus der Sichtweise eines Anfängers im Bereich Ansible ist es wahrscheinlich ratsam, die erst genannte Variante zu benutzen. Die letztere Lösung einer geordneten Struktur ist möglicherweise im Nachhinein leichter umzusetzen, weil man nach der ausführlichen Einarbeitung ein besseres Grundverständnis bezüglich Ansible besitzt. Dies ist jedoch der Eindruck des

[38] https://spring.io/guides/gs/spring-boot/

Verfassers der Arbeit, welcher aufgrund dieser gewählten Vorgehensweise entstanden ist.

Deshalb wird damit angefangen, das Playbook des Prototyps zu nutzen und zu erweitern. Dazu werden hier nun zwei Plugins eingeführt, mit denen die Entwicklung der CI Pipeline deutlich erleichtert wird. Das erste Plugin ist das „Job-DSL-Plugin“. Mit diesem Plugin ist es, nachdem es für den Jenkins installiert wurde, möglich, durch die DSL „Groovy“ im Jenkins die sogenannten Jobs zu definieren. Das zweite Plugin nennt man „Seed-Plugin“ und wird in Kombination mit dem vorher genannten Plugin verwendet, was daran zu erkennen ist, dass unter den Abhängigkeiten des Seed-Plugins auch das Job-DSL-Plugin zu finden ist. Diese zwei Plugins werden nun mit in das bereits vorhandene Playbook mit aufgenommen, indem die entsprechenden Definitionen hinzugefügt werden. Für die Installation von Jenkins Plugins gibt es ein Ansible Extra Modul[39], welches Unterstützung für deren Installation bietet. Danach sollte man die Installation der Plugins testen, indem das Playbook ausgeführt wird (Jenkins muss nach einer Plugin Installation in manchen Fällen neugestartet werden). Testen lässt sich das Ganze nun am besten erst einmal durch ein Manuelles Vorgehen, um die Arbeitsweise innerhalb von Jenkins und auf dem Dateisystem zu verstehen.

Um hier starten zu können, erstellt man im Jenkins nun ein Projekt unter „Element anlegen“ und vergibt einen Namen. Anschließend gibt es verschiedene Konfigurationsmöglichkeiten, von denen aber nur die der „Buildverfahren“ benötigt wird. Wurden die zuvor genannten Plugins korrekt installiert, hat man hier die Möglichkeit, Job-DSLs einzubinden. Füllt man den gerade beschriebenen „Seed-Job“ mit einer Job-DSL, entstehen bei einem Build dieses Jobs, die in ihm angegebenen weiteren Jobs. Erweitert man das Szenario nun um ein Versionskontrollsystem, müssen je nach Anwendungsfall noch weitere Plugins hinzugefügt werden (Beispiele sind in dieser Arbeit Git und Maven). Auch diese zusätzlichen Plugins werden in dem Playbook aufgenommen. Dabei gilt ebenfalls, dass nach erfolgreicher Installation

[39] Ein Extra Modul ist ein Modul, dass bisher standardmäßig mit der Ansible Installation vorhanden ist, aber zum Großteil von der Community gewartet wird. https://docs.ansible.com/ansible/modules_extra.html

ein Neustart erforderlich sein kann, was dann ebenfalls innerhalb des Playbooks definiert wird.

Damit die manuelle Konfiguration dieser Jobs nun auch in mit Infrastructure-as-Code realisiert wird, werden genau diese Schritte mit in das Ansible Playbook geschrieben. Dazu kann man sich in der internen Jenkins Struktur die Jobs anschauen und findet bei jedem Job eine „config.xml“ Datei (aus dem manuellen Prozess entnehmen). Diese XML Datei kann nun den Wünschen entsprechend angepasst werden und beinhaltet die benötigten Informationen für die Jenkins Jobs. Jetzt muss man vom Jenkins unter einer definierten URL die Datei für das Jenkins Command Line Interface (CLI) herunterladen und beim Jenkins ablegen. Mittels des Jenkins CLI kann man nun den Seed-Job erstellen, basierend auf der zuvor genannten config.xml Datei, die an beliebiger Stelle abgelegt werden kann. Für die Erstellung wird dann mittels korrekter Shell Befehle im Playbook definiert, an welche URL (inklusive der benötigten Login Informationen) das Kommando gesendet wird. Mit dem Befehl „create-job“ wird zuerst die Ordnerstruktur angelegt. Ist der Befehl erst einmal erfolgreich ausgeführt, kann man den erstellten Job auf der Jenkins GUI sehen. Weiterführend lässt sich, wie in dieser Arbeit durchgeführt, durch den Befehl „build“ der Seed-Job bauen. Sobald dieses Kommando ausgeführt wurde, werden aus der im Seed-Job definierten Job-DSL die zu erstellenden Jobs generiert. Wurden diese Jobs schließlich auch generiert, kann hier ebenfalls der „Build-Befehl“ ausgeführt werden.

In den beigelegten Skripten und Anleitungen kann nachvollzogen werden, wie die einzelnen Befehle aussehen und verwendet wurden. Aus dem in dieser Arbeit vorhandenen Seed-Job werden zwei weitere Jobs generiert, die das Beispielprojekt von Spring, was zu Beginn erwähnt wurde, Kompilieren und Testen. Abschließend sollte das einzelne große Playbook in mehrere kleine Teile überführt werden, damit man eine übersichtliche Struktur für das Projekt besitzt.

### 4.3.2 Entwicklung mit SaltStack

Nachdem die Entwicklung der Pipeline durch Ansible nun ausführlich beschrieben wurde, folgt die Darstellung der entwickelten Pipeline durch SaltStack. Ebenso wie bei Ansible kann auch hier wieder wie zuvor der entwickelte Prototyp zur weiteren Entwicklung für die Pipeline verwendet werden.

Begonnen wird mit der Erweiterung der vorhandenen State Datei von Salt. Hierbei wird zu Beginn der gesamte Code hineingeschrieben, damit auch das Verständnis für SaltStack gefestigt werden kann. Im Nachhinein wird die große Datei dann in mehrere, kleinere und strukturiertere Dateien aufgeteilt. Zunächst wird die gewohnte Entwicklungsumgebung wieder eingerichtet, damit die Entwicklung der Pipeline beginnen kann. Hierzu werden erneut die zwei Vagrant Boxen so eingerichtet, sodass auf der administrativen Box einen funktionierenden Salt-Master gibt und auf der zu provisionierenden Box den Salt-Minion. Aufgrund der bereits besseren Kenntnisse dieser Technologien, kann dieses Wissen genutzt werden, um auch diese Schritte der Konfiguration zu automatisieren (zuvor war es für den Lerneffekt, gut dies manuell zu machen). Möglich ist also die Verwendung von Ansible, um die zwei Maschinen aufzusetzen oder man erleichtert sich das Ganze, indem man die ständig benötigten Konfigurationen über das Vagrantfile definiert. Sind Master & Minion eingerichtet, müssen die Voraussetzungen für den Jenkins auf der Maschine eingerichtet werden. Dazu verwendet man das aus Salt dafür vorgesehenes Salt State und installiert alle Abhängigkeiten, die notwendig sind. Diese sind auch hier Git, Maven und Java. Nachdem diese auf dem System vorhanden sind, müssen noch drei weitere Packages hinzugefügt werden, die bisher bei Salt nicht benötigt wurden. Als erstes muss hier wie schon von vorher bekannt EPEL wieder installiert werden (siehe *Kapitel 4.2.1 Prototypenentwicklung mit Ansible*). Die EPEL werden benötigt, um ein Problem, welches später folgen würde, zu vermeiden. Anschließend fügt man nun python-pip hinzu, was dazu benötigt wird, dass man mittels pip die Library für Python Jenkins installieren kann. Warum genau diese Punkte noch hinzukommen, wird an der passenden Stelle erläutert. Im Anschluss folgt jetzt der Teil, bei dem der Jenkins vorbereitet sowie Betriebsbereit gemacht wird.

Die Integration des Jenkins Repository und die Installation werden, genau wie im Prototypen auch, hintereinander ausgeführt. Ist der Jenkins installiert, muss man wieder ein wenig warten, damit der Service genutzt werden kann. Wenn man den Jenkins nun verwenden kann, muss als nächstes das Jenkins-CLI gedownloadet werden. Abgespeichert wird die Datei für das Jenkins-CLI in dem „Jenkins_Home“ Verzeichnis. Sobald dieses ebenfalls vorhanden ist, muss für das weitere Vorgehen mit SaltStack nun ein Nutzer im Jenkins angelegt werden. Auch hierzu folgen später genauere Details. Damit man einen Nutzer in Jenkins anlegen kann, muss man sich hier wieder mit der Jenkins Groovy DSL befassen. Verwendet man nun ein spezielles Groovy Skript zum Erstellen eines Nutzers, so ist es möglich, dieses mithilfe des Jenkins-CLI auszuführen. Bei dessen Benutzung muss man jedoch die URL des gestarteten Jenkins angeben, wobei der Start notwendig ist, da es sonst zu Fehlern kommt. Weiterhin muss man sich innerhalb dieser URL mit den Administrator Daten anmelden, weil diese zu dem Zeitpunkt die einzigen sind, die vorhanden sind. Dadurch, dass man auch dies nicht gern manuell machen möchte, ist es notwendig, hier eine Verbindung mehrerer CMD Befehle durchzuführen. Als erstes speichert man das initial angelegte Passwort aus der hierfür vorgegebenen Datei in eine Variable in der Kommandozeile. Durch ein Semikolon fügt man nun einen „Echo-Befehl“ hinzu, der das Groovy Skript beinhaltet, welcher dann an das Kommando zur Ausführung des Jenkins-CLI über eine Pipe gesendet wird. Danach muss nur noch das Passwort in Form der Variable in die URL geschrieben werden, wonach der User dann angelegt werden kann.

Jetzt folgt der Abschnitt, bei dem es um die Installation der Jenkins Plugins geht. Speziell dieser Teil der Entwicklung der Pipeline war deutlich schwieriger umzusetzen als die bisherigen, da es für die Installation keine richtige Unterstützung seitens SaltStack gibt. Eine Möglichkeit wäre gewesen, die Plugins mittels Jenkins-CLI zu installieren, jedoch wurden hier wichtige Abhängigkeiten zu anderen Plugins nicht berücksichtigt. Dadurch hätten erst die im Abhängigkeitsbaum untersten Plugins installiert werden müssen (Blätter des Baumes). Nach und nach müsste man also die Plugins von den Blättern ausgehend bis hin zur Wurzel installieren um das eigentliche Plugin installieren zu können. Um dem entgehen zu können, gibt es eine Python Library, mit der die Installation samt Abhängigkeiten machbar ist. Salt hat

hier auch einige Methoden übernommen, jedoch fehlte diese innerhalb des Salt Modules leider. Deswegen musste man vorher die drei zusätzlichen Packages beifügen, damit man ein eigenes Salt Module einbinden kann. Hierzu kann man das von Salt bereitgestellte Module kopieren und um die nötige Methode erweitern. Ist diese verfügbar, können die Plugins für den Jenkins nun ohne weitere Probleme eingerichtet werden. Aufpassen muss man jedoch bei jenen Plugins, welche viele Abhängigkeiten besitzen wie z.B. das Seed-Plugin. Eine zu schnelle nächste Aktion beeinflusst die Installation der Abhängigkeiten dadurch, dass nicht alle benötigten installiert werden, was bedeutet, dass Wartezeiten zwischen den einzelnen Aktionen notwendig sind. Hat man diese aber eingebaut, ist dieser Schritt auch abgeschlossen.

Daraufhin werden nun die abschließenden Continuous Integration Jobs eingerichtet. Um dies zu erreichen, muss als erstes die config.xml Datei in einem Verzeichnis abgelegt werden, welches beliebig gewählt werden kann, da es als temporäres Verzeichnis dient. Danach nutzt man anstatt des Jenkins-CLI nun das Python Module, welches vorher erstellt wurde. Dort enthalten ist nämlich bereits eine Methode, die es erlaubt einen Jenkins-Job zu erstellen, indem man den Namen des Jobs und die dazugehörige Konfigurationsdatei als Parameter mitgibt. Sobald der Job generiert wurde, kann der nächste Befehl zum Bauen des Seed-Jobs verwendet werden. Auch dieser befindet sich in dem eigenen Python Module, wobei dort nur die oben genannte Funktion selbstständig erstellt werden muss und der Rest einem bereits geliefert wird. Der Funktion zum Bauen des Seed-Jobs wird hier nur der Name als Parameter übergeben. Wurden schließlich die zwei weiteren Jobs zum Kompilieren und Testen des Projekts erstellt und sofern das Ergebnis beim Bauen jener erfolgreich ist, hat man die Pipeline durch SaltStack fertiggestellt.

Nicht zu vergessen ist jetzt jedoch, dass sich der gesamte Aufbau der Pipeline innerhalb einer einzigen Datei befindet. Dieser Zustand ist tatsächlich nicht besonders gut und auch irgendwann sehr unübersichtlich. Darum sollte das eine State File aufgeteilt werden und durch mehrere ersetzt werden. Für diese Arbeit wurde zu diesem Zweck eine Struktur verwendet, die einen besseren Überblick über die verschiedenen States liefert.

Zu Beginn muss in der Konfigurationsdatei des Masters ein weiterer Pfad aktiviert werden, indem die Kommentare dort entfernt werden. Hierbei handelt es sich um die Pfade für die Salt Pillar Dateien. Sofern die Kommentare hier entfernt sind, müssen die dort angegebenen Verzeichnisse im System erstellt werden, da dies nicht automatisch geschieht. Anschließend können in dem Verzeichnis die Dateien top.sls[40] und pillar.sls[41] erstellt und mit Funktionalität gefüllt werden. Danach bewegt man sich wieder in das gewohnte Verzeichnis wie auch im Prototypen *(„/srv“)*. Hier muss nun ein Ordner erstellt werden, der die Jenkins Dateien enthalten soll. Darin werden zwei weitere Verzeichnisse benötigt und sollten „files“ und „_modules“ heißen. In „files“ befinden sich Dateien, die auf den Minion transportiert werden sollen und „_modules“ beinhaltet das selbst erstellte Python Module. Aufgeteilt werden können die Schritte beliebig, jedoch wurde das vorherige Kapitel extra so beschrieben, dass man erkennen kann, welche Abschnitte ein einzelnes state file darstellen sollen. Diese werden dann in dem gerade genannten Verzeichnis der Jenkins Dateien abgelegt. Zusätzlich, um die Ausführung generischer zu machen, erstellt man eine init.sls Datei, welche alle anderen inkludiert. So muss nur eine Datei zur Realisierung der Konfiguration gestartet werden. Eine Ausnahme ist hierbei jedoch die Konfiguration der vorhin genannten Python Unterstützungen. Diese müssen, bevor das selbst erstellte Python Modul verwendet wird, vorhanden sein, was prinzipiell auch mittendrin geschehen kann, aber nicht funktionsfähig ist. Der Grund ist, dass man Salt mitteilen kann, dass die Umgebung neu geladen werden muss, um Änderungen aufnehmen zu können. Dieses erneute Laden funktioniert in diesem Fall leider nicht für genau die Python Unterstützung. Zum Schluss braucht man noch eine Erweiterung der Dateien um ein sogenanntes map.jinja[42], welches in den unterschiedlichen State files eingebunden werden kann um bequem auf die Daten des Pillar files zugreifen zu können. Hierzu ist vielleicht noch erwähnenswert, dass die Jinja Blöcke in einer SaltStack Umgebung beim Start der Ausführung ausgewertet werden (Eager Evaluation). Auch für dieses

[40] Die top.sls Datei beinhaltet hier alle pillar.sls Dateien und kann steuern für welche Minion die einzelnen Pillar Dateien bestimmt sind (mehrere Pillar sind möglich).

[41] In einer Pillar Datei kann man sensible Daten wie Passwörter, Benutzernamen etc. unterbringen. Möglich sind aber auch Variablen und andere Konfigurationsdaten.

[42] http://jinja.pocoo.org/

Kapitel wird im Anhang genug Material vorhanden sein um die Entwicklung besser verstehen zu können.

### 4.3.3 Entwicklung mit Docker

Schlussendlich folgt die letzte der drei Entwicklungen mit Docker. Aufgrund der bisher gesammelten Kenntnisse und der Erstellung des Prototyps ist erkennbar, dass man mit Docker eine ganz andere Variante zu diesem Thema besitzt. Gleich zu Anfang ist hier zu sagen, dass der entwickelte Prototyp mit Docker für diesen Teil der Entwicklung nicht weiter verwendet wird. Durch den Prototypen sollte aber folglich ein gewisses Ausprobieren der Engine gegeben sein, mit dessen Hilfe der Einstieg ein bisschen leichter fällt. Unterdessen ist es jedoch trotzdem möglich die nachfolgende Entwicklung auch eigenständig durchzuführen, was jedoch den zeitlichen Rahmen dieser Arbeit deutlich überschreiten würde.

Daher kann in diesem Fall aus der prototypischen Entwicklung nur die Konfiguration der benötigten Docker-Umgebung verwendet werden. Auch hierbei gilt wieder, dass die Einrichtung nun automatisch erledigt werden kann. Ist man soweit und hat die vorausgesetzte Vagrant Box eingerichtet, sodass Docker funktioniert, beginnt die Entwicklung der Pipeline.

Der Startpunkt ist auch hier, wie im Prototypen ein Verzeichnis, indem die Datei Dockerfile enthalten ist. Ist die Datei erstellt, beginnt man erneut mit der aus dem Prototypen aus *Kapitel 4.2.3 Prototypentwicklung mit Docker* gewohnten Syntax und fängt an das Basis Image, welches verwendet werden soll, als ersten Befehl hinzuzufügen. Für dieses Basis Image wurde beim Prototyp das CentOS Image genutzt, was bei der Entwicklung dieser Pipeline jedoch nun ausgetauscht wird. Das neue Image, das in diesem Fall verwendet werden sollte, ist von Jenkins selbst und beinhaltet durch dessen Konfiguration schon viele nützliche und bereitgestellte Funktionalitäten. Dieses Image wurde, wie schon am Anfang des Kapitels erwähnt, ausgesucht, weil eine Eigenentwicklung dieses Ziels zeitlich im Rahmen dieser Arbeit nicht möglich ist. Bei genauerer Betrachtung der Jenkins Umsetzung ist schnell zu erkennen, dass die Unterstützten Funktionalitäten dort von erhöhter

Komplexität sind. Auszuschließen ist die selbstständige Entwicklung trotzdem nicht, da man dadurch eigene Funktionen möglicherweise anders umsetzen kann oder bestehende Elemente von Jenkins entfernen möchte.

Nachdem nun das Docker-Image von Jenkins verwendet wird, ist der größte Teil jetzt schon fertig. Zu zeigen ist dies, indem man aus dem Dockerfile nun das daraus resultierende Docker-Image erstellt, woraufhin dann der Container gestartet werden kann. Sobald der Docker-Container gestartet wurde, hat man einen funktionierenden Jenkins CI-Server, beinhaltend mit den Basis Funktionen, die durch die Entwickler bereitgestellt werden. Für den Zugriff von außerhalb müssen auch hier die Port Weiterleitungen von virtueller Maschine und Docker-Container beachtet werden.

Als nächstes folgen die Einrichtung der Jobs sowie die Installation der benötigten Plugins. Wegen der Verwendung von Docker muss die Konfiguration dafür im Dockerfile selbst geschehen. Um diesen Schritt realisieren zu können, sollte zuerst in der Dokumentation der Jenkins-Docker Unterstützung nachgelesen werden, wie man spezielle Funktionen einbauen kann[43]. Da man sich für ein Drittanbieter Docker-Image entschieden hat, ist es wichtig, dass man die Strukturen hierbei beibehält. Einhalten kann man dies bei Jenkins am besten, indem die Umsetzung der Lösung genauer untersucht wird. Herausfinden sollte man daher, wie die Jobs eingerichtet und Plugins installiert werden. Für die Installation der Plugins gibt es durch die Nutzung dieses Images bereits eine Umsetzung, mit dessen Hilfe alle Plugins samt Abhängigkeiten zu anderen Plugins installiert werden. Dazu wird wie von Jenkins vorgeschlagen der Befehl dem Dockerfile hinzugefügt, wonach die Installation problemlos funktionieren wird.

Der Abschluss für diese Entwicklung ist das Einrichten der Jenkins-Jobs. Dieser Schritt ist etwas schwieriger als bisher, da das einfache Hinzufügen der config.xml für den Seed-Job nun auf dem bisherigen Weg nicht mehr ohne weiteres möglich ist. Die Docker-Lösung für Jenkins richtet ein initiales Verzeichnis unter

[43] https://github.com/jenkinsci/docker
https://hub.docker.com/_/jenkins/

„/usr/share/jenkins/ref“ ein, welches dazu vorhanden ist, alle notwendigen Dateien und Konfigurationen zu halten[44]. Mithilfe dieses Verzeichnisses wird bei einer komplett neuen Installation alles in den Jenkins herein konfiguriert. Für den Seed-Job, der angelegt werden soll, wird in dem Dockerfile definiert, dass eine Groovy Datei in das initiale Verzeichnis für die Jenkins Installation kopiert wird. Diese Datei enthält nun in Groovy geschrieben, die Erstellung des Seed-Projekts, welches nach einem erfolgreichen Build die zwei Jobs erzeugt. Diese zwei Jobs müssen aber ebenfalls in einer weiteren Groovy Datei definiert werden (mit der Job-DSL). Ausgehend von der Groovy Datei für den Seed Job, wird die Groovy Datei für die Generierung der zwei weiteren Jobs ausgeführt.

Anschließend, nachdem die beiden Jobs zum Kompilieren und Testen des Projekts generiert wurden, wird noch jeweils ein Build angestoßen. Vorweg ist es noch notwendig, in dem Dockerfile ein Update der vorhandenen Linux Packages, sowie eine anschließende Installation von Maven durchzuführen. Grund hierfür ist ein Fehlschlagen der Jobs zum Kompilieren und Testen des Projekts, weil das Kommando zum Ausführen von Maven nicht möglich ist. Wenn nach der Ausführung der zwei Builds die Ergebnisse erfolgreich sind, hat man ebenso wie für Ansible und SaltStack jetzt eine funktionierende CI-Pipeline durch Docker.

[44] Die Konfiguration der Jobs durch das Kopieren in das gegebene Verzeichnis schaltet leider die eingebaute Security des Jenkins ab.

## 4.4 Gegenüberstellung der Technologien

Abgeschlossen wird die Entwicklung der jeweiligen Continuous Integration Pipelines durch einen Vergleich. Dabei soll es primär darum gehen, allgemein die gesammelten Erfahrungen darzustellen sowie einen speziellen Vergleich bezogen auf die gezielte Entwicklung dieser Arbeit anzustellen. Betrachtet man die aus der Arbeit resultierenden Continuous Integration Pipelines, kann im Vorfeld gesagt werden, dass eine Erleichterung der Einrichtungsprozesse in jedem Fall gegeben ist. Die Schwierigkeiten liegen hierbei in erster Linie beim Unverständnis gegenüber dem Produkt, das man verwendet. Genau auf diese Unwissenheit wird es im ersten Abschnitt des Vergleichs gehen.

Angefangen mit Ansible hat man für die Entwicklung für die Pipeline mit Jenkins eine gute Unterstützung. Die Verwendung ist hier sehr angenehm und gut erlernbar. Wenn Funktionalitäten zuerst unverständlich waren, hat man durch genaueres Nachlesen innerhalb der Dokumentation ein solches Problem schnell lösen können. Hierbei sind die Beschreibungen der Funktionen auch gut zu verstehen und lassen sich leicht finden. Die erste Umsetzung einer großen Datei, die man hinterher dann in die von Ansible vorgeschlagenen Einzelteile zerlegt, war ebenfalls gut handzuhaben und auch für Einsteiger gut umsetzbar. Aber es sind auch bei diesem Tool kleine Schwierigkeiten aufgetreten, um die man sich auch als Einsteiger in dem Gebiet kümmern muss. Als erstes zu nennen ist hier die Konfiguration von SSH. Für eine solche Konfiguration ist es anfänglich nicht so leicht, herauszufinden, wie die Einrichtung konkret umgesetzt wird. Sobald man sich aber in die Materie eingelesen hat, ist eine Verbindung der Maschinen durch SSH, bis auf die korrekte Rechteverwaltung durch Linux weniger problematisch. Das zweite Problem, was in manchen Fällen zur Verwirrung bei der Entwicklung beiträgt, ist eine missverständliche Rückgabe von Fehlermeldungen. Explizit durch solch irreführende Fehlermeldungen versucht man Fehler zu beheben, die gar nichts mit dem Problem zu tun haben. In diesem Beispiel ging es um den Fehler, dass Zugriffsrechte angeblich nicht vorhanden waren, jedoch lag das Problem in einem syntaktischen Fehler. Hätte man frühzeitig auf diesen Fehler hingewiesen, wäre der Aufwand der Fehlersuche möglicherweise nicht entstanden. Aufgrund dieser recht

geringen Anzahl an Problemen mit dem Tool Ansible fällt eine Bewertung hier sehr positiv aus.

Als nächstes folgt die Bewertung der Entwicklung durch SaltStack. Ebenso wie mit Ansible besteht durch SaltStack auch eine gute Variante der automatisierten Einrichtung von Jenkins. Die Installation und Vorbereitung waren bei Salt für dieses Szenario vergleichbar gut. Hat man die Installation dort jedoch abgeschlossen, ging es mit dem positiven Eindruck der Entwicklung stetig bergab. Weil Jenkins 2.0 für den initialen Start ein spezielles Passwort benötigt, ist dies besonders bei der Entwicklung problematisch geworden. Bis zu diesem Zeitpunkt fehlte die Möglichkeit, dieses Passwort zur Laufzeit auszulesen und zu verwenden[45]. Damit man diese Schwierigkeit lösen konnte, wurden viele Möglichkeiten gesucht und ausprobiert. Viele der Versuche haben unter SaltStack nicht so funktioniert, dass genau dieses Problem behoben werden konnte. Als Beispiel kann hier genannt werden, dass das von Salt genutzte Jinja zu Beginn der Ausführung benötigt wird. Dies ist im Falle einer Provisionierung ein nicht zu unterschätzendes Problem, da vieles erst zur Laufzeit entsteht und somit noch nicht vorhanden ist. Letztendlich konnte man dieses Problem durch einen etwas komplizierteren Kommandozeilen Befehl lösen. Alles Weitere war mit SaltStack im Nachhinein dann auch wie mit Ansible wieder deutlich einfacher und angenehmer. Für die Lösung des eben genannten Problems gibt es möglicherweise eine gute Methode, die genau das lösen kann. Diese Möglichkeit wurde in dieser Arbeit jedoch nicht gefunden, sollte aber falls vorhanden, verwendet werden. Weiterhin ist zu erwähnen, dass man mit Salt eine Menge an Funktionalität mitgeliefert bekommt, was für viele Anwendungsfälle sicherlich eine große Entlastung ist. Jenkins selbst hat hier ebenfalls umgesetzte Funktionalitäten durch ein Jenkins Python Module, jedoch fehlte hier leider die Funktion zur Installation von Plugins. Dafür muss man bei Bedarf ein eigenes SaltStack Module erstellen, wonach man die fehlende Funktionalität dann implementieren kann. Dies war im weiteren Verlauf aber nicht mehr problematisch.
Auch mit Salt war die Überführung einer großen einzelnen Ausführungsdatei in das von SaltStack gegebene Schema einfach und gut beschrieben. Lediglich das Suchen

[45] Diese Möglichkeit ist eventuell vorhanden, jedoch schwer zu finden.

der korrekten Informationen innerhalb der großen Dokumentation dieses Tools ist dabei etwas schwieriger als bei Ansible.

Zum Schluss folgt noch eine kurze Bewertung der Docker-Umsetzung. Vergleicht man nun die drei Entwicklungen, ist die durch Docker erstellte Implementierung wesentlich schlanker als die zwei anderen. Der Grund hierfür ist die starke Präsenz der Docker-Community und deren Entwicklungen. Das Resultat der Docker-Entwicklung der Jenkins CI-Pipeline ist sehr gut gelungen und ist ebenso gut wie die von SaltStack und Ansible. Wegen des extremen Hypes wird einem hier bereits zu Beginn viel Arbeit abgenommen und vereinfacht. Für Jenkins gibt es in diesem Fall direkt zum Starten eine Docker Basis Entwicklung für den Jenkins. Durch dieses bereitgestellte Docker-Image werden einem auch die grundlegenden Konfigurationen für Docker abgenommen. Begleitend sollte man hierzu die Jenkins Installationsanweisung nutzen. Der schwierige Teil bei dieser Entwicklung besteht bei der Einrichtung der Jenkins Jobs. Damit man diese korrekt einrichten kann, ist es notwendig die interne Struktur der Jenkins Docker-Lösung zu analysieren. Begründet ist das damit, dass dort spezielle Pfade und Basis Verzeichnisse festgelegt werden, die für eine frische Installation alle benötigten Dateien und Konfigurationen beinhalten und dann mit einspielen. Genau diese Analyse ist zu Anfang recht schwierig, weil man diverse CMD Dateien hat, die man durchgehen sowie verstehen muss. Wenn Konfigurationsdateien an einem falschen Platz abgelegt werden, schaltet der Jenkins sich automatisch in einen unsicheren Modus, was für die Entwicklung nicht zu empfehlen ist. Hält man sich aber an die vorgegebenen Pfade, Verzeichnisse und die Struktur, ist eine Jenkins Installation wie in dieser Arbeit angestrebt, schnell und einfach bereitgestellt.

Abschließend kann man sagen, dass eine gezielte Verwendung durch eines der drei Tools gut analysiert werden muss, um den zuvor genannten Problemen in der Entwicklung aus dem Weg gehen zu können. Den Zweck der Automatisierung erfüllen hier alle drei Produkte jedoch gleichermaßen gut, was durch das positive Entwicklungsergebnis zu begründen ist.

# 5 Fazit

Schlussendlich wurde durch diese Arbeit eine Zeitersparnis für die Provisionierung eines CI-Servers realisiert. Begründen lässt sich das damit, dass ein vorhandenes manuelles Vorgehen nun durch eine automatisierte Vorgehensweise ausgetauscht werden kann. Zudem hat man durch die automatisierte Lösung mit IaC mittels verschiedener Softwareprodukte eine Reduzierung von potentiellen Fehlerquellen erreicht. Hierbei ist es wichtig zu erwähnen, dass mögliche Fehler bereits bei der Entwicklung erkannt, behoben und getestet werden können. Weiterhin Ist eine deutliche Verbesserung zu erkennen, da man die entwickelte CI-Pipeline auf unterschiedliche Projekte übertragen kann. Außerdem hat man durch die in dieser Arbeit realisierte Pipeline, drei verschiedene Möglichkeiten eine neue Pipeline einzurichten, oder aber bestehende Pipelines auf das entwickelte Schema zu übertragen. Besteht jedoch nur der Bedarf zur Auswahl eines Tools zur Provisionierung, so kann man sich an den genannten Aspekten sowie Vor- und Nachteilen der einzelnen Produkte orientieren um die eigene Analyse zu erweitern.

# 6 Ausblick

Mit dem Ergebnis dieser Arbeit ist für den Bereich Infrastructure-as-Code nur ein kleines Gebiet abgedeckt worden. Dieses Ergebnis wurde durch die Verwendung dreier verschiedener Vertreter realisiert, die für ihre Anwendungsgebiete noch weitere Optimierungen bieten. Zu nennen ist hier z.B. die Verwendung der verschiedenen Docker-Technologien, welche für den Zweig der Containertechnologie noch einiges an Parallelisierung, Performance, Deployment oder die Skalierung optimieren können. Genauer sind hier Systeme wie Kubernetes, Apache Mesos oder aber Packer[46] zu nennen. Des Weiteren wird es wegen des aktuell großen Hypes um die Containertechnologie rund um Docker weiterhin Verbesserungen sowie Erweiterungen geben. Ebenso wie bei Docker ist auch bei Ansible und SaltStack nicht alles an Performance etc. in dieser Arbeit angesprochen worden. Bereits die vielfältigen Möglichkeiten der Provisionierung erlauben hier verschiedene Modelle bei der tatsächlichen Ausführung. Außerdem gibt es auch bei diesen zwei Produkten noch verschiedene Erweiterungen zur Vereinfachung mancher Prozesse.

Den Abschluss macht ein Ausblick in ein Gebiet, welches nicht direkt in dieser Arbeit behandelt, innerhalb von Recherche und Gespräch mit anderen Entwicklern gesichtet wurde. Dabei handelt es sich um die sogenannten Unikernel. Eine Betrachtung der Unikernel ist für die weitere Bearbeitung dieses Themengebiets in gewisser Weise sehr sinnvoll, da die Unikernel in das Anwendungsgebiet von Docker passen[47]. Die Entscheidung, ob eine Nutzung der genannten Möglichkeiten letztendlich sinnvoll ist liegt jedoch bei den Stakeholdern dieses Bereichs.

---

[46] https://jaxenter.de/8-containertools-im-vergleich-docker-kubernetes-rkt-mesos-packer-shipyard-cloudslang-linux-container-40654

[47] https://m.heise.de/ix/meldung/Jenseits-von-Containern-Docker-uebernimmt-Unikernel-Systems-3081268.html

# Abbildungsverzeichnis

# Abkürzungsverzeichnis

| | |
|---|---|
| **IaC** | Infrastructure-as-Code |
| **CI** | Continuous Integration |
| **CD** | Continuous Delivery |
| **EPEL** | Extra Packages for Enterprise Linux |
| **DSL** | Domain Specific Language |
| **http** | Hypertext Transfer Protocol |
| **SSH** | Secure Shell |
| **WinRM** | Windows Remote Management |
| **YAML** | YAML Ain’t Markup Language |

# Glossar

### Ansible

Ansible ist ein Softwareprodukt, welches für den Bereich der IT-Automatisierung verwendet wird.

### Ansible play

Ein Ansible play ist Bestandteil eines Playbooks. Ein play beschreibt den Zustand eines bestimmten Hosts, den man konfigurieren möchte. Ein Playbook kann ein oder mehrere plays enthalten.

### Ansible Playbook

Die Ansible Playbooks sind ein Kernelement von Ansible und sind für die Aufgabenbereiche von Ansible vorhanden. Hierzu gehören z.B. die Orchestrierung oder das Konfigurationsmanagement.

### Ansible role

Ansible roles sind dazu da, um die Ansible tasks in mehrere Playbooks gruppieren zu können. Dies bietet den Vorteil einer strukturierten Übersicht über viele Systeme.

### Ansible task

Ansible tasks sind diejenigen Elemente bei Ansible, mit denen die Arbeit durchgeführt wird. Die tasks sind die Zustandsbeschreibungen der einzelnen Hosts.

### Deployment

Das Deployment beschreibt die Installation sowie Konfiguration von Software auf bestimmten Systemen.

### Design-Pattern

Design-Pattern dienen der Softwareentwicklung als Musterschablonen für die Entwicklung von Software.

### Docker

Docker ist eine Software, mit der es möglich ist Software in einem Linux Container laufen zu lassen. Diese Container besitzen alle notwendigen Voraussetzungen für das Betreiben der Software in dem Container.

### Docker daemon

Der Docker daemon ist eine zentrale Komponente von Docker und ist für die Hauptaufgaben von Docker zuständig. Zu diesen Aufgaben gehören beispielsweise das Erstellen sowie Starten von Containern oder das Erstellen der Images.

### Docker image

Docker-Images sind für die Erstellung von Docker-Containern notwendig. Ein Docker-Image entsteht aus einem Dockerfile und kann Abhängigkeiten zu anderen Images besitzen.

### Docker machine

Docker-Machine ist eine Unterstützung für die Installation der Docker-Umgebung auf virtuellen Maschinen eines Host-Systems. Diese Unterstützung wird beispielsweise für Windows oder Mac Systeme benötigt.

### Docker registry

Die Docker registry ist eine Art Repository für Docker-Images. Hierzu gibt es eine registry von Docker, jedoch sind auch private registries für die Verwaltung von Images möglich.

### Docker-Swarm

Docker-Swarm ist ein Produkt von Docker, mit dem es möglich ist mehrere Docker Umgebungen in einem Cluster zu betreiben.

### Domain Specific Language (DSL)

Eine Domain Specific Language ist eine Sprache, die für das Lösen von bestimmten Problemen gedacht ist.

### Eager evaluation

Die eager evaluation ist eine Auswertung, die gierig arbeitet. Das bedeutet, dass eine Auswertung direkt zu Beginn bzw. so schnell wie möglich stattfindet.

### Extra Packages for Enterprise Linux (EPEL)

Die Extra Packages for Enterprise Linux enthalten zusätzliche Packages für Enterprise Linux.

### Infrastructure-as-Code

Infrastructure-as-Code ist eine Technologie, mit deren Hilfe IT-Infrastrukturen automatisiert eingerichtet werden können.

### Jinja

Jinja ist eine template engine für die Programmiersprache Python.

### Orchestrierung

Die Orchestrierung beschreibt den Prozess, bei dem auf einer Systemlandschaft verschiedene Software miteinander, gemäß der Reihenfolge ihrer Abhängigkeiten in Betrieb genommen wird.

### Provisionierung

Unter der Provisionierung versteht man das Installieren und konfigurieren von IT-Infrastrukturen oder Systemen.

### SaltStack

SaltStack ist ein Softwareprodukt, welches für den Bereich der IT-Automatisierung verwendet wird.

### Salt Module

Salt Module sind die Bestandteile von SaltStack, welche die Funktionalitäten beinhalten. Die Module werden verwendet um die mit SaltStack zu bewältigenden Aufgaben zu erledigen.

### Salt Pillar

Pillar Dateien sind bei SaltStack dazu da, um Daten zu verwalten. Diese Daten können z.B. Minion Konfigurationen, Variablen oder Login Informationen sein.

### Salt State

Salt states sind die Elemente, die die jeweiligen Zustände des zu konfigurierenden Systems darstellen.

### Union file system

Ein Union file system ist ein Dateisystem, bei dem verschiedene Dateisysteme zu einem logischen Dateisystem verschmolzen werden. Hierbei liegen Dateien aus zwei verschiedenen Dateisystemen aus dem gleichen Verzeichnis in dem selben Verzeichnis.

### GoCD

GoCD ist ein open source Continuous Delivery Server.

# Quellenverzeichnis

[Morris 2016] Kief Morris: Infrastructure as Code Managing Servers in the Cloud. 1. Release Auflage Juni 2016: O'Reilly Media Inc. - ISBN 978-1-4919-2435-8

[Ludewig 2013] Jochen Ludewig; Horst Lichter: Software Engineering – Grundlagen, Menschen, Prozesse, Techniken. 3., korrigierte Auflage April 2013: dpunkt.verlag, ISBN 978-3-86490-092-1

[Humble 2010] Jez Humble; David Farley: Continuous Delivery – Reliable Software Releases Through Build, Test and Deployment Automation.

[Mouat 2015] Adrian Mouat: Using Docker - Developing and Deploying Software with Containers. Release Dezember 2015: O'Reilly Media, Inc. – ISBN 9781491915752

[Hochstein 2014] Lorin Hochstein: Ansible Up & Running – Automating Configuration Management and Deployment the Easy Way. 2. Auflage Mai 2015: O'Reilly Media Inc. – ISBN 9781491915318

[Hatch u. Sebenik 2015] Thomas Hatch; Craig Sebenik: Salt Essentials. 1. Auflage Juni 2015: O'Reilly Mdeia Inc. – ISBN 9781491914427

Dr. Daniel Beimbron; Thomas Miletzki, B.Sc.; Dipl.-Wirt.Inf. Stefan Wenzel (2011): Platform as a Service (PaaS), S. 381-384, URL: http://link.springer.com/article/10.1007/s12599-011-0183-3

Prof. Dr. Peter Buxmann; Dipl.-Wirtsch.-Ing. Sonja Lehmann; Prof. Dr. Thomas Hess (2008): Software as a Service, S. 501-503, URL: http://link.springer.com/article/10.1007/s11576-008-0095-0

Virmani Manish (2015): Undeerstanding DevOps & Bridging the gap from Continuous Integration to Continuous Delivery, S. 78-82, URL: http://ieeexplore.ieee.org/document/7173368/

## Webquellen

Red Hat Inc.: Get Started. URL: https://www.ansible.com/get-started (zuletzt abgerufen am 28.07.2016)

Red Hat Inc.: Use Case: Continuous Delivery. URL: https://www.ansible.com/continuous-delivery (zuletzt abgerufen am 16.11.2016)

Red Hat Inc.: Use Case: Configuration Management. URL: https://www.ansible.com/configuration-management (zuletzt abgerufen am 26.09.2016)

Red Hat Inc.: Overview How Ansible Works. URL: https://www.ansible.com/how-ansible-works (zuletzt abgerufen am 16.11.2016)

Red Hat Inc.: Documentation. Stand: 15.02.2017. URL: http://docs.ansible.com/ (zuletzt abgerufen am 16.11.2016)

Jaynes Matt: Ansible: Post-Install Setup. URL: https://valdhaus.co/writings/ansible-post-install/ (zuletzt abgerufen am 14.11.2016)

YAML.org: About. URL: http://www.yaml.org/about.html (zuletzt abgerufen am 26.09.2016)
Geerling Jeff: Running Ansible Within Windows. Stand: 13.01.2014 (aktualisiert 2016). URL: https://www.jeffgeerling.com/blog/running-ansible-within-windows (zuletzt abgerufen am 28.07.2016)

SaltStack Inc.: Salt documentation URL: https://docs.saltstack.com/en/latest/contents.html (zuletzt abgerufen am 16.11.2016)

SaltStack Inc.: SaltStack Platform Support. URL: https://saltstack.com/product-support-lifecycle/ (zuletzt abgerufen am 14.11.2016)

SaltStack Inc.: Supported Operating Systems. Stand: 16.02.2016. URL: http://saltstack.com/wp-content/uploads/2016/08/SaltStack-Supported-Operating-Systems.pdf (zuletzt abgerufen am 14.11.2016)

Rusev Martin: SaltStack – Review and how it fares against Ansible and Puppet? URL: https://www.amon.cx/blog/saltstack-review/ (zuletzt abgerufen am 17.10.2016)

Kramm Thorsten: SaltStack. Stand: Juni 2015. URL: https://www.heinlein-support.de/sites/default/files/it-automatisierung-mit-saltstack.pdf (zuletzt besucht am 05.10.2016)

Ellingwood Justin: An Introduction to SaltStack Terminology and Concepts. Stand 05.10.2015. URL: https://www.digitalocean.com/community/tutorials/an-introduction-to-saltstack-terminology-and-concepts (zuletzt abgerufen am 28.07.2016)

SaltStack Inc.: salt-winrepo. URL: https://github.com/saltstack/salt-winrepo (zuletzt abgerufen am 21.07.2016)

Cane Benjamin: Getting started with SaltStack by example: Automatically installing nginx. Stand: 03.09.2013. URL: http://bencane.com/2013/09/03/getting-started-with-saltstack-by-example-automatically-installing-nginx/ (zuletzt abgerufen am 28.07.2016)

Jaber German: Salt – Beginners Tutorial. Stand: 25.07.2014. URL: https://blog.talpor.com/2014/07/saltstack-beginners-tutorial/ (zuletzt abgerufen am 28.07.2016)

Aymen El Amri: SaltStack tutorial for beginners. Stand: 18.11.2014. URL: http://eon01.com/blog/salt-stack-tutorial-for-beginners/ (zuletzt abgerufen am 21.07.2016)

Docker Inc.: Docker Swarm. URL: https://www.docker.com/products/docker-swarm (zuletzt abgerufen am 23.11.2016)

Docker Inc.: What is Docker. URL: https://www.docker.com/what-docker# (zuletzt abgerufen am 23.11.2016)

Docker Inc.: Docker Cheat Sheet. Stand: 08.09.2016. URL: https://www.docker.com/sites/default/files/Docker_CheatSheet_08.09.2016.pdf (zuletzt abgerufen am 28.09.2016)

Roden Golo: Anwendungen mit Docker transportabel machen. Stand: 28.02.2014. URL: https://www.heise.de/developer/artikel/Anwendungen-mit-Docker-transportabel-machen-2127220.html (zuletzt abgerufen am 23.11.2016)

Mouat Adrian: Swarmv. Fleet v. Kubernetes v. Mesos. Stand: 21.10.2015. URL: https://www.oreilly.com/ideas/swarm-v-fleet-v-kubernetes-v-mesos (zuletzt abgerufen am 23.11.2016)

Büst René: Docker in a Nutshell. Stand: 16.10.2015 URL: https://www.crisp-research.com/docker-nutshell/ (zuletzt abgerufen am 22.11.2016)

Madhavapeddy Anil: Improving Docker with Unikernels: Introducing Hyperkit, VPNKit And DataKit. Stand: 18.05.2016. URL: https://blog.docker.com/2016/05/docker-unikernels-open-source/ (zuletzt abgerufen am 15.11.2016)

Marks Mano: Announcing The Docker For Mac And Windows Public Beta. Stand: 20.06.2016. URL: https://blog.docker.com/2016/06/docker-mac-windows-public-beta/ (zuletzt abgerufen am 15.11.2016)

Faler Wille: How Ansible and Docker Fit: Using Ansible to Bootstrap and Coordinate Docker Containers. Stand: 09.09.2014. URL: https://blog.docker.com/2016/06/docker-mac-windows-public-beta/ (zuletzt abgerufen am 19.10.2016)

Dr. Oliver Diedrich: Container: Apps für Server. Stand: 19.02.2016. URL: https://www.heise.de/ct/ausgabe/2016-5-Wie-Docker-Container-die-IT-industrialisieren-3102933.html (zuletzt abgerufen am 10.10.2016)
Janschitz Mario: Docker-Hosting: 10 Anbieter im Überblick. Stand 30.09.2016. URL: http://t3n.de/news/docker-hosting-613802/ (zuletzt abgerufen am 28.09.2016)

Hines Chris: Docker Basics Webinar Q&A: Understanding Union Filesystems, Storage And Volumes. Stand: 01.10.2015. URL: https://blog.docker.com/2015/10/docker-basics-webinar-qa/ (zuletzt abgerufen am 25.08.2016)

Sitakange Jafari: Infrastructure as Code: A Reason to Smile. Stand: 14.03.2016. URL: https://www.thoughtworks.com/de/insights/blog/infrastructure-code-reason-smile (zuletzt abgerufen am 23.11.2016)

Fowler Martin: InfrastructureAsCode. Stand: 01.03.2016. URL: https://martinfowler.com/bliki/InfrastructureAsCode.html (zuletzt abgerufen am 23.11.2016)

Morris Kief: Infrastructure as Code – Automation Is Not Enough. Stand: 01.01.2014. URL: http://kief.com/infrastructure-as-code-versus-automation.html (zuletzt abgerufen am 22.08.2016)

Büst René: Infrastructure as Code: Programmierung für Administratoren. Stand: 19.07.2016. URL: http://www.silicon.de/41631054/infrastructure-as-code-programmierung-fuer-administratoren/ (zuletzt abgerufen am 21.07.2016)

Morgenthal JP: Infrastructure-as-code, new rules for the old game. Stand: 30.06.2014. URL: https://devops.com/infrastructure-as-code/ (zuletzt abgerufen am 21.07.2016)

Seiwald Christopher: Infrastructure as Code: Agile Entwicklung muss über den Softwarecode hinausgehen, Stand: 04.01.2016. URL: https://www.heise.de/developer/artikel/Infrastructure-as-Code-Agile-Entwicklung-muss-ueber-den-Softwarecode-hinausgehen-3046561.html (zuletzt abgerufen am 28.07.2016)

Boucha Dave: How To Create Your First Salt Formula. Stand: 20.08.2013. URL: https://www.digitalocean.com/community/tutorials/how-to-create-your-first-salt-formula (zuletzt abgerufen am 30.11.2016)

Ellingwood Justin: How To Install and Configure Salt Master and Minion Servers on Ubuntu 14.04. Stand: 05.10.2015. URL: https://www.digitalocean.com/community/tutorials/how-to-install-and-configure-salt-master-and-minion-servers-on-ubuntu-14-04 (zuletzt abgerufen am 28.07.2016)

SaltStack Inc.: jenkins.py. URL: https://github.com/saltstack/salt/blob/develop/salt/modules/jenkins.py (zuletzt abgerufen am 27.12.2016)
Ronacher Armin: Jinja. URL: http://jinja.pocoo.org/ (zuletzt abgerufen am 02.01.2017)

Garage Willow: API reference python jenkins. URL: https://python-jenkins.readthedocs.io/en/latest/api.html (zuletzt abgerufen am 04.01.2017)

URL: https://github.com/geerlingguy/ansible-role-jenkins/issues/50 (zuletzt abgerufen am 03.01.2017)

URL: http://stackoverflow.com/questions/17716242/creating-user-in-jenkins-via-api (zuletzt abgerufen am 29.12.2016)

URL: https://issues.jenkins-ci.org/browse/JENKINS-32358 (zuletzt abgerufen am 29.12.2016)

Walsh Daniel: Running systemd within a Docker Container. Stand: 05.05.2014. URL: http://developerblog.redhat.com/2014/05/05/running-systemd-within-docker-container/ (zuletzt abgerufen am 05.12.2016)

Crosby Michael: Dockerfile Best Practices. Stand: 14.07.2013. URL: http://crosbymichael.com/dockerfile-best-practices.html (zuletzt abgerufen am 05.12.2016)

Tezer O.S.: Docker Explained: Using Dockerfiles to Automate Building of Images. Stand: 13.12.2013. URL: https://www.digitalocean.com/community/tutorials/docker-explained-using-dockerfiles-to-automate-building-of-images (zuletzt abgerufen am 05.12.2016)

Tezer O.S.: How To Install and Use Docker: Getting Started. Stand: 11.12.2013. URL: https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-getting-started (zuletzt abgerufen am 05.12.2016)

URL: https://ogavrisevs.github.io/2016/02/22/docker-in-scale/ Stand: 22.02.2016. (zuletzt abgerufen am 03.01.2017)

Inman Hannah: Get Started with Jenkins 2.0 with Docker. Stand: 29.04.2016. URL: https://www.cloudbees.com/blog/get-started-jenkins-20-docker (zuletzt abgerufen am 03.01.2016)

Van der Ploeg Nik: How To Work With Docker Data Volumes on Ubuntu 14.04. Stand: 17.11.2016. URL: https://www.digitalocean.com/community/tutorials/how-to-work-with-docker-data-volumes-on-ubuntu-14-04 (zuletzt abgerufen am 03.01.2017)

EPEL. Stand: 15.11.2016. URL: https://fedoraproject.org/wiki/EPEL (zuletzt abgerufen am 28.11.2016)

Anicas Mitchell: How To Configure SSH Key-Based Authentication on a FreeBSD Server. Stand: 14.01.2015. URL: https://www.digitalocean.com/community/tutorials/how-to-configure-ssh-key-based-authentication-on-a-freebsd-server (zuletzt abgerufen am 28.11.2016)

URL: http://stackoverflow.com/questions/25093168/how-do-i-create-a-new-project-in-jenkins-using-groovy (zuletzt abgerufen am 16.12.2016)

URL: https://github.com/martinmosegaard/vigilant-sniffle/blob/master/jenkins/createSeedJob.groovy (zuletzt abgerufen am 16.12.2016)

UpGuard Inc.: Declarative vs. Imperative models for configuration management: which is really better? URL: https://www.upguard.com/blog/articles/declarative-vs.-imperative-models-for-configuration-management (zuletzt abgerufen am 21.07.2016)

Fowler Martin: DeploymentPipeline. Stand 30.05.2013. URL: https://martinfowler.com/bliki/DeploymentPipeline.html (zuletzt abgerufen am 14.10.2016)

Fowler Martin: Continuous Delivery. Stand: 30.05.2013. URL: https://martinfowler.com/bliki/ContinuousDelivery.html (zuletzt abgerufen am14.10.2016)

Rantil Jens: Salt vs. Ansible. Stand: 17.03.2014. URL: http://jensrantil.github.io/salt-vs-ansible.html (zuletzt abgerufen am 14.10.2016)

UpGuard Inc.: Ansible vs Salt. URL: https://www.upguard.com/articles/ansible-vs-salt (zuletzt abgerufen am 16.11.2016)

Neumann Alexander: Jenseits von Containern: Docker übernimmt Unikernel Systems. Stand: 21.01.2016. URL: https://m.heise.de/ix/meldung/Jenseits-von-Containern-Docker-uebernimmt-Unikernel-Systems-3081268.html (zuletzt abgerufen am 30.01.2017)

Mohilo Dominik: 8 Containertools im Vergleich: Docker – Kubernetes – Rkt – Mesos – Packer – Shipyard – CloudSlang – Linux Container. Stand: 25.05.2016. URL: https://jaxenter.de/8-containertools-im-vergleich-docker-kubernetes-rkt-mesos-packer-shipyard-cloudslang-linux-container-40654 (zuletzt abgerufen am 20.01.2017)

Bundesamt für Sicherheit in der Informationstechnik: Man-in-the-Middle-Angriff. URL: https://www.bsi.bund.de/DE/Themen/ITGrundschutz/ITGrundschutzKataloge/Inhalt/_content/g/g05/g05143.html (zuletzt abgerufen am 16.11.2016)

URL: https://de.statista.com/ (zuletzt abgerufen am 16.11.2016)

ThoughtWorks Inc.: TechnologyRadar. URL: https://www.thoughtworks.com/de/radar (zuletzt abgerufen am 14.11.2016)

Oracle: http://www.oracle.com/technetwork/articles/java/deployment-illo-2228199.jpg (zuletzt abgerufen am 04.12.2016)

Docker-Container: http://nebulaworks.com/blog/wp-content/uploads/2016/08/01-docker-container.jpg

TeamCity Logo: http://inedo.com/den/logos/TeamCity-large.png

Bamboo Logo: http://images.google.de/imgres?imgurl=https%3A%2F%2Fwac-cdn.atlassian.com%2Fdam%2Fjcr%3A4f99ae3f-808f-44f1-9647-2b7cb87bb0e6%2Fbamboo_rgb_blue.png%3FcdnVersion%3Ded&imgrefurl=https%3A%2F%2Fde.atlassian.com%2Fsoftware%2Fbamboo&h=192&w=801&tbnid=Bau1epmj0G74HM%3A&vet=1&docid=iyc7l6N9wYnTRM&ei=xPM3WNzmEoqYgAbDpL1I&tbm=isch&client=firefox-b-ab&iact=rc&uact=3&dur=626&page=0&start=0&ndsp=42&ved=0ahUKEwjc2NiavMPQAhUKDMAKHUNSDwkQMwgbKAAwAA&bih=1073&biw=1920#h=192&imgrc=Bau1epmj0G74HM:&vet=1&w=801

Jenkins Logo: https://wiki.jenkins-ci.org/download/attachments/2916393/logo.png?version=1&modificationDate=1302753947000

Git Logo: https://git-scm.com/images/logos/logomark-orange@2x.png

Fehling Christoph, Professor Dr. Frank Leymann: Cloud Computing. URL: http://wirtschaftslexikon.gabler.de/Archiv/1020864/cloud-computing-v9.html (zuletzt abgerufen am 14.11.2016)

URL: http://www.rightscale.com/lp/devops-trends-report

URL: https://jenkinsci.github.io/job-dsl-plugin/# (zuletzt abgerufen am 16.12.2016)

URL: https://github.com/jenkinsci/job-dsl-plugin (zuletzt abgerufen am 16.12.2016)

URL: https://github.com/jenkinsci/jenkins/blob/master/pom.xml (zuletzt abgerufen am 16.12.2016)

URL: https://wiki.jenkins-ci.org/display/JENKINS/Groovy+plugin (zuletzt abgerufen am 16.12.2016)

URL: https://wiki.jenkins-ci.org/display/JENKINS/Jenkins+Script+Console (zuletzt abgerufen am 16.12.2016)

URL: https://wiki.jenkins-ci.org/display/JENKINS/Jenkins+CLI (zuletzt abgerufen am 16.12.2016)

URL: https://github.com/jenkinsci/job-dsl-plugin/wiki/Tutorial---Using-the-Jenkins-Job-DSL (zuletzt abgerufen am 16.12.2016)

URL: https://wiki.jenkins-ci.org/display/JENKINS/Seed+Plugin (zuletzt abgerufen am 16.12.2016)

URL: https://wiki.jenkins-ci.org/display/JENKINS/Job+DSL+Plugin (zuletzt abgerufen am 16.12.2016)

URL: https://wiki.jenkins-ci.org/display/JENKINS/Installing+Jenkins+on+Red+Hat+distributions#InstallingJenkinsonRedHatdistributions-ImportantNoteonCentOSJava (zuletzt abgerufen am 16.12.2016)

URL: http://pkg.jenkins-ci.org/redhat/ (zuletzt abgerufen am 16.12.2016)

URL: https://github.com/jenkinsci/docker (zuletzt abgerufen am 24.11.2016)

Docker Inc.: Official Repository Jenkins. URL: https://hub.docker.com/_/jenkins/ (zuletzt abgerufen am 27.11.2016)

# Anhang

# Installationsanleitung Ansible

## How to für die Installation/Konfiguration von Ansible

- Diese Installationsanleitung dient der Beschreibung zur Konfiguration von Ansible auf zwei verschiedenen Vagrant-Boxen innerhalb des selben Host-Systems.

### 1. Benötigte Installationen

1. Vagrant
   i. Downloaden der latest Version von Vagrant
   ii. Installation von Vagrant
2. Virtualbox
   i. Falls nicht in Vagrant vorhanden, downloaden von Virtualbox
   ii. Installation von Virtualbox

- Die Konfiguration der Ansible Umgebung muss für das Testen zwei mal durchgeführt werden, da eine Kontrollmaschine und eine Maschine zur Konfiguration benötigt werden.
- Schritt 5 und sechs werden nur auf der Kontrollmaschine benötigt.

### 1. Konfiguration der Ansible Umgebung

1. Konfiguration von Vagrant
   i. Verzeichnis für die Box erstellen und hinein wechseln
      a. `mkdir Verzeichnisname`
      b. `cd Verzeichnisname`
   ii. Vagrant Initialisierung des Verzeichnis
      a. `vagrant init`
   iii. Hinzufügen der folgenden zwei Parameter in dem Vagrantfile
      a. config.vm.box="NAME DER BOX"
      b. config.vm.box_url="DOWNLOADLINK"
   iv. Hochfahren und einloggen in die Vagrant Box
      a. Zum Hochfahren der Box `vagrant up`
      b. Zum einloggen in die Box `vagrant ssh`
   v. Updaten der Packages bei Linux (nur auf der Kontrollmaschine)
      a. `sudo yum update`
   vi. Ansible Installation auf der Kontrollmaschine
      a. `sudo yum install epel-release`
      b. `sudo yum install ansible`
      c. `sudo yum install nano`
      d. `sudo yum update`
      e. `sudo ssh-keygen` -> Mit sudo ausführen! wichtig um als remote_user: root agiren zu können
      f. `cat /root/.ssh/id_rsa.pub`
      g. Test durchführen um zu schauen ob der spezifizierte Host erreicht werden kann: `sudo ansible all -m ping`

vii. Konfiguration der Ansible Installation
    a. `sudo nano hosts` -> Hier muss der zu konfigurierende Host hinzugefügt werden: [jenkins] xxx.yyy.zz.aa
    b. `sudo touch jenkins.yml`
    c. `sudo nano jenkins.yml` -> Inhalt für die Provisionierung hinzufügen!
        a. Angabe der Hosts, die provisioniert werden sollen
        b. Angabe über den remoteuser, den man für die Provisionierung nutzen möchte (z.B. root)
        c. Abzuarbeitende Tasks des Playbooks:
            a. Aktualisieren der Packages
            b. Maven installieren
            c. Git installieren
            d. Java installieren
            e. Extra packages for Enterprise Linux (EPEL) installieren
            f. Jenkins Repository hinzufügen
            g. Authentifizierung für das Jenkins Repo durchführen
            h. Jenkins installieren
            i. Jenkins starten (für den Prototypen bis hier)
            j. Installation des Job-DSL Plugin
            k. Installation des Seed Plugin
            l. Installation des Git Plugin
            m. Herunterladen des Jenkins CLI für dessen Benutzung
            n. Neustarten des Jenkins Service! (muss zwischen den einzelnen Plugin Installationen ggf. auch gemacht werden)
            o. Ordner für den Seed Job erstellen
            p. Kopiere die config.xml in den Ordner des Seed Jobs
            q. Erstelle den Jenkins Seed Job
            r. Jenkins neustarten
            s. Seed Job Bauen
            t. Compile-Project Job Bauen -> nach dessen Beenden startet automatisch der Test-Project Job
    d. Ausführung des Playbooks: `sudo ansible-playbook jenkins.yml` !

### 1. Konfiguration von SSH auf beiden Maschinen

1. SSH-Key Generierung durchführen (auf beiden Maschinen oder allen wenn mehr vorhanden)
    i. `ssh-keygen`
    ii. Die weitere Konfiguration ist ausführlich im folgenden Link bescherieben:
        - https://www.digitalocean.com/community/tutorials/how-to-configure-ssh-key-based-authentication-on-a-freebsd-server
2. Innerhalb der Erläuterung von DigitalOcean wird ssh-copy verwendet. Eine manuelle Eintragung des Keys ist hier ebenfalls möglich:
    i. Maschine 1:
        a. `cat ~/.ssh/id_rsa.pub`
        b. Kopiere den ausgegebenen SSH Key
        c. Wechsle in Maschine 2
        d. Wechsle in Maschine 2 in folgendes Verzeichnis `cd ~/.ssh`
        e. Öffne mit einem Editor nach Wahl `authorized_keys`
        f. Einfügen des kopierten SSH Keys
    ii. Maschine 2:
        a. `cat ~/.ssh/id_rsa.pub`
        b. Kopiere den ausgegebenen SSH Key
        c. Wechsle in Maschine 1
        d. Wechsle in Maschine 1 in folgendes Verzeichnis `cd ~/.ssh`
        e. Öffne mit einem Editor nach Wahl `authorized_keys`
        f. Einfügen des kopierten SSH Keys

- Zum Testen von Ansible kann nun beispielsweise git auf der zu konfigurierenden Maschine eingerichtet werden
- Dieser Ansible Befehl kann wie folgt aussehen:

```
ansible all -s -m yum -a "name=git-all state=latest"
```

- all bedeutet, dass alle hosts angesprochen werden sollen
- -s steht hier für sudo
- -m steht für das Module, welches genutzt werden soll
- -a steht für Argumente, die dem angegebenen Module üebrgeben werden müssen

## Installationsanleitung SaltStack

# How-to für die Installation/Konfiguration von Saltstack

- Diese Installationsanleitung dient der Beschreibung zur Konfiguration von Saltstack auf zwei verschiedenen Vagrant-Boxen innerhalb des selben Host-Systems.

### 1. Benötigte Installationen

1. Vagrant
   i. Downloaden der latest Version von Vagrant
   ii. Installation von Vagrant
2. Virtualbox
   i. Falls nicht in Vagrant vorhanden, downloaden von Virtualbox
   ii. Installation von Virtualbox

### 1. Konfiguration der Salt-Master Umgebung auf der Kontrollmaschine

1. Konfiguration von Vagrant
   i. Verzeichnis für die Box erstellen und hinein wechseln
      a. `mkdir Verzeichnisname`
      b. `cd Verzeichnisname`
   ii. Vagrant Initialisierung des Verzeichnis
      a. `vagrant init`
   iii. Hinzufügen der folgenden zwei Parameter in dem Vagrantfile
      a. config.vm.box="NAME DER BOX"
      b. config.vm.box_url="DOWNLOADLINK"
   iv. Hochfahren und einloggen in die Vagrant Box
      a. Zum Hochfahren der Box `vagrant up`
      b. Zum einloggen in die Box `vagrant ssh`
   v. Updaten der Packages bei Linux (nur auf der Kontrollmaschine)
      a. `sudo yum update`
   vi. Installation des Salt-Master
      a. Hinzufügen des offiziellen Salt-Repository `sudo yum install https://repo.saltstack.com/yum/redhat/salt-repo-latest-1.el7.noarch.rpm`
      b. `sudo yum clean expire-cache`
      c. Installation vom Salt-Master `sudo yum install salt-master`
   vii. Suche die folgenden Zeilen innerhalb Datei /etc/salt/master und kommentiere sie aus. (Achtung auskommentieren der Zeilen nach dem example!)
      a. `file_roots: base: - /srv/salt`
      b. Das verzeichnis /salt muss in /srv/ noch erstellt werden -> `sudo mkdir salt`
   viii. Als nächstes muss der Salt-Master neu gestartet werden, damit die Konfiguration greifen kann (erst neustarten wenn die Minion Konfiguration auch abgeschlossen ist. Anschließend beide zusammen neustarten)
      a. `sudo systemctl restart salt-master`

ix. Wenn die Konfiguration bis hierhin abgeschlossen ist (auch für den Minion) kann die Verbindung samt erster Befehle von Master & Minion durchgeführt werden.
    a. Bite beachten: Ab hier muss die Konfiguration des Salt-Minion bis Schritt 7 vollständig sein!
    b. Wenn Punkt 9.1 zutrifft versucht der Minion bereits sich mit dem Master zu verbinden
    c. Um diese Verbindung zu bestätigen muss der Key des Minions innerhalb des Masters akzeptiert werden
        a. Auflisten aller Varianten der Salt-Keys `sudo salt-key -L`
        b. Akzeptieren aller Salt-Keys die beim Master eingegangen sind `sudo salt-key -A`
x. Ist Schritt 9 erfolgreich gewesen, kann man damit beginnen die ersten Salt Befehle auszuführen wie z.B. `sudo salt '*' test.ping`
xi. Für die spätere Ausführung wird in dem Verzeichnis /srv/salt/ ein weiteres benötigt:
    a. `sudo mkdir _modules`
    b. Dieser Name ist von Saltstack aus vorgegeben und wird für die Verwendung eigener Execution Modules benötigt
xii. In dem in Schritt 11. erstellten Verzeichnis wird dann ein eigenes Module abgelegt (für die spätere Installation der Jenkins Plugins.
    a. Das Execution Module was hier abgelegt wird ist eine Python datei `DATEI.py` und besteht hauptsächlich aus dem bereits vorhandenen Execution Module jenkins von saltstack
        a. https://github.com/saltstack/salt/blob/develop/salt/modules/jenkins.py
    b. Jedoch muss man dem custom module auch noch etwas hinzufügen, damit die Plugin Installation funktioniert:
        a. Benötigt wird hier aus dem python-jenkins das Feature zum installieren von Plugins
        b. Hat man den Code hierfür hinzugefügt muss man nur noch den **virtualname** auf ein eigenen ändern z.B. s_jenkins
        c. Innerhalb der Connect Methode dieses Modules sind dann die jenkins url, user und password angegeben. Dort muss man jeweils das vorherige jenkins durch das neue s_jenkins ersetzen, da dieser Name dann auch in der Konfigurationsdatei des Minions angegeben ist.
xiii. Ist man mit der Einrichtung für das Custom Execution Module fertig muss man den Minions diese Änderung mitteilen:
    a. `sudo salt '*' saltutil.sync_all`
    b. Dieser Befehl Teilt den Minions mit, dass ein selbst erstelltes Modul in dem vorgesehenen Ordner _modules/ vorhanden ist
xiv. Hat man allen Minions mitgeteilt, dass es ein Custom Execution Module gibt, kann man das state file mit der Konfiguration des Minion ausführen.

xv. Dazu müssen aber noch zwei Zusätze in das State file eingefügt werden:
    a. Als erstes muss man mit dem von Saltstack eingebauten python-pip die python-jenkins libary installieren
    b. Der zweite Punkt ist, die spezielle Art, wie man ein Execution Module aus einem State file heraus verwendet
    c. Sind die oberen zwei Punkte abgeschlossen kann man nun folgenden Befehl ausführen um den Jenkins auf der Minion Maschine zu Provisionieren und die nötigen Plugins zu installieren
        a. `sudo salt '*' state.apply jenkins`

vii. Salt-Minion mit dem Salt-Master verbinden
   a. In das Verzeichnis /etc/salt wechseln
      a. `cd /etc/salt`
   b. Ändern der Minion Konfigurationsdatei wie folgt
      a. `sudo nano minion`
      b. Suche die folgende Zeile, entferne das beginnende Kommentar und füge folgendes hinzu
         a. `master: IP_ADDRESSE_DES_SALT_MASTER`
         b. Hinzufuegen des folgenden Blocks:
            a.
```
##########################################
######       Jenkins Information      ######
##########################################
s_jenkins:
  user: admin
  password: JENKINS_ADMIN_PASSWORD
  url: http://localhost:8080
```
               a. Der obige Platzhalter JENKINS_ADMIN_PASSWORT muss aus der Datei /var/lib/jenkins/secrets /initialAdminPassword entnommen werden.
         c. Als nächstes muss der Salt-Minion neu gestartet werden, damit die Konfiguration greifen kann
            a. `sudo systemctl restart salt-minion`

### 1. Erstellung des Salt State Files zum aufsetzen des Jenkins Prototyps

1. Als erstes muss mit Saltstack für den Jenkins eine Java Version installiert werden.
2. Danach kann man das Jenkins Repository hinzufügen.
   i. Hierbei muss man die Authentifizierung durchführen.
3. Nachdem man das Jenkins Repository hinzugefügt hat kann man mit dem richtigen Befehl der Jenkins installiert werden.
4. Jetzt ist es möglich den Jenkins Service zu starten und der jenkins kann verwendet werden
5. Mittels der eingerichteten Saltstack Umgebung kann man nun das Jenkins.sls mit folgenden Befehl ausführen, um die definierten Zustände auf dem Minion zu installieren.
   i. `sudo salt '*' state.apply jenkins` -> Hierbei ACHTUNG: das .SLS sollte bei dem der Ausführung weggelassen werden, da sonst ein Error entsteht, der sagt, dass die Datei nicht in env, base vorhanden ist (ist der Ort, für den man in der master Datei die Kommentare entfernt hat)

### 1. Konfiguration der Salt-Minion Umgebung auf der Testmaschine

1. Konfiguration von Vagrant
   i. Verzeichnis für die Box erstellen und hinein wechseln
      a. `mkdir Verzeichnisname`
      b. `cd Verzeichnisname`
   ii. Vagrant Initialisierung des Verzeichnis
      a. `vagrant init`
   iii. Hinzufügen der folgenden zwei Parameter in dem Vagrantfile
      a. config.vm.box="NAME DER BOX"
      b. config.vm.box_url="DOWNLOADLINK"
   iv. Hochfahren und einloggen in die Vagrant Box
      a. Zum Hochfahren der Box `vagrant up`
      b. Zum einloggen in die Box `vagrant ssh`
   v. Updaten der Packages bei Linux (nur auf der Kontrollmaschine)
      a. `sudo yum update`
   vi. Installation des Salt-Minion
      a. Hinzufügen des offiziellen Salt-Repository `sudo yum install https://repo.saltstack.com/yum/redhat/salt-repo-latest-1.el7.noarch.rpm`
      b. `sudo yum clean expire-cache`
      c. Installation vom Salt-Minion `sudo yum install salt-minion`

## Installationsanleitung Docker

# How-to für die Installation/Konfiguration von Docker

- Diese Installationsanleitung dient der Beschreibung zur Konfiguration von Docker inenrhalb einer Vagrant Box.

### 1. Benötigte Installationen

1. Vagrant
   i. Downloaden der latest Version von Vagrant
   ii. Installation von Vagrant
2. Virtualbox
   i. Falls nicht in Vagrant vorhanden, downloaden von Virtualbox
   ii. Installation von Virtualbox

### 1. Konfiguration der Docker Umgebung auf der Testmaschine

1. Konfiguration von Vagrant
   i. Verzeichnis für die Box erstellen und hinein wechseln
      a. `mkdir Verzeichnisname`
      b. `cd Verzeichnisname`
   ii. Vagrant Initialisierung des Verzeichnis
      a. `vagrant init`
   iii. Hinzufügen der folgenden zwei Parameter in dem Vagrantfile
      a. config.vm.box="NAME DER BOX"
      b. config.vm.box_url="DOWNLOADLINK"
   iv. Hochfahren und einloggen in die Vagrant Box
      a. Zum Hochfahren der Box `vagrant up`
      b. Zum einloggen in die Box `vagrant ssh`
   v. Installation des Editors (verwende am liebsten Nano) `sudo yum install nano`
   vi. Updaten der Packages bei Linux
      a. `sudo yum update`
   vii. Hinzufügen des Docker Repository für CentOS
      a. `cd etc/yum.repos.d/`
      b. `sudo touch docker.repo`
      c. `sudo nano docker.repo`
      d. Hinzufügen des folgenden Blocks: `[dockerrepo] name=Docker Repository baseurl=https://yum.dockerproject.org/repo/main/centos/7/ enabled=1 gpgcheck=1 gpgkey=https://yum.dockerproject.org/gpg`
   viii. Installation der Docker Engine
      a. `sudo yum install docker-engine`
      b. `sudo systemctl enable docker.service`
      c. `sudo systemctl start docker`
      d. Testen ob Docker funktionsfähig ist: `docker run --rm hello-world`

**Einrichtung des Jenkins unter Docker innerhalb von Linux CentOS (als Prozess)**

1. Ein Verzeichnis an beliebiger Stelle erstellen (z.B. /home/)
   i. `sudo mkdir jenkins`
2. Erstellen des Dockerfiles
   i. `sudo touch Dockerfile`
      a. Inhalt des Dockerfiles:

      a.
```
FROM centos:7
MAINTAINER Patrick Steinhauer
RUN echo "Starte Installation benoetigter Software..."
RUN yum -y update
RUN yum -y install java-1.8.0-openjdk-devel
RUN yum -y install wget
RUN wget -O /etc/yum.repos.d/jenkins.repo http://pkg.jenkins-ci.org/redhat/jenkins.repo
RUN rpm --import https://jenkins-ci.org/redhat/jenkins-ci.org.key
RUN yum -y install jenkins
CMD java -jar /usr/lib/jenkins/jenkins.war
```

3. Jetzt wird aus dem erstellten Dockerfile ein Image erstellt:
   i. `sudo docker build -t jenkins .`
4. Als letztes muss der Container nur noch gestartet werden:
   i. `docker run jenkins`

## Ansible Skripte

```
<?xml version='1.0' encoding='UTF-8'?>
<project>
  <keepDependencies>false</keepDependencies>
  <properties/>
  <scm class="hudson.scm.NullSCM"/>
  <canRoam>false</canRoam>
  <disabled>false</disabled>
  <blockBuildWhenDownstreamBuilding>false</blockBuildWhenDownstreamBuilding>
  <blockBuildWhenUpstreamBuilding>false</blockBuildWhenUpstreamBuilding>
  <triggers/>
  <concurrentBuild>false</concurrentBuild>
  <builders>
    <javaposse.jobdsl.plugin.ExecuteDslScripts plugin="job-dsl@1.53">
      <scriptText>
          def repoURL = &apos;https://github.com/xKrieger135/Spring-Boot-Example.git&apos;

          job(&apos;Compile-Project&apos;) {
              scm {
                  git(repoURL)
                  }

              triggers {
                  scm(&apos;H/15 * * * *&apos;)
              }

              steps {
                  maven(&apos;compile&apos;)
              }
          }

          job(&apos;Test-Project&apos;) {
              customWorkspace(&apos;/var/lib/jenkins/workspace/Compile-Project&apos;)
                  triggers {
                      upstream(&apos;Compile-Project&apos;, &apos;SUCCESS&apos;)
                  }

                  steps {
                      maven(&apos;test&apos;)
                  }
          }
      </scriptText>
      <usingScriptText>true</usingScriptText>
    </javaposse.jobdsl.plugin.ExecuteDslScripts>
  </builders>
  <publishers/>
  <buildWrappers/>
</project>
```

**config.xml**

```yaml
---
- hosts: jenkins
  remote_user: vagrant
  become_user: root
  become: yes
  become_method: sudo

  tasks:
    - include_vars:
        file: roles/common/vars/control_maschine_vars.yml
    - include: "{{ JENKINS_PREPARATION_TASKS }}"
    - include: "{{ JENKINS_INSTALLATION_TASKS }}"
    - include: "{{ JENKINS_PLUGINS_TASKS }}"
    - include: "{{ JENKINS_CONFIGURATION_TASKS }}"
```

**Site.yml**

```yaml
---
- name: Restart
  service:
    name: jenkins
    state: restarted
```

**Jenkins_handlers.yml**

```yaml
---
- include_vars:
    file: roles/common/vars/jenkins_vars.yml

- include_vars:
    file: roles/common/vars/remote_maschine_vars.yml

- include_vars:
    file: roles/common/vars/control_maschine_vars.yml

- name: Erstellen eines Seed Jobs
  shell: mkdir -p tmp
  args:
    chdir: "{{ USER_HOME_DIRECTORY }}"

- name: Kopiere config.xml in den vorgesehenen Ordner des Seed-Jobs
  copy:
    src: "{{ SEED_JOB_CONFIG_FILE }}"
    dest: "{{ TEMP_LOCATION_FOR_SEED_JOB_CONFIG }}"
    mode: 0777

- pause:
    seconds: 5

- name: Erstellen des Jenkins Jobs
  shell: java -jar jenkins-cli.jar -s "{{ JENKINS_URL_WITH_CREDENTIALS }}"
         create-job "Seed-Job" < "{{ TEMP_SEED_JOB_CONFIG_FILE }}"
  args:
    chdir: "{{ JENKINS_CLI_JAR_DESTINATION }}"

- name: Jenkins Neustart
  service:
    name: jenkins
    state: restarted

- pause:
    seconds: 15

- name: Ausfuehren des Jenkins Seed Jobs.
  shell: java -jar jenkins-cli.jar -s "{{ JENKINS_URL_WITH_CREDENTIALS }}" build Seed-Job
  args:
    chdir: "{{ JENKINS_CLI_JAR_DESTINATION }}"

- name: Ausfuehren der aus dem Seed Job generierten Jobs.
  shell: java -jar jenkins-cli.jar -s "{{ JENKINS_URL_WITH_CREDENTIALS }}" build Compile-Project
  args:
    chdir: "{{ JENKINS_CLI_JAR_DESTINATION }}"
```

(Umbruch in Zeile 26-27 muss weg! Dieser wurde aus Unleserlichkeit eingefügt)

**Jenkins_configuration.yml**

```yaml
---
- include_vars:
    file: roles/common/vars/jenkins_vars.yml

- include_vars:
    file: roles/common/vars/remote_maschine_vars.yml

- name: Hinzufuegen des Jenkins Repository.
  get_url:
    url: "{{ JENKINS_REPOSITORY_URL }}"
    dest: "{{ JENKINS_REPOSITORY_DESTINATION }}"

- name: Durchfuehren der Authentifizierung.
  rpm_key:
    state=present
    key="{{ JENKINS_AUTHENTIFICATION_KEY }}"

- name: Installiere die neuste Version von Java.
  yum:
    name: java
    state: latest

- name: Installiere die neuste Version des Continuous Integration Tools Jenkins.
  yum:
    name: jenkins
    state: latest

- name: Starten des Jenkins-Service.
  service:
    name: jenkins
    state: started
```

**Jenkins_installation.yml**

```yaml
---
- include_vars:
    file: roles/common/vars/jenkins_vars.yml
- include_vars:
    file: roles/common/vars/remote_maschine_vars.yml

- name: Installiere das Jenkins Job DSL Plugin.
  jenkins_plugin:
    name: job-dsl
    params: '{{ jenkins_params }}'
  register: jenkins_plugin_installation

- name: Ueberpruefe, ob ein Neustart fuer den Jenkins erforderlich ist.
  set_fact:
    jenkins_restart_required: yes
  when: jenkins_plugin_installation.changed
  with_items: '{{ jenkins_plugin_installation }}'

- name: Fuehre den Neustart des Jenkins durch,
        wenn Aenderungen lokalisiert werden, die einen Neustart erfordern.
  service:
    name: jenkins
    state: restarted
  when: jenkins_restart_required

- pause:
    seconds: 15

- name: Installiere das Jenkins Seed Plugin.
  jenkins_plugin:
    name: seed
    params: '{{ jenkins_params }}'
  register: seed_plugin

- name: Ueberpruefe, ob ein Neustart fuer den Jenkins erforderlich ist.
  set_fact:
    jenkins_restart_required: yes
  when: seed_plugin.changed
  with_items: '{{ seed_plugin }}'

- name: Installiere das Jenkins Git Plugin.
  jenkins_plugin:
    name: git
    params: '{{ jenkins_params }}'
  register: git_plugin

- name: Ueberpruefe, ob ein Neustart fuer den Jenkins erforderlich ist.
  set_fact:
    jenkins_restart_required: yes
  when: git_plugin.changed
  with_items: '{{ git_plugin }}'

- name: Herunterladen des Jenkins CLI.
  get_url:
    url: "{{ JENKINS_CLI_JAR }}"
    dest: "{{ JENKINS_CLI_JAR_DESTINATION }}"

- name: Fuehre den Neustart des Jenkins durch,
        wenn Aenderungen lokalisiert werden, die einen Neustart erfordern.
  service:
    name: jenkins
    state: restarted
  when: jenkins_restart_required
```

**Jenkins_plugins.yml**

```yaml
---
- include_vars:
    file: roles/common/vars/remote_maschine_vars.yml
- include_vars:
    file: roles/common/vars/control_maschine_vars.yml

- name: Kopiere die Jenkins initialAdminPassword Datei vom remote System
       in die Kontrollmaschine
  fetch:
    src: "{{ JENKINS_INITIAL_ADMIN_PASSWORD }}"
    dest: "{{ CONFIG_FILES }}"
    flat: yes

- include_vars:
    file: roles/common/vars/jenkins_vars.yml

##############################################################################
##############################################################################

# Das Deinstallieren der spaeter installierten Plugins wird benoetigt,
# um einen Fehler aus dem
# jenkins_plugin Module zu vermeiden, der ein erneutes installieren mit
# einem FATAL unterbricht.
- name: Deinstalliere das Jenkins Seed Plugin.
  jenkins_plugin:
    name: seed
    params: "{{ jenkins_params }}"
    state: absent

- name: Deinstalliere das Jenkins Job Plugin.
  jenkins_plugin:
    name: job-dsl
    params: '{{ jenkins_params }}'
    state: absent

```

**Jenkins_preparation.yml**

**Teil 1**

```
- name: Deinstalliere das Jenkins Git Plugin
  jenkins_plugin:
    name: git
    params: '{{ jenkins_params }}'
    state: absent

# Damit die zu deinstallierenden Plugins auch wirklich entfernt werden
# wird ein Neustart benötigt,
# da diese erst dann wirklich deinstalliert werden. Vorher werden sie
# nur vorgemerkt.
- name: Neustarten des Jenkins Service.
  service:
    name: jenkins
    state: restarted

################################################################################
################################################################################

- name: Aktualisiere die bereits vorhandenen Linux Packages.
  yum:
    name=*
    state=latest

- name: Installiere Maven
  yum:
    name: maven2
    state: latest

- name: Installiere Git.
  yum:
    name: git-all
    state: latest

- name: Installiere fuer das CentOS die Extra Packages for Enterprise
      Linux (EPEL).
  yum:
    name: epel-release
    state: latest
```

**Jenkins_preparation.yml**

**Teil 2**

```yaml
---
JENKINS_PREPARATION_TASKS: roles/common/tasks/jenkins_preparation.yml

JENKINS_INSTALLATION_TASKS: roles/common/tasks/jenkins_installation.yml

JENKINS_PLUGINS_TASKS: roles/common/tasks/jenkins_plugins.yml

JENKINS_CONFIGURATION_TASKS: roles/common/tasks/jenkins_configuration.yml

CONFIG_FILES: /etc/ansible/roles/common/files/

JENKINS_VARS: roles/common/vars/jenkins_vars.yml

CONTROL_MASCHINE_VARS: roles/common/vars/control_maschine_vars.yml

REMOTE_MASCHINE_VARS: roles/common/vars/remote_maschine_vars.yml

SEED_JOB_CONFIG_FILE: /etc/ansible/roles/common/files/config.xml
```

**Control-machine-vars.yml**

```yaml
---
jenkins_params:
  url_username: admin
  url_password: "{{lookup('file', '/etc/ansible/roles/common/files/initialAdminPassword') }}"
  url: http://localhost:8080

JENKINS_REPOSITORY_URL: http://pkg.jenkins-ci.org/redhat-stable/jenkins.repo

JENKINS_AUTHENTIFICATION_KEY: https://jenkins-ci.org/redhat/jenkins-ci.org.key

JENKINS_CLI_JAR: http://localhost:8080/jnlpJars/jenkins-cli.jar


JENKINS_URL_WITH_CREDENTIALS: http://"{{ jenkins_params.url_username }}":
                              "{{ jenkins_params.url_password }}"@localhost:8080
```

**Jenkins_vars.yml**

```yaml
---
JENKINS_INITIAL_ADMIN_PASSWORD: /var/lib/jenkins/secrets/initialAdminPassword

JENKINS_REPOSITORY_DESTINATION: /etc/yum.repos.d/jenkins.repo

JENKINS_CLI_JAR_DESTINATION: /var/lib/jenkins

TEMP_LOCATION_FOR_SEED_JOB_CONFIG: /home/vagrant/tmp

USER_HOME_DIRECTORY: /home/vagrant/

TEMP_SEED_JOB_CONFIG_FILE: /home/vagrant/tmp/config.xml
```

**Remote_machine_vars.yml**

## Saltstack Skripte

```
{% set multipleVariables = {
    'jenkins_admin_user':          'jenkinsadmin',
    'jenkins_admin_user_password': 'jenkins',
} %}

Jenkins:
  minion_paths:
    jenkins_initial_admin_password: /var/lib/jenkins/secrets/initialAdminPassword
    jenkins_cli_jar: /var/lib/jenkins/jenkins-cli.jar
    jenkins_files_from_master: salt://files/
    jenkins_seed_job_config: /etc/salt/files/config.xml
    jenkins_files_for_minion: /etc/salt/files/

  urls:
    jenkins_repository_url: http://pkg.jenkins.io/redhat-stable
    jenkins_repository_key_url: https://pkg.jenkins.io/redhat/jenkins.io.key
    jenkins_cli_jar_url: http://localhost:8080/jnlpJars/jenkins-cli.jar

  credentials:
    jenkins_user: {{ multipleVariables.jenkins_admin_user }}
    jenkins_password: {{ multipleVariables.jenkins_admin_user_password }}
    jenkins_admin: admin

  groovy:
    jenkins_create_user_groovy: echo
      'jenkins.model.Jenkins.instance.securityRealm.createAccount("jenkinsadmin", "jenkins")'

s_jenkins:
  user: {{ multipleVariables.jenkins_admin_user }}
  password: {{ multipleVariables.jenkins_admin_user_password }}
  url: http://localhost:8080
```

**Pillar.sls**

```
base:
  '*':
    - pillar
```

**Top.sls**

```
include:
  - packages
  - install
  - plugins
  - jobs
```

**Init.sls**

```
{% from "map.jinja" import variables with context %}

# Hinzufügen des jenkins repository für Redhat / CentOS Systeme
# Wichtige Notiz: die Baseurl ist jene, die man findet wenn man
# folgenden Link aufruft:
# https://pkg.jenkins.io/redhat-stable/jenkins.repo
jenkins-repository:
  pkgrepo.managed:
    - name: jenkins
    - baseurl: {{ variables.jenkins_repository_url }}
    - gpgkey: {{ variables.jenkins_repository_key_url }}
    - gpgcheck: 1

# Nachdem das Jenkins repository hinzugefügt wurde kann der Jenkins
# CI Service installiert und gestartet werden
jenkins:
  pkg.installed:
    - name: jenkins
  service.running:
    - enable: true

Sleep bis Jenkins verfuegbar ist:
  module.run:
    - name: test.sleep
    - length: 20

jenkins-cli:
  file.managed:
    - source: {{ variables.jenkins_cli_jar_url }}
    - name: {{ variables.jenkins_cli_jar }}
    - skip_verify: true

add new user:
  cmd.run:
    - name: initialAdminPassword=$(sudo cat /var/lib/jenkins/secrets/initialAdminPassword);
            {{ variables.jenkins_create_user_groovy }} | java -jar /var/lib/jenkins/jenkins-cli.jar
            -s http://admin:$initialAdminPassword@localhost:8080 groovy =
    - skip_verify: true
```

**Install.sls**

```
{% set pwd = salt['cmd.run']('cat /var/lib/jenkins/secrets/initialAdminPassword')%}

maven:
  pkg.installed:
    - name:
      - maven2

# Dieser Befehl installiert das VCS git auf den angegebenen Minion Maschinen
git:
  pkg.installed:
    - name:
      - git-all

# Mit diesem Befehl wird das Java JDK installiert, welches für eine jenkins
# Installation notwendig ist
Java:
  pkg.installed:
    - name: java-1.8.0-openjdk-devel
# Wird nicht benötigt, war lediglich ein Test
epel-release:
  pkg.installed:
    - name: epel-release

# Hinzufügen des jenkins repository für Redhat / CentOS Systeme
# Wichtige Notiz: die Baseurl ist jene, die man findet wenn man folgenden
# Link aufruft:
# https://pkg.jenkins.io/redhat-stable/jenkins.repo
jenkins-repository:
  pkgrepo.managed:
    - name: jenkins
    - baseurl: http://pkg.jenkins.io/redhat-stable
    - gpgkey: https://pkg.jenkins.io/redhat/jenkins.io.key
    - gpgcheck: 1
```

**Jenkins.sls**

**Teil 1**

```
34
35 # Nachdem das Jenkins repository hinzugefügt wurde kann der Jenkins
36 # CI Service installiert und gestartet werden
37 jenkins:
38   pkg.installed:
39     - name: jenkins
40   service.running:
41     - enable: true
42 #    - reload: true
43 #    - watch:
44 #      - file: /var/lib/jenkins/*
45
46 Sleep bis Jenkins verfuegbar ist:
47   module.run:
48     - name: test.sleep
49     - length: 20
50
51 jenkins-cli:
52   file.managed:
53     - source: http://localhost:8080/jnlpJars/jenkins-cli.jar
54     - name: /var/lib/jenkins/jenkins-cli.jar
55     - skip_verify: true
56
57 # http://stackoverflow.com/questions/17716242/creating-user-in-jenkins-via-api
58 add new user:
59   cmd.run:
60     - name: echo 'jenkins.model.Jenkins.instance.securityRealm.createAccount("admin2", "admin")'
61             | java -jar /var/lib/jenkins/jenkins-cli.jar -s http://admin:{{pwd}}@localhost:8080
62             groovy =
63     - skip_verify: true
64
65 seed plugin installieren:
66   module.run:
67     - name: s_jenkins.install_plugin
68     - m_name: seed
69
```

**Jenkins.sls**

**Teil 2**

```
70  pause2:
71    module.run:
72      - name: test.sleep
73      - length: 60
74
75  Jenkins-Neustart2:
76    cmd.run:
77      - name: service jenkins restart
78
79  Sleep after seed plugin installation:
80    module.run:
81      - name: test.sleep
82      - length: 20
83
84  git plugin installieren:
85    module.run:
86      - name: s_jenkins.install_plugin
87      - m_name: git
88
89  pause3:
90    module.run:
91      - name: test.sleep
92      - length: 30
93
94  Jenkins-Neustart3:
95    cmd.run:
96      - name: service jenkins restart
97
```

**Jenkins.sls**
**Teil 3**

```
Warten bis Jenkins neugestartet ist:
  module.run:
    - name: test.sleep
    - length: 20

/home/vagrant/:
  file.recurse:
    - source: salt://files/

jenkins seed job generieren:
  module.run:
    - name: s_jenkins.create_job
    - m_name: seed
    - config_xml: /home/vagrant/config.xml

Build jenkins seed job:
  module.run:
    - name: s_jenkins.build_job
    - m_name: seed
```

**jenkins.sls**

**Teil 4**

```
{% from "map.jinja" import variables with context %}

copy files for jenkins to minion:
  file.recurse:
    - name: {{ variables.jenkins_files_for_minion }}
    - source: {{ variables.jenkins_files_from_master }}

jenkins seed job generieren:
  module.run:
    - name: s_jenkins.create_job
    - m_name: seed
    - config_xml: /etc/salt/files/config.xml

Build jenkins seed job:
  module.run:
    - name: s_jenkins.build_job
    - m_name: seed
```

**Jobs.sls**

```
# Innerhalb dieser Datei werden Variablen festgelegt,
# die in dem state file fuer den Jenkins verwendet werden.

{% set variables = {
    'jenkins_admin':                                salt['pillar.get']
                    ('Jenkins:credentials:jenkins_admin', 'admin'),

    'path_to_jenkins_initial_admin_password':       salt['pillar.get']
                ('Jenkins:minion_paths:jenkins_initial_admin_password',
                '/var/lib/jenkins/secrets/initialAdminPassword'),

    'jenkins_files_from_master':                    salt['pillar.get']
    ('Jenkins:minion_paths:jenkins_files_from_master', 'salt://files/'),

    'jenkins_seed_job_config':                      salt['pillar.get']
    ('Jenkins:minion_paths:jenkins_seed_job_config', '/etc/salt/files/config.xml'),

    'jenkins_files_for_minion':                     salt['pillar.get']
    ('Jenkins:minion_paths:jenkins_files_for_minion', '/etc/salt/files/'),

    'jenkins_repository_url':                       salt['pillar.get']
    ('Jenkins:urls:jenkins_repository_url', 'http://pkg.jenkins.io/redhat-stable'),

    'jenkins_repository_key_url':                   salt['pillar.get']
    ('Jenkins:urls:jenkins_repository_key_url', 'https://pkg.jenkins.io/redhat/jenkins.io.key'),

    'jenkins_cli_jar_url':                          salt['pillar.get']
    ('Jenkins:urls:jenkins_cli_jar_url', 'http://localhost:8080/jnlpJars/jenkins-cli.jar'),

    'jenkins_user':                                 salt['pillar.get']
    ('Jenkins:credentials:jenkins_user', 'jenkinsadmin'),

    'jenkins_password':                             salt['pillar.get']
    ('Jenkins:credentials:jenkins_password', 'jenkins'),

    'jenkins_create_user_groovy':                   salt['pillar.get']
    ('Jenkins:groovy:jenkins_create_user_groovy'),
} %}
```

**Map.jinja**

```
maven:
  pkg.installed:
    - name:
      - maven2

git:
  pkg.installed:
    - name:
      - git-all

Java:
  pkg.installed:
    - name: java-1.8.0-openjdk-devel
```

**Packages.sls**

```
seed plugin installieren:
  module.run:
    - name: s_jenkins.install_plugin
    - m_name: seed

pause2:
  module.run:
    - name: test.sleep
    - length: 60

Jenkins-Neustart2:
  cmd.run:
    - name: service jenkins restart

Sleep after seed plugin installation:
  module.run:
    - name: test.sleep
    - length: 20

git plugin installieren:
  module.run:
    - name: s_jenkins.install_plugin
    - m_name: git

pause3:
  module.run:
    - name: test.sleep
    - length: 30

Jenkins-Neustart3:
  cmd.run:
    - name: service jenkins restart

Warten bis Jenkins neugestartet ist:
  module.run:
    - name: test.sleep
    - length: 20
```

**Plugins.sls**

```
epel-release:
  pkg.installed:
    - name: epel-release

python-pip:
  pkg.installed:
    - name: python2-pip
    - reload: True

python-jenkins:
  pip.installed:
    - name: python-jenkins
    - reload_modules: True
```

**Python.sls**

```
def _connect():
    '''
    Return server object used to interact with Jenkins.
    :return: server object used to interact with Jenkins
    '''
    jenkins_url = __salt__['config.get']('s_jenkins.url') or \
                  __salt__['config.get']('s_jenkins:url') or \
                  __salt__['pillar.get']('s_jenkins.url')

    jenkins_user = __salt__['config.get']('s_jenkins.user') or \
                   __salt__['config.get']('s_jenkins:user') or \
                   __salt__['pillar.get']('s_jenkins.user')

    jenkins_password = __salt__['config.get']('s_jenkins.password') or \
                       __salt__['config.get']('s_jenkins:password') or \
                       __salt__['pillar.get']('s_jenkins.password')

    if not jenkins_url:
        raise SaltInvocationError('No Jenkins URL found.')

    return jenkins.Jenkins(jenkins_url,
                           username=jenkins_user,
                           password=jenkins_password)

```

**Salt_jenkins.py**

**(Das veränderte Salt Jenkins module. Hier die Anpassung der Pillar Konfiguration.)**

```
394 def install_plugin(name):
395
396     server = _connect()
397
398     plugin_exists = plugin_installed(name)
399
400     if plugin_exists:
401         return 'The plugin which should be installed already exists!'
402     else:
403         restart_needed = server.install_plugin(name)
404
405         time.sleep(5)
406
407         if restart_needed:
408             return 'Plugin: ' + name + ' installed. Restart needed: ' + str(restart_needed)
409         else:
410             return 'Plugin: ' + name + ' installed. Restart needed: ' + str(restart_needed)
```

**Salt_jenkins.py**

**(Das veränderte Salt Jenkins module. Hier die selbst erstellte Funktion)**

## Docker Skripte

```
FROM jenkins
MAINTAINER Patrick Steinhauer

#################### Variante, welche aus der jenkins Docker Lösung hervorgeht ####################
# Diese Variante schaltet jedoch die eingerichtete Security vom Jenkins aus!                     #
##################################################################################################
                                                                                                  #
#RUN mkdir /usr/share/jenkins/ref/jobs/; mkdir /usr/share/jenkins/ref/jobs/seed                   #
#COPY config.xml /usr/share/jenkins/ref/jobs/seed/config.xml                                      #
                                                                                                  #
##################################################################################################

#################### Weitere Variante, die ebenfalls funktioniert ################################
# Schaltet ebenfalls die Security ab!                                                            #
##################################################################################################
                                                                                                  #
#RUN mkdir -p /var/jenkins_home/jobs/seed/                                                        #
#COPY config.xml /var/jenkins_home/jobs/seed/                                                     #
                                                                                                  #
##################################################################################################

# Verwenden des root user um die Installation von Maven durchzuführen
USER root

# Maven muss installiert werden, da die Maven Commandos von Jenkins sonst nicht funktionieren
RUN apt-get update && apt-get install -y maven
RUN mkdir /usr/share/jenkins/ref/init.groovy.d/job-dsl/

COPY compile_and_test_project.groovy /usr/share/jenkins/ref/init.groovy.d/job-dsl/
COPY seed.groovy /usr/share/jenkins/ref/init.groovy.d/

# Hier wird der jenkins user wieder genutzt (Vorgehen von der Jenkins-Docker Doku)
# https://github.com/jenkinsci/docker
USER jenkins

RUN /usr/local/bin/install-plugins.sh maven git seed
```

**Dockerfile**

```groovy
import hudson.model.*
import jenkins.model.*;
import javaposse.jobdsl.plugin.*;

Thread.start {
    // In Anlehnung an diesen Code (einiges funktionierte jedoch
    // nicht mehr und musste ausgetauscht werden!
    // https://ogavrisevs.github.io/2016/02/22/docker-in-scale/

    sleep 15000
    def jenkinsInstance = Jenkins.getInstance()
    def seedJob = new FreeStyleProject(jenkinsInstance, "seed")

    seedJob.setCustomWorkspace("/usr/share/jenkins/ref/init.groovy.d/job-dsl/")

    def ExecuteDslScripts.ScriptLocation scriptlocation =
    new ExecuteDslScripts.ScriptLocation();
    scriptlocation.setValue('false');
    scriptlocation.setTargets("*.groovy");
    scriptlocation.setScriptText(null);

    def ExecuteDslScripts execute = new ExecuteDslScripts(scriptlocation);

    seedJob.buildersList.add(execute)

    jenkinsInstance.add(seedJob, seedJob.getName());
    jenkinsInstance.reload();

    seedJob.schedule();
}
```

**Seed.groovy**

```
def repoURL = 'https://github.com/xKrieger135/Spring-Boot-Example.git'

job('Compile-Project') {
    scm {
        git {
            remote {
                url(repoURL)
            }

            branch('master')
        }
    }

    triggers {
        scm('H/15 * * * *')
    }

    steps {
        maven('compile')
    }
}

job('Test-Project') {
    customWorkspace('/var/jenkins_home/workspace/Compile-Project')
    triggers {
        upstream('Compile-Project', 'SUCCESS')
    }

    steps {
        maven('test')
    }
}
```

**Compile_and_test_project.groovy**

Wichtig: Alle Umbrüche in den Skripten sind nur gemacht worden, weil die Zeilen sonst zu lang gewesen wären! Diese müssen wieder entfernt werden.

# Versicherung über Selbstständigkeit

*Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.*

*Hamburg, den 20. Februar 2017* _____