DHBW
Duale Hochschule
Baden-Württemberg
Mannheim

Eine Evaluation von Docker

als Ersatz herkömmlicher Virtualisierungskonzepte im PLM Umfeld

Praxisarbeit

für die Prüfung der T2000 Praxisarbeit

Studiengang *Angewandte Informatik*

Duale Hochschule Baden-Württemberg Mannheim

von

Lukas Retschmeier

Abgabedatum:	28. August 2017
Bearbeitungszeitraum:	12 Wochen
Matrikelnummer, Kurs:	1339518, TINF15/AI-BI
Ausbildungsfirma:	Atos Information Technology GmbH
Betreuer der Ausbildungsfirma:	Dipl.-Phy. Norbert Hranitzky

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich meine Praxisarbeit mit dem Thema

> *Eine Evaluation von Docker als Ersatz herkömmlicher Virtualisierungskonzepte im PLM Umfeld*

selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Mannheim, den 16. August 2017

_____

Lukas Retschmeier

Abstract

German *Containerisierung* verändert aktuell die Art, wie Software verteilt und virtualisiert wird grundlegend. Hierbei wird die schwergewichtige Virtualisierung mittels Hypervisor durch eine leichtgewichtige Betriebssystemvirtualisierung abgelöst. Ein Vorreiter dieser Technologie ist *Docker*. Es stellt sich die Frage, inwiefern dies die Arbeit in Projekten im Product Lifecycle Management (PLM) Umfeld des Unternehmens *Atos Information Technology GmbH* erleichtern und in welchen möglichen Einsatzgebieten dies nützlich sein kann. Hierfür werden die Komponente der PLM Applikationen *IBM Websphere* und *Siemens Teamcenter 11.2* auf Docker Container verteilt und das Ergebnis evaluiert.

English *Containerization* is currently changing the manner on *how* software is being distributed and virtualized. Therefore, the heavy virtualisition using Hypervisor is replaced by lightweighted Operating System Virtualization. The most popular pioneer is called *Docker*. It raises the question on *how* this can improve the work of projects in the PLM Environment of *Atos IT Solutions and Services GmbH*. Therefore, Siemens' PLM software *Teamcenter* is distributed to Docker Containers and the results evaluated.

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

AIS	Atos It Solutions and Services GmbH
BIOS	Basic Input Output System
CLI	Command Line Interface (Kommandozeile)
DBMS	Database Management System
DHBW	Duale Hochschule Baden-Württemberg
FCC	FMS Client Cache
FMS	File Management System
FSC	FMS Server Cache
GUI	Graphical User Interface
JDK	Java Development Kid
JRE	Java Runtime Environment
J2EE	Java Enterprise Edition
OCI	Open Container Initiative
OOTB	Out-Of-The-Box
PLM	Product Lifecycle Management
RC	Rich Client
SLES	SUSE Linux Enterprise Server
SOA	Serviceorintierte Architektur
SSH	Secure Shell

RAM	Rapid Access Memory
RDP	Remote Desktop Prokoll
TC	Teamcenter
UFS	Union File System
VBox	Oracle Virtual Box
VNC	Virtual Network Connect
VM	Virtual Machine
VMM	Virtual Machine Monitor
LCX	Linux Containers
2T-RC	2Tier Rich Client

1 Einleitung

Container haben bereits einmal die Welt revolutioniert. 1966 haben die ersten Containerschiffe, die die Fracht eines Schiffes in einen genormten Metallbehälter packten erste europäische Häfen angefahren und vorerst wenig aufsehen erregt. [14] Wurden sie von Fachleuten zuerst müde belächelt, da diese den großen Mehrwert nicht erkennen konnten, hat sich dies doch in kürzester Zeit verändert.

Der Vorteil entgegen einem unstrukturiertem Beladen besteht nämlich nun darin, dass die Ware an jedem Hafen *automatisiert* in einer *hohen Geschwindigkeit* abgefertigt werden kann. Ein Containerumschlag von etwa 132 Millionen Containern weltweit und einer durch die Globalisierung bedingte Steigerung auf prognostiziert 174 Millionen im Jahr 2020 [10, vgl.] spricht eine deutliche Sprache: Container haben den globalen Warenaustausch gravierend verändert und beschleunigt.

In letzter Zeit deutet sich an, dass eine ähnliche Revolution auch im IT Umfeld Einzug nehmen wird. *Containerisierung* beschreibt eine Verlagerung von Anwendungen in sogenannte Container. Diese stellen eine isolierte Laufzeitumgebung dar, die wie Bausteine miteinander kombiniert werden können. Diese sind, ähnlich der vor einem halben Jahrundert eingeführten Fracht, hochportabel.

Damit wird es die Art, wie Anwendungen entwickelt und ausgeliefert werden nachhaltig verändern. Im Mai 2016 wurden 310 Entwickler von *DevOps.com* und *ClusterHQ* zum Einsatz von Containertechnologien in ihrem Unternehmen befragt. 79% geben an, dass Sie die Technologie in irgend einer Weise nutzen, bereits 76% davon nutzen sie auch in Produktionsumgebungen - eine Steigerung um 38% zur Umfrage des Vorjahres. [2]

Die Open Source Platform *Docker* gilt als ein Vorreiter dieses Wandels. 94% der Teilnehmer geben an, dass ihr Unternehmen auf diese Container Technologie setzt.

Es ist daher sehr interessant zu untersuchen, wie bestehende Applikationen mit Docker Containern zusammenspielen. In dieser Arbeit wird die Siemens Software *Teamcenter* in Container aufgespalten.

1.1 Motivation

Diese Arbeit ist im Rahmen einer Praxisphase in der deutschen PLM Abteilung des Unternehmens *Atos Information Technology* entstanden und wird als T2000 einer Prüfungskommision der *Dualen Hochschule Baden-Württemberg* Mannheim vorgelegt.

Mithilfe von *Docker* können Deployment Prozesse in Projekten optimiert und speziell im Umfeld der Continous Integration neue Projekte gewonnen werden.

1.2 Ziel der Arbeit

Das Ziel dieser Arbeit ist es, nach einer Einführung in die *Docker*-Thematik zu skizzieren, wie sich *Teamcenter 11* in eine Orchestrierung von Containern aufspalten lässt. Abschließend wird untersucht, welche Vorteile dies speziell für die deutsche PLM Abteilung der *Atos Information Technology GmbH* (*AIT*) mit sich bringt.

1.3 Aufbau der Arbeit

Den Leser erwarten in dieser Arbeit drei größere Blöcke:

1. Eine **Exposition**, die eine grundlegende Einführung in die Virtualisierung und speziell in die Dockertechnologie bereitstellt
2. Einer **Durchführung**, die eine technische Vorgehensweise beschreibt, wie *Teamcenter 11* in Docker Container verlagert wird
3. Einen **Diskussionsteil**, der anhand verschiedener Kriterien klären soll, welche Vorteile diese Technik nun für das PLM Umfeld mit sich bringt

2 Theorie

Im folgenden Abschnitt werden die theoretischen Grundsteine gelegt, die im weiteren Teil vom Leser vorrausgesetzt werden.

2.1 Virtualisierung

> "Who in the world am I? Ah, that's the great puzzle.
>
> — Alice from *Alice aus Wonderland*,
> Lewis Carroll

Es ist nicht übertrieben zu sagen, dass jeder, der bereits einen Computer oder ein Smartphone in der Hand gehalten hat, bereits mit Virtualisierungstechniken in Berührung gekommen ist. So werden beispielsweise die meisten Webseiten auf *virtuellen Servern* oder iOS Apps in *Sandboxen* betrieben. Bereits auf Hardwareebene gibt es das Konzept der *Speichervirtualisierung*, welche physikalische Speicherplatzgrenzen dynamisch macht. Es gibt unzählige Einsatzgebiete dieses Schlagwortes.

Doch was versteht man unter diesem Begriff überhaupt? Diese Frage wollen wir in diesem Kapitel klären und man wird an einigen Stellen merken, dass man, genau wie Alice in Carolls Klassiker nicht mehr so ganz genau weiß, wo man sich gerade befindet...

Definition 2.1.1 (Virtualisierung und Virtuelle Maschine) *Unter Virtualisierung versteht man die Abstraktion physikalisch vorhandener Hardware"[5, S.9] oder Software. Diese zusammengefasst bezeichnet man als virtuelle Maschine. [5, vgl. S.9]*

Hierbei gibt es nun zwei grundsätzliche Ansätze: **Partitionierung** und **Aggregation**, wobei letzterer für diese Arbeit nicht relevant ist.

Definition 2.1.2 (Partitionierung und Aggregation) *Partitionierung beschreibt eine Aufteilung von Komponenten ähnlich wie in der Mathematik in mehrere vollständige Systeme, welche eigenständig angesprochen werden können. Bei der Aggregation ist dies genau das Gegenteil: Mehrere Komponente werden zu einem großen ganzen System zusammengezogen.*[1]

Durch Partitionierung ist es möglich, auf einer physischen Hardware mehrere emulierte zu nutzen. Dafür spaltet man die vorhandenen Ressourcen wie einen Kuchen auf verschiedene Abnehmer auf. Aggregation kann beispielsweise dafür genutzt werden, mehrere Rechner zu *einem* logischen Betriebssystem zusammenzufassen.

2.1.1 Arten der Virtualisierung

Grob lässt sich die Virtualisierung in drei Arten unterteilen: einer *Vollständigen-*, *Para-* und *Betriebssystem*virtualisierung. Diese unterscheiden sich voneinander in dem Grad, wie stark virtualisiert wird; mit anderen Worten: Wie stark die einzelnen Komponenten von der Hardware abgekapselt werden und auf eine Abstraktionsschicht zugreifen. Diese werden im Folgenden näher beleuchtet.

Definition 2.1.3 (Gast- und Hostsysteme) *Unter einem Hostsystem verstehen wir fortan genau das System, auf welchem eine Virtuelle Maschine läuft. Als Gastsystem bezeichnen wir die Virtuelle Maschine selbst.*

Definition 2.1.4 (Kernel) *Der Kernel stellt grundlegende Dienste für die Interaktion mit der Hardware zur Verfügung und stellt eine direkte Schnittstelle zu dieser dar.*

Der Kernel zählt zum Hauptbestandteil eines Betriebssystems. Er kümmert sich um Aufgaben wie die *Speicherverwaltung*, *Prozessverwaltung*, *Geräteverwaltung* und die *Verwaltung der Dateisysteme*.

Definition 2.1.5 (Virtual Machine Monitor/Hypervisor) *Der Virtual Machine Monitor (VMM) ist diejenige Schnittstelle, die das Gastsystem mit den Hostressourcen verbindet.*

[1] Verteilte Systeme, z.B: *Apache Hadoop*

Je nachdem, wie stark diese Schicht auf die physikalischen Ressourcen zugreifen darf, spricht man von einem *Typ-I-VMM* (Läuft direkt auf der Hardware) und einem *Typ-II-VMM* (Benötigt ein Host Betriebssystem). Bekannte Vertreter hierfür sind *VMWare ESX* oder *Microsoft Hyper-V* für Typ-I und *VMWare Workstation* oder *Sun Virtual Box* für Typ-II [5, vgl. S.10].

2.1.1.1 Vollständige Virtualisierung

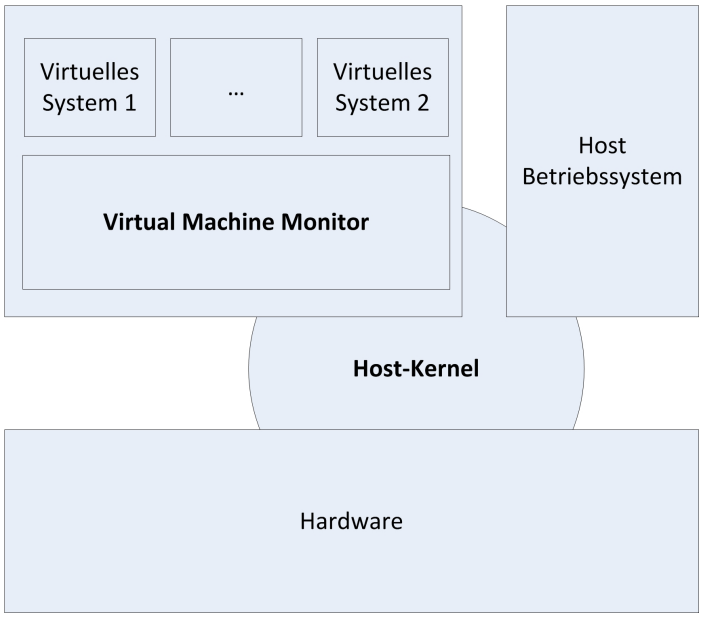


Abbildung 2.1: Vollständige Virtualisierung

Definition 2.1.6 (Vollständige Virtualisierung) *Eine Virtuelle Maschine mit einer virtuellen Nachbildung der gesamten Hardwareumgebung bezeichnen wir als eine Vollständige Virtualisierung.*

Bei der vollständigen Virtualisierung[2] besteht kein direkter Kontakt zwischen der Hardware und den virtualisierten Gastkomponenten. Wie in 2.1 zu sehen ist, läuft das gesamte Betriebssystem mit allen Komponenten parallel zum Hostsystem und wird von einem Type-II-VMM gesteuert. Dem Gastsystem werden alle

[2] Auch: *Echte* oder *Voll*-Virtualisierung

Hardwarekomponente von BIOS bis RAM virtualisiert, so dass dieses von der physischen Hardware komplett losgelöst ist. Infolgedessen muss das Gastsystem alle Komponenten, die es braucht, selbst verwalten.

2.1.1.2 Paravirtualisierung

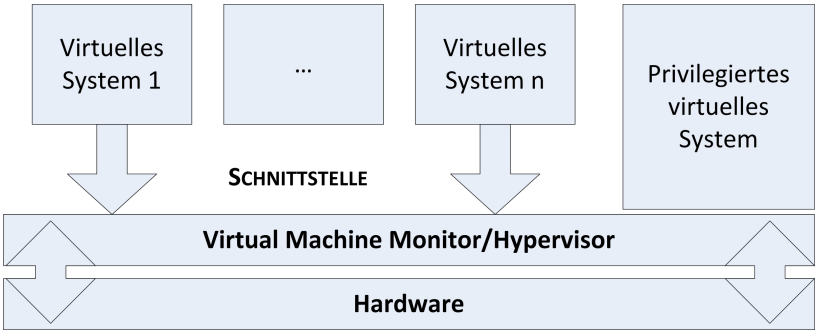


Abbildung 2.2: Paravirtualisierung

Definition 2.1.7 (Paravirtualisierung) *Bei Bereitstellung einer eigenen Schnittstelle für den Zugriff einer Virtuelle Maschine auf die Hardware spricht man von einer Paravirtualisierung.*

Bei der Paravirtualisierung (von lat. pars: *Teil*) weiß die VM, dass sie nur auf virtuelle Hardware zugreift. Zu diesem Zweck stellt die VMM eine gesonderte Schnittstelle zur Verfügung, die durch eine explizite Anpassung des Kernels der VM angesprochen werden kann. Folglich muss nicht mehr für jede Virtuelle Maschine die Hardware einzeln virtualisiert werden, sondern alle Maschinen greifen, wie in 2.2 zu sehen ist quasi auf die gleichen virtuellen Ressourcen, die vom Hypervisor organisiert werden zu. Das Problem ist, dass man für eine Modifikation des Kernels auch auf den Sourcecode zugreifen muss, was zwar bei Open-Source Linux Distributionen wie *Ubuntu* keines, jedoch bei kommerzieller Software wie *Microsoft Windows* ein Problem darstellt, da der Quellcode von den Unternehmen nicht veröffentlicht wird.

2.1.1.3 Betriebssystemvirtualisierung

Definition 2.1.8 (Betriebssystemvirtualisierung nach [15]) *Von Betriebssystemvirtualisierung wird gesprochen, wenn die Virtualisierungsebene als Anwendung des Be-*

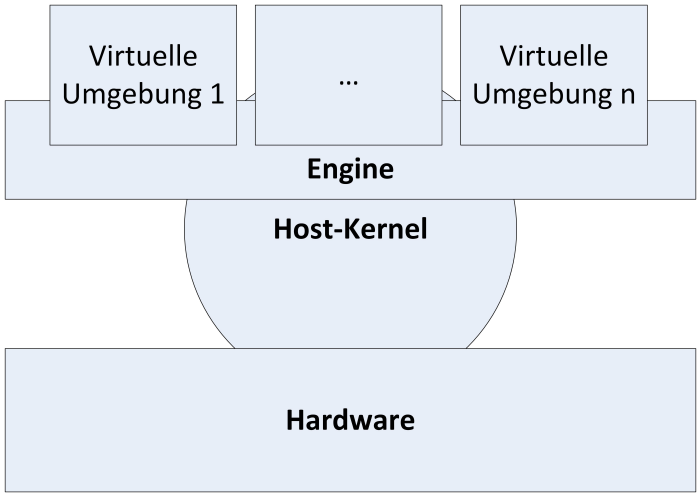


Abbildung 2.3: Betriebssystemvirtualisierung

triebssystems läuft und die virtuellen Umgebungen den Kernel des Hauptsystemes mitnutzen und teilen.

Einen etwas anderen Weg geht die Betriebssystemvirtualisierung. Bei dieser werden wie in Abbildung 2.3 dargestellt, virtuelle Umgebungen geschaffen, die auf Hauptressourcen des Hostsystemes zugreifen dürfen. Diese werden meist Container genannt und bestehen nur aus den Daten, die jede individuelle Umgebung benötigt. Somit können auch alle Bibliotheken des Hauptsystemkernels mitbenutzt werden und müssen nicht extra dupliziert werden. Jedoch sind die Prozesse in einem Container natürlich komplett isoliert. Der kleine Unterschied, dass wir nun von sogenannten *Umgebungen* sprechen, ist sehr wichtig, denn es wird nicht mehr das vollständige System emuliert. Vielmehr werden in das bestehende ganze Dateisysteme getrennt voneinander eingebunden und die Ressourcen, die auch dem Hostsystem zur Verfügung stehen genutzt. Durch diesen minimalen Virtualisierungsaufwand liegen die Verluste[3] lediglich bei einem bis drei Prozent. [5, vgl. S.17]

Diese Technik schränkt das System jedoch soweit ein, dass auf den Containern nur ein Betriebssystem, welches auf dem gleichen Kernel wie das Hostsystem basiert laufen kann. Eine Engine ist dafür verantwortlich, die virtuellen Umgebungen zu verwalten, die Schnittstelle zum Systemkernel zur Verfügung zu stellen und

[3] Auch: Virtualisierungsschwund

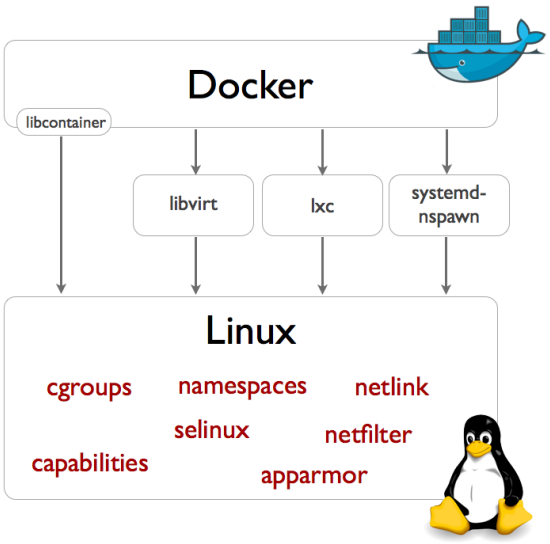


Abbildung 2.4: Docker und die Containertechnologie

sorgt auch dafür, dass die einzelnen Container isoliert werden, also die Inhalte eines Container nicht von außen manipuliert werden können.

Die *Dockertechnologie*, die in dieser Arbeit noch Wichtigkeit erlangen wird, basiert auf den Prinzipien der Betriebssystemvirtualisierung.

Definition 2.1.9 (Linux Containers (LXC)) *Eine Suite von Systemfunktionen für Linux Kernel, die speziell für Containerisierung und die gemeinsame Nutzung von Systemressourcen gedacht sind.*

LXC is ein freies Projekt für eine Containerverwaltung auf einem Linux Kernel. Es nutzt bereits bestehende Funktionen des Linux Kernels wie *namespaces*, *cgroups*, *netlink*, *apparmor* usw. aus, um eine Containerumgebung zur Verfügung zu stellen.[23, Vgl.] Bis Version 0.9 nutzte auch Docker LXC, bis sie mit *libcontainer* LXC durch eine eigene Entwicklung ersetzen.[12] Abbildung 2.4 deutet an, wie das noch näher zu betrachtende Docker auf die Containerisierungsfunktionen, mitunter auch LXC, zurückgreift, um eine Betriebssystemvirtualisierung zu erreichen.

2.1.2 Ziele einer Virtualisierung

Folgende Ziele wollen mithilfe von Virtualisierungstechniken erreicht werden:

1. **Sicherheit** durch Isolation der einzelnen Anwendungsumgebungen
2. Nutzung von **mehreren verschiedenen** Betriebssystemen auf einem Rechner, vor allem für Testentwicklungssysteme
3. **Bessere Ausnutzung** bestehender Ressourcen, durch hohe Auslastung der Systemkomponenten und damit **Energiesparungen**
4. **Schnelles Einrichten** von neuen virtuellen System
5. Einfaches **Backup** und **Verteilen** von Systemen
6. Lauffähigkeit **alter**, nicht mehr unterstützter Applikationen durch Virtualisierung älterer Systeme

2.2 Docker

Abbildung 2.5: Offizielles Docker Logo

Dieser Abschnitt beschäftigt sich nach einer kurzen Einführung mit den grundlegenden Konzepten der Dockertechnologie.

2.2.1 Einführung in Docker

Docker etabliert sich aktuell zum de-facto Standard der Softwareverbreitung. Mit dieser Technologie ist es möglich, fast beliebige Applikationen in kürzester Zeit zu starten und zu stoppen. Docker unterstützt die LXC Technik des Linux Kernels

und einige eigene Weiterentwicklungen um per Betriebssystemvirtualisierung Anwendungen in Container zu verlagern und parallel zu betreiben. Durch die Open-Source Lizenz [21, S.7] ist es jedem möglich, Docker im privaten, als auch im kommerziellen Umfeld zu benutzen. Das englische Wort *docker* stammt zum einen aus dem nautischen und bedeutet so viel wie "Hafenarbeiter". Man kann es aber auch vom englischen *to dock: anlegen*, was etymologisch aus dem Lateinischen *docere: belehren/instruieren* hervorging ableiten. Beide Erklärungen passen sehr gut in Nachfolgende Metapher, die die Technologie beschreibt.

Die Metapher vom Hafen Man stelle sich einen Hafen vor, bei dem jeden Tag viele Schiffe ankommen und wieder wegfahren. Bis auf ein paar kleine Besonderheiten bei bestimmten Schiffstypen, können alle Schiffe die gleichen Stellen ansteuern. Sie brauchen dazu einfach eine Zuordnung, wo sie hinfahren sollen und können sich dann mit einem Tau ganz einfach festhängen. Auch wenn die Schiffe an sich sehr verschieden sein können - es gibt von kleinen Containerschiffen bis riesigen Frachtern alles, was man sich vorstellen kann - nutzen diese alle die gleiche Schnittstelle am Hafen: Die Anlegestelle. Die Hafenverwaltung selber koordiniert die Schiffe und die Zuweisungen zu den Anlegestellen.

Die Frachtcontainer, die ein Schiff beladen hat können problemlos auf ein Anderes verlagert und auf diesem weiter transportiert werden. Auch besitzt jedes Schiff selbst einen individuellen Bau- und Befrachtungsplan, mit dem es die fleißigen Mitarbeiter an einem Port nachbauen und beladen können.

Die Schiffe kann man als Applikationen mit mehreren Komponenten sehen. Diese bestehen aus der Fracht, die sie transportieren und sind die zentralen Bestandteile des Schiffes. Sie werden nachfolgend auch *Container* genannt und stellen aus der Sicht von Docker weitere Teilkomponenten da, die zusammen ein großes Ganzes ergeben. Jeder Container hat seine spezifische Aufgabe und kann mit anderen Containern in Verbindung stehen. So gehört beispielsweise zu einem Webserver häufig noch eine enge Verbindung zu einer Datenbank hinzu. Die abstrahierende Schicht namens *Docker Engine* - die “Hafenaufsicht” - kann Applikationen mithilfe eines Planes eindeutig zusammensetzen lassen. Auch die Container selbst können nach gewissen Mustern aufgesetzt werden und sind wiederverwendbar, d.h sie können in den verschiedensten Schiffen erneut verbaut werden. Solche wiederverwendbare Container nennt Docker *Images*. Somit ist es möglich, beispielsweise

einen Datenbankcontainer mit den verschiedensten Applikationen zu verbinden. Die Struktur der Datenbank selbst bleibt ja in jedem Fall gleich, nur der Aufbau dieser unterscheidet sich von Anwendung zu Anwendung. Die Hafenarbeiter sind äquivalent mit den Ressourcen, die ein Hostsystem zur Verfügung hat.

Zusätzlich ist die Docker Engine eine Verwaltungsinstanz. So ist diese dafür zuständig, Container und Schiffe zu starten und stoppen, umbauen zu lassen, zu zerstören oder zu kopieren. Auch fremde Schiffe können den Hafen einfach anfahren. Die Applikationen lassen sich daher von der eigentlichen Dockerumgebung loslösen und können auf jeder anderen *Docker Engine* installiert werden.

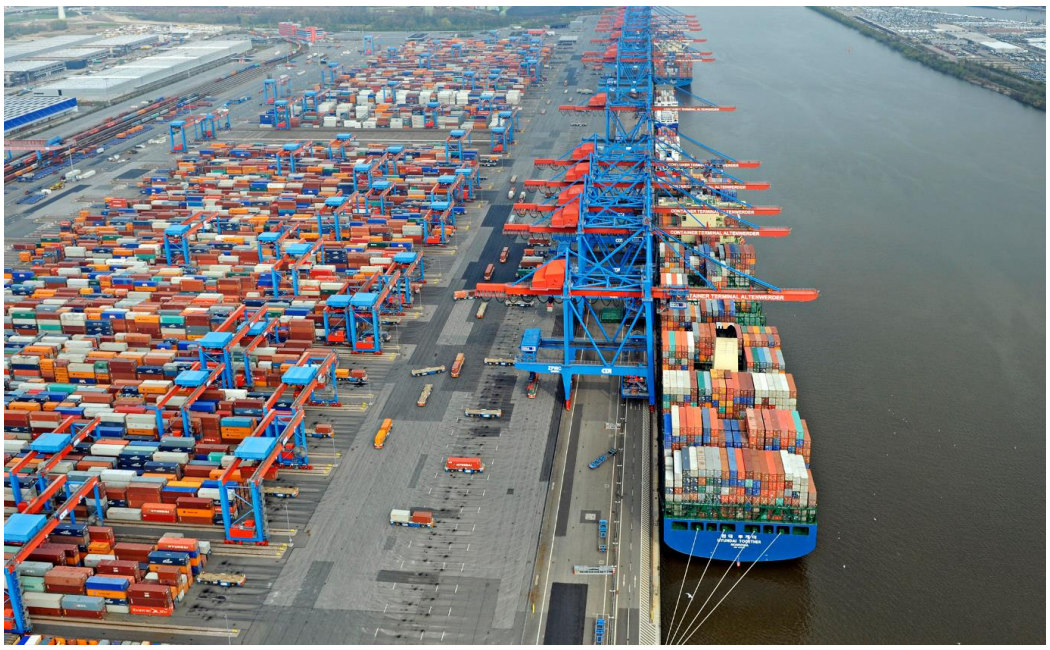


Abbildung 2.6: Frachthafen

Die Aufgabe der Docker Engine ist es also, eine reibungslose Organisation und Koordination mehrerer verschiedener Schiffe zu gewährleisten, also auch unter der Nutzung eines einzelnen Hafens gleichzeitige Applikationen zu erlauben. Dafür darf jedes Schiff auch auf die gleichen freien Hafenmitarbeiter zurückgreifen oder hat fest zugewiesene Arbeiter, um sicherzugehen, dass die Arbeiten in einem bestimmten Zeitintervall sicher durchgeführt werden. (dynamische und statische Ressourcenallokation)

2.2.2 Ziele von Docker

Die meisten Ziele, die Docker verfolgt, kann man bereits in der vorangegangen Herleitung herauslesen. Diese werden nun kurz analysiert.

Plattformunabhängigkeit: fremde Schiffe Es wird die Plattformunabhängigkeit betont, denn - sofern es sich um ein Schiff handelt - wird dieses vom Hafen akzeptiert. Auf diese Weise können auf einer Dockerumgebung fast alle Container eingebunden werden[4]. Wie wir jedoch später feststellen werden und bereits im Abschnitt "Betriebssystemvirtualisierung" angeschnitten wurde, muss jedoch das System, auf dem die *Docker Engine* läuft, ähnlich sein um den fremden Container einbinden zu können, denn dieser nutzt ja den Kernel des Hostsystemes direkt mit. Ein Schiff fährt auch in der Realität von Hafen zu Hafen um dort seine Arbeit zu verrichten. Genauso kann ein bereits bestehendes Systeme einfach mitgenommen und in eine andere Docker Engine importiert werden.

Verteilbarkeit: Baupläne und Schiffe Docker unterstützt sogenannte *Dockerfiles* und *Compose Files*. Diese stellen einen Bauplan der Applikation dar und sind eine einfache Textdatei, in der die Struktur und Abhängigkeiten einer Applikation beschrieben wird. Dieses ist sehr leichtgewichtig und lässt sich sehr schnell verbreiten.

Einfachheit, Schnelligkeit, Effizienz und Leichtgewichtigkeit: Genau ein Hafen Jeder einzelne Hafen kann dazu benutzt werden, mehrere Schiffe gleichzeitig zu behandeln. Dies ist eine wichtige Eigenschaft von Docker, denn Docker unterstützt es, mehrere Anwendungen parallel laufen zu lassen und kümmert sich um die komplette Ressourcenverteilung. Dies macht das Ganze sehr leichtgewichtig. Durch die Betriebssystemvirtualisierung ist dies sehr leichtgewichtig und kann geringe Virtualisierungsverluste vorweisen, da kein neuer Kernel virtualisiert werden muss, sondern der Bestehende des Hostsystems mitbenutzt werden kann.

[4] In echt muss tatsächlich ein Hafen erst eine gewisse Größe haben, um auch alle Schiffe zu untersützen.

Isolation von Anwendung und Ausführumgebung: Platz auf den Schiffen Jeder Container ist in sich isoliert, da er mithilfe einer Metallwand von den anderen abgeschottet ist. Man stelle sich vor, ein kleines motiviertes Männchen sitze in jedem Container. Durch eine kleine Luke teilt es seine Ergebnisse mit der Außenwelt. So kann es diese beispielsweise den Hafenarbeitern, die auch auf dem Boot arbeiten, geben und in einen anderen Container, dessen Männchen bereits sehnsüchtig auf neue Arbeit warten weiterreichen lassen.

Nutzung gemeinsamer Ressourcen: Hafenarbeiter Das Ziel bei jeder Form von Virtualisierung ist auch die Nutzung von gemeinsamen Ressourcen. Jeder Hafenarbeiter kann von jedem Container oder Schiff angefordert werden. Dass jede Applikation Zugriff auf die gleichen Ressourcen hat, ist ein wichtiges Konzept. Das Load Balancing wird von der Docker Engine vorgenommen. Hat eine Applikation nun eine schwierigere Rechenaufgabe, dann können die Ressourcen auf diese verlagert werden und dort mithelfen. In der Informatik spricht man hierbei von einer *dynamischen* Ressourcenallokation. Zusätzlich wird aber auch die *Statische* unterstützt. Hierbei kann man sich vorstellen, dass ein renommierter Unternehmer den Hafen betritt und natürlich sofort bedient werden will. Er hat fest zugewiesene Arbeiter, die seine Arbeit schnellstmöglich erledigen, ohne ihn aufgrund lange "Ladezeiten" zu verärgern.

Modularisierung: einheitliche Container Container sind auch in der realen Welt standardisiert. Diese können nach belieben, wie bei einem Lego Baukasten aufeinander gestapelt werden. Genauso können die einzelnen Teilkomponenten der Software beliebig miteinander kombiniert werden.
Der Name *Docker* kann auch als eine Hommage an die Erfindung des Containers sein. Container haben bereits einmal den Schiffsverkehr revolutioniert und diesen standardisiert. Auch Docker möchte, wie damals, die Container für die Fracht eine Möglichkeit schaffen, Software ohne große Hürden schnell zu verbreiten und effizient auf Bestehendem auszuführen.

2.2.3 Geschichte

Laut einer Umfrage von *Cloud Zone* wurden bereits mehr als 450.000 Applikationen nach Docker portiert, während über vier Milliarden Container von der offiziellen Distribution vergeben wurden. [1]. Diese Zahlen wurden 2016 während der *DockerCon* veröffentlicht.

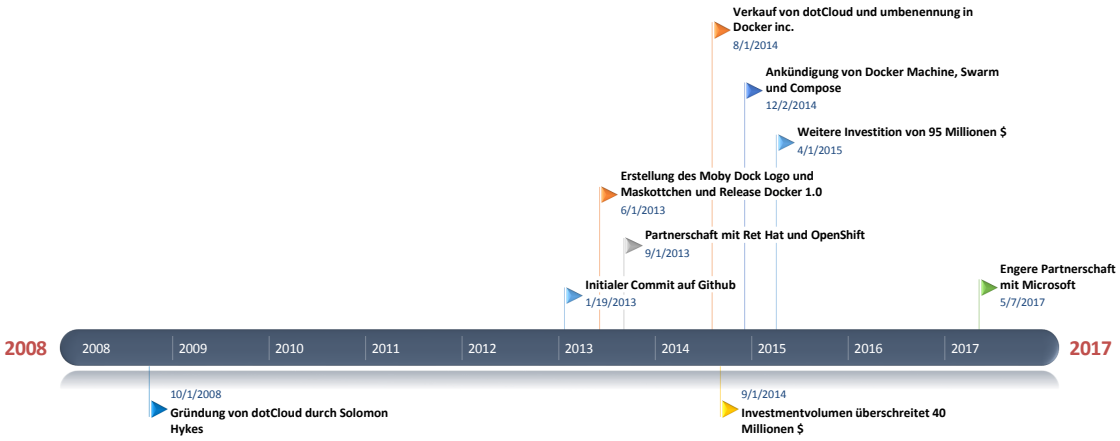


Abbildung 2.7: Geschichtlicher Verlauf

Im Jahre 2008 gründete *Solomon Hykes* das Unternehmen *dotCloud*, um ein sprachunabhängiges Platform-as-a-Service-(PaaS-)Angebot aufzubauen. Dies hatte zuerst nur mäßigen Erfolg, konnte es sich doch nicht klar von bestehenden kostenpflichtigen PaaS-System absetzen. Die Erfolgsgeschichte begann im März 2013, als *dotCloud* Docker als Open Source Software auf *GitHub*[5] bereitgestellt hat. Es ist nämlich so, dass ein Softwareunternehmen meistens ihre eigene Software als größtes Wirtschaftsgut ansehen und daher mit allen Mitteln den Sourcecode geheimhalten. Auf diese Weise hat sich Docker deutlich von anderen Konkurrenten abgesetzt.

Innerhalb von sechs Monaten gab es bereits mehr als 6700 Stars von Personen,

[5]Onlinedienst zur Bereitstellung von Software-Entwicklungsprojekten auf Basis des Versionsverwaltungssystems *Git*

denen das Projekt gefällt und 175 Contributors [7, vgl.] - freiwillige Unterstützer des Projektes. Daraufhin hat sich *dotCloud* in *Docker Inc.* umbenannt und die Aufmerksamkeit von anderen großen Unternehmen auf sich gezogen, die damit das Potenzial sehr schnell erkannt haben. *Red Hat*, *Google*, *Amazon* und *DigitOcean* [7] haben bereits sehr früh eine Integration von Docker in ihr bestehendes Cloud Angebot durchgeführt. Zeitgleich veröffentlichte Docker ein eigenes öffentliches Repository für Docker Container namems *Docker Hub*. Im Juni 2014 wurde dann eine erste Version 1.0 veröffentlicht. Auch Microsoft hat im März 2017 auf der *DockerCon* eine enge Partnerschaft mit Docker angekündigt und will die Technologie in ihren zukünfigten Windows Betriebssystemen (vor allem serverseitig) nativ unterstützen. [22]

Docker hat seine Erfolgsgeschichte der Open-Source Gemeinde zu verdanken. Aktuell zählt Docker mehr als 32000 Github Stars und über 3000 Personen, die einen Beitrag für die Software beigetragen haben. Über 8 Milliarden heruntergeladene Container und mehr als eine halbe Million Applikationen, die nach Docker portiert wurden unterstreichen die Erfolgsgeschichte der Dockertechnologie. [8, vgl.]

Open Container Initiative (OCI) Im Juni 2015 hat Docker die *Open Container Initiative* ins Leben gerufen. Ziel dieser Institution ist es, einen standardisierten Rahmen für die Containertechnologie zu bieten. Aktuell (Stand August 2017) werden zwei verschieden Spezifikationen entwickelt: Eine Laufzeitspezifikation, die auf eine abstrakten Ebene erklärt, wie Container aus einem Dateibündel erzeugt werden sollen und einer Imagespezifikation, die beschreibt, welche Bestandteile ein Image haben muss.

Eine Implementation einer Containertechnologie nach OCI würde bedeuten, dass man sich zuerst ein Image nach OCI Standard herunterlädt und dieses dann nach OCI Manier in eine Laufzeitumgebung packt.[6]

Der Grund für die Notwendigkeiten einer derartigen Institution ist die schnelle Verbreitung der Containertechnologie vor allem in der Wirtschaft. Um nun einen einheitlichen Rahmen, der auch andere Technologien als Docker abdeckt zu schaffen wurde die OCI gegründet.

2.2.4 Architektur

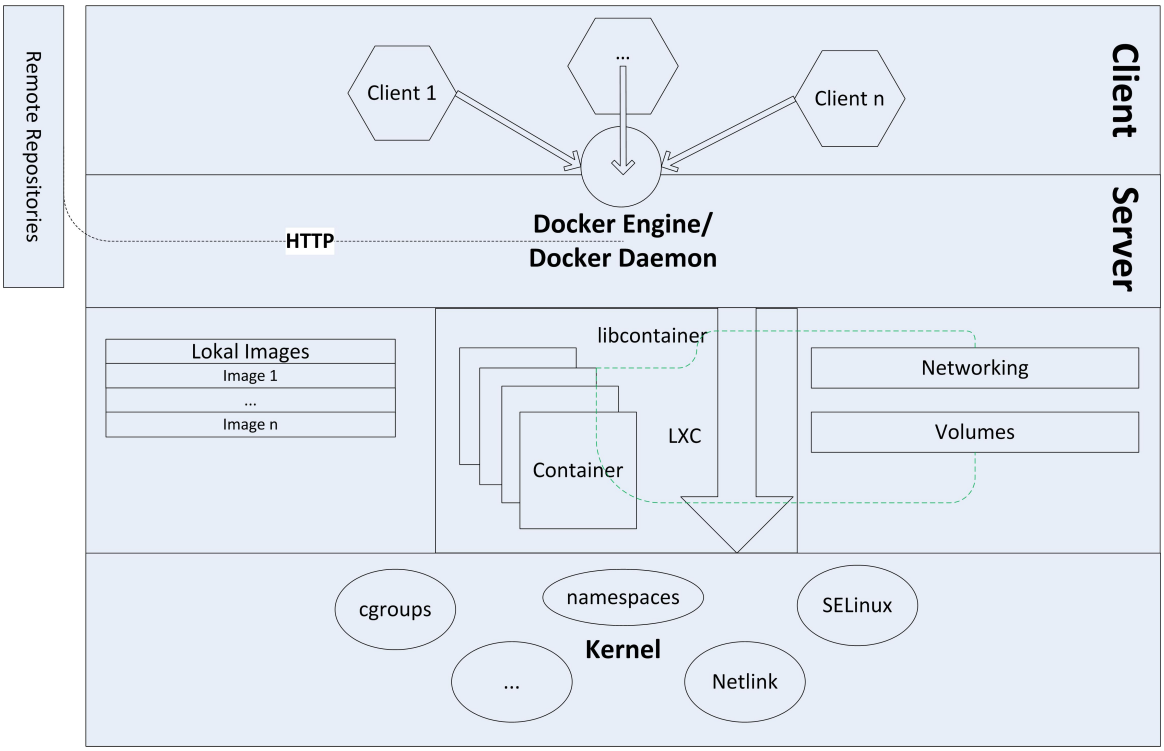


Abbildung 2.8: Docker Architektur

Docker besteht aus einer Client-Server-Architektur. Der Client kommuniziert über einen Endpunkt der Docker Engine mit dem Server. Der bekannteste Client ist die von Docker selbst veröffentlichte *Docker CLI*, mit deren Hilfe ein Zugriff auf den Docker Server direkt bei der Installation mitgeliefert wird. Die dadurch entstehende Flexibilität kann genutzt werden, um eine gesamte Dockerumgebung lokal einzurichten oder aber mit einem Clienten eine Verbindung zu einem entfernen Docker Server aufzubauen. Eine *RESTful API* stellt diese Schnittstelle bereit.

Definition 2.2.1 (Docker Engine) *Die Docker Engine steuert und protokolliert alle Vorgänge innerhalb einer Dockerumgebung und stellt eine Schnittstelle nach außen dar.*

Alle weiter beschriebenen Funktionen werden von dieser Schnittstelle geregelt. Über einen Endpunkt - beispielweise ein Socket unter */var/run/docker.sock* - kann diese von außen aufgerufen werden. Die wichtigsten Funktionen von ihr sind das Erstellen und Zerstören von Containern, das Starten und Stoppen oder das Verwalten von Images.

Definition 2.2.2 (Docker Container) *Lauffähige isolierte Umgebungen, welche benutzt werden können, um Anwendungen über den Hostkernel auszuführen.*

Container stellen das Herz einer Dockerumgebung dar, denn diese sind die eigentlichen Anwendungen, die wir virtualisieren wollen. Auf Anforderung können sie mithilfe der Docker Engine auf einem System gestartet und verwaltet werden. Der Zugriff auf diese erfolgt meistens über eine *bash*, in manchen Fällen, wenn die Anwendung bereits fertig ausgeliefert wird auch über eine lokale Webschnittstelle. Zusätzlich können neuen Containern weitere Parameter übergeben werden. So können zum Beispiel lokale Dateipfade in den Container gemountet, Portweiterleitungen in den Container definiert, Umgebungsvariablen und Hostnamen gesetzt oder der Container einem virtuellen Netzwerk zugeordnet werden, um diesen mit anderen Containern zu verlinken.

Im Gegenssatz zu Images sind Container lauffähig und beinhalten das Dateisystem, eine Menge an Operationen und eine Laufzeitumgebung [21, S.11 unter "Docker images"] um Applikation zu betreiben. Entgegen den Images, gibt es beim Container eine oberste Schicht, die beschreibbar ist.

Man erkennt bereits den Vorteil: Da es nur ein Dateisystem ist, welches eingebunden wird, nicht jedoch alle Prozesse, die ein vollwertiges Betriebssystem braucht kann so auch bei parallel laufenden Containern die Virtualisierung sehr performant gehalten werden.

Docker & Microservices Ein Microservice ist ein ausführbares Programm, welches alle notwendigen Abhängigkeiten mit sich selbst mitbringt und über eine fest definierte Schnittstelle mit anderen Services kommuniziert.[11] Docker ist hierfür eine optimale Lösung, denn die Container sind in sich geschlossen und bringen alle Abhängigkeiten selbst mit. Außerdem können die isolierten Container miteinander kombiniert werden, was genau den Anforderungen an einen Microservice erfüllt.

Definition 2.2.3 (Docker Images) *Schritt-für-Schritt aufgesetztes geschichtetes Dateisystem, aus welchem ein lauffähiger Container erstellt werden kann.*

Docker Images sind die Baublöcke von Containern. Da diese auf Layer aufbauen, also jede Modifikation an einem Basisimage (häufig Ubuntu, CentOS, usw...)

diesem eine neue Ebene aufsetzt, kann viel Speicherplatz gespart werden, da das Grundgerüst bei ähnlichen Images nur einmal heruntergeladen werden muss. Diese sind hochportabel und können einfach verteilt über eine ⇒*Repository* oder einen lokalen Datenträger verteilt werden.

Definition 2.2.4 (Docker Network) *Ermöglicht die Nutzung und Verwaltung virtueller Netzwerke für Container.*

Docker Networks sind wichtig für jegliche Kommunikation zwischen den Containern. Diese können einem virtuellen Netzwerk zugewiesen werden, über das sie dann untereinander kommunizieren können. Docker stellt zusätzlich noch einen externen *Domain Name Service* (DNS) Dienst zur Verfügung mit deren Hilfe es möglich ist, Container ganz komfortabel über deren Hostnamen oder einen Alias-Hostnamen (Netzwerkname) zu erreichen.

Aufgrund der Komplexität kann der interessierte Leser hier nur auf die offiziellen Quellen verwiesen werden.

Definition 2.2.5 (Docker Volumes) *Stellen spezielle persistente Dateiverzeichnisse zur Verfügung, auf die mehrere Container gemeinsam zugreifen können.*

Container sind aus Sicherheitsgründen grundsätzlich gegenseitig abgeschottet. Das heißt, dass es einem Container nicht möglich ist, Daten direkt mit anderen Container auszutauschen. Um eine Möglichkeit zu haben, ohne große Umwege auf gemeinsame Daten zuzugreifen wurden die *Volumes* eingeführt. Dabei handelt es sich lediglich um ein Verzeichnis auf der Hostmaschine, welches an einen bestimmten Einhängepunkt in die Container eingebunden wird.

Definition 2.2.6 (Docker Registry) *Eine serverseitige Open-Source Applikation mit deren Hilfe Docker Images gespeichert, archiviert und verteilt werden können.*

Der Docker Client verwendet eine *Remote Repository* um mittels eines *pull*-Befehls ein entferntes Image in den lokalen Speicher zu laden. Unter einem *Repository* versteht man das Archiv, in welchem die Abbilder liegen und ist fester Bestandteil der Registry. Eine Registry ist eine Basisquelle, über die man sich verschiedenste Abbilder von Anwendungen oder Betriebssystemen herunterladen kann. Die bekannteste Online Registry ist die offizielle *Dockerhub*, die von jedermann frei verfügbar

verwendet werden kann. Mit aktuell über 400.000 öffentlichen Images [3] findet sich für jeden erdenklichen Anwendungsfall das passende Abbild. Unternehmen können für ihre Zwecke auch eigene sichere Repositories auf ihren Servern einrichten und betreiben.

Definition 2.2.7 (Dockerfile) *Eine Textdatei, die Befehle beinhaltet, mit deren Hilfe ein Image aufgebaut wird.*

Alle Befehle in einem Dockerfile werden der Reihe nach abgearbeitet. Neben Kommandozeilenbefehlen, die auf dem neuen Image ausgeführt werden existieren noch eine ganze andere Reihe Befehle - wie beispielsweise das Öffnen von Ports oder setzen von Umgebungsvariablen, die ein Image genauer beschreiben. Jeder einzelne Befehl erzeugt eine neue Ebene auf einem Grundimage, welches mit einem *FROM* definiert wird. Code 2.1 zeigt ein simples Beispiel einer solche Datei. In diesem wird auf einem aktuellen Ubuntu mit dem Nutzer *demo* ein einfacher Webserver installiert und der Port 8080 freigegeben. Zusätzlich wird für den Containerstart eine Datei *Entrypoint.sh* vom Host kopiert, die bei jedem Starten des Containers ausgeführt wird. In diesem könnte man weitere Dienste definieren, die beim Starten des Containers ausgeführt werden sollen.
Mit Hilfe *docker build* Befehls kann das neue Image erzeugt werden.

Definition 2.2.8 (Docker Compose nach [9]) *Docker Compose ist ein Werkzeug, um Multi-Container Docker Applikationen zu erstellen und zu betreiben.*

Unter Multi-Container Applikation versteht man solche Applikationen, bei denen mindestens zwei Container fest miteinander vernetzt sind. Ein gutes Beispiel ist das anfangs erwähnte Datenbank-Webserver Konstrukt, bei dem in einer gewissen Weise diese zwei Dinge fest verbunden sind, denn die Webapplikationen des Webservers nutzen die Daten der Datenbank. Auf diese Weise können einzelne Komponenten logisch ausgelagert, jedoch die gesamte Struktur einer Anwendung zusammengehalten werden.

Ein solcher Aufbau wird mithilfe eines einfachen Textfiles (*Composefile*) definiert, welches dann von Docker interpretiert und zusammengebaut wird.

Zusätzlich können in diesem noch weitere Parameter für die individuellen Container übergeben werden.

Code 2.1: Ein Beispiel *Dockerfile*

```
1 FROM ubuntu:latest  # uses latest ubuntu as baseimage
2 USER demo # sets UID 'demo'
3
4 RUN apt-get update  # Updates Ubuntu Packages
5 RUN apt-get install httpd  # install simple webserver
6
7 ADD entrypoint /  # copies entrypoint to root
8
9 WORKDIR /  # Following commands from home dir
10
11 [...]
12
13 EXPOSE 8080  # opens Port 8080
14 ENTRYPOINT /entrypoint.sh
15         # a new Container from that image will always run
16         # entrypoint.sh
```

Definition 2.2.9 (Union File System (UFS)) *Ein UFS fügt Dateisysteme verschiedener Orte zu einem Einzigen, Logischen zusammen.*

Docker bedient sich der UFS-Technik um eine Menge Speicherplatz zu sparen. Dies bedeutet, dass "ähnliche" Images - die, die ein gleiches Basisimage benutzen - das Basisimage nur einmal abspeichern müssen und jeder Container, der daraus erzeugt wird auf die bereits vorhandenen Daten zugreifen kann. Für jede neue Version muss daher nur das Delta abgespeichert werden.

Jeder Container besteht aus dem darunterliegenden Image und einem beschreibbaren Layer, welches ganz oben liegt. Es sind also Schichten gestapelt. Jede Änderung an einem aktiven Container wird in dieser Schicht gespeichert. Auf diese Weise bleiben die unteren Schichten unveränderlich, was es möglich macht, mehrere verschiedene Container auf das gleiche Basisimage zugreifen zu lassen, obwohl jeder Container selbst seinen eigenen IST-Zustand hat, ohne das Basisimage kopieren zu müssen.

Der *Docker Storage Driver* steuert die Verwendung von Image und Container Layern. Er bedient sich einer Copy-On-Write Strategie, bei der die Daten nur kopiert werden, wenn wirklich eine Änderung vorliegt. Wenn eine Datei in einer tieferlie-

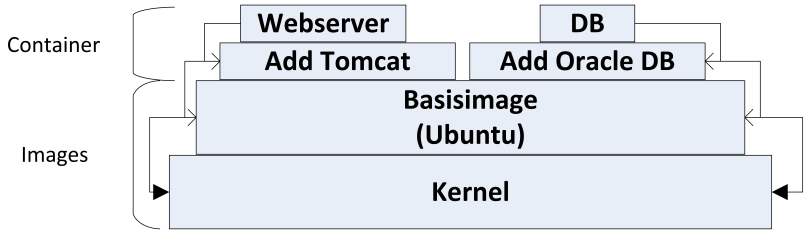


Abbildung 2.9: Union File System (UFS)

genden Schicht bereits vorhanden ist, wird beim Lesevorgang einfach auf dieses verwiesen. Muss nun eine Änderung durchgeführt werden, dann wird diese Datei in die oberste beschreibbare Schicht kopiert und modifiziert.

Will man nun diesen Container commiten, dann wird die oberste Schicht eingefroren und als neue unbeschreibbare Schicht gespeichert.

Wenn man nun zwei Container erzeugt, die auf das gleiche Basisimage zurückgreifen, dann referenzieren diese auch auf den gleichen Speicherplatz, an dem dieses liegt. Beide Container bekommen nun eine eigene Schicht, die sich zur Laufzeit verändert und somit ist gewährleistet, dass jeder Container seinen eignen Zustand besitzt.

Abbildung 2.9 soll das Prinzip exemplarisch verdeutlichen. Hier werden aus einem gemeinsamen Grundimage ein Datenbank und ein Webserver Image erstellt, aus denen sich dann die entsprechenden Container bilden lassen. Hierfür wird auf die mit Pfeilen gekennzeichneten Abhängigkeiten zurückgegriffen.

Alle Container verwenden den gleichen Kernel und das gleiche Ubuntu Basisimage. Der oberste Baustein dieser Pyramide ist beschreibbar. Der Kernel ist nur der Vollständigkeitshalber aufgeführt. Dieser stellt keinen Speicherplatz dar, der verwendet wird, sondern die physikalischen Ressourcen, die diese erst lauffähig machen.

Docker CLI Client Der Client stellt eine Menge von Kommandozeilentools zur Verfügung mit deren Hilfe der Docker Server gesteuert werden kann. Diese Suite wird bei einer Installation automatisch mitgeliefert. Nachfolgend sind einige der wichtigsten Funktionen aufgelistet.

- **pull**: Download eines Images von einem Repository Server

- **run**: Erstellt einen neuen Container mit dem Imageargument. Falls das Image nicht lokal vorhanden ist, wird zuerst *docker pull* aufgerufen
- **stop/start**: Starten/Stoppen eines vorhanden Containers
- **rm/rmi**: Löschen eines Containers/Images
- **ps/info**: Listet erstellte Container/Systeminformationen der Docker Engine auf
- **commit**: Speichert einen Container als Image ab, von dem neue Container erstellt werden können
- **export/save**: Speichern eines Container/Images auf einem physischen Datenträger

Zusätzlich gibt es aber auch grafische, webbasierte Klienten, wie *Portainer*, mit deren Hilfe eine Verwaltung der Docker Engine vereinfacht wird. Interessanterweise werden solche Applikationen selbst als Docker Container ausgeliefert und der Endpunkt der Dockerumgebung hineingemountet.

2.2.5 Windows Container

Windows Container sind Container, deren grundlegendes Filesystem das Windows Filesystem ist, welches sich sehr von Linux unterscheidet. Seit einigen Monaten stellt Microsoft eine native Containerumgebung für die Windows Systeme zur Verfügung. Da die Docker Container den gleichen Kernel wie das Gastsystem benutzen, sind die *Images* vom Typ des Hostsystems abhängig. Damit können Windows Container nur auf Windows Kernel und Linux Container nur auf Linux Kernel laufen.

2.3 Product Lifecycle Management (PLM)

Definition 2.3.1 (Product Lifecycle Management [16]) *"Beim Product Lifecycle Management (PLM) handelt es sich um einen Ansatz zur ganzheitlichen, unternehmensweiten Verwaltung und Steuerung aller Produktdaten und Prozesse des gesamten Lebenszyklus – von der Entwicklung und Produktion über den Vertrieb bis hin zur Wartung. Ziel dabei ist es, den Produktentstehungsprozess durch Datenmanagement zu unterstützen und die Entwicklungsproduktivität zu erhöhen. Zur Unterstützung dieses Ansatzes*

existieren Informationstechnologie (IT-) basierende PLM-Lösungen, die mit ihren Funktionen die Umsetzung des PLM-Ansatzes in großen Teilen erst ermöglichen."

Diese Definition beschreibt bereits sehr gut, um was es sich bei PLM handelt. Mithilfe von (rechengesteuerten) Prozessen wird der gesamte Lebenszyklus eines Produktes erfasst und gesteuert. Der Lebenszyklus besteht aus allen Phasen, die ein Produkt durchläuft: Von der *Planung und Entwicklung* zur *Montage* und dem *Vertrieb* bis hin in den After-Sales Bereich mit *Service & Wartung* und der *Demontage*.

2.4 Siemens Teamcenter 11.2

Dieser Abschnitt beschäftigt sich mit der von der von *Siemens* entwickelten PLM Software *Teamcenter*, die im weiteren verlauf der Arbeit nach Docker portiert wird.

2.4.1 Geschichtliche Hintergründe

Nach der Übernahme von *UGS PLM Solutions (UGS)* von Siemens im Mai 2007 gehört die *Siemens PLM Software* als eine Geschäftsbereich zur Division des *Siemens Industry Sector*s. Diese ist für die Entwicklung einer PLM Software unter dem Namen *Teamcenter* zuständig. [19, S.2]. Anfängliche Versionen von *Teamcenter* sind bereits in den 1980ern aus der Motivation entstanden, ein reines CAD Dateisystem für die damalige Zeit zur Verfügung zu stellen.

In den darauffolgenden Jahren wurde dieser Softwarekern stetig durch zusätzliche Komponenten erweitert, um damit ein vollständiges PLM Umfeld verwalten zu können. Um die Software auch weiterhin auf dem technisch aktuellsten Stand zu halten wurde ab den 1990er Jahren *Teamcenter* in mehreren Versionen die Grundelemente komplett überarbeitet und modernisiert. Folglich bestehen die aktuelleren (ab Version 11) Versionen aus einer Vielzahl von unterschiedlichen Technologien und Programmiersprachen, die ineinander verflochten sind.

Teamcenter zählt heute gemeinhin als eine der weit verbreitesten PLM Lösungen weltweit.

2.4.2 Funktionen von Teamcenter nach [19]

Teamcenter stellt eine Reihe von Funktionen zur Verfügung, die speziell für das PLM Umfeld gedacht sind. Folgende Aufzählung soll dem Leser eine grobe Vorstellung geben, welche Funktionsbereiche von Teamcenter abgedeckt werden.

- Lieferantenmanagement
- Systems Engineering
- Management der Fertigungs-und Simulationsprozesse
- Wartung, Reparatur und Überholung-
- Reporting und Analyse
- Zusammenarbeit
- Portfolio und Projektmanagement
- (...)

2.4.3 Softwarearchitektur

Wir werden uns im Nachfolgenden mit der genauen Architektur von Teamcenter beschäftigen.

Tiers Wie in Abbildung 2.10 zu erkennen ist, besteht Teamcenter aus einer vierstufigen (*4Tiers*) Client-Server Architektur. Die unterste Stufe, der *Ressource Tier* ist für die physische Ablage der Daten zuständige Schicht. In ihr liegt die Datenbank, die alle Konfigurationen und Einstellungen der Teamcenter Instanzen beschreiben. Sie muss nicht mit der Teamcenter Installation auf einer Maschine betrieben werden, sondern kann auch verteilt im Netzwerk betrieben werden. In Verzeichnissen (Volumens) daneben liegen die externen Dateien selbst, die beispielsweise von den Nutzern hochgeladen wurden.

Im *Enterprise Tier* laufen die Teamcenter Server selbst, die aus mehreren Instanzen bestehen können. Ein Server Pool Manager ist dafür zuständig, die Instanzen zu verwalten. Teamcenter selbst wurde in C/C++ entwickelt. Die Schicht steht in einem direkten Verhältnis mit der Datenbank, die alle für den Betrieb von Teamcenter notwendigen Daten beinhaltet und mit diesen interagiert. Für Administratoren sehr wichtig ist der sogenannte *2-Tier Richt Client*. Diese auf *Eclipse RCP* basierte

Anwendung kann über einen X-Server von einem Clienten aufgerufen werden und stellt eine Benutzeroberfläche zur Verfügung, die es erlaubt, die gesamte Teamcenter Umgebung direkt an der betreibenden Schicht zu verwalten.

Der *Webtier* stellt einen Webserver zur Verfügung, auf dem die Clientanwendungen laufen. Als Webserver können verschiedene (freie) Anbieter verwendet werden, die das Deployment einer Java Enterprise Edition (J2EE) Anwendung auf einer Java Runtime Environment (JRE) erlauben: Beispielsweise *Redhat JBoss*, *Apache Tomcat* oder *IBM Websphere*. Darauf kann man nun die einzelnen Clients aufsetzen.

Im *Client Tier* befindet sich die Anwendungen, die auf den Webserver der darunterliegenden Schicht aufgesetzt wurde. Diese kommunizieren über HTTP(S) mit dem Webserver. Zur Darstellung der Inhalte in einem Browser werden bekannte Technologien wie JavaScript/AJAX/HTML/CSS verwendet, es existieren jedoch auch Plugins, die eine Einbindung der Oberfläche direkt in Eclipse erlauben. Seit neustem stellt Siemens mit *Active Workspace* (AWC) eine komplett überarbeitete Benutzeroberfläche basierend auf HTML5 zur Verfügung. Diese ersetzt durch ihre an moderne Standards angepasste Oberfläche den schon etwas veralteten Webclienten, nutzt aber einige Webclient-Schnittstellen, um eine Verbindung mit Teamcenter aufzubauen, weswegen dieser noch lange nicht obsolet ist und zumindest weiter installiert werden muss.

File Management System Das File Management System (FMS) ist ein Framework, welches dafür sorgt, eine Schnittstelle zu den Volumes und einem Volume Server herzustellen. Somit kann von jeder Ebene direkt mit den Daten gearbeitet werden. Laut Tiwari [20] besteht das FMS aus zwei Komponenten, dem *FMS Server Cache (FSC)* und dem *FMS Client Cache (FCC)*. Beide haben die Aufgabe, auf ihrer jeweiligen Ebene die Daten zu Cachen und Dateitransaktionen auf diese Weise zu beschleunigen. Der FSC ist zusätzlich für die persistenten Schreibvorgänge zuständig und steht in enger Verbindung mit dem FCC des Clients. Jedem Client wird ein kleiner lokaler FCC installiert, während meistens nur ein FSC pro Server existiert.

Die Speicherung der Daten erfolgt getrennt: In einer Datenbank werden die Metadaten der in den Volumes abgelegten Daten gespeichert. Diese können von CAD-Zeichnungen über Office Dokumente bis hin zu einfachen Textdaten alles mögliche sein.

Abschließend lässt sich noch anmerken, dass Teamcenter aus historischen Gründen auf die verschiedensten Technologien basiert, die sich im Laufe der Zeit miteinander vermischt haben.

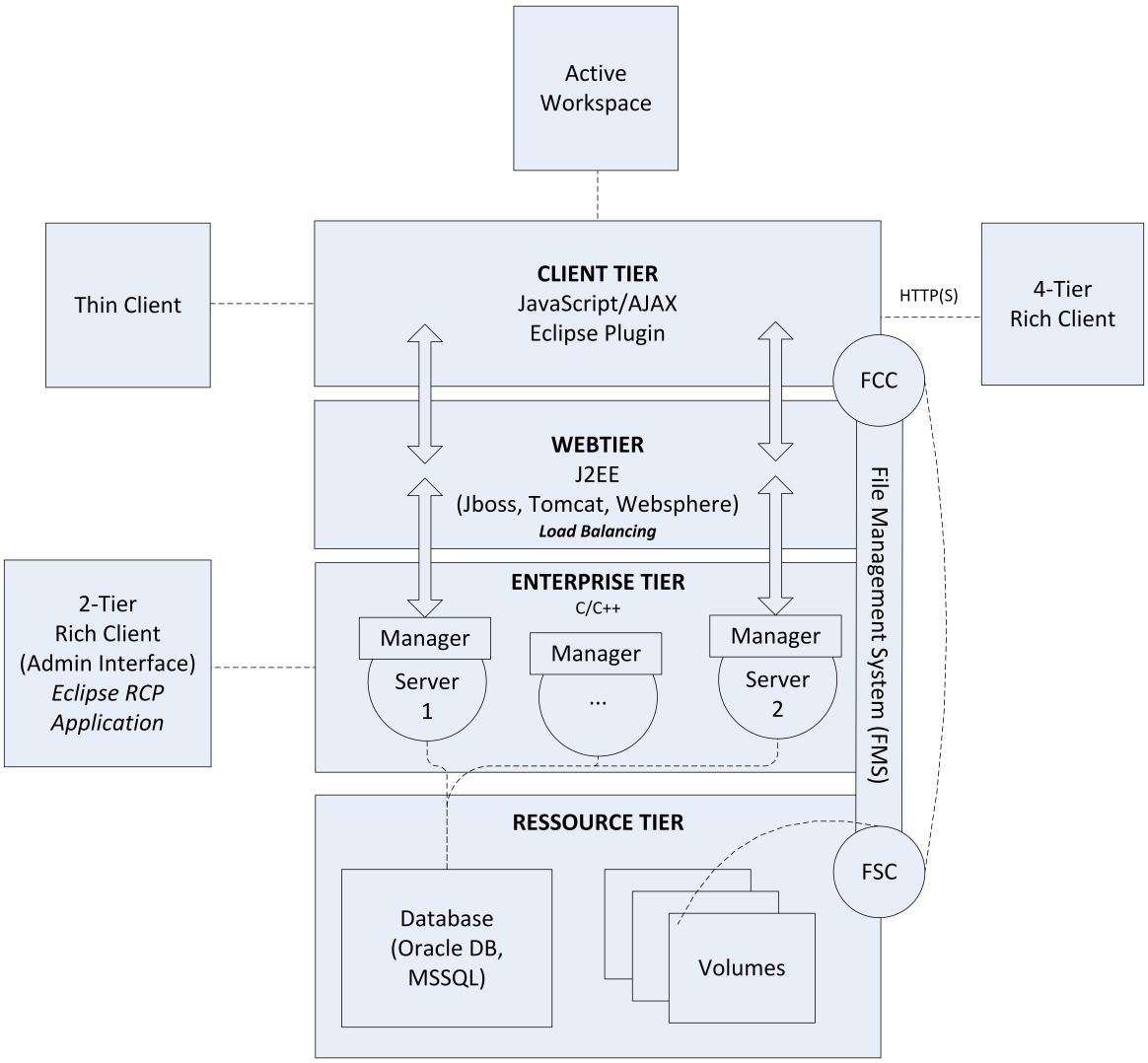


Abbildung 2.10: Teamcenter 11.2 Architektur

3 Methodenteil

Im Folgenden wird eine lauffähige Teamcenter Umgebung in einer Docker Orchestrierung installiert. Nach einem Aufbau der virtuellen Entwicklungsumgebung mit *Vagrant* wird untersucht, wie sich die Teamcenter Architektur auf unterschiedliche Container verteilen lässt. Im nächsten Schritt werden die Container erstellt und eine vollständige verteilte Teamcenter Installation durchgeführt. Hierfür spalten wir Teamcenter auf mehrere Komponenten, angelehnt an die *4-Tier* Architektur auf: Der *Datenbank*, dem *Teamcenter Server* und dem *Webclient*, bzw. Webserver.

Daraufhin wollen wir die einzelnen Komponenten soweit zusammenfügen, dass sie auch als ein einzelnes Bündel gestartet werden können, also eine komplette neue Teamcenter Instanz von Docker automatisch aufgesetzt werden kann und somit der ursprüngliche Deployment Prozess um ein Vielfaches beschleunigt wird. Hierfür benutzen wir ein *Docker Compose*.

Zum Abschluss wird nun untersucht, welche Vorteile sich nun durch diese Art der Architektur ergeben.

4 Durchführung

Dieses Kapitel beschäftigt sich mit der Entwicklung einer Teamcenter Multi-Containeranwendun
(Orchestrierung) für Docker. Wir beschreiben zunächst den Aufbau der Entwicklungsumgebung.

4.1 Anlegen der Entwicklungsumgebung

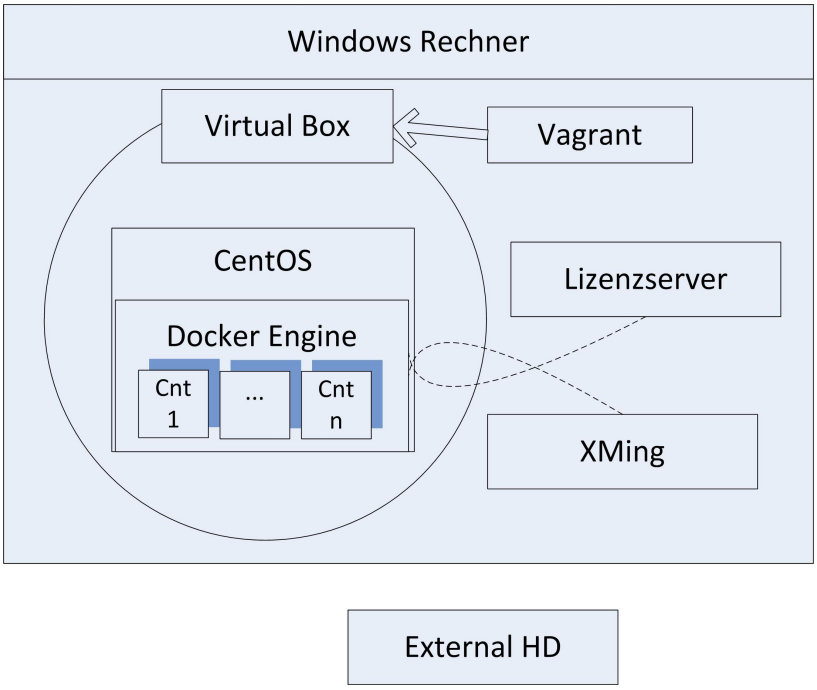


Abbildung 4.1: Der Aufbau der Entwicklungsumgebung

Die Entwicklung erfolgt in einer eigens dafür vorgesehenen virtuellen lokalen Entwicklungsumgebung.

Virtualisierungssoftware Um eine in sich geschlossene Entwicklungsumgebung, in der dann die Teamcenter Docker Container installiert werden zu haben, wird diese vollständig auf einem Windows Rechner virtualisiert. Die Wahl fiel hierbei auf *Oracles Virtual Box* (VBox). Man könnte genauso gut mit *VM Ware Workstation* oder anderen "Vollvirtualisierern" arbeiten, jedoch war bereits einiges an Erfahrung im Umgang mit dieser Software vorhanden. Zusätzlich ist VBox unter einer Open Source Lizenz[13] veröffentlicht, die es uns erlaubt, auch ohne hohe Lizenzosten eine vollwertige Virtualisierungssoftware zu verwenden.

Virtuelle Maschinen mit Vagrant Da, wie bereits angedeutet, mehrere Entwicklungsumgebungen verwendet werden, sei es, weil sich eine Maschine aufgrund irgendeines Fehlers verabschiedet oder sei es, weil neben Linux auch ein Windows Setup ausprobiert wird, ist es von Vorteil, eine Möglichkeit zu haben, diese einfach zu verwalten und gegebenenfalls auch zu ex-und importieren. Mithilfe von *HashiCorp Vagrant*[1] ist es möglich, seine Maschinen mithilfe eines einfachen beschreibenden Textfiles zu verwalten und aufzusetzen. Zusätzlich bietet Vagrant die Möglichkeit, ganze vordefinierte Virtuelle Maschinen aus einer globalen Hub herunterzuladen und aufzusetzen. Dadurch fällt das mühevolle Suchen von speziellen Betriebssystemimages (*.iso) im Internet weg.

In dieses kann man nun alle Dinge schreiben, die man auch manuell über die VBox GUI einstellen könnte, jedesmal aber einen großen Aufwand bedeuten würde. Die wichtigsten Funktionen sind:

- Wahl und Download des Basisbetriebssystems
- Erstellung Portweiterleitungen und Networking
- Definition von geteilten Ordnern (Mounts)
- Verändern der Allokation von Hardwareressourcen
- Setzen von Proxy und SSH Einstellungen
- usw...

Im Anhang 7.1 findet sich eine solches *Vagrantfile*, welches das im folgenden verwendete *CentOS* beschreibt.

[1] https://www.vagrantup.com

CNTLM Um auch Applikationen eine Internetverbindung im Firmennetz zu geben, die keine *NT LAN Manager*[2] (NTLM) Authentifizierung unterstützen, wird ein lokaler NTLM Proxy installiert, der den Datenverkehr umleitet: CNTLM. Die geforderte Authentifizierung wird dann vom Proxy übernommen. In 7.1 (Z. 4-8) wird der Proxy breits direkt von Vagrant genutzt und an das virtuelle System weitergegeben. Für manche Applikationen oder beispielsweise den Docker Containern muss dieser manuell in die entsprechenden Proxyfiles eingetragen werden.

Xming für X11 Da das *centOS* Gastsystem, um Ressourcen zu sparen, ohne Graphical User Interface (GUI) aufgesetzt wurde, wird für einige Applikationen eine Möglichkeit benötigt, Grafiken darzustellen.

Viele Anwendungen können ihre grafische Benutzeroberfläche auf einen X-Server exportieren. Hierfür wird bei UNIX-artigen Systemen und installiertem X11 Client eine Display Variable auf die IP des X-Servers gesetzt. In dieser Arbeit wird der freie X-Server *X-Ming* benutzt.

Lizenzserver Um Teamcenter verwenden zu können, muss ein Dienst eingerichtet werden, der eine Softwarelizenz zur Verfügung stellt. Nach Erhalt einer Lizenz wird diese auf dem Windows Host eingetragen. Die VM selbst und damit auch die dort ansässigen Docker Container, die eine installierte Teamcenterumgebung beinhalten können diesen dann, wie in 4.1 dargestellt ansprechen und als Lizenzserver hinterlegen. Beispielsweise kann ein Virtualbox Gast über die IP Adresse 10.0.2.2 den Host direkt ansprechen.

Natürlich wäre es alternativ möglich, den Lizenzserver auch im Gastsystem zu installieren um keine Abhängigkeiten vom Windows Host zu erzeugen. Dies ist aber insofern kein Problem, da wir innerhalb der VM mit einer dynamischen Namensauflösung (via /etc/hosts) arbeiten und so im Nachhinein der Lizenzserver noch geändert werden kann. Ein gravierender Nachteil ist zusätzlich, dass für jede neue Entwicklungsumgebung, die eventuell erzeugt werden muss, ein neuer kostenpflichtiger *Product Key* verwendet werden muss. Die Verwendung eines zentralen Lizenzservers verhindert dies jedoch, da alle VMs auf den gleichen Aktivierungsschlüssel zurückgreifen.

[2] Authentifizierungsverfahren für Rechnernetze

```
root@lukas-VirtualBox:~# docker info
Containers: 14
 Running: 0
 Paused: 0
 Stopped: 14
Images: 10
Server Version: 1.12.6
Storage Driver: aufs
 Root Dir: /var/lib/docker/aufs
 Backing Filesystem: extfs
 Dirs: 136
 Dirperm1 Supported: true
Logging Driver: json-file
Cgroup Driver: cgroupfs
Plugins:
 Volume: local
 Network: bridge null overlay host
Swarm: inactive
Runtimes: runc
Default Runtime: runc
Security Options: apparmor seccomp
Kernel Version: 4.4.0-83-generic
Operating System: Ubuntu 16.04 LTS
OSType: linux
Architecture: x86_64
CPUs: 6
Total Memory: 12.46 GiB
Name: lukas-VirtualBox
ID: E5XR:W7CQ:ZJ3T:4OSG:PKIR:ZL6A:QTUB:YWHR:FOO5:47TK:WV2R:QF7G
Docker Root Dir: /var/lib/docker
Debug Mode (client): false
Debug Mode (server): false
Registry: https://index.docker.io/v1/
WARNING: No swap limit support
Insecure Registries:
 127.0.0.0/8
```

Abbildung 4.2: Docker Umgebung auf einem Ubuntu System

4.1.1 Installation von Docker

Docker selbst kann sehr einfach über die offiziellen Pakete installiert werden. *centOS* bietet hierfür den Paketinstaller *yum*. Zusätzlich müssen wir noch manuell *Docker-Compose* hinzufügen, denn dieses wird standardmäßig nicht mit eingerichtet.

```
1 $ yum install docker-io
2 $ curl -L https://github.com/docker/compose/releases/download/1.14.0/
3   |docker-compose-`uname -s`-`uname -m` > /usr/local/bin/docker-compose
4 $ sudo chmod +x /usr/local/bin/docker-compose
```

Abbildung 4.2 zeigt eine installierte Docker Umgebung.

4.2 Aufteilung von TC auf Docker Container

Im Folgenden wird die Struktur von Teamcenter auf ein Docker Umfeld angepasst und überlegt, welche Basisimages am besten zu verwenden sind. Natürlich können hier nur die entscheidenden Schritte skizziert werden.

Es biete sich an, die logische Einteilung wie in 2.10 beschrieben leicht anzupassen und die Container auf dieser Weise aufzubauen. Daher spalten wir, wie in 4.3

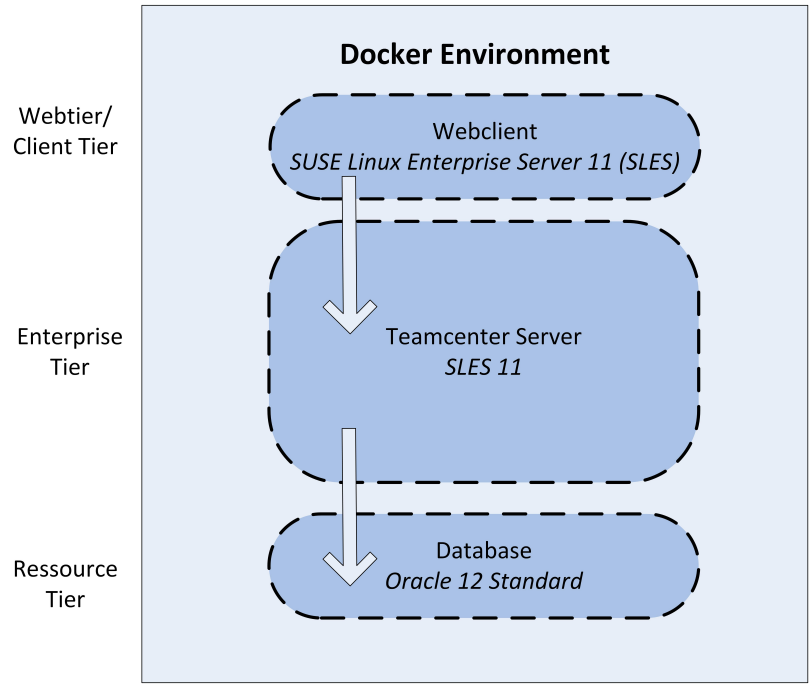


Abbildung 4.3: Aufbau der Container

gezeigt die TC Architektur auf drei unabhängige Teilkomponenten auf, dem *Web- und Client Tier*, dem *Teamcenter Server* im Enterprise Tier und einer *Datenbank*. Dass Web und Client Tier zusammengezogen sind hat schlicht den Grund, dass der Client Tier einfach nur ein kleines Datenpaket ist, welches vom Webclient automatisch aufgesetzt wird damit die Daten physisch am gleichen Ort gespeichert werden sollten.

Abbildung 4.3 zeigt diese Aufteilung in einzelne Container. Die Pfeile symbolisieren Abhängigkeiten. So benötigt der Webclient eine Verbindung mit dem Teamcenter Server und der Teamcenter Server mit der Datenbank. Diese Verhältnisse werden über eine automatische Namensauflösung erzeugt, wodurch ein Container über einen Hostnamen auf einen anderen zugreifen kann.

4.2.1 Basisimages

Für die einzelnen Container werden Basisimages benötigt, auf denen dann die jeweiligen Komponenten installiert werden.

Oracle DB 11 Standard Als Datenbank verwenden wir ein offizielles Oracle DB Image, welches aus der Docker Hub heruntergeladen und nach eigenen Bedürfnissen angepasst wird. Auf diesem ist dann bereits eine voll funktionsfähige Datenbank vorinstalliert-und konfiguriert.

Wir könnten natürlich auch einen eigenen Datenbank Container selber erstellen, aber warum sollten wir das Rad neu erfinden, wenn es bereits eine optimierte Lösung gibt?

SUSE Linux Enterprise Server 11 Um unseren Teamcenter und Tomcat Server aufzusetzen benutzen wir ein angepasstes SUSE Linux Enterprise Server Image (SLES). Dieses hat von sich aus keinerlei Programme installiert und wird dann manuell um das nötigste aufgestockt. Dadurch kann eine Menge Speicherplatz gespart werden.

4.3 Installation von TC in Container

Wir werden nun die Installation durchführen. Hierfür werden wir die Schichten (Tiers) von unten nach oben aufbauen und miteinander verknüpfen. In einem letzten Schritt wird das starten und stoppen dieser Instanzen mithilfe von *Docker Compose* automatisiert.

4.3.1 Installation der Oracle DB

Wir starten zunächst einen neuen Datenbankcontainer und ergänzen ein leeres Verzeichnis für die Teamcenter Daten.

```
1 $ docker run --d tcdb_oracle sath89/oracle-12c
2 $ docker exec -it tcdb_oracle /bin/bash
3 $ mkdir -p /teamcenter/db && chown -R oracle/teamcenter
```

Hierbei benutzen wir ein von *Oracle* offiziell bereitgestelltes Datenbankimage. In ihm sind bereits alle Datenbankdienste, die wir später benötigen werden vorinstalliert und ein administrativer Benutzer eingerichtet.

Ferner ist bereits ein Entrypoint, der beim Starten des Containers die Datenbank hochfährt enthalten.

Um diesen Container später zu erreichen, werden wir ihn mit den andern dynamisch verlinken. Später werden wir dieses Linksystem aufheben und mit einem virtuellen Netzwerk und einem externen Domain Name Service (DNS) Dienst ersetzen.

Port	1521
SID	XE
Username	system
Password	oracle

4.3.2 Installation des Teamcenter Servers

Als nächstes installieren wir das Herz einer jeden Teamcenter Umgebung: Den **Teamcenter Server**. Hierfür füttern wir einen speziellen Container mit den Teamcenter Daten und starten die Installation. Zusätzlich werden dann notwendige Dienste konfiguriert und an die Laufzeitumgebung angepasst.

4.3.2.1 SLES Image für Teamcenter

Um das Basisimage für unsere ohnehin sehr wachsende Teamcenter Instanz möglichst klein zu halten erstellen wir ein eigenes Image speziell für unsere Teamcenterumgebung, die nur die Dienste und Programme bereitstellt, die von ihr auch benötigt werden. Dafür rüsten wir ein minimales *SUSE Linux Enterprise Server 11* System mit den benötigten Komponenten, Nutzern und Verzeichnissen auf.

Über einen einfachen geteilten Ordner unter */tcrepo* können weitere Hilfsprogramme, die von Teamcenter benötigt werden (z.B. *Java*) direkt von der Festplatte installiert werden.

```
1 $ docker run -i -t -v /tcrepo/:/tcrepo/ --name sles11-4-tc
2  | gbeutner/sles11-sp4_64 /bin/bash
3         $ useradd -m infodba -p infodba
4         $ zypper --non-interactive in ksh, tcsh, vim, libelf, libaio,
5                 | curl elfutils, libstdc++-devel libcanberra-gtk xorg-x11 gawk
6                 sudo xorg-fonts-core xorg-x11-fonts
7           ...
8 $ docker commit sles11-4-tc tcsles
```

Nach einem Commit in die lokale Repository können wir dieses Image nun für unseren Teamcenter Server benutzen. Hierfür brauchen wir nichts weiter zu tun, als aus dem Image einen Container zu erzeugen.

4.3.2.2 Starten des Containers

Nun werden wir unsere Teamcenter Umgebung ein erstes mal starten. Wir verbinden diesen Container über den *–link-* Befehl mit unserer bereits erzeugten Datenbank. Damit kann diese nun über den Namen *tcdb* erreicht werden. Diese Technik des verlinkens setzt einfach die IP des verbundenen Containers dynamisch in die DNS-Hostdatei */etc/hosts*. Später werden wir auf einen externen DNS umsteigen, damit auch bidirektionale Verbindungen erzeugt werden können.

```
$ docker run -i -t --name sles11-4-tc --hostname teamcenter
| --link tcdb_oracle:tcdb
| --add-host tclicense:10.0.2.2 -v /tcrepo:/tcrepo tcsles /bin/bash
```

4.3.2.3 .bash_profile-Script

Wir wollen nun einige Routineaktionen für den Teamcenter Nutzer vereinfachen. Die Datei *.bash_profile* findet sich unter linuxartigen Systemen im Home Verzeichnis der User (/home/usrName). Dieses Script wird immer ausgeführt, wenn sich der zugehörige Nutzer anmeldet. Dieses ist nützlich für Aufgaben, die ein Nutzerkonto ausführen muss, wenn es angemeldet wird.

Für Teamcenter sind einige Umgebungsvariablen wichtig, die bei jedem Anmelden gesetzt werden müssen. Zusätzlich kann man auch automatisch die *DISPLAY*-Variable setzen lassen, damit das manuelle exportieren vor jeder X-Applikation wegfällt.

```
1 export DISPLAY=10.0.2.2:0.0 #Display
2 export TC_NO_TEXTSRV_SHARED_MEMORY=TRUE #Teamcenter Trigger
3 export TC_USE_LOV_SHARED_MEMORY=FALSE
4 export TC_USE_METADATA_SHARED_MEMORY=FALSE
5 export TC_USE_PREFS_SHARED_MEMORY=FALSE
6 export TC_ROOT=/teamcenter/11.2 # Teamcenter Installation Directories
7 export TC_DATA=/teamcenter/tcdata
8 . /teamcenter/tcdata/tc_profilevars #Environmentvariables
```

4.3.2.4 Grundinstallation

Die *tem.sh* ist ein einfaches Scriptfile, dass die Installation von Teamcenter initiiert. Ferner ist *tem.sh* die zentrale Anlaufstelle für alles, was mit der Administration des Teamcenter Kerns zu tun hat: So kann man bestehende Konfigurationen mit Updates versorgen, einzelne Features nachinstallieren und Konfigurationen ändern.

Wenn wir nun zuvor die *DISPLAY*-Variable auf unseren X-Ming Server setzen, können wir die grafische Benutzerführung der Installation benutzen.

```
1 $ export DISPLAY=10.0.2.2:0.0
2 $ ./tem.sh -jre /usr/java8/jre
```

Nun erscheint auf dem Hostrechner aus dem Container der VM heraus die Oberfläche, die in 4.4 zu sehen ist.

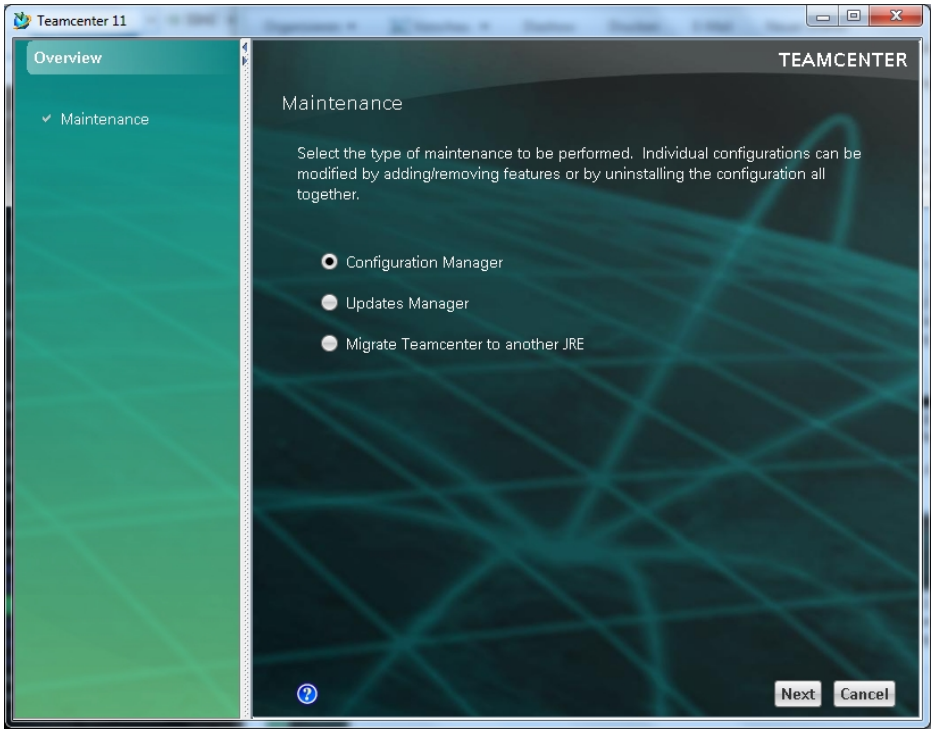


Abbildung 4.4: Teamcenter 11 Installations Umgebung

Nutzung anderer Container Da die Hostnamen der anderen Container dynamisch von einem DNS-Dienst aufgelöst werden, können diese einfach in der Installation verwendet werden. Auf diese Weise wird beispielsweise der Datenbankcon-

tainer unter dem Hostnamen *tcdb* in die Installation eingetragen und dynamisch aufgelöst.

Eine genaue Beschreibung der Teamcenter Installation sei an dieser Stelle ausgelassen, denn diese ist sehr komplex und die benötigten Komponenten sind sehr von den verschiedenen Anforderungen abhängig. Für genauere Informationen sei der interessierte Leser auf die offizielle Dokumentation von *Siemens* verwiesen.

4.3.2.5 Konfiguration der Dienste

Eine minimale Teamcenter Umgebung besteht aus zwei wesentlichen Diensten: Dem FMS, der für alle Interaktionen mit den Dateisystemen zuständig ist und dem Pool Manager, der sich um die Teamcenter Instanz selbst kümmert. Diese werden von Teamcenter als Service mitgebracht, die einfach in das System eingebunden werden können. Dies hat den Vorteil, dass diese dann sofort als Hintergrunddienste laufen.

Nach dem Einrichten können die Dienste ganz einfach als Service gestartet werden.

```
1 $ service rc.ugs.FSC_teamcenter_infodba start
2 $ service rc.tc.mgr_tc_sles01_PoolA start
```

Wenn man diese Services nun mit dem Container starten will, dann kann man diese nicht, wie von normalen Linux Distributionen gewöhnt in einen Autostart eintragen. Auf diesen hat Docker nämlich zu Gunsten eines *entrypoint*-Scriptes verzichtet. Der Start dieser Dienste kann dort ganz einfach eingetragen werden, wobei darauf zu achten ist, dass am Ende mindestens eine aktive Applikation im Vordergrund weiterläuft. Mehr dazu im Punkt ⇒*Entrypoint* auf S. 42

4.3.2.6 Teamcenter auf Version 11.2.3 patchen

Wir wollen natürlich unsere Teamcenter Version auf den aktuellsten Stand bringen. Um auf die aktuellste Version 11.2.3 zu updaten, arbeiten wir wieder mit der *tem.sh*. Im Hauptmenü der Installation wählen wir dieses mal *Updatemanager* (Vergleiche Abbildung 4.4) und wählen auf der nächsten Seite den Ordner aus, in dem sich die rohen Updatedateien befinden. Daraufhin werden die Daten aktualisiert.

4.3.2.7 Warten auf Datenbankinitialisierung

Im ersten Schritt (4.3.1) wurde ein eigener Container für die Datenbank angelegt. Beim Starten dieser wird ein Startskript ausgeführt, welches die vorhanden Daten in die Datenbank einbindet und das Datenbank Management System (DBMS) hochfährt. Dabei ergibt sich ein Problem: Wenn sich nun der Teamcenter Container schneller initialisiert als die Datenbank, werden natürlich Fehler geworfen, dass die Datenbank noch nicht erreichbar ist und Ereignisse ausgeführt, die eigentlich noch warten sollen.

Um nun Fehlfunktionen beim Starten der Teamcenter Services vorzubeugen wäre es vorteilhaft zu warten, bis die Datenbank vollständig erreichbar ist. Im folgenden werden einige Lösungsansätze für diese Problematik untersucht und entwickelt.

Porttest auf den Datenbankcontainer Ein nicht zielführender Ansatz ist es mit speziellen Portuntersuchungstools (Beispielsweise nmap) zu warten, bis sich der entsprechende Port der Datenbank geöffnet hat. Nun wurde sehr schnell festgestellt, dass die Datenbank die Ports zwar sofort nach dem Starten öffnet, das jedoch noch lange nicht heißt, dass die Datenbank auch initialisiert worden ist.

Das macht auch Sinn, denn sonst könnte auch die Fehlermeldung, dass *die Datenbank noch nicht bereit ist* nicht geworfen werden.

Festes Zeitlimit Der wohl primitivste funktionierende Ansatz wäre, einfach eine gewisse Zeit zu warten, bis man mit Sicherheit davon ausgehen kann, dass die Datenbank initialisiert und erreicht wurde.

Das Problem ist, dass je nach Leistungsfähigkeit und Auslastung der Maschine nicht vorausgesagt werden kann, wie lange diese Zeitspanne sein muss. Man könnte zwar durch aufwändiges ausprobieren ungefähre Zeiten herausfinden, jedoch unterscheiden sich diese von Rechner zu Rechner, bzw. von Auslastungsgrad zu Auslastungsgrad. Außerdem könnte auch das Gegenteil auftreten: Man könnte einige Minuten warten müssen, obwohl die Datenbank bereits fertig initialisiert wurde. Ein solcher Leerlauf ist natürlich nicht gewünscht.

Daher wird auch dieser Ansatz verworfen, denn es wäre schön, wenn es für dieses Problem eine dynamische Lösung gäbe.

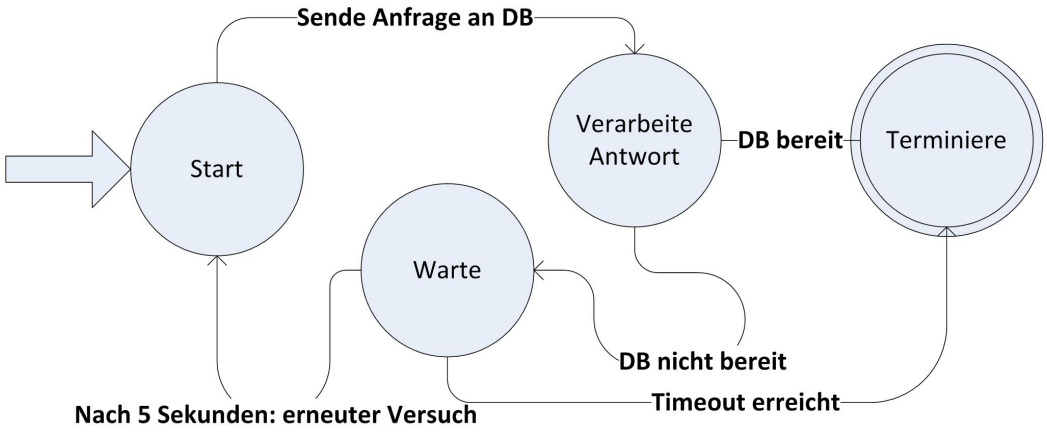


Abbildung 4.5: Java Datenbanktest: Zustände

Java Programm Die Wahl fällt auf eine eigene Lösung. Da wegen Teamcenter sowieso Java installiert ist sorgt nun ein kleines Kommandozeilenprogramm dafür, der Datenbank in regelmäßigen Abständen Anfragen zu schicken, bis diese akzeptiert werden oder ein Timeout erreicht wird. In Abbildung 4.5 wird das grobe Konzept dieses Programmes veranschaulicht. Der Vorteil liegt nun darin, dass wirklich nur solange gewartet wird, bis die Datenbank sicher erreichbar ist. Sollte dies nicht möglich sein, wird nach einer gewissen Zeit abgebrochen.

4.3.2.8 Entrypoint-Skript

Der Entrypoint ist der zentrale Einstiegspunkt der Teamcenter Instanz. Dieses wird ausgeführt, sobald der Container gestartet wird. Speziell für unser Teamcenter hat es die folgenden Aufgaben:

- Warten bis die Datenbank initialisiert wurde
- Starten der Teamcenter Dienste
- Modifikation von Teamcenter Parametern

Die ersten beiden Punkte wurde bereits in den vorherigen Abschnitten abgearbeitet. Der letzte wird wichtig, sobald wir mehr als einen Teamcenter auf einer Maschine betreiben möchten. Dazu jedoch später mehr unter dem Punk 4.3.4. Hierbei werden wir die Parameter, die von Docker-Compose übergeben wurden in einige Konfigurationsdaten übernehmen.

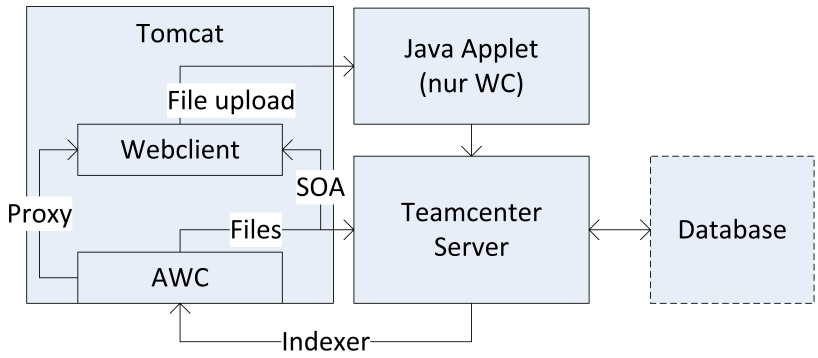


Abbildung 4.6: Architektur des Webclienten

4.3.3 Installation des Webclients und ActiveWorkspace (AWC)

Als nächstes wollen wir den 4-Tier Rich Client installieren. Diese Weboberfläche ist die direkte Interaktion des Endnutzers mit Teamcenter. Im ersten Schritt erstellen wir den normalen Webclienten und erweitern diesen anschließend um die neue Oberfläche des AWC. Diese packen wir dann in einen eigenen Container, der auf den Teamcenter Server zugreifen kann.

4.3.3.1 Architektur des Webclienten

Abbildung 4.6 versucht, die einzelnen Abhängigkeiten genauer darzustellen. Der Webclient kommuniziert über einer SOA Schnittstelle mit dem Teamcenter Server. Auf diese Weise gelangt er an die Teamcenter Daten. Der Fileupload läuft hingegen über ein eigenes Java Applet. Dieses holt sich eine FMS Bootstrap Adresse, um zu wissen, wo sich der FMS befindet. Damit können dann beispielsweise Dateien hochladen kann werden. Die meisten Browser haben ihren Support für Java Applets in den vergangen Jahren eingestellt. Einzig der *Internet Explorer* unterstützt weiterhin dieser alte Technik. Ferner werden nächste Versionen von Teamcenter komplett ohne Java Applets auskommen.

Im Grunde ist der AWC Client nur eine Erweiterung der Webclienten, denn dieser nutzt die gleiche SOA Schnittstelle. Der Umgang mit Dateien hat sich geändert: Diese werden nun über HTML5 auf den Tomcat Server hochgeladen, der diese dann an den FMS weitergibt.

4.3.3.2 Erstellen eines *Apache Tomcat 8*-Images

Das Erstellen eines Images, welches Tomcat installiert hat kann automatisiert werden, da nur in ein paar kleinen Schritten die Tomcat Dateien heruntergeladen und entpackt werden müssen. Dadurch kann man schnell auch Container bauen, die auf einer anderen Tomcat Version basieren.

Abhilfe schafft uns hierfür ein eigenes *Dockerfile*.

```
1  FROM sles11-4-tc
2
3  WORKDIR [/usr]
4  RUN wget --no-check-certificate
5          |https://archive.apache.org/dist/tomcat-8/[Version]/...
6  RUN wget --no-check-certificate -no-cookies --header
7          |"Cookie: oraclelicense=accept-securebackup-cookie;"
8          |http://download.oracle.com/otn-pub/java/jdk/[Version]/...
9  RUN tar -xvf apache-tomcat-[Version]
10 RUN rm -R apache-tomcat-[Version].tar.gz
11 RUN rpm -i java-[Version].rpm
12
13 ENV JAVA_HOME=/usr/java/jre[Version]
14 EXPOSE 8080
```

Wir benutzen wieder das SLES-Image, welches wir auch für die Teamcenter Installation benötigt haben als Basisimage. *Tomcat* benötigt eine *Java* Laufzeitumgebung, weswegen wir beides installieren müssen. Hierfür laden wir die entsprechenden Daten aus dem Internet herunter und installieren sie. Daraufhin wird der Java Pfad als Umgebungsvariable gesetzt, damit *Tomcat 8* weiß, wo sich die Java Installation befindet.

Nun können wir einen Tomcat-Container erstellen und die Webclienten aufsetzen.

```
1 $ docker build webclient:1.0
2 $ docker run -it -v /tcrepo:/tcrepo -p 8080:8080 webclient:1.0 /bin/bash
```

Es ist wichtig, einen lokalen Port auf 8080 weiterzuleiten. Auf diese Weise kann der Docker Host, bzw. auch der Windowsrechner dahinter auf den Tomcat Server zugreifen.

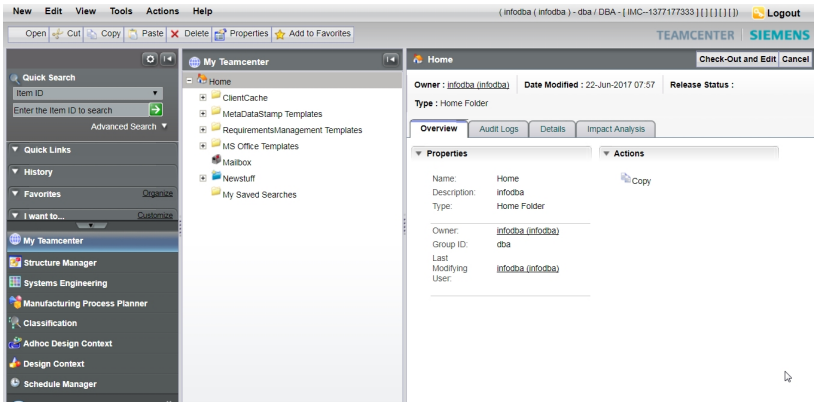


Abbildung 4.7: Teamcenter Standard Webclient

4.3.3.3 Erstellen des Webclienten und AWC

Beide Weboberflächen können als **.war* generiert und von Tomcat selbstständig aufgesetzt werden.

4Tier Webclient Der Webclient kann über eine eigene grafische Oberfläche gebaut werden. In dieser werden dann die entsprechenden Komponente ausgewählt und links definiert. Abbildung 4.7 zeigt einen Ausschnitt dieses Clients.

Installation von ActiveWorkspace Entgegen dem normalen Webclienten benötigt AWC auch serverseitig einige Erweiterungen. Diese werden zunächst über die Updatefunktion eingespielt.

AWC selbst kann mithilfe der Teamcenter Installationsoberfläche erzeugt werden. Leider unterstützt dieses Tool keine linuxbasierten Systeme, weswegen auf eine Out-Of-The-Box Lösung, die auf dem Windows Host ohne irgendwas installieren zu müssen, ausgeführt werden kann zurückgegriffen.

In einer *tem.properties*-Datei können bestimmte Parameter gesetzt werden, die einen Zugriff auf den Teamcenter Container ermöglichen. Folgend sind einige der Wichtigsten dargestellt.

```
1 ProxyServlet.redirectURL=http://localhost:8080/tc/
2 FmsProxyServlet.bootstrapFSCURLs=http://teamcenter:4544/
3 VisPoolProxy.soaPath=http://localhost:8080/tc/
4 ...
```

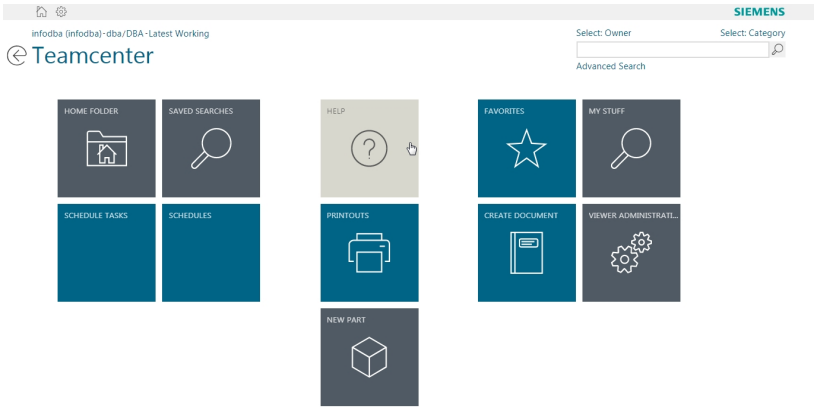


Abbildung 4.8: Teamcenter 11.2 AWC

Hier ist gut zu sehen, dass der AWC Client den normalen Webclienten über den containereigenen Localhost als Proxyschnittstelle benutzt, um auf den Teamcenter Server zuzugreifen.

Die *bootstrapFSCURLs* sind jene Verweise auf die FSCs des Teamcenter Servers. Damit greift AWC auf das Filesystem zu. Der Unterschied zum Webclient liegt hierbei darin, dass der Standard Webclient das Up-und Downloaden von Daten auf ein eigenes *Java Applet* auslagert, indem die Addresse des FMS angefordert und damit interagiert wird. Die Hostnamen werden wie immer vom DNS aufgelöst und stellen somit einen Verweis auf den korrekten Container dar.

Ein weiterer Unterschied zum normalen Webclienten ist es, dass AWC eine bidirektionale Verbindung mit dem Teamcenter Server benötigt. Dieses Problem lässt sich jedoch mithilfe eins virtuellen Netzwerkes, welches von Haus aus einen externen DNS besitzt lösen.

Theoretisch könnte man auch AWC in einen eigenen Tomcat Container auslagern. Abbildung 4.8 zeigt einen Ausschnitt des AWC Clienten.

4.3.3.4 Deployment

Das schöne ist, dass Tomcat **.war*-Files automatisch deployed. Dafür müssen diese nur in das Verzeichnis *Webapps* kopiert und der Tomcat Server gestartet werden.

```
1 $ cp /tcrepo/*.war /usr/apache-tomcat-8/Webapps/
2 $ /usr/apache-tomcat-8/bin/catalina.sh run
```

Catalina, der Tomcat Prozess, kümmert sich nun darum, die Verzeichnisse zu entpacken und aufzusetzen. Dies zweite Zeile kommt später in einen Entrypoint und wird bei jedem Start des Containers ausgeführt.

4.3.4 Starten von mehreren Teamcenter Instanzen auf einer Maschine

Motivation Um vorhandene Ressourcen optimal auszunutzen wäre es interessant, mehrere Teamcenter Instanzen auf einer Maschine nebeneinander laufen zu lassen. Dies ist sehr interessant, weil Docker Container als abgekapselte Umgebungen problemlos nebeneinander existieren können, ohne großartig einen Virtualisierungsschwund zu erzeugen und sich gegenseitig zu blockieren.

Zusätzlich würde eine Menge Speicherplatz gespart werden, da durch das *Union File System* wirklich nur die Änderungen an der verschiedenen Teamcenter Instanzen gespeichert werden müssen, das Grundsystem jedoch nur einmal Speicherplatz benötigt.

4.3.4.1 Probleme

Obwohl es bereits möglich ist, ein weiteres Teamcenter hochzuziehen, ergeben sich Probleme, sobald man von außen auf diese Container zugreifen will, denn man muss die einzelnen Installationen irgendwie unterscheiden können.

A) Gleiche Ports Alle Teamcenter Instanzen nutzen sowohl intern, als auch nach außen die gleichen Ports. 'Nach außen' sorgt natürlich für Komplikationen, da nicht mehr unterschieden werden kann, ob beispielsweise Port 8080 auf den Webclienten der ersten, oder zweiten Instanz zeigen soll.

B) FSC Bootstrap URL Wie bereits bei der Architektur des Webclients 4.3.3.1 erläutert, nutzt dieser ein Java Applet, welches eine Bootstrap URL, die auf den Teamcenter Container verweis nach außen schickt. Diese URL besteht aus einem symbolischen Hostnamen und einem Port, über den die Daten ausgetauscht werden. Wenn wir dieses nicht abändern, können die einzelnen Teamcenter Installation nicht mehr unterschieden werden.

4.3.4.2 Lösung

A) Gleiche Ports Eine Lösung für das erste Problem ist es, die Portweiterleitungen in den Container abzuändern. Beispielsweise kann man ganz einfach definieren, dass Port 8081 des Dockerhosts auf Port 8080 in einem Tomcat Container verweist. Damit hätte man, ohne an der internen Struktur etwas ändern zu müssen einen Zugriff auf die einzelnen Container gewährt.

B) FSC bootstrap URL Dieses Problem ist etwas verzwickter. In verschiedenen **.xml*-Konfigurationsdateien sind diese Einstellungen in der AWC Webapp und im Teamcenter Server verankert.

Die Lösung hierfür ist es, beim Starten der Teamcenter Instanz, diese Parameter mithilfe eines Scriptes manuell anzupassen. Hierfür eignet sich das Kommandozeilentool *sed* (*s*tream *ed*itor), welches auf fast jedem Linux bereits vorinstalliert ist. Mit diesem Werkzeug kann man Texte mithilfe regulären Ausdrücken (RegEx) manipulieren.

Zuerst verändern wir die FMS Einstellungen. Folgendes Codeschnipsel sucht in einer gegebenen Datei den String *address="xxx"* und füllt diesen mit den aktuellen Werten, die zuvor in die Umgebungsvariablen gesetzt gesetzt wurden.

```
1 $ sed -r -i
2         | 's#(address=")[^"]+"#\1'http:\/\/\$NEW_NAME:$NEW_PORT'"#'
3         | /teamcenter/11.2/fsc/fmsmaster_FSC_teamcenter_infodba.xml
```

Zusätzlich müssen wir einen Datenbankeintrag aktualisieren. Hierfür importieren wir die benötigten Daten in **.xml*-Form, verändern die entsprechende Stelle und exportieren sie wieder in die Datenbank.

```
1  $ su - infodba -c "/teamcenter/11.2/bin/preferences_manager"
2          | -u=infodba -p=*******
3          | -mode=export -scope=SITE -file=/pathTo/preferences_FMS_bootstrap
4          | -out_file="/pathTo/preferences_teamcenter.xml"
5
6  $ sed -r -i
7          | 's#(address=")[^"]+"#\1'http:\/\/\$NEW_NAME:$NEW_PORT'"#'
8          | /teamcenter/11.2/fsc/fmsmaster_FSC_teamcenter_infodba.xml
9  $ su - infodba -c /teamcenter/11.2/bin/preferences_manager
10         | -u=infodba -p=*******
```

```
11          | -mode=import -scope=SITE -file=/pathTo/preferences_teamcenter.xml
12          | -action=OVERRIDE
```

4.3.5 Docker Compose

Anforderung Wir wollen nun alle Komponenten so konfigurieren, damit diese auch gleichzeitig hochgefahren werden können, denn es war bisher nur möglich, jede Teilkomponente einzeln zu starten, was auf Dauer doch etwas mühsam ist. Ferner wird es möglich sein, eine komplett neue Teamcenter Instanz mit all ihren Einstellungen und Querverbindungen, die bisher nur sehr mühsam über die Kommandozeile definiert werden mussten zu erzeugen. Dabei muss die Lösung von außen Parameter akzeptieren, um Ports und Hostnamen nach außen dynamisch zu halten. Dies wird spätestens dann notwendig, wenn mehr als ein komplettes Teamcenter auf einer einzelnen Maschine laufen soll.

Zu diesem Zweck verwenden wir ein Docker Compose File. Dieses sorgt dafür, dass alle Container mit ihren Abhängigkeiten initialisiert und gestartet werden. Dieses ist größtenteils selbsterklärend und kann im Anhang unter 7.2 gefunden werden. Trotzdem sollen zwei interessante Punkte beleuchtet werden.

Networks Damit die Container untereinander kommunizieren können stecken wir diese in ein gemeinsamen virtuelles Netzwerk.

```
1 networks:
2         net:
3                 driver: bridge
```

Jetzt weißt man die Container diesem Netzwerk zu und gibt ihnen Aliasnamen. Dies hat den Vorteil, dass nun jeder Container dieses Konstrukts die andern erreichen kann. Docker stellt einen eigenen DNS Dienst zur Verfügung, der die Namen für jeden Container auflöst.

Damit wäre auch eine bidirektionale Verbindung zwischen AWC und Teamcenter geschaffen:

```
1 services:
2         teamcenter:
```

```
3                   networks:
4                       net:
5                           aliases:
6                           - teamcenter
7                           - $tc_name
8           webclient
9                   ...
```

Parameter in Compose File Wie eingangs erwähnt, wollen wir das Compose file mit Parametern von außen versorgen, um Ports und Namen nach außen dynamisch zu halten. Obwohl ein Compose File (bisher) noch keine direkten Argumente unterstützt, gibt es doch eine sehr elegante Lösung.

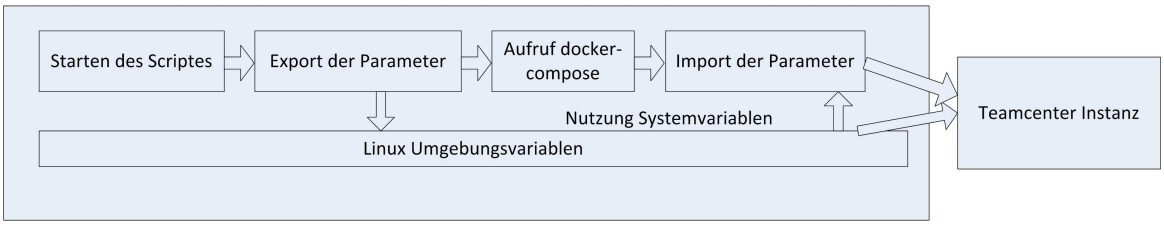


Abbildung 4.9: Parametrisierung Compose File

Mit Hilfe von **Umgebungsvariablen** können Argumente indirekt übergeben, indem sie temporär im System abgelegt und darauf von Docker-Compose verarbeitet werden. Dafür wird ein Script vorgeschaltet, welches zuerst die erforderlichen Variablen systemweit exportiert und im Anschluss über *docker-compose* die Teamcenter Instanz startet. Das *Compose-File* selbst ist in der Lage, Systemvariablen zu verwenden und auf diese Weise Namen dynamisch anzupassen. Die Variablen können wie bei Bash-Scripten gewohnt mit einem $-Zeichen verwendet werden.

Im folgenden Beispiel wird der FMS-Port gesetzt und dem Teamcenter ein Hostname gegeben. Diese Parameter werden dann beim Containerstart verarbeitet und die entsprechenden Dateien angepasst.

```
1 services:
2         teamcenter:
3             ports:
4                 - $fmsport:$fmsport
```

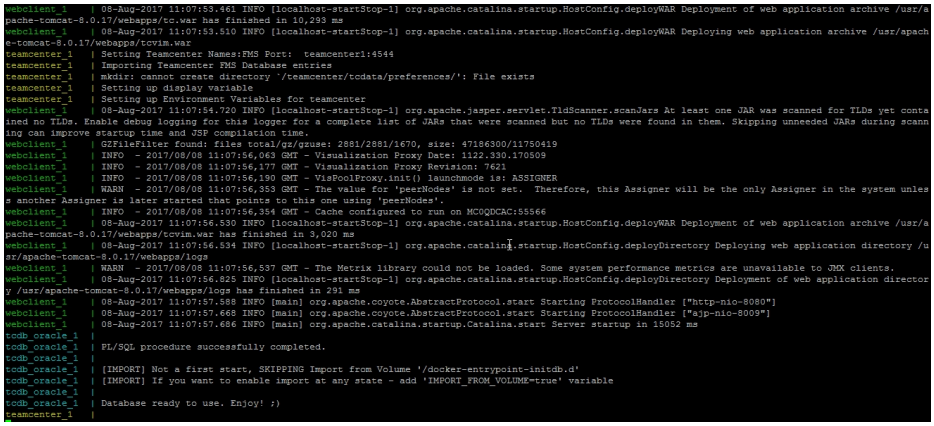


Abbildung 4.10: CLI Startup Compose Ausgabe

```
5                   extra_hosts:
6                   - "tclicense:10.0.2.2"
7                   environment:
8                   - tc_name=$tc_name
9                   - tc_fmsport=$fmsport
```

Das fertige Script kann im Anhang unter 7.2 gefunden werden.

Management Script Da wir unser Compose File parametrisiert haben, müssen wir zuvor noch die verwendeten Variablen setzen. Weil dies von Hand sehr mühselig ist, erstellen wir ein Script, welches die erforderlichen Variablen exportiert und darauf *Docker Compose* mit einem übergebenen Parameter aufruft.

```
1 #!/bin/bash
2 export fmsport=4544
3 export client_port=8080
4 export tc_name=teamcenter1
5 export fsc_url="http://$tc_name:$fmsport/"
6 export DB_TIMEOUT=6000
7
8 docker-compose -f ./teamcenter-compose.yml $1
```

Nun kann mithilfe des folgenden Befehls eine ganze Teamcenter Umgebung auf einmal gestartet werden.

```
1 $ ./startup.sh up
```

4.4 Ausblick: Installation in einem Windows Server Container

Die Installation selbst ist möglich, aber noch nicht komfortabel. Windows Container sind noch sehr neu und damit auch noch etwas unausgereift. Es fehlen einige Funktionalitäten, die speziell für unseren Awwendungsfall sehr wichtig wären.

4.4.1 Probleme

Zum Stand vom 30. Juni 2017 ergeben sich aber noch einige Probleme, die eine produktive Nutzung von Teamcenter in Windows Containers verhindern.

Kein RDP/X11 Support Dies ist wohl das zur Zeit das größte Problem. Es ist nicht möglich, grafische Oberfläche aus dem Container heraus zu erzeugen. Die Installation selbst kann über einen Umweg auch ohne Grafik erzeugt werden: Man muss nur auf einem Betriebssystem, das Grafik Darstellen kann eine *silent.xml* erzeugen, die die Installationsparameter beinhaltet. Diese kann dann per Kommandozeile an die tem.sh übergeben werden und über ein CLI installiert werden

Jedoch Kann nicht auf den 2-Tier Rich Client (2.10) zugegriffen werden, dieser ist jedoch für die Administration essentiell.

Keine SMB Unterstützung SMB wäre eine weitere Option, zumindest den Installer grafisch zu starten. Damit könnte man die Teamcenter Daten aus dem Container heraus auf das Hostsystem, welches über eine grafische Oberfläche besitzt teilen und von dort starten.

4.4.2 Zukunft

Microsoft hat für das kommende Jahr angekündigt SMB zu unterstützen Damit könnte man eventuell das Problem mit den grafischen Oberflächen lösen.[18]

Ferner will Microsoft in einer kommenden Version das Windows Betriebssystem mittels Hypervisor-Virtualisierung um eine native Linux Container Unterstützung erweitern. Technisch wäre dies keine große Herausforderung. Microsoft will dem Benutzer die Wahl lassen, auf welchem Hintergrundbetriebssystem die Container

laufen sollen. Microsoft hat sich mit *Canonical*, *Red Hat*, *Intel* und *SUSE* bereits einige Partner mit ins Boot geholt, die darauf hindeuten lassen, dass der Kunde aus einer breiten Auswahl von bekannten Linux Distributionen auswählen kann.

Damit könnte man ganz einfach unser bereits unter Linux entwickeltes Teamcenter in einen Windows Host einbinden, ohne wie bisher eine eigene virtuelle Linux Maschine dafür aufsetzten zu müssen.[17]

5 Diskussion

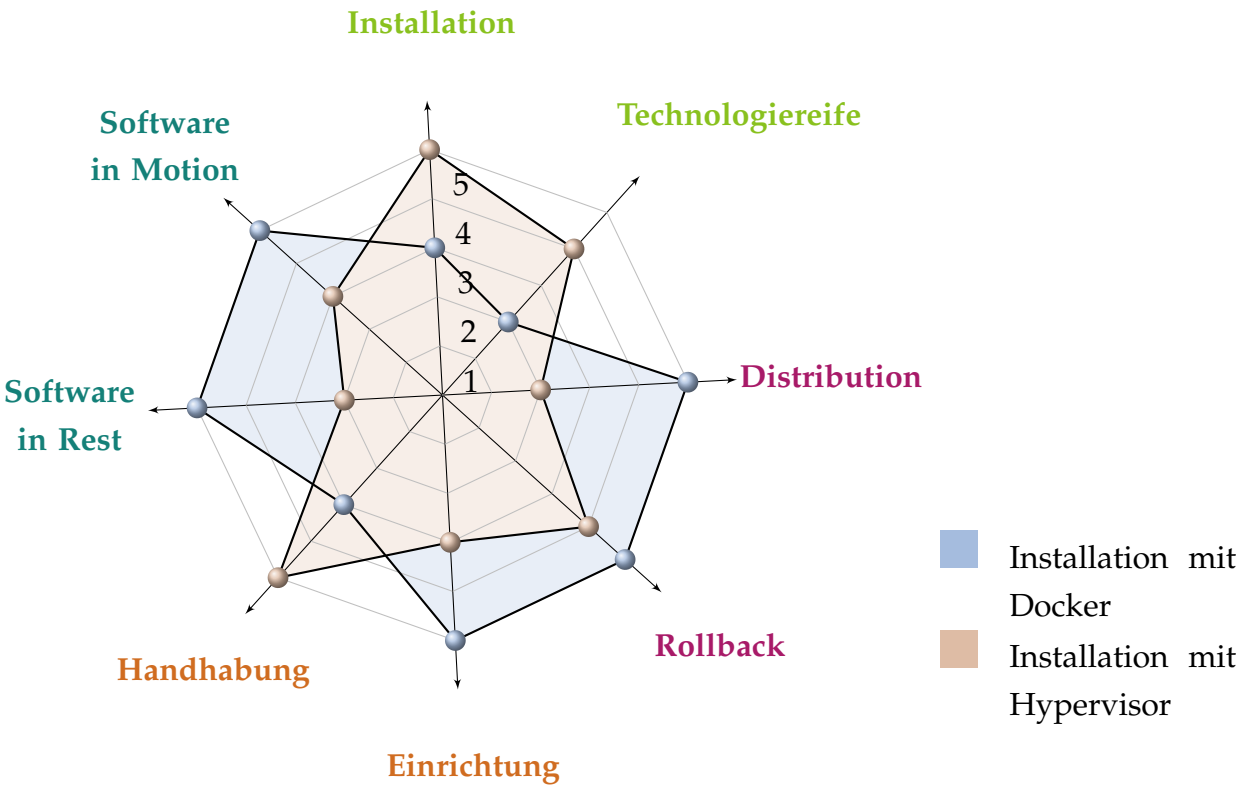
Abgrenzung Wir wollen nun untersuchen, welche Vor-und Nachteile sich durch die in dieser Arbeit vorgestellten Methode ergeben. hierfür werden einige Ausgewählte Punkte benutzt um eine Teamcenter Installation in Docker mit einer bloßen Installation auf einer (Hypervisor) VM entgegen zu stellen.

Obwohl in Serverlandschaften oft Virtualisierungen bereits als Basis verwendet werden, wollen wir an dieser Stelle davon ausgehen, dass Docker rein auf einer physischen Maschine und die vollvirtualsierte VM eine lokale Entwicklungsumgebung darstellt.
Zum Abschluss werden noch einige Mögliche Einsatzgebiete erläutert.

1. **Installation** Wie einfach lässt sich das System erstmalig installieren?
2. **Technologiereife** Wie ausgereift ist die verwendete Technologie? Wie fehleranfällig?
3. **Software in Motion** Wieviele Laufzeitressourcen werden für eine Instanz benötigt?
4. **Software in Rest** Wieviel Speicherplatz werden für eine ruhende Teamcenter Instanz benötigt?
5. **Rollback** Wie gestaltet sich das zurückrudern auf ältere Versionen?
6. **Distribution** Wie einfach lassen sich verschieden Versionen verteilen? Bereitstellen?
7. **Einrichtung** Wie einfach lässt sich ein bereits konfiguriertes System benutzen? Für Administratoren?
8. **Handhabung** Wie einfach ist der Umgang mit Teamcenter?

Wir wollen diese Themen nun grob einordnen: **Integrationsmöglichkeit**, **Ressourcennutzung**, **Umgang mit den Technologien**, **Versionsverwaltung**

5.1 Gegenüberstellung



5.1.1 Installation

Docker Entgegen einer nativen Teamcenter Installation auf einer VM müssen bei einer Aufteilung in Docker Container einige Abhängigkeiten berücksichtigt werden. Hierfür müssen virtuelle Netzwerke erzeugt werden, die eine Kommunikation unterhalb dieser Container erlauben. Diese können zwar sehr einfach erstellt, müssen aber dennoch in die Planung mit aufgenommen werden und erzeugen auf diese Weise zusätzliche Komplexität.

Container sind vom eigentlichen System abgeschottet. Deshalb muss auf ein *mounting* von Hostordnern geachtet werden, damit die einzelnen Container über-

haupt auf die Installationsdateien zugreifen können. Dies ist ein weiterer Punkt, den man beachten muss.

Die Installation selbst wird über einen X-Server durchgeführt, wobei es auch möglich wäre, einen Container mit einer grafischen Benutzeroberfläche (*KDE*, *GNOME*,...) nachzurüsten. Dies fällt aber nicht in die Wertung mit ein, weil auch virtuelle Server meistens ohne Grafik installiert werden.

Die Tatsache, dass eine Menge Applikationen bereits vorgefertigt und einsatzfähig aus der Docker Hub heruntergeladen werden können, vereinfacht die Installation enorm, da man einige Komponenten, beispielsweise die Datenbank, einfach wie bei einem Lego Baukasten mitbenutzen kann ohne diese Komponenten eigens konfigurieren zu müssen.

Da die Installation doch einige Punkte beinhaltet, an die gedacht werden müssen und dies die Komplexität der Installation noch weiter steigert werden an dieser Stelle drei von fünf Punkten verliehen.

VM Auch bei einer Installation in einer VM müssen die Installationsdaten zuerst irgendwie in die Maschine gebracht werden. Entgegen den Container kann diese aber zu jeder Zeit vom Provider[1] erledigt werden. Auch sind hier über *Guest Extensions* bidirektionale Drag-and-Drop Verbindungen möglich.

Den Container kann wirklich nur beim erzeugen gesagt werden, welche Daten gemeinsam genutzt werden können. Wenn dies also vergessen wurde, muss der Container zuerst in ein Image gespeichert und dann aus diesem ein neuer Container erzeugt werden.

Meistens kommen Betriebssysteme, wie sie in der VM benutzt werden bereits mit vorinstallierter grafischer Oberfläche. Dies macht die Installation von Teamcenter sehr leicht, da auf die normale Menüführung zurückgegriffen werden kann.

Da die Installation in Docker Container um einiges komplizierter und zeitaufwändiger ist, gibt es für die VM volle Punktzahl.

5.1.2 Technologiereife

Docker Die Praxis hat gezeigt, dass eine Docker Umgebung sehr fehleranfällig auf Manipulationen ist. So kann es passieren, dass bei unachtsamen Operationen

[1] VBox, VMWare

des Docker-Users die ganze Engine den Betrieb einstellt, weil aus versehen eine falsche Datei verändert wurde. Da es (noch) keine wirklichen Reparaturfunktionen gibt, müssen bei einem Ausfall Verzeichnisse neu erstellt, bzw. Docker neu installiert werden. Dies führt dazu, dass Images/Container, die nicht gesichert sind neu eingelesen werden müssen und im schlimmsten Falle verloren gehen. Daher ist es auch enorm wichtig, Rechte für Docker-Benutzer richtig zu verteilen.

Auch ist Docker selbst eine sehr neue Technologie (OpenSource seit etwa 2013), die zwar monatlich mit neuen Versionen versorgt wird, jedoch an vielen Stellen noch an der Stabilität arbeiten muss. Somit verändern sich in regelmäßigen Abschnitten auch Dinge und es kann sein, dass Funktionen, die heute noch State-of-The-Art sind im nächten Release bereits als veraltet gelten.

Des weiteren verfügen Windows Container noch nicht über den Funktionsumfang, den wir für unsere Zwecke bräuchten. So fehlt zum Beispiel eine Möglichkeit per Remote Desktop (RDP) den 2T-RC aufzurufen. Diese Funktionen werden zwar nach Meinung des Autors in nächster Zeit nachgeliefert, werden aber dennoch einige Monate brauchen, bis sie komplett stabil laufen.

Deswegen fällt auch die Entscheidung Docker hier nur zwei Punkte zu verleihen.

VM Die VM selbst läuft sehr stabil. Beispielsweise wurde die erste Version von *Oracle Virtual Box* bereits 2004[4] veröffentlicht und blickt nun über eine Dekade Updates und Verbesserungen zurück.

Die meisten schwerwiegenden Fehler rühren von falsch manipulierten Daten her und sind meistens die Schuld des Anwenders. Jedoch sind natürlich auch hier Fehler im virtuellen System nicht ausgeschlossen und es kann immer passieren, dass etwas nicht mehr so funktioniert, wie es eigentlich sollte, obwohl man den Grund nur sehr schwer findet.

Zusätzlich ist mit den Jahren ein sehr großer Erfahrungsschatz gewachsen, der sich im Umfang der Dokumentationen und Einträge in Online Foren auszeichnet.

5.1.3 Distribution

Docker Die Container können sehr einfach benutzt werden. Mit einem

```
1 $ docker load imageName.tar.gz
```

können sie von externen Speichermedien importiert werden. Ferner können mit dem

```
1 $ docker pull imageName
```

pull-Befehl sogar externe Repositories genutzt werden, von denen man immer die aktuellste Version herunterladen kann. Hierfür müssen nur die tatsächlichen Dateiänderungen gegenüber den Versionen geladen werden und können so sehr schnell allein mit offiziellen Methoden auf eine große Menge an Personen Verteilt werden.

Daher kann man sagen, dass die Verteilung der Images sehr einfach und effizient möglich ist.

VM VMs werden meistens über externe Speichermedien von Rechner zu Rechner gebracht. Verschieden Versionen werden geteilt, indem die ganze VM kopiert wird, damit jeder wieder auf dem gleichen Stand ist und nehmen damit eine Menge Speicherplatz ein. Die vollen Kopiervorgänge dauern auch häufig sehr lange und verhindern eine strukturierte Verteilung der einzelnen Versionen. Damit drifften die verschieden VM Stände innerhalb verschiedener Projekte sehr schnell auseinander und es gibt meistens keine konsolidierte Version, bis sich jeder wieder einen aktuellen Stand komplett kopiert.

5.1.4 Rollback

Docker Durch das Git-ähnliche Versionierungssystem können auch ältere Stände im System behalten werden, ohne viel Speicherplatz zu fressen. Dies ergibt sich erneut aus der Lösung, dass nur die Änderungen an den einzelnen Dateischichten gespeichert werden.

Ein weiterer großer Vorteil ist jedoch auch die Geschwindigkeit, mit der alte Versionen eingespielt werden müssen. Es muss nämlich hierfür nicht eine neuer Snapshot oder eine VM neu gebootet werden, sondern aus dem alten Image wird einfach nur ein Container erstellt, der sofort in das Dateisystem eingebunden und gestartet wird. Auf diese Weise können auch ältere Versionen in unter sekundenschnelle eingespielt werden.

Zusätzlich können verschiedene Versionen *kombiniert* werden. Da jeder einzelne

Container getrennt gespeichert wird können auch einzelne Container ausgetauscht werden. Man kann beispielsweise verschiedene Versionen der Teamcenter Komponenten abspeichern, die verschiedenen zusätzliche Funktionen und Versionen bereitstellen und dann aus diesen ausprobieren, wie sich diese untereinander vertragen. Man kann also wie bei einem Lego Baukasten verschiedene Images derart kombinieren, dass sie den eigenen Anforderungen bestmöglich entsprechen.

VM Auch hier gibt es Snapshots, die eine Speicherung der Deltas erlauben. Auf diese Weise können die Snapshots speichertechnisch niemals größer als die Haupt VM werden. Jedoch muss bei einem zurückrudern auf ältere Versionen die Maschine komplett neu gebootet werden und eventuell eine Zeit, die der Snapshot zum laden braucht in Anspruch genommen werden. Auch ist es nicht möglich die einzelnen Komponente zusammenzustecken, weil nur Teamcenter als ganzen in dem Snapshot festgehalten wird. Bei den Containern sind die einzelnen Bestandteile strikt getrennt.

Auf diesem Gebiet kann sich daher Docker ganz klar durchsetzen.

5.1.5 Einrichtung

Docker Die Images selber können sehr einfach benutzt werden. Über einen einzigen Befehl können verschiedene Teamcenter Container verbunden und gestartet werden. Das Erstellen und Aufsetzen des Containers funktioniert in Sekundenschnelle und so hat man bereits nach kürzester Zeit - abhängig von der Leistung der Maschine - eine vollständige Teamcenter Umgebung.

Da Docker Container abgekapselte Umgebungen sind, ist es auch möglich, mehr als eine Teamcenter Instanz auf einer Maschine gleichzeitig laufen zu lassen. Ferner müssen diese nicht einmal seperat Konfiguriert werden, sondern können mit Hilfe der in dieser Arbeit vorgestellten Lösung für einige Probleme einfach nebeneinander betrieben werden, ohne für jede Umgebung eigenes Ressourcen emulieren zu müssen, wie dies bei der VM der Fall ist.

VM Das starten einer VM ist sehr leicht. jedoch müssen Parameter für die VM selbst konfiguriert werden - beispielsweise Portweiterleitungen oder Ressourcenallokationen. Dies kann mit Tools wie *Vagrant* vereinfacht werden.

Mehrere Instanzen auf einer VM laufen zu lassen ist zwar möglich, jedoch sehr aufwändig, da jede Instanz eigens Installiert. Man könnte nun für jede Instanz eine eigene VM starten. Der Nachteil dieser Methode ist jedoch, dass dann zwei mal emulierte Ressourcen zur Verfügung gestellt werden müssten.

5.1.6 Handhabung

Docker Der **Enduser** sieht den Webclienten so, wie er es gewohnt ist, ohne zu bemerken, dass im Hintergrund ein Dockersystem läuft. Etwas Komplizierter ist es für die Infrastrukturadministratoren, die direkt mit dem System in Kontakt kommen. Dieser muss seine Containerstruktur genau kennen und die einzelnen Containern immer erst mit

```
1 $ docker exec -it xxx /bin/bashg
```

betreten, bevor er direkt mit dem Dateisystem in Berührung kommt und die Dienste starten kann, die er benötigt. Ferner wird er immer einen X-Client oder zumindest eine RDP/VNC Programm benötigen, um grafisch erscheinen lassen zu können.

VM Eine normale VM erlaubt es häufig bereits bei der Erstinstallation der Maschine, eine Grafik mit zu installieren, weswegen man sich im weiteren Verlauf keine Gedanken mehr in diese Richtung machen muss und direkt auf einer Oberfläche gearbeitet werden kann. Damit auch volle Punktzahl.

5.1.7 Software in Rest

Docker Dank des Union File Systems (UFS) speichert Docker nur das Delta gegenüber anderen ab. Beispielsweise läuft der Webclient und die Teamcenter Instanz auf dem gleichen SLES-Linux Grundimage. Dieses liegt aber nur einmal im System gespeichert und beide Container greifen auf die gleichen Grundschichten zu. Nur die wirkliche Differenz der beiden Container muss abgespeichert werden.

Auch mehrere Teamcenter Instanzen profitieren sehr von dieser Eigenschaft: Erstellt man nun zwei parallel auf einer Maschine, dann nutzen diese die gleichen Grundimages und benötigen nur einmal den Speicherplatz für das Teamcenter ”Grundgerüst” im Umfang von etwa 20 GB. Sie speichern wirklich nur neu ab,

was an Daten hinzukommt. Dies ist ein riesige Vorteil gegenüber einer echten Virtualisierung, die dies nicht erlaubt. Höchstwertung!

VM Die echte Virtualisierung benötigt für jede Installation den Speicherplatz, den die Installation benötigt. Es kann bei mehreren Instanzen nichts gespart werden und der Speicherplatz wird doppelt benötigt. Hierfür ziehen wir drei Punkte ab.

5.1.8 Software in Motion

Docker Wie in 2.1.1.3 erklärt, nutzt Docker den Kernel des Hosts und es entsteht quasi[2] kein Virtualisierungsschwund. Dies liegt daran, dass für Docker keine Laufzeitressourcen emuliert werden müssen und ist eines der Hauptvorteile bei der Entscheidung für ein Dockersystem.

Daher auch die volle Punktzahl in diesem Bereich.

VM Eine Voll-/Paravirtualisierung erzeugt einen Overhead an Laufzeitressourcen, da alle Ressourcen mindestens einmal komplett emuliert werden müssen. Die Technik ist heutzutage jedoch soweit, dass allerhöchstens ein Verlust von 5-10% auftritt.

Hierfür ziehen wir jedoch ein paar Punkte ab und vergeben drei von fünf.

5.2 Fazit

Insgesamt lässt sich sagen, dass Docker seine Stärken in der Ressourcennutzung und der Versionsverwaltung ausspielt, wohingegen die Hypervisor VM der Docker Technologie in Sachen Integration der Technologien und dem Umgang mit ihnen ausspielt.

Man muss aber sagen, dass es im echten Umfeld nur sehr selten ist, eine Docker Engine direkt auf einer physischen Maschine zu installieren, obwohl dies natürlich möglich ist. In den meisten Fällen liegt aus Gründen der Aufteilung und Isolation bestehender Ressourcen doch ein Virtuelles System im Hintergrund. Daher

[2] Wie bereits erwähnt etwa 3%

wird in der Realität auch eine kombinierte Mischlösung eingesetzt werden, die die Portabilität und Mobilität der Container mit den "sauber" getrennten virtuellen Maschinen verbindet.

5.3 Mögliche Anwendungsszenarien

Es gibt nun viele Möglichkeiten, die Kombination aus Teamcenter und Docker vorteilhaft zu nutzen. In diesem Abschnitt wollen wir einfach kurz zwei Fälle skizzieren.

5.3.1 Kontinuierliche Integration

Bei einer kontinuierlichen Integration will man in der Software Entwicklung einzelne Komponenten der Applikation fortlaufend zusammenfügen. Mithilfe eines Repository Servers kann man alle Änderungen, die sich in der letzten Zeit ergeben haben, auf effiziente Weise allen Teammitgliedern zur Verfügung stellen. Diese müssen nicht mehr riesige Virtuelle Maschinen kopieren, sondern brauchen sich nur die jeweiligen Änderungen vom Server holen und sind damit alle wieder auf dem gleichen Stand. Wenn diese nun eine Änderung gemacht haben, können Sie diese auf den Server hochladen und andere Mitarbeiter können sich die Änderungen anschauen, indem Sie einfach aus dem neuen Image eine Teamcenter Instanz starten. Wenn diese Änderung nicht die gewünschten Funktionen aufweist, kann sie auch einfach wieder verworfen werden.

Somit existiert immer ein Hauptversion, die fortlaufend mit den aktuellsten Ständen aktualisiert werden kann und voll funktionsfähig ist.

Mit den Ergebnissen dieser Arbeit wurde bereits ein kleines Projekt bei einem bayrischen Automobilhersteller gewonnen, welches untersucht, wie ein kontinuierliches Deployment von Teamcenter auf Docker in einer *OpenStack* Plattform funktioniert.

5.3.2 Bereitstellung lokaler Entwicklungsumgebungen

Da die Docker Images portabel und quasi auf jeder Linux Distribution laufen können, kann man auf diese Weise jedem Experimentierfreudigen, der gerne einen

aktuellen Stand hätte diese, Images geben, damit er sich eine kleine lokale Testumgebung aufbauen kann. Der große Vorteil ist, dass er sich nicht mit irgendwelchen Konfigurationen herumschlagen muss, sondern eine fertige Instanz einfach einbinden und starten kann, was innerhalb von Sekunden passiert. Sollte er einmal einen Fehler machen, braucht er nur die Image neu zu erzeugen um den Stand zurück zu rollen.

5.3.3 Besserer Ausnutzung leistungsstarker Maschinen durch Parallelbetreibung

Wie wir bereits gesehen haben können wir mithilfe von Docker auch mehrere Teamcenter Instanzen sehr performant auf einer einzigen Maschine betreiben. Damit können Leistungsstärkere Maschinen höher ausgelastet werden.

Durch das UFS wird nur der jeweilige Unterschied der Teamcenter Instanzen zu den Basis Images extra abgelegt. Das Grundimage hingegen benötigt nur einmal den Speicherplatz. Wenn man nun bedenkt, dass eine minimale Teamcenter Installation schnell 20 bis 30 GB an Speicherplatz benötigt, sieht man sehr schnell, dass man hierbei eine Menge Ressourcen sparen kann.

6 Schluss

Es ist sehr Wahrscheinlich, dass Container ein weiteres Mal "die Welt verändern" werden. Die steigende Popularität und Unterstützung, nicht zuletzt durch *Microsoft* lässt auf hohe Erwartungen und Akzeptanz dieser Technologie schließen.

In dieser Arbeit wurde gezeigt, wie die Container Technologie genutzt werden kann, um Teamcenter zu modularisieren und zu isolieren. Dies macht es uns nun sehr leicht, dieses Softwarepaket auch anderen Personen zur Verfügung zu stellen.

Die Container Technologie wird zwar bestehende Virtualisierungstechniken, vor allem im Server Umfeld nicht ablösen, jedoch als Ergänzung seinen Beitrag leisten, denn durch die nun mögliche Modularisierung werden Container und Images die bevorzugte Art werden, Software auszuliefern und zu warten.

Auch der Warenumschlag bei echten Containerschiffen steigt seit Jahren rapide an. Nehmen wir das in unsere Metapher mit auf und erwarten, dass das selbe mit (Docker) Containern passieren wird, um eine *Containerrevolution 2.0* in die Wege zu leiten.

7 Anhang

7.1 Vagrantfile

Code 7.1: Beschreibung der virtuellen Maschine mittels Vagrant

```
1  Vagrant.configure("2") do |config|
2
3    # Setting up Proxy Environment
4    if Vagrant.has_plugin?("vagrant-proxyconf")
5          config.proxy.http = "http://10.0.2.2:3128/"
6          config.proxy.https = "http://10.0.2.2:3128/"
7                              # We use local NTLM proxy
8      config.proxy.no_proxy ="localhost,127.0.0.1"
9    end
10   config.vm.box = "vagrant_docker"
11   # Mounts external Files into VM
12   config.vm.box_url = 'file:///D:/vbox/centos7_vagrant_docker.box'
13   config.vm.synced_folder  "D:", "/mnt/DDrive"
14   config.vm.synced_folder "D:/tc11.2_docker", "/tcrepo"
15
16   config.vm.provider "virtualbox" do |vb|
17     # Display the VirtualBox GUI when booting the machine
18     vb.gui = true
19     vb.name ="vagrant_docker"
20     vb.customize ["modifyvm", :id, "--memory", "4000"]
21     vb.customize ["modifyvm", :id, "--cpus", "2"]
22     vb.customize ["modifyvm", :id, "--clipboard", "bidirectional"]
23     vb.customize ["modifyvm", :id, "--draganddrop", "bidirectional"]
24     # Customize the amount of memory on the VM:
```

```
25      vb.memory = "4000"
26    end
27
28    config.vm.network "forwarded_port", guest: 9000, host: 9000
29                    # Portainer Port
30    config.vm.network "forwarded_port", guest: 8080, host: 8080
31                    # Apache2
32  # ... more forwarded ports...
33    config.ssh.username = "vagrant"
34    config.ssh.password = "vagrant"
35  end
```

7.2 Docker Compose

```
1  version: "2"
2  services:
3          teamcenter:
4                  build:
5                  context: ./Dockerfiles
6                  dockerfile: teamcenter
7                  ports:
8                  - $fmsport:$fmsport
9                  extra_hosts:
10                 - "tclicense:10.0.2.2"
11                 environment:
12                 - tc_name=$tc_name
13                 - tc_fmsport=$fmsport
14                 - DB_TIMEOUT=$DB_TIMEOUT
15                 #volumes:
16                 #- /tcrepo/:/tcrepo/
17                 networks:
18                         net:
19                                 aliases:
20                                 - teamcenter
21                                 - $tc_name
22         webclient:
23                 build:
```

```
24                  context: ./Dockerfiles
25                  dockerfile: tomcat7
26                  ports:
27                  - $client_port:8080
28                  - $TCFTSIndexer:$TCFTSIndexer
29                  networks:
30                          net:
31                                  aliases:
32                                  - webclient
33                  environment:
34                  - tc_fmsport=$fmsport
35                  - client_port=$client_port
36                  - fsc_url=$fsc_url
37                  #volumes:
38                  #- /tcrepo/:/tcrepo
39          tcdb_oracle:
40                  build:
41                  context: ./Dockerfiles
42                  dockerfile: tcdb_oracle
43                  networks:
44                          net:
45                                  aliases:
46                                  - tcdb
47                                  - tcdb_oracle
48                  entrypoint: "/entrypoint.sh"
49  networks:
50          net:
51                  driver: bridge
```

Literaturverzeichnis

[1] Peter Arijs. Docker usage stats: Adoption up in the enterprise, and production. https://dzone.com/articles/docker-usage-statistics-increased-adoption-by-ente, 2016. (05/10/17).

[2] DevOps.com & ClusterHQ. Container market adoption - survey 2016. https://clusterhq.com/assets/pdfs/state-of-container-usage-june-2016.pdf, 2016. (07/28/17).

[3] Anne Curie. How many public images are there on docker hub? https://medium.com/microscaling-systems/how-many-public-images-are-there-on-docker-hub-bcdd2f7d6100, 2016. (06/06/17).

[4] Dr. Oliver Diedrich. Virtualbox. https://www.heise.de/ct/artikel/VirtualBox-222035.html, 2007. (08/08/17).

[5] Björn Stelte Dr. Udo Helmbrecht, Prof. Dr. Gunnar Teege. Virtualisierung: Techniken und sicherheitsorientierte anwendungen. https://www.unibw.de/inf2/pub-en/getFILE?fid=5811473&tid=pub-en, 2010. (06/06/17).

[6] The Linux Foundation. Open container initiative. https://www.opencontainers.org/, 2017. (08/14/17).

[7] Docker Inc. Docker company. http://content.schweitzer-online.de/static/catalog_manager/live/media_files/representation/zd_std_orig__zd_schw_orig/036/442/834/9783864903847_content_pdf_4.pdf, 2016. (05/10/17).

[8] Docker Inc. Docker company. https://www.docker.com/company, 2016. (05/10/17).

[9] Docker Inc. Docker compose. https://docs.docker.com/compose/, 2017. (06/06/17).

[10] IHS Global Insight. Umgebaute containertransportmenge in der weltweiten seeschifffahrt von 2010 bis 2022. https://de.statista.com/statistik/daten/studie/259570/umfrage/containertransportmenge-weltweit/, 2017. (07/28/17).

[11] Oliver Tigges Jerry Preissler. Docker - perfekte verbindung von microservices. https://www.sigs-datacom.de/uploads/tx_dmjournals/preissler_tigges_OTS_Architekturen_15.pdf, 2017. (08/14/17).

[12] Kristian Kißling. Docker 0.9 ersetzt lxc. http://www.linux-magazin.de/NEWS/Docker-0.9-ersetzt-LXC, 2014. (08/08/17).

[13] Oracle. Licensing: Frequently asked questions. https://www.virtualbox.org/wiki/Licensing_FAQ, 2017. (08/13/17).

[14] ORF.at. Umgebaute 'fairyland' war nur der beginn. http://orf.at/stories/2337768/2337767/, 2016. (07/28/17).

[15] Margaret Rouse. Definition: Containerbasierte virtualisierung. http://www.searchdatacenter.de/definition/Container-basierte-Virtualisierung, 2015. (06/12/17).

[16] Prof. Dr. Günther Schuh. Docker 0.9 ersetzt lxc. http://www.linux-magazin.de/NEWS/Docker-0.9-ersetzt-LXC, 2014. (08/08/17).

[17] Peter Siering. Linux-container bald nativ unter windows. https://blogs.technet.microsoft.com/hybridcloud/2017/04/18/dockercon-2017-powering-new-linux-innovations-with-hyper-v-isolation-and-windo 2017. (08/14/17).

[18] Wolfgang Sommergut. Neu im windows server: Linux-subsystem, smb für container, insider program. https://www.windowspro.de/wolfgang-sommergut/neu-windows-server-linux-subsystem-smb-fuer-container-insider-program, 2017. (08/08/17).

[19] Manoj Tiwari. Teamcenter fms overview. https://www.plm.automation.siemens.com/de_de/Images/21848_tcm73-100227.pdf, 2013. (06/12/17).

[20] Manoj Tiwari. Teamcenter fms overview. http://teamcenterplm.blogspot.com/2013/06/teamcenter-fms-overview.html, 2013. (06/12/17).

[21] James Turnbull. *The Docker Book* -. Lulu.com, Raleigh, North Carolina, 2014.

[22] Unknown. Microsoft announces native linux containers for windows. https://mspoweruser.com/microsoft-announces-native-linux-containers-windows/, 2016. (05/10/17).

[23] Unknown. Whatś lxc. https://linuxcontainers.org/lxc/introduction/, 2017. (08/08/17).