Masterarbeit im Studiengang Computer Science and Media

Portierung einer Enterprise OSGi Anwendung auf eine PaaS

Thomas Uhrig
(Matrikelnummer 25826)

(Cessna EC-2, Quelle: Wikimedia Commons, 2013)

1. Juli 2014

Hochschule der Medien Stuttgart
Informatica Corporation

HOCHSCHULE DER MEDIEN

Erstprüfer: Prof. Dr. Martin Goik
Zweitprüfer: Dipl.-Ing. Thomas Kasemir

Erklärung an Eides statt

Hiermit versichere ich, Thomas Uhrig, an Eides statt, dass ich die vorliegende Masterarbeit mit dem Titel: „Portierung einer Enterprise OSGi Anwendung auf eine PaaS“ selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.

Ich habe die Bedeutung der eidesstattlichen Versicherung und die prüfungsrechtlichen Folgen (§26 Abs. 2 Bachelor-SPO (6 Semester), § 23 Abs. 2 Bachelor-SPO (7 Semester) bzw. § 19 Abs. 2 Master-SPO der HdM) sowie die strafrechtlichen Folgen (gem. § 156 StGB) einer unrichtigen oder unvollständigen eidesstattlichen Versicherung zur Kenntnis genommen.

____

(Datum, Unterschrift)

Abstrakt

Cloud Computing ist ein vieldiskutiertes Thema der Softwareindustrie. Anwendungen und Dienste werden in die Cloud verlagert und ein Ökosystem neuer Technologien und Anwendungsarchitekturen entwickelt sich. Die vorliegende Arbeit untersucht, wie dieses Ökosystem für eine bestehende (OSGi) Anwendung genutzt werden kann.

Gegenstand der Evaluation ist der *Informatica PIM Server*, ein Softwareprodukt der *Informatica Corporation* (ehemals *Heiler Software AG*) zum Management von Produktdaten. Dieser wird zunächst vorgestellt und dessen technische Umsetzung mit OSGi beleuchtet.

Anschließend wird Cloud Computing als Technologie eingeführt. Die Servicemodelle *IaaS*, *PaaS* und *SaaS* werden definiert und ein Schema zur Einordnung von Cloud-Angeboten vorgestellt. Nachfolgend wird eine Deployment-Strategie für den PIM Server in die Cloud entwickelt. Hierzu werden verschiedene Methoden verglichen und hinsichtlich ihrer Eigenschaften bewertet. Die Anbindung von Cloud-Diensten wird ebenfalls diskutiert.

Als Ergebnis der Untersuchung wird im weiteren Verlauf *Docker* als Anwendungscontainer vorgestellt und verwendet. Ein prototypisches Werkzeug zeigt, wie ein konkretes Deployment des PIM Servers und angegliederter Komponenten mit Hilfe von Docker auf die Amazon Web Services umgesetzt werden kann. Das Werkzeug verfügt über eine Kommandozeilenschnittstelle und eine Weboberfläche zum Überwachen des Deployments. Deployment-Einheiten und Konfigurationen werden lokal und in einem Cloud-Speicher verwaltet.

Abschließend werden die Ergebnisse bewertet und hinsichtlich ihrer PaaS-Eigenschaften eingeordnet. Des weiteren wird ein Ausblick für anschließende Untersuchungs- und Entwicklungsschritte gegeben.

Schlagwörter: *Cloud*, *OSGi*, *Docker*, *AWS*

Abstract

Cloud computing is a heavily discussed topic in the software industry. Applications and services are shifted towards the cloud and an ecosystem of new technologies and system architectures is evolving. The presented thesis evaluates how this ecosystem can be utilized for an existing (OSGi) application.

The subject of the evaluation is the *Informatica PIM server*, a software product of the *Informatica Corporation* (formally *Heiler Software AG*) for product data management. This product as well as its technical implementation based on OSGi will be introduced in the first part of the thesis.

Afterward, cloud computing as a technology will be introduced. The service models *IaaS*, *PaaS* and *SaaS* will be defined and a model for categorizing cloud offers will be presented. A deployment strategy for the PIM server in the cloud will be developed in the following. Different approaches will be compared and assessed according to their capabilities. Besides, the integration of cloud services will be discussed.

As the result of the evaluation, *Docker* will be presented as the application container hereinafter. By using Docker, a prototypical tool demonstrates an actual deployment scenario for the PIM server and related components on the Amazon Web Services. The tool provides a command line interface as well as a web interface for monitoring the deployment process. Deployment units and configurations are stored locally as well as in a cloud storage.

Finally, the results will be assessed and categorized according to their PaaS characteristics. Additionally, an outlook for subsequent evaluation and development steps will be given.

Keywords: *Cloud*, *OSGi*, *Docker*, *AWS*

Vorwort

Die vorliegende Abschlussarbeit entstand im Masterstudiengang *Computer Science and Media* (Master) der *Hochschule der Medien Stuttgart* in Zusammenarbeit mit der *Informatica Corporation*. Ziel der Abschlussarbeit ist die Portierung einer bestehenden Enterprise OSGi Anwendung auf eine *Platform as a Service* (kurz *PaaS*).

Ich bedanke mich bei Andreas Bühler, Andreas Itzelberger, Achim Grosshans und Ralf Engel für die zahlreichen Diskussionen und Hilfestellungen zum Informatica PIM Server. Viele Ideen wären ohne euch nicht entstanden. Thomas Klingler und Alexander Altmayer gilt mein Dank für ihre Unterstützung in den alltäglichen Untiefen der Softwareentwicklung - egal ob eine Datenbank nicht erreichbar war oder ein Projekt nicht kompilierte, sie standen mir jederzeit gerne zur Seite. Ein herzlicher Dank geht an dieser Stelle auch an Nenad Jovanovic für das aufschlussreiche Telefoninterview über die Amazon Web Services - thank you for your insights and tips on AWS! Thomas Waible danke ich für die zahlreichen Links und Hinweise zur Oracle Datenbank unter Linux. Dem gesamten Team des PIM Servers, egal ob Plattform, Web oder QA, gilt gleichsam mein Dank für die (mittlerweile) fast eineinhalbjährige Unterstützung in Werksstudententätigkeit und Masterarbeit.

Ein besonderer Dank geht an meine Betreuer Stefan Röck und Thomas Kasemir. Sie standen mir während meiner gesamten Zeit bei Heiler bzw. Informatica jederzeit mit viel Engagement zur Seite.

Ebenso möchte ich mich bei meinem betreuenden Dozenten an der HdM Stuttgart, Prof. Martin Goik, bedanken.

Mein abschließender Dank gilt meiner Familie, meiner Freundin Katharina (Danke für die Kommas!) und meinen Freunden, die mich in den vergangenen sechs Jahren meines Studiums unterstützt haben. Euer Rückhalt hat mir sehr geholfen.

Stuttgart, im Sommer 2014.

Thomas Uhrig

Inhaltsverzeichnis

I. Grundlagen **13**

1. Der Informatica PIM Server **14**
1.1. Funktionale Aspekte . . . 14
1.2. Technische Aspekte . . . 15

2. Einführung in OSGi **17**
2.1. Grundlagen von OSGi . . . 17
2.1.1. Bundles . . . 18
2.1.2. Features . . . 19
2.1.3. Products . . . 19
2.1.4. OSGi Services und Extension Points . . . 20
2.2. Modularität in OSGi . . . 20
2.3. OSGi und Modularität im Informatica PIM Server . . . 21

II. Cloud Computing **22**

3. Cloud Computing **23**
3.1. Servicemodelle . . . 24
3.1.1. Infrastructure as a Service (IaaS) . . . 24
3.1.2. Platform as a Service (PaaS) . . . 25
3.1.3. Software as a Service (SaaS) . . . 26
3.1.4. Model zur Beschreibung von Cloud-Services . . . 26
3.2. Liefermodelle . . . 28
3.3. Cloud-Komponenten . . . 29
3.4. Abgrenzung und Ausschluss von Themen . . . 30
3.4.1. Grid Computing . . . 30
3.4.2. Virtualisierung . . . 30
3.4.3. SaaS . . . 31
3.4.4. Mandantenfähigkeit . . . 31
3.4.5. Outsourcing . . . 31
3.4.6. Wirtschaftlichkeit . . . 31
3.4.7. Sicherheit und Datenschutz . . . 31

III. Portierung in die Cloud 32

4. Motivation 33

5. Deployment in die Cloud 34
5.1. Deployment Strategien . . . 36
5.1.1. VM Images . . . 36
5.1.2. Manuelle Installation . . . 37
5.1.3. Docker . . . 38
5.1.4. JEE Archive . . . 39
5.1.5. OSGi-Bundles . . . 40
5.1.6. Source Code . . . 41
5.2. Deployment-Strategie für den Informatica PIM Server . . . 43
5.3. Alternativen für modulare OSGi-Anwendungen . . . 45

6. Skalierung 50
6.1. Vertikale Skalierung (*scale up*) . . . 50
6.2. Horizontale Skalierung (*scale out*) . . . 50
6.3. Elastizität . . . 51
6.4. Skalierung des Informatica PIM Servers . . . 51

7. Cloud-Services 52

IV. Prototypische Umsetzung 55

8. Cloud-Plattform 56
8.1. Amazon Web Services . . . 56
8.1.1. AWS REST API . . . 57
8.1.2. AWS EC2 . . . 57
8.1.3. AWS S3 . . . 57
8.1.4. AWS RDS . . . 57
8.1.5. AWS Elastic IP . . . 58
8.2. Vergleichbare Angebote . . . 58

9. Docker als Application-Container 60
9.1. Docker . . . 60
9.2. Einordnung von Docker in die Cloud-Landschaft . . . 62
9.3. Vergleichbare Angebote . . . 64
9.4. Entwicklungsabläufe mit Docker . . . 64
9.4.1. Tests . . . 64
9.4.2. Entwicklung . . . 65
9.4.3. Dokumentation . . . 65
9.5. Elastic Beanstalk und Docker . . . 66

10. Deployment **67**
10.1. Deployment-Werkzeuge 67
10.2. Cloud-Init 69
10.3. Entwicklung eines Deployment-Tools 69
10.3.1. Implementierung 69
10.3.2. Ablauf des Deployments 70
10.4. Logging, Monitoring und Konfigurationsmanagement 70
10.4.1. Logging 71
10.4.2. Monitoring 72
10.4.3. Konfigurationsmanagement 73

V. Diskussion und Fazit **75**

11. Bewertung der Ergebnisse **76**

12. Hindernisse und Probleme **77**

13. Ausblicke und nächste Schritte **78**

VI. Anhang **80**

Literaturverzeichnis **90**

Abbildungsverzeichnis

1.1. Informatica PIM (aus Informatica, 2014) 15
1.2. Web- und Desktopoberfläche des Informatica PIM Servers 16

2.1. Model der OSGi-Laufzeitumgebung (aus OSGi Alliance, 2014a) 17
2.2. Bundle Life Cycle (nach McAffer et al., 2010, Seite 23) 18
2.3. Products, Features und Bundles 19
2.4. Anzahl der OSGi-Bundles im Informatica PIM Server 21

3.1. Google Trends Chart für das Suchwort „Cloud“ 23
3.2. Cloud-Servicemodelle 24
3.3. Model zur Beschreibung von Cloud-Services (nach Haan, 2013) 27
3.4. Cloud-Liefermodelle 28

5.1. Deployment-Strategien für die Cloud 34
5.2. Deployment mittels VM Images 36
5.3. Manuelle Installation 37
5.4. Deployment mittels Docker (nach Docker Inc., 2014a, Abbildung 8) 38
5.5. Deployment mittels JEE-Archiven 39
5.6. Deployment mittels OSGi-Bundles 41
5.7. Deployment mittels Quellcode 41
5.8. Deployment-Strategie für den PIM Server mit Docker 45
5.9. OSGi Provisioning (nach de Vreede and Offermans, 2013, Folie 22) 46
5.10. Deployment-Oberfläche von Apache ACE (aus Apache ACE, 2014) 46

7.1. AWS Cloud-Dienste für den Informatica PIM Server 52

8.1. Einordnung von AWS in das Cloud-Modell von Haan, 2013 56

9.1. Vergleich von virtuellen Maschinen und Docker (nach Docker Inc., 2014a) 61
9.2. Einordnung von Docker in das Cloud-Modell von Haan [2013] 62
9.3. Docker in Entwicklung, Tests und Produktion 65

10.1. Übersicht der Deployment-Schritt im Web-Interface 71
10.2. Kibana mit Log-Information des PIM Servers 72
10.3. Push Updates vs. Pull Updates 74

11.1. Einordnung der entwickelten Lösung in das Modell von Haan, 2013 76

Tabellenverzeichnis

5.1. Kriterien zur Bewertung von Deployment-Strategien 35
5.2. Vergleich von Deployment-Werkzeugen für OSGi 49

8.1. Vergleich von Cloud-Anbietern 59

9.1. Vergleich von Docker und Heroku 63

10.1. Vergleich von Provisioning-Werkzeugen 68

13.1. Kriterien zur Bewertung von Deployment-Strategien 82
13.2. Bewertung von Deployment-Methoden 83

Nomenklatur

AWS	Amazon Web Services, das Cloud-Angebot von Amazon
Deployment	Verteilen und Installieren von Software
Deployment	Verteilung und Installation von Software
Docker	Anwendungscontainer, der in dieser Arbeit vorgestellt wird
Docker Hub	Offizielle und öffentliche Registry mit Weboberfläche
Docker Registry	Repository zum Speichern von Docker Images
EC2	Virtuelle Maschinen von Amazon (Linux und Windows)
Heroku	Bekannter Cloud-Anbieter, auf den Code via Git deployt wird
IaaS	Infrastructure as a Service
OSGi	Open Services Gateway initiative
PaaS	Platform as a Service
PIM	Produktinformationsmanagement
S3	Cloud-Speicher (Key-Value) von Amazon
SaaS	Software as a Service

Teil I.

Grundlagen

1. Der Informatica PIM Server

Der Informatica PIM Server ist ein Softwareprodukt zur Verwaltung von Produktdaten (engl. *Product Information Management*, kurz PIM). Er wurde bis 2013 von der *Heiler Software AG*[1] entwickelt, die nunmehr von der *Informatica Corporation*[2] akquiriert wurde. Die Entwicklung findet überwiegend am Standort Stuttgart statt. Der PIM Server wird auf Basis von Nutzerlizenzen vertrieben.

1.1. Funktionale Aspekte

Zusammen mit Desktop- und Webinterfaces bildet der PIM Server eine Umgebung zur Steuerung von Produktdaten. Abbildung 1.1 zeigt den funktionalen Aufbau dieser Umgebung.

Über das *Onboarding* werden Daten in das System eingespeist und durch eine *Staging Area* geleitet. Hier werden beispielsweise Datenqualitätsregeln geprüft (z.B. Feldtypen) und etwaige Konvertierungen vorgenommen. Die Daten werden dann in einem zentralen Katalog persistiert, in dem sie bearbeitet, durchsucht, administriert und exportiert werden können. Am Ende der Verarbeitungskette stehen Web Shops, Druckerzeugnisse oder der Austausch mit anderen Systemen.

Der PIM Server bietet eine Anbindung an Datenqualitätssysteme und Mediendatenbanken von Informatica. Für externe Anwendungen bietet er eine REST-Schnittstelle (die sogenannte *Service API*). Der Zugriff für Nutzer erfolgt überwiegend über ein Desktop-Programm, jedoch stehen auch Weboberflächen mit eingeschränkter Funktionalität zur Verfügung. Abbildung 1.2 zeigt die Web- und Desktopoberfläche des Informatica PIM Servers für interne Mitarbeiter (des Kunden). Systeme von Zulieferern können ebenfalls angeschlossen werden, wodurch Produktdaten von außerhalb an den Server weitergeleitet werden können. Über eine eigene Nutzer- und Rechteverwaltung können interne Mitarbeiter sowie externe Partner abgebildet werden.

In dieser Arbeit steht jedoch die technische Umsetzung des Servers im Vordergrund, nicht dessen Funktionalität als solche. Daher wird auf eine ausführlichere Beschreibung der Funktionalität an dieser Stelle verzichtet und stattdessen auf die offizielle Produktbeschreibung[3] verwiesen.

[1] Die *Heiler Software AG* war ein mittelständisches Softwareunternehmen aus dem Raum Stuttgart und entwickelte vor allem E-Commerce Produkte. Mehr unter www.heiler.com.

[2] Informatica ist ein U.S.-amerikanisches Softwarekonzern mit mehr als 2700 Mitarbeitern weltweit. Zum Kerngeschäft gehören Produkte zum Datenmanagement. Mehr unter www.informatica.com.

[3] www.informatica.com/de/products/master-data-management/product-information-management

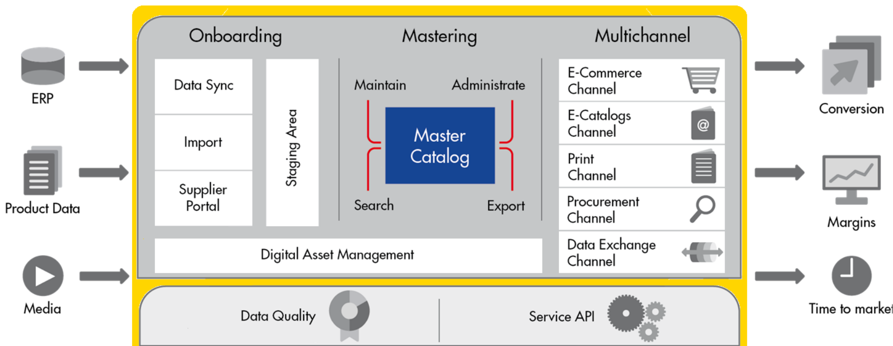

Abbildung 1.1.: Informatica PIM (aus Informatica, 2014)

1.2. Technische Aspekte

Der PIM Server ist eine klassische 3-Tier-Architektur. Er besteht aus einer Datenhaltungsschicht (unterstützt wird eine Oracle- oder MSSQL-Datenbank), dem eigentlichen Server und diversen Clients (beispielsweise einem Rich Client für den Desktop und einem Webinterface). Server und Rich Client basieren beide auf *OSGi* (und dessen Implementierung *Equinox*[4]) und teilen sich einen Großteil des Programmcodes (d.h. dass dieselben OSGi-Bundles in Client und Server verwendet werden). Die Webinterfaces basieren teilweise auf OSGi, aber auch auf *Spring*[5] und verwenden *Vaadin*[6] als Framework für die Oberflächen. Der Rich Client (welcher auf der *Eclipse Rich Client Plattform*, kurz RCP, basiert) wird jedoch als primäres Interface eingesetzt.

Rich Client und Server kommunizieren über ein proprietäres Protokoll (dem *Heiler-TCP*). Dieses verwendet Socket-Verbindungen und ermöglicht, Objekte zwischen Client und Server zu serialisieren, aber auch Nachrichten innerhalb des Servers zu versenden.

Der PIM Server läuft (abgesehen von OSGi) völlig selbstständig. Er verwendet keinen Servlet- oder Application Container, sondern implementiert diese Funktionalität selbst[7].

Alle Produkte werden seit mehreren Jahren nur noch auf Windows (sowohl als Client- als auch als Serverplattform) entwickelt. Die Pflege einer Linux-Variante wurde eingestellt, soll jedoch wieder aufgenommen werden.

Zur Auslieferung wird der Server in ein ZIP-Archiv gepackt, welches die Anwendung und ihre Abhängigkeiten (etwa ein JRE und Bibliotheken) beinhaltet. Dieses ZIP-Archiv wird beim Kunden entpackt und enthält wiederum eine Ordnerstruktur mit OSGi-Bundles und Ressourcen. Nach Einrichtung der nötigen Konfigurationen wird der Server über (Batch/Bash) Skripte und Wrapper-Applikationen[8] gestartet.

[4]Siehe Kapitel 2.1.
[5]Spring ist ein Framework für JEE-Anwendungen. Mehr unter http://spring.io.
[6]Vaadin ermöglicht das Erstellen von *Rich Internet Applications*, die mit Hilfe des *Google Web Toolkits* (GWT) von Java in HTML und JavaScript kompiliert werden. Mehr unter https://vaadin.com.
[7]Die Webinterfaces werden innerhalb des Servers in einem *Jetty* Container betrieben.
[8]Unter Windows wird der Server mit einem Launcher der Firma Tanukisoftware als Windows-Service gestartet.

Abbildung 1.2.: Web- und Desktopoberfläche des Informatica PIM Servers

2. Einführung in OSGi

2.1. Grundlagen von OSGi

Die *Open Services Gateway Initiative* (kurz *OSGi*) ist eine Spezifikation der *OSGi Alliance*[1] für ein modulares Komponentensystem für Java. Die Spezifikation beschreibt eine Softwareplattform oberhalb der JVM mit der Möglichkeit, Komponenten zur Laufzeit zu installieren[2] und deren Abhängigkeiten aufzulösen (OSGi Alliance, 2014a). Außerdem unterstützt OSGi eine serviceorientierte Architektur von Applikationen (siehe Kapitel 2.1.4).

Die OSGi-Laufzeitumgebung ist schematisch in Abbildung 2.1 dargestellt und wird im Folgenden erklärt. Für die Laufzeitumgebung existieren verschiedene Implementierungen wie etwa *Apache Felix*[3], *Knopflerfish*[4] und *Equinox*[5]. Alle Implementierungen unterstützen die OSGi-Spezifikation, bieten darüber hinaus aber proprietäre Features. Es wird insbesondere die Equinox-Implementierung von OSGi näher betrachtet, da diese die Grundlage für den Informatica PIM Server bildet.

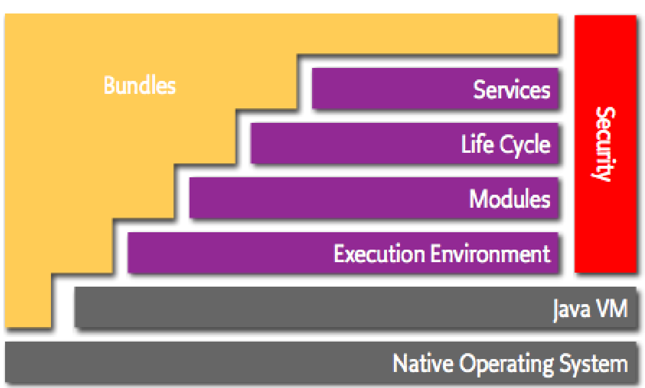

Abbildung 2.1.: Model der OSGi-Laufzeitumgebung (aus OSGi Alliance, 2014a)

[1]Die *OSGi Alliance* ist ein Industriekonsortium zur Spezifikation von OSGi. Mitglieder sind unter Anderem IBM, Oracle, Siemens und viele weitere (vgl. OSGi Alliance, 2014b). Mehr unter www.osgi.org.
[2]OSGi nennt diese Komponenten *Bundles*. Bundles können zur Laufzeit dynamisch installiert, deinstalliert, gestartet und gestoppt werden (siehe Kapitel 2.1.1).
[3]Das Projekt findet sich unter https://felix.apache.org.
[4]Mehr auf www.knopflerfish.org.
[5]*Equinox* ist die Laufzeitumgebung von Eclipse. Mehr unter www.eclipse.org/equinox.

2.1.1. Bundles

Bundles bilden die kleinste Einheit einer OSGi-Anwendung, d.h. dass sich jede OSGi-Anwendung aus einem Set von Bundles zusammensetzt.

Technisch gesehen sind OSGi-Bundles JAR-Dateien, wobei sie im Fall von Equinox auch als Ordner vorliegen können. Dies kann nützlich sein, wenn etwa Ressourcen innerhalb eines Bundles (z.B. Textdateien) zugänglich gemacht werden sollen. Jedoch ist dies nicht Teil des OSGi-Standards und Equinox-spezifisch. Diese Technik wird im Informatica PIM Server für ca. 20 Bundles genutzt.

Anders als JAR-Dateien müssen Bundles zwingend eine Manifest-Datei (`MANIFEST.MF`) enthalten. Diese Manifest-Datei folgt der Standardsyntax für JAR-Dateien[6], beinhaltet jedoch zusätzliche OSGi-spezifische Einträge (vgl. McAffer et al., 2010, Seite 18-21). Diese Einträge legen beispielsweise fest, welche Packages das Bundle zur Laufzeit exportiert und welche Abhängigkeiten es hat (etwa zu andere Bundles oder Packages, welche es importiert). Außerdem enthalten sie den Namen und die Version[7] des OSGi-Bundles.

In OSGi gilt als *Best Practice*, nur einzelne Packages eines Bundles zu exportieren bzw. zu importieren, keine kompletten Bundles. So kann ein Bundle leicht durch ein anderes ersetzt werden, solange es dieselben Packages exportiert. Direkte Abhängigkeiten zwischen Bundles brechen das Konzept der Modularität von OSGi (vgl. Bakker and Ertman, 2013, Seite 76-77).

Mit Hilfe der `MANIFEST.MF` löst das OSGi-Framework die Abhängigkeiten eines Bundles zur Laufzeit auf. Ein Bundle wird nur dann gestartet, wenn alle seine Abhängigkeiten erfüllt sind. So sollen Klassenpfadfehler (z.B. `ClassNotFoundException`) vermieden werden.

In Equinox werden Bundles auch als *Plugins* bezeichnet. Diese Plugins können neben der Manifest-Datei eine zusätzliche (Equinox-spezifische) `plugin.xml` enthalten. Diese Datei beschreibt sogenannte *Extension Points*, die das Plugin anbietet bzw. nutzt (vgl. McAffer et al., 2010, Seite 275-278). Extension Points sind vergleichbar mit OSGi-Services, werden aber hauptsächlich deklarativ (in der `plugin.xml`) und nicht programmatisch definiert (vgl. Bartlett [2007]). Sie werden intensiv im Informatica PIM Server genutzt (siehe Kapitel 2.1.4).

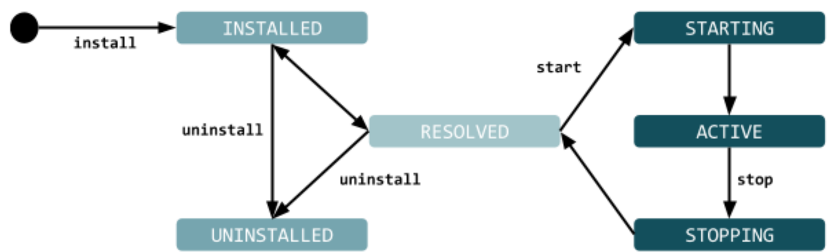

Abbildung 2.2.: Bundle Life Cycle (nach McAffer et al., 2010, Seite 23)

[6] Weitere Informationen hierzu finden sich unter goo.gl/pCO6vU.
[7] Die Versionierung von OSGi-Bundles wird im Informatica PIM Server nicht genutzt.

OSGi Bundles haben einen definierten *Lebenszyklus* (engl. *Life Cycle*). Dieser ist in Abbildung 2.2 dargestellt. Wird ein Bundle (zur Laufzeit) zum Framework hinzugefügt, so ist es `INSTALLED`. Sobald alle Abhängigkeiten eines Bundles erfüllt sind, gilt es als `RESOLVED` und kann gestartet werden, wodurch es in `STARTING` und `ACTIVE` übergeht. Wird ein Bundle gestoppt, geht es in den Zustand `STOPPING` und `RESOLVED` über. Bundles können auch vom Framework entfernt werden (`UNINSTALLED`). Dieser Life Cycle macht Bundles zur Laufzeit dynamisch. Bundles können dabei entweder manuell (z.B. über die OSGi Console), durch Konfigurationsdateien (z.B. die `config.ini` in Equinox) oder durch Abhängigkeiten zu anderen Bundles gestartet werden.

2.1.2. Features

Features sind ein Equinox-spezifischer Mechanismus, um Bundles (bzw. andere Features) zu gruppieren (vgl. McAffer et al., 2010, Seite 222). Sie werden durch eine `feature.xml` definiert, welche alle Bundles (respektive Features) auflistet, die das Feature enthält. Bundles können so zu logischen Komponenten zusammengefasst werden. Features dienen vor allem dem Build und Starten der Anwendung.

Features werden im Informatica PIM Server zwar genutzt, jedoch nur rudimentär. Die etwa 380 Bundles des Servers werden durch gerade einmal fünf Features gruppiert, wobei das größte Feature (eingebettete Features ausgenommen) alleine schon über 370 Bundles umfasst[8].

2.1.3. Products

Products gruppieren Bundles, Features, Ressourcen und Konfigurationen zu einem fertigen Eclipse-Produkt. Sie sind ebenfalls Equinox-spezifisch und dienen dazu, Applikationen, die auf Equinox und Eclipse basieren, zu bauen. Sie fassen alle Bestandteile einer Applikation in einer Konfigurationsdatei (`*.product`) zusammen, welche z.B. mithilfe von Eclipse als fertiges Produkt (inklusive Equinox) exportiert werden kann. Die `*.product`-Datei ist selbst nicht Teil des fertigen Produkts.

Products werden im Informatica PIM Server nicht genutzt. Der Build der Applikation basiert auf eigenen ANT-Skripten.

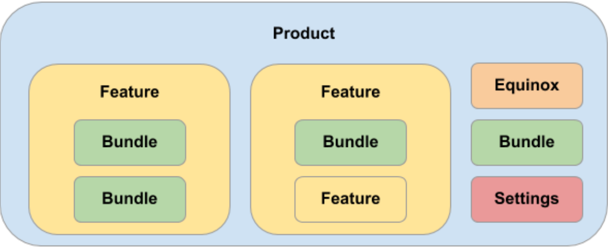

Abbildung 2.3.: Products, Features und Bundles

[8] Dieses Feature umfasst quasi den gesamten Server. Lediglich einige betriebssystemspezifische Bibliotheken finden sich in anderen Features.

2.1.4. OSGi Services und Extension Points

OSGi fördert eine serviceorientierte Architektur (kurz *SOA*) für Applikationen. Dies bedeutet, dass es Bundles möglich ist, Services zur Verfügung zustellen, die wiederum von anderen Bundles genutzt werden können (vgl. McAffer et al., 2010, Seite 67-69). OSGi sieht hierfür die sogenannten *OSGi Services* vor. Equinox bietet darüber hinaus den Mechanismus der *Extension Points*. Beide Technologien funktionieren jedoch ähnlich.

Sowohl OSGi Services als auch Extension Points geben einem Bundle die Möglichkeit, einen oder mehrere Services zu registrieren (und somit für andere Bundles bereitzustellen) bzw. bereitgestellte Services zu konsumieren. OSGi Services werden meist programmatisch in der *Service Registry* erfasst, Extension Points deklarativ in der `plugin.xml`[9]. Beide Mechanismen beschreiben das Interface der jeweiligen Services, beispielsweise durch Angabe der Input-Parameter für den Service.

Der wesentliche Unterschied zwischen OSGi Services bzw. Extension Points zu herkömmlicher SOA ist, dass erstere immer nur innerhalb einer JVM verfügbar sind. Sie müssen lediglich über eine zentrale Registry abgerufen werden, sind dann aber direkte Methodenaufrufe auf einem Objekt. Dies hat den Vorteil, dass OSGi Services sehr leichtgewichtig und performant sind (vgl. Bakker and Ertman, 2013, Seite 13-16). Jedoch gibt es mit dieser Technologie keine Möglichkeit, die Services auf verschiedene Maschinen zu verteilen (wie es mit SOA möglich ist).

OSGi Services bzw. Extension Points sollten direkten Abhängigkeiten zwischen Bundles vorgezogen werden, da diese das Konzept der Modularität brechen. Beide Mechanismen sind so konzipiert, dass sie es erlauben, Services dynamisch zu aktualisieren oder zu entfernen. Das OSGi-Framework löst dabei Services auf und verwaltet diese zur Laufzeit. Daher macht erst ihre Verwendung eine OSGi Applikation modular.

OSGi Services werden im Informatica PIM Server nicht genutzt, Extension Points werden hingegen verwendet. Jedoch dienen diese überwiegend als Erweiterungspunkte (z.B. zum Registrieren von Listenern), nicht dem Entkoppeln von Komponenten. Der PIM Server ist daher nicht modular im Sinne von OSGi (siehe Kapitel 2.2)[10].

2.2. Modularität in OSGi

OSGi soll die Komponenten einer Applikation trennen und diese zur Laufzeit modular machen. Mit OSGi ist es möglich, Code in Bundles aufzuteilen und diese Bundles lose über Services zu koppeln. Dadurch werden Bundles zur Laufzeit dynamisch. OSGi erzwingt diese Modularität jedoch nicht, sondern unterstützt sie durch die Bereitstellung von Servicemechanismen. Werden diese nicht genutzt, so ist die Applikation auch nicht modular[11].

[9] Extension Points können ebenfalls programmatisch registriert werden, OSGi Services je nach Framework auch deklarativ. Dies ist aber unüblich.

[10] Der PIM Server ist insofern modular, als dass er in logische Projekte unterteilt ist. Komponenten lassen sich so leicht anpassen und austauschen. Dies wird jedoch nicht zur Laufzeit (durch OSGi) unterstützt.

[11] OSGi verliert so seinen Hauptzweck und dient nur noch der Aufteilung von Code in (logische) Bundles.

2.3. OSGi und Modularität im Informatica PIM Server

Der Informatica PIM Server basiert auf der OSGi-Implementierung Equinox in der Version 3.8. Alle Komponenten des PIM Servers liegen als OSGi-Bundles vor, externe Bibliotheken sind als Bundles *repacked*[12]. Der Server besteht aus ca. 200 OSGi Bundles, zuzüglich externen Bibliotheken (ca. 90 Bundles), Tests, Features und Equinox selbst (ca. 60 Bundles).

Der PIM Server nutzt OSGi als Laufzeitumgebung, ist aber nicht modular (zur Laufzeit) konzipiert[13]. Zwar werden intensiv Extension Points verwendet[14], OSGi Services finden sich in der Anwendung jedoch nicht. Die einzelnen Bundles sind stark voneinander abhängig. Meist werden von einem Bundle mehrere andere Bundles *direkt* importiert. Dies bricht das Konzept der Modularität von OSGi (vgl. Bakker and Ertman, 2013, Seite 76-77) und führt zu Kopplungseffekten, wie in Kapitel VI beispielhaft beschrieben.

Der PIM Server nutzt außerdem ein internes Kommunikationsprotokoll (*Heiler-TCP*), mit dem Objekte zwischen Komponenten verschickt werden können[15]. Dieses Kommunikationsprotokoll schafft zusätzliche Abhängigkeiten zwischen Komponenten (Sender und Empfänger), die erst zur Laufzeit auftreten und weder durch den Java Compiler noch durch OSGi aufgelöst werden können. Dadurch kann die korrekte Auflösung von Bundles umgangen werden[16].

Weitere Abhängigkeiten des Projekts, etwa zwischen geteilten Bibliotheken von Client und Server, werden im Kontext dieser Arbeit nicht betrachtet. Die starke Abhängigkeit zwischen den Bundles wird jedoch in Kapitel 5.1.5 hinsichtlich ihrer Folgen für das Deployment wieder aufgegriffen.

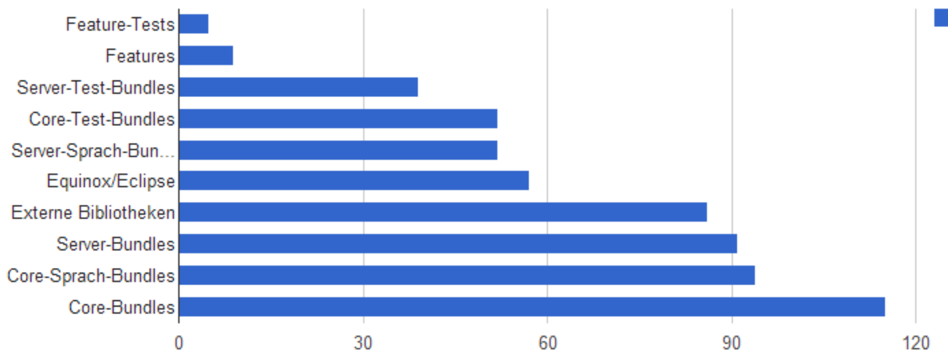

Abbildung 2.4.: Anzahl der OSGi-Bundles im Informatica PIM Server

[12] Externe Bibliotheken werden manuell in OSGi Bundles gepackt, exportiert und in SVN versioniert.

[13] Der Server ist jedoch sehr wohl modular zur Compile-Zeit. Er ist in logische Projekte unterteilt, die zusammengehörige Funktionalität kapseln.

[14] Es werden ca. 100 Extension Points im Server definiert.

[15] Heiler-TCP serialisiert Java-Objekte zwischen Sender und Empfänger. Dazu müssen auf beiden Seiten dieselben Klassen vorhanden sein.

[16] Dieses Problem kann generell überall dort auftreten, wo Klassen anhand von Strings dynamisch geladen werden.

Teil II.

Cloud Computing

3. Cloud Computing

Cloud Computing (oder kurz *die Cloud*) ist eines der meist gehörten Schlagwörter der Softwareindustrie in den vergangenen Jahren. Auf den Weg gebracht durch Firmen wie *Salesforce.com*, *Amazon* und *Google*[1] brachte es die Cloud von IT Abteilungen und Entwicklerbüros bis in die Schlagzeilen der Massenmedien. Doch was bedeutet *Cloud Computing*?

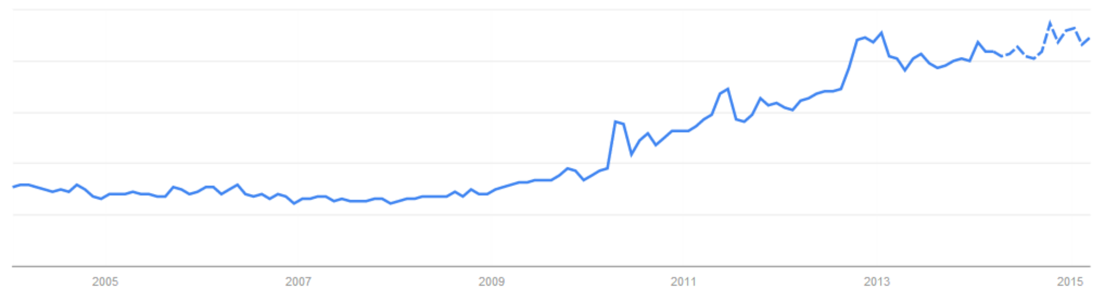

Abbildung 3.1.: Google Trends Chart für das Suchwort „Cloud“

Die folgenden Kapitel geben eine Antwort auf die Frage, was Cloud Computing ist und wie es sich definieren lässt. Jedoch ist Cloud Computing nicht nur ein sich stetig wandelnder Trend, es ist auch eine heterogene Technologie. Teilweise verbirgt sich hinter Cloud Computing sogar weniger eine Technologie, als viel mehr eine Sammlung von Prinzipien für das Design, das Deployment und die Verfügbarmachung von Anwendungen. Daher gibt es keine einheitliche Definition von Cloud Computing.

Nichtsdestotrotz kategorisiert das folgende Kapitel die Cloud in verschiedene Angebote und Servicelevels. Jede dieser Kategorien richtet sich an eine eigene Problemdomäne und Benutzergruppe. Obwohl diese Kategorien weder erschöpfend noch allgemeingültig sein können, werden sie im Folgenden herangezogen, um die Zielplattform für die Portierung der OSGi Applikation, die Gegenstand dieser Arbeit ist, zu beschreiben.

[1] Salesforce.com startete 1999 sein Angebot an *Software as a Service*. Amazon führte 2002 sein Cloud-Angebot *Amazon Web Services* ein. Im Jahr 2006 folgte das Angebot *EC2*, welches auch in dieser Arbeit verwendet wird. Mit der Einführung von *Google Docs* (heute *Google Drive*), stieg Google 2002 in die Cloud ein (vgl. Mohamed, 2009).

3.1. Servicemodelle

Cloud Computing unterscheidet zwischen drei *Servicemodellen*: *Infrastructure as a Service*, *Platform as a Service* und *Software as a Service* (vgl. Metzger et al., 2011, Seite 21-22). Diese Servicemodelle decken das gesamte Spektrum an Cloud-Angeboten ab und werden im Folgenden vorgestellt. Wie sich zeigen wird, sind diese Kategorien häufig zu weit gefasst um jeden Cloud-Service hinreichend zu beschreiben. Daher wird in Kapitel 3.1.4 ein weiteres Modell zur Kategorisierung von Cloud Angeboten nach Haan [2013] eingeführt. Dieses erlaubt eine genauere Einordnung von Cloud-Angeboten.

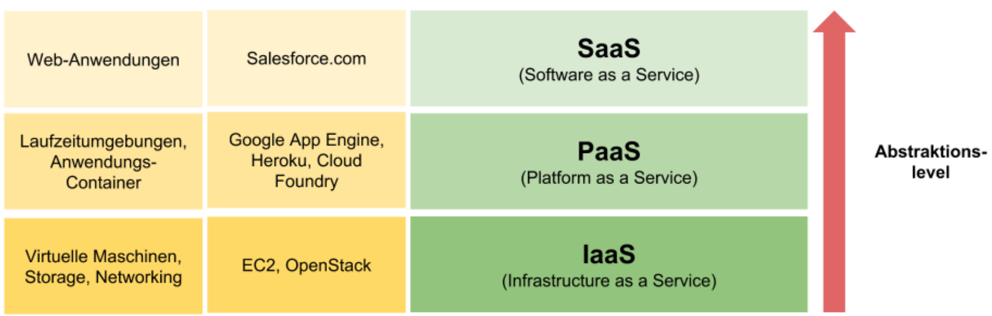

Abbildung 3.2.: Cloud-Servicemodelle

3.1.1. Infrastructure as a Service (IaaS)

Die grundlegendste Form von Cloud-Computing ist die sogenannte *Infrastructure as a Service* (IaaS). IaaS virtualisiert IT-Infrastrukturen wie CPU, RAM oder Plattenspeicher und stellt diese als Services über ein Netzwerk bereit. Anders als PaaS (oder SaaS) ist IaaS unabhängig von der darauf betriebenen Software.

Ähnlich wie virtuelle Maschinen abstrahiert IaaS die eigentliche Hardware. Doch anstatt lediglich virtuelle Maschinen über das Internet anzubieten (was Web- und Anwendungshoster schon seit einigen Jahren tun), bildet IaaS ein weiteres Abstraktionslevel oberhalb von virtuellen Maschinen (VMs). Während VMs auf einem einzelnen (bekannten) Host laufen, versteckt IaaS die Herkunft von Ressourcen (vgl. Chee and Franklin, 2010, Seite 55-56). CPU, RAM oder Plattenspeicher werden so zu Services, die von mehreren Maschinen oder VMs bereitgestellt werden können. Diese Abstraktion macht es möglich, virtuelle Maschinen *On-The-Fly* zu erstellen oder zu skalieren und sie in ein zusätzliches Set von Diensten und APIs (wie etwa Lastverteilung und Auto Scaling) einzubinden.

IaaS richtet sich an IT-Abteilungen und DevOps[2], die sich nicht mit der physikalischen Hardware und dem Management von virtuellen Maschinen auseinandersetzen wollen. Anbieter von IaaS sind zum Beispiel Amazon Web Services (mit EC2), Microsoft Azure und Google Compute Engine.

[2]DevOps steht für *Development and Operations* und beschreibt die Überlappung von Softwareentwicklung, IT-Administration und Qualitätssicherung.

3.1.2. Platform as a Service (PaaS)

Platform as a Service (PaaS) stellt nicht nur Infrastruktur als Service zur Verfügung, sondern eine komplette Applikationsumgebung. Solch eine Umgebung kann z.B. eine JVM oder ein (JEE) Application Container sein. Einige PaaS-Angebote setzen ein eigenes SDK voraus (etwa Google App Engine), andere setzen auf die Nutzung von Standardtechnologien (wie etwa *Maven* oder *Spring*). Anbieter wie Heroku[3] stellen nicht nur eine Applikationsumgebung, sondern übernehmen zusätzlich den gesamten Build- und Deployment-Prozess.

PaaS-Angebote bieten für gewöhnlich einen Arbeitsablauf, im Rahmen dessen entweder eine bereits gepackte Applikation (etwa ein *WAR-Archiv*) oder Quellcode (meist via *Git*) zum Anbieter hochgeladen werden. Dieser baut und deployt die Anwendung dann automatisch auf eine entsprechende Umgebung. PaaS-Angebote sind damit abhängig von der darauf betriebenen Software und der Funktionalität des Anbieters.

Zwar bildet PaaS die mittlere der drei Cloud-Schichten, jedoch bedeutet dies nicht, dass eine PaaS zwingend auf einer IaaS aufsetzen muss. Dies ist zwar häufig der Fall (etwa bei Amazons PaaS *Elastic Beanstalk*, welche auf dem IaaS-Angebot *EC2* aufsetzt), jedoch können PaaS-Implementierungen wie *Deis* oder *Flynn*[4] auch auf herkömmlichen Maschinen installiert werden.

Ein wesentlicher Unterschied zwischen IaaS und PaaS, sind die angebotenen Cloud-Services. Dies sind Softwaremodule wie beispielsweise Datenbanken, die vom Nutzer gestartet und in der eigenen Anwendung genutzt werden können. Der Anbieter übernimmt dabei deren Betrieb, was den Aufwand für den Anwendungsentwickler reduziert.

Anbieter von PaaS bieten darüber hinaus Werkzeuge zum Monitoring, Logging und Skalieren der Anwendung. Plattformen können so über Weboberflächen, Kommandozeilenwerkzeuge oder IDE-Plugins administriert werden. Insbesondere Programmierschnittstellen ermöglichen eine hohe Integration und Automatisierung von Plattformen in Entwicklungsprozessen.

Teilweise setzen PaaS Programmiermodelle voraus, z.B. zustandslose Server oder bestimmte Regeln für Threads. Dadurch sollen sich Anwendungen konform zur Plattform des Anbieters verhalten, etwa um automatisch repliziert werden zu können[5]. Dies kann bei bestehenden Applikationen zu Inkompatibilität und bei Neuentwicklungen zu Abhängigkeiten zur jeweiligen Plattform führen. Jedoch ist dies stark vom konkreten Anbieter abhängig.

PaaS richtet sich an Softwareentwickler, die sich nicht um Deployment und die Betreuung von Applikationsumgebungen kümmern möchten.

[3] *Heroku* ist einer der bekanntesten PaaS-Anbieter und wird in dieser Arbeit öfter als Beispiel genannt. Anwendungen können via Git an Heroku übertragen werden, welches diese automatisch baut und deployt. Das Angebot findet sich unter www.heroku.com.

[4] *Deis* und *Flynn* sind offene PaaS-Implementierungen für Docker (siehe Kapitel 9 und folgende). Mit ihnen kann Code aus einem Git Repository in Docker Container gepackt und deployt werden.

[5] Anbieter wie Amazon bieten eine automatische horizontale Skalierung (siehe Kapitel 6.2). Diese repliziert die Anwendung unter Last. Jedoch kann diese Art der Skalierung nur genutzt werden, wenn der Server zustandslos ist und es erlaubt, dynamisch neue Instanzen hochzufahren.

3.1.3. Software as a Service (SaaS)

Software as a Service (SaaS) bietet nicht nur Applikationsumgebungen, sondern komplette (Web-) Anwendungen für Endkunden über das Internet an (vgl. Metzger et al. [2011], Seite 21-22). Dies können (mehr oder weniger) einfache Webseiten sein, Webanwendungen (wie etwa Google Docs) oder ganze Software-Ökosysteme (wie *Salesforce.com*). Anbieter von SaaS verbergen die Installation, das Hosting und die Betreuung ihrer Anwendungen vor ihren Kunden. Diese können sich so auf die Nutzung der Applikation konzentrieren, anstatt auf deren Einrichtung.

SaaS richtet sich an Endkunden (sowohl privater Personen als auch Firmen), die Software zwar nutzen wollen, sie aber nicht selbst betreiben möchten (Installation, Hosting, Aktualisierung, etc.).

3.1.4. Model zur Beschreibung von Cloud-Services

Die zuvor vorgestellte (traditionelle) Einteilung von Cloud-Services in *IaaS*, *PaaS* und *SaaS* hat nach Haan [2013] zwei Einschränkungen:

- Es ist nicht möglich, zwischen dem Level (IaaS, PaaS oder SaaS) von unterschiedlichen Bestandteilen eines Angebots zu unterscheiden. Angebote wie Heroku bieten z.B. vorkonfigurierte Laufzeitumgebungen (was einer PaaS entspräche), sowie die Möglichkeit, direkt auf der darunterliegenden virtuellen Maschine zu arbeiten (via SSH-Zugriff, was einer IaaS entspräche). Heroku bietet außerdem Datenbanken als Service, was einem Angebot zwischen PaaS und SaaS entspräche.
- Es ist nicht möglich, zwischen den Domänen von Cloud-Services zu unterscheiden. Alle Services, egal ob Rechen-, Speicher- oder Kommunikationsservices, werden in die drei Kategorien IaaS, PaaS und SaaS unterteilt.

Eine genauere Einteilung von Cloud-Angeboten bietet das Klassifikationsmodell von Haan [2013]. Dieses ist in Abbildung 3.3 dargestellt. Es bietet eine differenziertere Einordnung von Cloud-Angeboten, wie sie in dieser Arbeit behandelt werden.

Das Model unterscheidet zwischen drei Arten von Cloud-Services:

- *Computation Services* wie virtuelle Maschinen und Applikations-Containern.
- *Communication Services* wie Netzwerke, Routing und Load Balancing.
- *Storage Services* wie Dateisysteme und Datenbanken.

Layer 6	SaaS	Applications			*End-users*
Layer 5	App Services	Compute App Services	Communicate App Services	Store App Services	*Citizen developers*
Layer 4	Model-Driven PaaS	bpmPaaS, Model-Driven aPaaS	Model-Driven iPaaS	baPaaS	*Business engineers*
Layer 3	PaaS	aPaaS	iPaaS	dbPaaS	*Professional developers*
Layer 2	Foundational PaaS	Application containers	Routing, messaging	Object storage	*DevOps*
Layer 1	Software Defined Datacenter	Virtual Machines	Software Defined Networking (SDN)	Software Defined Storage (SDS)	*Infrastructure engineers*
		Compute	Communicate	Store	

Abbildung 3.3.: Model zur Beschreibung von Cloud-Services (nach Haan, 2013)

Zusätzlich beinhaltet das Modell sechs Abstraktionslevels, beginnend bei virtuellen Maschinen auf Level eins, bis zu Applikationen (SaaS) auf Level sechs. Die eigentliche Hardware entspricht einem gedachten Level null unterhalb der virtuellen Maschinen. Das Modell verweist außerdem auf die Zielgruppe der jeweiligen Schicht (zu sehen auf der rechten Seite von Abbildung 3.3). Ein einzelnes Cloud-Angebot (wie etwa Heroku oder Amazon Web Services) kann dabei mehrere Schichten abdecken.

IaaS deckt Schicht eins und zwei ab, PaaS Schicht zwei und drei. IaaS- und PaaS-Angebote überlappen sich also teilweise, was die unklaren Grenzen zwischen diesen Definitionen aufzeigt. Eine IaaS (wie Amazons EC2) kann etwa mit Images betrieben werden, die eine Anwendungsumgebung (z.B. einen Apache Tomcat[6]) vorinstalliert haben. Ähnlich wie bei einer PaaS kann so eine Cloud-Instanz gestartet und ein Artefakt direkt (in Tomcat) deployt werden[7].

Schicht vier und fünf decken Angebote ab, die es ermöglichen, über (Web-) Oberflächen oder domänenspezifische Sprachen (kurz *DSLs*), Anwendungen oder Teile davon zu definieren. Hierzu gehören graphische Programmierwerkzeuge und modellgetriebene Plattformen. Schicht sechs bildet die fertige *Software as a Service*.

Im Zuge dieser Arbeit sind die ersten drei Schichten des Models von Interesse. Die verwendeten Technologien *Docker* und *Amazon Web Services* werden in Kapitel 9.2 bzw. in Kapitel 8.1 in das Model eingeordnet. Die Ergebnisse der Arbeit werden in Kapitel 11 hinsichtlich des Modells betrachtet.

[6] *Apache Tomcat* ist ein quelloffener Servlet Container der Apache Foundation. Mehr unter http://tomcat.apache.org.

[7] Tomcat erlaubt es sogar Anwendungen als WAR-Archive über eine Weboberfläche hochzuladen und zu deployen. PaaS-Angebote wie Amazons Elastic Beanstalk machen letztlich nichts anderes, bieten aber weitaus mehr Kontrolle (etwa über die VM oder den Application Server selbst).

3.2. Liefermodelle

Cloud-Angebote lassen sich in vier sogenannte *Liefermodelle* unterteilen. Diese Liefermodelle bestimmen, wie das Cloud-Angebot den Nutzer erreicht (vgl. Metzger et al., 2011, Seite 18-20). In dieser Arbeit wird vor allem die *Public Cloud* betrachtet.

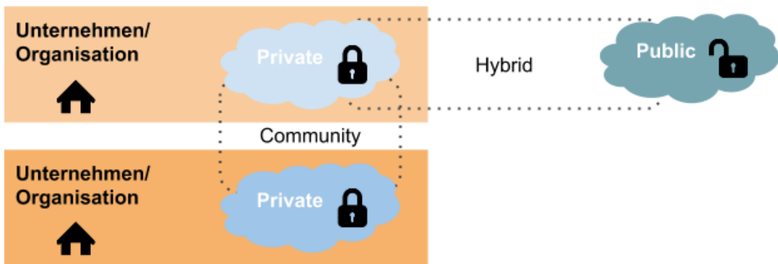

Abbildung 3.4.: Cloud-Liefermodelle

Private Clouds werden auf eigenen, hausinternen Servern der Nutzer (z.B. einer Firma oder Universität) betrieben. Ein Beispiel hierfür ist *OpenStack*[8], eine quelloffene Cloud-Software, die eine IaaS bereitstellt. Private Cloud-Plattformen sind besonders sicher, da Daten und Anwendungen im Unternehmen bleiben.

Public Clouds stehen öffentlich (im Internet) zur Verfügung. Sie werden gegen Gebühr (oder teilweise kostenlos[9]) von Anbietern offeriert und können von jedem genutzt werden. Sie stellen den Großteil der Cloud-Angebote dar. Das wohl bekannteste Beispiel ist *Amazon Web Services*. Öffentliche Cloud-Plattformen bieten einen hohen Komfort, da die Bereitstellung komplett vom Anbieter übernommen wird. Sie werden überwiegen im Rahmen dieser Arbeit betrachtet.

Community Clouds bezeichnen den Zusammenschluss mehrerer Private Clouds bzw. die Nutzung einer gemeinsamen Private Cloud in mehreren Unternehmen oder Organisationen. Ein Beispiel hierfür ist *Google Apps for Government*[10] welches die Nutzung von gemeinsamen Cloud-Diensten für Behörden bietet.

Hybrid Clouds setzen sich aus mehreren Cloud-Angeboten zusammen. Diese können privat oder öffentlich sein und bilden eine gemeinsame Plattform für Anwendungen. Ein typisches Beispiel ist eine Cloud-Anwendung, die zwar auf einer öffentlichen Plattform läuft, unternehmenskritische Daten aber in einer Private Cloud speichert. Hybride Cloud-Plattformen erlauben so einen Kompromiss aus der komfortablen Nutzung öffentlicher Cloud-Dienste und der privaten Speicherung von Daten.

[8] https://www.openstack.org
[9] Viele Cloud-Angebote sind im Rahmen gewisser Grenzen kostenlos (z.B. bis 100 MB Speicherplatz). Erst beim Überschreiten dieser Grenzen werden sie kostenpflichtig.
[10] *http://www.google.com/enterprise/apps/government*

3.3. Cloud-Komponenten

Die Cloud ist keine einheitliche Technologie mit klaren Grenzen. Vielmehr setzt sie sich aus verschiedenen Komponenten, Services, Design Patterns und Entwicklungsstrategien zusammen. Im Folgenden werden die wichtigsten Cloud-Komponenten benannt.

Virtuelle Maschinen bilden die Grundlage für Cloud Computing[11] und die IaaS. Dem Nutzer wird die Möglichkeit gegeben, Maschinen selbst (binnen weniger Minuten) zu erzeugen, zu starten und einzurichten. So können Anwendungen unter Last durch neue Instanzen skaliert werden oder deren Verfügbarkeit erhöht werden. Zur Administration stellen Anbieter (Web-) Oberflächen, IDE-Plugins und Programmierschnittstellen bereit.

Monitoring erlaubt es Nutzern (oder Anwendungen wie Load Balancern), den Status von Maschinen zu überwachen. Cloud-Anbieter stellen hierfür (Web-) Oberflächen und APIs zur Verfügung, um Metriken ihrer Plattform abzufragen. Services dieser Art können Systemmonitore wie Nagios[12] in der Cloud ersetzen.

Load Balancing macht es möglich, Anwendungen dynamisch zu skalieren. Je nach Last können VMs gestartet oder abgeschaltet werden (siehe die Begriffserklärung von *Elastizität* in Kapitel 6.3). Ähnlich wie *Monitoring* wird dieser Service von den meisten Cloud-Anbietern bereitgestellt. Ein Beispiel hierfür ist *Amazon CloudWatch*, ein Service um Amazon EC2 Maschinen[13] zu überwachen und bei Bedarf (automatisch) zu skalieren.

Cloud Services stellen den Schlüssel zu skalierbaren und schlanken Applikationen dar. Sie können Funktionalität aus der eigenen Anwendung auslagern und sie in die Verantwortung eines Cloud-Anbieters und dessen Infrastruktur geben. Cloud Services folgen den Designprinzipien serviceorientierter Architektur (vgl. Wik, 2011). Ein Beispiel für einen solchen Cloud Service ist Amazons *DynamoDB*[14]. DynamoDB ist eine NoSQL-Datenbank als Service in Amazons Cloud. Amazon übernimmt dabei nicht nur die Einrichtung der Datenbank, sondern auch deren Management. Die Datenbank wird bei Bedarf automatisch mit neuem Speicherplatz versehen, auf mehrere Maschinen verteilt und ausfallsicher gemacht.

Deployment ist ein zentraler Bestandteil vieler Cloud-Angebote. Insbesondere PaaS-Angebote versuchen sich durch ihr Deployment-Verfahren von der Konkurrenz abzuheben. Hiervon können vor allem Entwickler profitieren, die immer größere Teile im Entwicklungsprozess an Services externer Anbieter auslagern können.

[11] Mehr zu Abgrenzung von virtuellen Maschinen und Cloud Computing in Kapitel 3.4.2.

[12] *Nagios* ist ein verbreitetes Werkzeug zur Überwachung von IT-Infrastrukturen. Mehr unter http://www.nagios.org.

[13] Amazons Plattform für virtuelle Maschinen heißt *EC2*.

[14] Informationen zu DynamoDB finden sich unter http://aws.amazon.com/dynamodb.

3.4. Abgrenzung und Ausschluss von Themen

3.4.1. Grid Computing

Grid Computing bezeichnet den Zusammenschluss mehrerer Maschinen und deren Ressourcen. Diese Ressourcen werden innerhalb des *Grids* geteilt und dezentralisiert. Dadurch soll deren Nutzung verbessert und die Ressourcen akkumuliert werden (vgl. Foster, 2002). Grid Computing wird hauptsächlich im wissenschaftlichen Bereich für aufwendige Berechnungen genutzt (vgl. Metzger et al., 2011, Seite 24). Diese werden auf mehrere Rechner verteilt, was die Durchführung beschleunigt. Anders als Cloud Computing dient Grid Computing also nicht dem Betrieb/Hosting von Anwendungen, sondern dem verteilten Rechnen. Es wird daher in dieser Arbeit nicht betrachtet.

3.4.2. Virtualisierung

Virtualisierung ist die Grundlage für Cloud Computing und bietet seinerseits (virtualisierte) Ressourcen als Services. Cloud-Plattformen stellen daher immer eine virtualisierte Umgebung für Applikationen bereit (vgl. Chee and Franklin, 2010, Seite 46-47). Jedoch ist eine virtualisierte Umgebung nicht gleichbedeutend mit einer Cloud. Folgende Kriterien (aus Mell and Grance, 2011) differenzieren eine virtualisierte Umgebung von einer Cloud-Plattform (nach Martin, 2013):

- **On-Demand Self-Service**: Ein Nutzer kann Computerressourcen (z.B. Speicher oder CPU) einer Cloud ohne *zwischenmenschliche* Interaktion mit dem Service Provider steuern. Virtualisierte Umgebungen bieten zwar Zugriff auf eine VM, jedoch nicht auf deren Hypervisor.

- **Broad Network Access**: Cloud-Plattformen stehen über ein Netzwerk zur Verfügung. Dies ist bei virtualisierten Umgebungen nicht zwingend der Fall.

- **Resource Pooling**: Während Virtualisierung i.d.R. auf einem *einzelnem* Host aufsetzt, bestehen Cloud-Plattformen aus einem Pool von Ressourcen. Der Nutzer hat keine Kenntnis über die Herkunft der Ressourcen des Service.

- **Rapid Elasticity**: Nutzer können Cloud-Services (bzw. Cloud-Ressourcen) frei und scheinbar unlimitiert skalieren[15]. Virtualisierte Umgebungen sind meist fest konfiguriert und statisch.

- **Measured Service**: Cloud-Ressourcen können überwacht und kontrolliert werden. Häufig werden sie auf Basis geeigneter Metriken nach Nutzung abgerechnet.

Virtualisierung wird daher nicht explizit in dieser Arbeit thematisiert, sondern lediglich in seiner Eigenschaft als Grundlage für Cloud Computing betrachtet.

[15]Bei der Skalierbarkeit von Cloud-Services spricht man von Elastizität. Dies meint nicht nur das hochskalieren unter Last, sondern auch das runter-skalieren bei wenig Last (siehe Kapitel 6.3).

3.4.3. SaaS

SaaS ist zwar Teil der Cloud-Landschaft und viele Anwendungen, die in der Cloud betrieben werden, sind *SaaS-Applikationen*, jedoch ist dies nicht zwingend. Ein SaaS muss nicht in der Cloud laufen und eine Anwendung, die in der Cloud läuft muss kein SaaS sein. Die Portierung einer Anwendung in die Cloud kann zwar Grundlage dafür sein, die Anwendung als SaaS anzubieten, ist aber zunächst davon unabhängig. Im Zuge der Arbeit wird SaaS daher nicht betrachtet.

3.4.4. Mandantenfähigkeit

Mandantenfähigkeit meint die Eigenschaft einer Anwendung, mehrere Gruppen von Nutzern (sogenannte *Mandanten*) bedienen zu können, deren Daten aber strikt getrennt zu halten (Metzger et al., 2011, Seite 16-17). Ein Beispiel hierfür ist *GitHub*, in dem Anwender eigene Organisationen anlegen können, die wiederum über eine individuelle Mitgliederverwaltung verfügen. So können Firmen (als Mandanten) ihre eigen Struktur in GitHub abbilden.

Mandantenfähigkeit ist in erster Linie eine Eigenschaft von SaaS-Anwendungen und wird als solche nicht in dieser Arbeit betrachtet.

3.4.5. Outsourcing

Outsourcing ist zum Teil eng mit Cloud Computing verbunden. Insbesondere SaaS kann dazu dienen, Anwendungen und deren Betrieb auszulagern und durch einen externen Anbieter zu beziehen. Dies sind jedoch vorwiegend betriebswirtschaftliche Betrachtungen und werden im Kontext dieser Arbeit nicht untersucht.

3.4.6. Wirtschaftlichkeit

Wie in allen Unternehmensbereichen gilt es auch beim Cloud-Computing wirtschaftliche Aspekte zu beachten. Cloud-Angebote sind in der Regel kostenpflichtig und müssen finanziell mit anderen Angeboten verglichen werden. Die Anbieter von Cloud-Diensten stehen in zunehmender Konkurrenz, was eine Differenzierung des Marktes zur Folge hat. Ähnlich wie Outsourcing ist die Wirtschaftlichkeit aber kein Bestandteil dieser Arbeit. Außerdem hängt sie maßgeblich vom Anbieter und den in Anspruch genommenen Diensten ab. Daher kann sie nur in individuellen Fällen konkret evaluiert werden.

3.4.7. Sicherheit und Datenschutz

Sicherheit und Datenschutz spielen im Cloud-Umfeld eine wichtige Rolle. Je nach Unternehmen, Anwendung und Land ist es von großer Bedeutung, wo Daten gespeichert werden und wie mit ihnen verfahren wird. Dies kann auch bei Produkt- und Kundendaten, wie im Falle des Informatica PIM Servers, der Fall sein. Jedoch steht nicht die sicherheits- und datenschutzrechtliche Perspektive, sondern die technische Umsetzung des Cloud-Deployments im Zentrum dieser Arbeit.

Teil III.

Portierung in die Cloud

4. Motivation

Cloud-Computing ist derzeit ein populäres Thema in der IT und Software-Entwicklung (wie bereits eingangs in Kapitel 3 erwähnt). Von der Cloud verspricht man sich Lösungen für verschiedenste Probleme: Das Auslagern der IT-Infrastruktur soll Kosten senken, das Starten von Cloud-Instanzen *on demand* soll die Ausfallsicherheit erhöhen, Cloud-Services sollen Anwendungen skalieren und der administrative Komfort von Cloud-Plattformen soll Entwicklern und Kunden in ihrer täglichen Arbeit helfen.

Von der Portierung des Informatica PIM Servers in die Cloud wird sich vor allem letzteres versprochen. Das Aufsetzen von Servern (etwa für Entwickler- oder Demo-Systeme) ist ein zeitraubender Prozess. Virtuelle Maschinen müssen hausintern gepflegt und betreut werden, was regelmäßig zu Arbeitsaufwänden führt. Außerdem ist das Aufsetzen von kompletten Anwendungsstacks (z.B. PIM Server mit Web Oberfläche, Supplier Portal und weiteren Komponenten) teilweise auch für kleine Aufgaben nötig. Dabei übersteigt der Aufwand zum Einrichten der Umgebung bisweilen den eigentlichen Entwicklungsaufwand.

Die Firma Informatica sucht daher seit längerem eine neue Strategie zum Verteilen ihrer Anwendung. In einer vorangegangen Masterthesis wurde bereits eine „Installations- und Aktualisierungsstrategie von modularen [...] Enterprise Softwaresystemen“ (Titel der Arbeit von Nölke, 2013) untersucht. Dabei wurde *Eclipse p2*[1] als Deployment-Werkzeug evaluiert (vgl. Kapitel 5.1.5). Eine Werksstudentenstelle zur Implementierung eines *Installers* war ebenfalls ausgeschrieben. Die Arbeit und der *Installer* haben jedoch einen Fokus auf *inkrementellen Aktualisierungen* (auch in der Datenbank) und keinen Bezug zur Cloud.

Im Zuge dieser Arbeit soll daher eine adäquate Deployment-Strategie für die Cloud gefunden werden. Ziel ist eine *Platform as a Service*, die Entwicklern und Kunden das Einrichten von PIM-Instanzen in der Cloud erleichtert. Dabei sollen auch erste Erfahrungen mit der Software in der Cloud gesammelt werden, um etwaige Probleme in künftigen Versionen zu adressieren. Außerdem sollen Cloud-Services evaluiert werden, die im PIM Server genutzt werden können.

Die horizontale Skalierung des Servers wird derzeit von einem Entwicklerteam von Informatica unter dem Projekttitel *Multi-Server* vorangetrieben. Dabei soll ein *Master-Slave-Cluster* mit mehreren Serverinstanzen ermöglicht werden. Dieses zukünftige Feature macht den Einsatz in der Cloud ebenfalls attraktiv.

[1] Eclipse p2 ist ein Framework des Equinox Projekts zum Verteilen und Aktualisieren von OSGi-Applikationen. Es kommt etwa im Plugin-Mechanismus von Eclipse zum Einsatz.

5. Deployment in die Cloud

Der erste Schritt, um eine Applikation in die Cloud zu portieren, ist deren Deployment. Obwohl dieser Schritt offensichtlich ist, so ist er doch kritisch. Ohne ein adäquates Vorgehen für das Packen und Verteilen einer Applikation kann diese nicht in die Cloud portiert werden. Die Art und Weise, wie eine Anwendung auf ihrem Zielsystem eingerichtet wird, hat Einfluss auf Entwicklungsprozesse wie beispielsweise das Testen, Bauen oder Aktualisieren der Anwendungen.

Die Deployment-Strategie gewinnt im Besonderen im Umfeld von Cloud Computing an Bedeutung, da hier Applikationen (automatisch) auf *fremde* Systeme verteilt werden. Viele dieser Systeme sind aus Sicht der Entwickler Blackboxen[1], andere Systeme gewähren uneingeschränkte Kontrolle über ein Betriebssystem, wieder andere stellen sich als „Quasi-Blackboxen“ heraus, die in Problemsituationen (eingeschränkten) Zugriff auf ein Betriebssystem gewähren können[2].

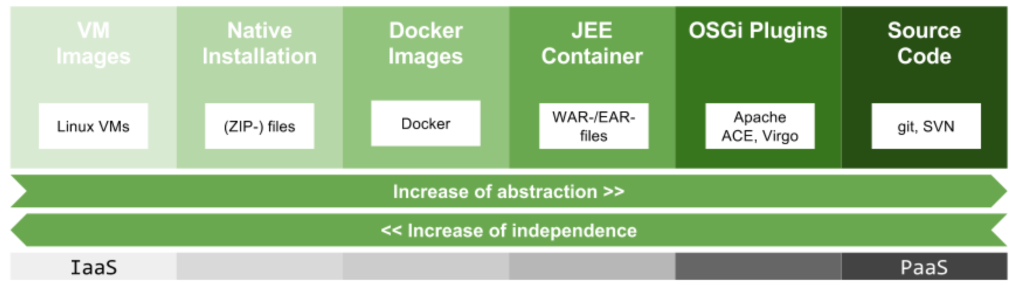

Abbildung 5.1.: Deployment-Strategien für die Cloud

Im Folgenden werden Deployment-Methoden für die Cloud im Hinblick auf Java und OSGi verglichen. Zuerst wird die systemnahste Methode (das Einrichten von VM-Images) betrachtet, zuletzt die abstrakteste Methode (das Hochladen von Quellcode zu einem Cloud-Anbieter). Mit steigendem Abstraktionslevel fällt jedoch der Grad der Unabhängigkeit und Portabilität der Methode. Während ersteres Vorgehen nicht an einen speziellen Anbieter (oder die Cloud im Allgemeinen) gebunden ist, hängt die letztere Methode maßgeblich vom jeweiligen Anbieter ab. Diese Abhängigkeit kann zu Lock-in-Effekten führen[3], befreit den Entwickler jedoch von Aufgaben wie dem Installieren von Bibliotheken, dem Aufsetzen von Tools oder dem Konfigurieren eines Betriebssystems (was bei der Einrichtung von VM-Images nötig wäre).

[1] Eine solche Blackbox ist etwa Google App Engine, die das eigentlich Betriebssystem verschleiert.
[2] Plattformen wie Heroku können Standardapplikation direkt verarbeiten und müssen nur in seltenen Fällen auf Betriebssystemebene angepasst werden.
[3] Lock-in-Effekte bezeichnen die Abhängigkeit zu einem Anbieter oder Produkt.

Dies macht die Wahl der Deployment-Strategie zu einer Abwägung verschiedener Faktoren. Während ein konkretes Vorgehen zu einer bestimmten Applikation passen kann, kann es unzureichend für eine andere sein. Alle Deployment-Strategien, die im Folgenden vorgestellt werden, werden anhand der in Tabelle 5.1 dargestellten Kriterien bewertet. Eine niedrige Wertung (gekennzeichnet durch einen Asterisk *) bedeutet, dass das jeweilige Kriterium nicht auf die Methode zutrifft. Eine hohe Wertung (gekennzeichnet durch drei Asteriske ***) bedeutet, dass das jeweilige Kriterium voll auf eine Methode zutrifft.

Kriterium	Beschreibung
Granularität	Eine Methode oder Plattform gilt als granulär, wenn sie es erlaubt, einzelne Teile einer Anwendung (z.B. OSGi-Bundles) zur Laufzeit zu deployen und zu aktualisieren.
Unabhängigkeit	Eine Plattform oder Methode ist unabhängig, wenn sie nicht an einen konkreten Anbieter oder eine Implementierung gebunden ist.
Kapselung	Eine Methode wird als gekapselt betrachtet, wenn für das Deployment keine detaillierten Kenntnisse über die Anwendung oder Plattform notwendig sind.
Komplexität	Ein Vorgehen wird als komplex betrachtet, wenn es sehr aufwändig in seiner Umsetzung ist, etwa durch das Einrichten von Servern oder Schreiben von Skripten.
Leistungsfähigkeit	Eine Plattform/Methode ist leistungsfähig, wenn mit ihr heterogene Anwendungen deployt werden können und sie nicht an eine spezielle Technologie (etwa Java oder OSGi) gebunden ist.
Level	Eine Plattform/Methode wird einem oder mehreren der drei Cloud-Level (IaaS, PaaS oder SaaS) zugeordnet.

Tabelle 5.1.: Kriterien zur Bewertung von Deployment-Strategien

5.1. Deployment Strategien

Im Folgenden werden die Deployment-Methoden *VM Images*, *manuelle Installation*, *Docker*, *JEE Archive*, *OSGi Bundles* und *Source Code* vorgestellt. Eine detaillierte Erklärung zu den Bewertungen der jeweiligen Methoden findet sich in Anhang VI.

5.1.1. VM Images

Granularität	Unabhängigkeit	Kapselung	Komplexität	Lstg.-fähigkeit	Level
*	***	*	**	***	IaaS

VM Images sind Dateien, die ein Abbild (engl. *Image*) eines Betriebssystems (meist Linux) und dessen Zustand enthalten. Als Zustand sind dabei alle installierten Programme, Dateien und Konfigurationen des Systems zu verstehen. Images können auf einer physischen oder einer virtuellen Maschine gestartet werden. Sie stellen die systemnahste, aber auch mächtigste Deployment-Strategie für (Cloud-) Applikationen dar.

Da VM Images das Einrichten eines kompletten Betriebssystems (manuell oder mittels Werkzeugen) notwendig machen, bedeutet ihre Erstellung einen administrativen Aufwand. Jedoch können sie genutzt werden, um praktisch jede Applikation zu packen und zu verteilen. Mit ihnen ist es außerdem nicht nur möglich die Applikation und ihre Abhängigkeiten, sondern auch Middleware wie etwa Server oder Caches in eine portable Datei zu packen. Ein Image enthält neben der eigentlichen Anwendung also auch deren komplette Umgebung.

Images bieten den Vorteil, portabel zu sein. Sie können lokal erstellt und getestet werden, um dann auf einen Cloud Anbieter deployt zu werden. In der Regel kann ein Image bei verschiedenen Anbieter installiert werden, was einen Lock-in verhindert. Images können auch als Basis für weitere Deployment-Schritte dienen, indem sie etwa eine anpassbare Basisinstallation oder einen („*leeren*") Application Container enthalten. Dieses Vorgehen wurde in dieser Arbeit genutzt, um das Deployment von Docker Images zu beschleunigen (siehe Kapitel 5.1.3 und 10).

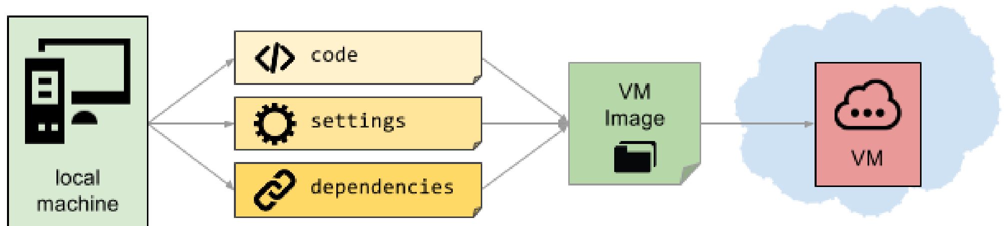

Abbildung 5.2.: Deployment mittels VM Images

5.1.2. Manuelle Installation

Granularität	Unabhängigkeit	Kapselung	Komplexität	Lstg.-fähigkeit	Level
*	***	*	***	***	IaaS

Die *manuelle Installation* meint die herkömmliche Installation einer Anwendung. Viele Anwendungen lassen sich mittels Installern, ZIP-Files oder anderer Mechanismen manuell auf einer Maschine einrichten. Dieser Installationsprozess ist abhängig von der jeweiligen Anwendung. Im Fall des Informatica PIM Servers muss etwa ein ZIP-Archiv entpackt, Konfigurationen in einem Properties-File angepasst und eine Skript zum Start des Server aufgerufen werden (siehe auch Kapitel 1). Ein JRE wird mitgeliefert, wodurch die Installation von Java entfällt (außer es soll eine andere JVM verwendet werden).

Diese „Installation von Hand" kann auch im Cloud-Umfeld genutzt werden. Anbieter wie Amazon bieten vollen Zugriff auf eine virtuelle Maschine, etwa über *SSH* (für Linux) oder *Remote Desktop* (für Windows). So können Anwendungen auf gewohnte Weise manuell eingerichtet werden.

Die Methode hat den Nachteil, dass Vorteile von Cloud-Plattformen praktisch nicht genutzt werden. Server müssen einzeln und von Hand aufgesetzt werden und es findet nur eine eingeschränkte Automatisierung statt. Zudem kann die Installation von einer Anwendung nicht auf eine andere übertragen werden, d.h. dass die Installationsschritte jeder Anwendung bekannt sein müssen.

Mit diesem Vorgehen lässt sich jedoch einfach testen, ob eine Anwendung mit dem System eines Anbieters kompatibel ist. Darüber hinaus hat es keinen Mehrwert.

Je nach Anbieter könne eingerichtete Maschinen auch als Vorlagen (Images) gespeichert werden und für neue Instanzen dienen. Ein Consulting-Team von Informatica aus Australien hat dieses Vorgehen bereits für Windows-Instanzen in der Amazon-Cloud evaluiert. Dabei wurden Instanzen von Hand eingerichtet und als Vorlage gespeichert. Wurde eine neue Instanz benötigt, wurde eine Vorlage gestartet und wiederum per Hand betriebsbereit gemacht (z.B. konfiguriert).

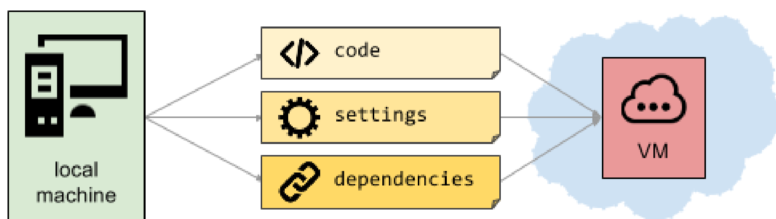

Abbildung 5.3.: Manuelle Installation

5.1.3. Docker

Granularität	Unabhängigkeit	Kapselung	Komplexität	Lstg.-fähigkeit	Level
**	***	**	**	***	IaaS/PaaS

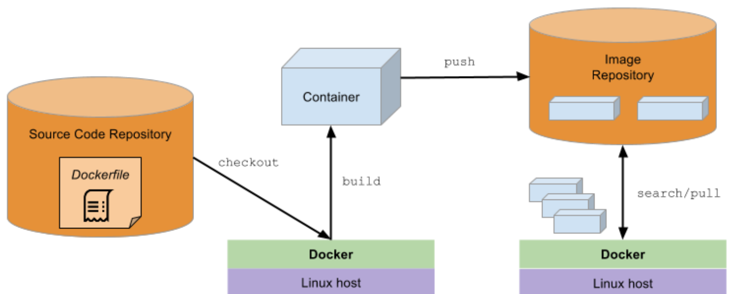

Abbildung 5.4.: Deployment mittels Docker (nach Docker Inc., 2014a, Abbildung 8)

Docker[4] ist ein Application-Container zur isolierten Ausführung von Anwendungen auf Linux. Mit Docker können *Images* von Anwendungen erstellt und verteilt werden. Diese lassen sich wiederum als *Container* starten. Docker wird als quelloffenes Projekt von *Docker Inc.*[5] entwickelt und ist aktuell in der Version 0.11 verfügbar (Stand Juni 2014).

Docker basiert auf *Linux-Containern*[6] (kurz *LXC*) und dem Dateisystem *AUFS*. LXC ist eine „*vom Betriebssystem bereitgestellte virtuelle Umgebung zur isolierten Ausführung von Prozessen*" (vgl. Roden, 2014). Es ist in verschiedenen Linux-Distributionen enthalten (z.B. in Ubuntu seit Version 13.10, siehe Ubuntu, 2013), kann aber auch nachträglich installiert werden[7]. AUFS (für *Advanced Multi Layered Unification Filesystem*) erlaubt das Schichten mehrere Dateisysteme. Ein *read-only* Dateisystem kann von einem *read-write* Dateisystem überdeckt werden, wobei beide dem Nutzer als *ein* Dateisystem präsentiert werden. Dadurch können Images Dateien teilen, ohne sich gegenseitig zu beeinflussen.

Im Gegensatz zu virtuellen Maschinen ist Docker wesentlich leichtgewichtiger, da sich mehrere Container zur Laufzeit einen gemeinsamen Linux-Kernel und ggf. Dateien teilen und nur deren Unterschiede von Docker (bzw. LXC und AUFS) verwaltet werden (siehe Abbildung 9.1). Die einzelnen Container bleiben dabei *stets isoliert*. Docker ist dadurch ressourcenschonender als virtuelle Maschinen[8] (vgl. Roden, 2014). Ein weiterer Vorteil von Docker ist die geringe Größe des Images: da die Container das Betriebssystem des Hosts nutzen, enthält das Image lediglich Bestandteile der Applikation[9].

[4]Mehr zu Docker in Kapitel 9 sowie unter www.docker.io.
[5]Ehemals *dotCloud Inc.*, siehe https://www.docker.com.
[6]Das Projekt findet sich unter https://linuxcontainers.org.
[7]Voraussetzung für die Installation ist ein kompatibler Linux-Kernel ab Version 2.6.
[8]Konkret bedeutet dies, dass Container etwa in wenigen Sekunden (oder schneller) starten und dutzende Container gleichzeitig auf einem Host betrieben werden können.
[9]Ein Image mit dem Informatica PIM Server und einer Java-Installation ist ca. 860 MB groß, wobei der Server etwa 530 MB umfasst. Ein vergleichbares VM-Image ist dagegen ca. 3360 MB groß.

Docker selbst ist im Wesentlichen ein Werkzeug zum Umgang mit LXC. Durch Docker können Container z.B. anhand von Konfigurationsdateien[10] erstellt und dann als Image[11] gespeichert werden. Es ist auch möglich, einen laufenden Container manuell einzurichten und das Resultat in Form eines Images zu *committen*[12]. Darüber hinaus bietet Docker die Möglichkeit, Images in einem (entfernten) Repository zu verwalten oder als TAR-Archive zu exportieren.

5.1.4. JEE Archive

Granularität	Unabhängigkeit	Kapselung	Komplexität	Lstg.-fähigkeit	Level
*	**	**	**	**	IaaS/PaaS

JEE Container bzw. JEE Archiv-Dateien (wie etwa WAR- oder EAR-Dateien) sind ein Java-spezifischer Weg, um (JEE) Applikationen zu packen. Solche Applikationen sind meist Web-Services, die dafür ausgelegt sind, innerhalb eines JEE Containers ausgeführt zu werden. Diese Container bieten der Applikation eine Laufzeitumgebung mit APIs, z.B. für Transaktionen, Dependency Injection, Persistenz oder Security.

Es ist jedoch möglich, beinahe jede Java-Anwendung in eine solche Archiv-Datei zu packen, solange sie mit dem Container verbunden werden kann. Im Falle einer OSGi-Anwendung ist dies über die *Equinox Servlet Bridge* möglich[13]. Auch wenn es wenig Vorteile bringt, eine Applikation in eine JEE-Archiv zu packen, die die Container APIs nicht nutzt, so kann dies dennoch ein praktikabler Weg für deren Deployment darstellen.

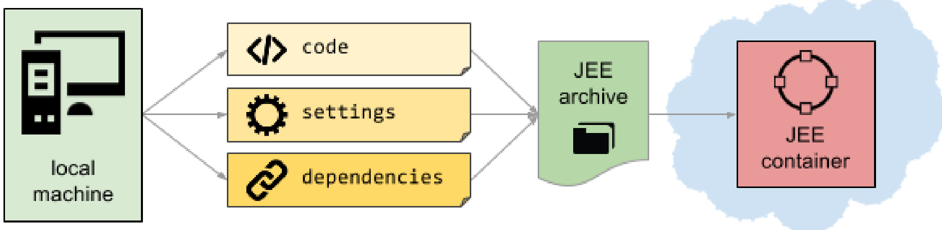

Abbildung 5.5.: Deployment mittels JEE-Archiven

Eine JEE-Datei ist ein Archiv, das vergleichbar mit einer ZIP-Datei, einer vorgegebenen Dateistruktur gehorcht (siehe Abbildung 5.1). Innerhalb dieser Dateistruktur befinden sich alle Klassen, Ressourcen (z.B. Bilder) und Abhängigkeiten einer Applikation

[10]Man bezeichnet diese Konfigurationsdateien als *Dockerfiles*. Sie geben z.B. an welche Dateistruktur oder Programme der Container enthalten soll.

[11]Ein Image ist die persistente Abbildung eines Containers, der wiederum eine laufende Instanz eines Images ist.

[12]Über den Befehl `docker run -t -i <image-id> /bin/bash` gelangt man in die Shell eines laufenden Containers. In diesem können nun Änderungen vorgenommen und über `docker commit <conainter-id> <image-name>` persistiert werden.

[13]http://mvnrepository.com/artifact/org.eclipse.equinox/servletbridge

sowie deren Konfiguration (z.B. `web.xml`). Daher ist ein solches Archiv in sich geschlossen und kann direkt in einen JEE Container deployt werden.

Algorithmus 5.1 Struktur eines WAR-Archivs

```
/static_content.html
/images/image_1.png
/WEB-INF/web.xml
/WEB-INF/classes/com/company/project/Servlet.class
/WEB-INF/lib/library_1.jar
/WEB-INF/lib/library_2.jar
/META-INF/MANIFEST.MF
```

JEE-Archive können als Grundlage für PaaS-Angebote wie etwa Amazons Elastic Beanstalk dienen. Hier ist es möglich, ein gepacktes Archiv hochzuladen und dieses mit Hilfe einer Weboberfläche in wenigen Schritten in einer VM zu starten.

5.1.5. OSGi-Bundles

Granularität	Unabhängigkeit	Kapselung	Komplexität	Lstg.-fähigkeit	Level
**	**	*	***	**	IaaS/PaaS

OSGi teilt Komponenten einer Software in sogenannte *Bundles* auf. Diese Bundles können zur Laufzeit in einem OSGi-Framework installiert und deinstalliert werden. Dies macht es möglich einzelne Bundles in ein laufendes OSGi-Framework zu deployen (wenn die Abhängigkeiten der Bundles erfüllt sind).

Es existieren zahlreiche Werkzeuge, welche sich diese Eigenschaft von OSGi zunutze machen und ein Bundle-basiertes Deplyoment von OSGi Applikationen unterstützen. Beispiele für solche Werkzeuge sind *Apache ACE*[14], *Equinox p2*[15], *Virgo*[16] oder *Apache Karaf*[17]. All diese Werkzeuge adressieren verschiedene Grundprobleme, bieten aber ein vergleichbares Vorgehen für das Deployment von Anwendungen. Der Fokus von Apache ACE liegt beispielsweise auf dem Deployment von Bundles aus einem entfernten Repository über ein Webinterface. Virgo hingegen ist in erster Linie ein OSGi-basierter Anwendungsserver, der jedoch auch Bundles aus einem Repository laden kann.

Abbildung 5.6 zeigt ein typisches Deployment-Szenario für eine OSGi-Anwendung. Ein Build-Server füllt ein Repository mit fertig gepackten OSGi-Bundles. Auf einem (Cloud-) Server läuft ein OSGi-Framework mit einer Managementkomponente, die ein Web- oder Kommandozeileninterface bietet. Ein Entwickler kann sich von einer lokalen Maschine mit diesem Interface verbinden, um Bundles aus dem (entfernten) Repository zu installieren. Die Managementkomponente innerhalb des OSGi Frameworks lädt diese Bundles aus dem Repository und installiert sie[18].

[14] https://ace.apache.org
[15] www.eclipse.org/equinox/p2
[16] www.eclipse.org/virgo
[17] https://karaf.apache.org
[18] Dieses Vorgehen wird in Kapitel 5.3 noch einmal detailliert aufgegriffen.

Bislang gibt es keine PaaS-Angebote für OSGi, d.h. kein Anbieter bietet ein OSGi-Framework (ggf. mit einem der zuvor erwähnten Deployment-Werkzeuge) in der Cloud. Dies macht OSGi als Deployment-Grundlage unattraktiv, da Repository, Framework und Managementkomponenten von Hand aufgesetzt werden müssen, bevor die eigentlichen Bundles verteilt werden können.

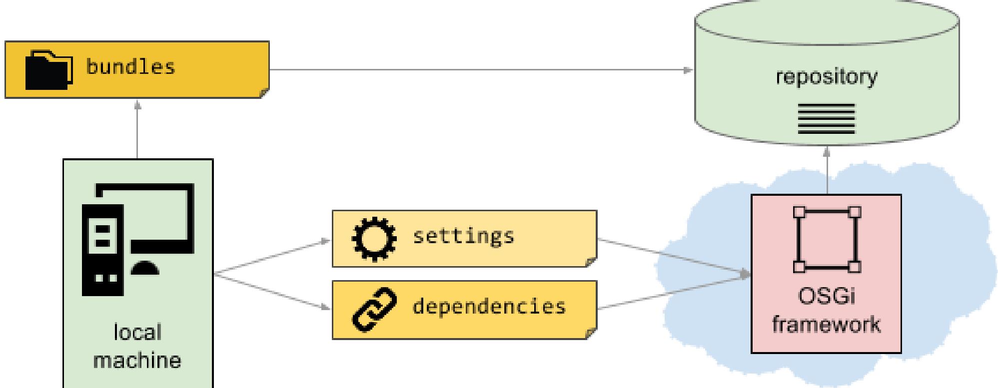

Abbildung 5.6.: Deployment mittels OSGi-Bundles

5.1.6. Source Code

Granularität	Unabhängigkeit	Kapselung	Komplexität	Lstg.-fähigkeit	Level
***	*	***	**	**	PaaS

Cloud Services wie *Heroku* bieten die Möglichkeit, den kompletten Build-Prozess auf ihre Infrastruktur auszulagern. Hierzu wird ein Repository (meist *Git*) zur Verfügung gestellt, in welches der Code einer Anwendung committet wird. Das Committen löst einen Build aus, welcher die Anwendung kompiliert, packt und auf die Infrastruktur des Anbieters (sprich *in die Cloud*) deployt.

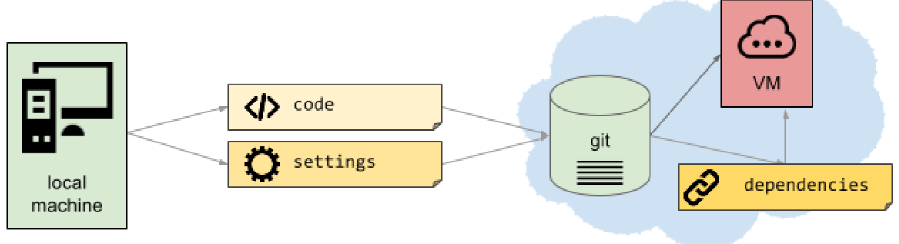

Abbildung 5.7.: Deployment mittels Quellcode

Algorithmus 5.2 Ausschnitt des Heroku Build Packs für Java

```
# install JDK
javaVersion=$(detect_java_version ${BUILD_DIR})
echo -n "----->␣Installing␣OpenJDK␣${javaVersion}..."
install_java ${BUILD_DIR} ${javaVersion}
jdk_overlay ${BUILD_DIR}
echo "␣done"
```

Diese Art des Deployments ist insbesondere für kleine oder mittlere Anwendungen[19] geeignet, die von Standardtechnologien (wie z.B. Maven oder Spring) Gebrauch machen. Solche Applikationen können automatisch und ohne weitere Konfiguration von den meisten Anbietern gebaut werden. Anwendungen, die nicht automatisch gebaut werden können, können mit Hilfe sogenannter *Build Packs* deployt werden. Dies sind Sammlungen von (Bash-) Skripten, Abhängigkeiten und Konfigurationen, die in den Build-Prozess beim Cloud-Anbieter integriert werden. Sie dienen dazu, eine virtuelle (Linux) Maschine einzurichten und sind anbieterspezifisch. Darstellung 5.2 zeigt einen Auszug aus dem Java Build Pack für Heroku[20], welcher das benötigte JDK installiert. Die Java-Version wird dabei aus dem hochgeladenen Code ermittelt.

Diese Art des Deployment bietet den Vorteil, den Entwickler vom eigentlichen Einrichten der Anwendung zu entlasten (er muss nur noch Code committen). Sie ist besonders bei reinen PaaS-Anbietern (wie Heroku) zu finden. Jedoch führt dieses Vorgehen zu starken Lock-in-Effekten und ist für große Anwendungen kaum umsetzbar.

[19] Viele Anbieter haben Limitierungen für die Größe der Anwendungen und Build-Prozesse. Heroku erlaubt etwa nur Build-Prozesse bis zu 10 Minuten und Anwendungen bis zu 300 MB, was u.U. nicht ausreichend ist (vgl. Heroku Dev Center, 2014a).

[20] Der Build Pack findet sich unter https://github.com/heroku/heroku-buildpack-java. Die angegebene Stelle ist der Datei `/bin/compile` (Zeile 44-49) entnommen.

5.2. Deployment-Strategie für den Informatica PIM Server

Die Evaluation der Deployment-Strategien in Kapitel 5.1 zeigt Vor- und Nachteile der jeweiligen Methoden. Diese Charakteristika sowie die Anforderungen des *Informatica PIM Servers* an Deployment und Laufzeitumgebung sind ausschlaggebend für die Wahl der Deployment-Strategie und werden im Folgenden dargelegt.

– Der PIM Server nutzt *keinen* (JEE) Application Container (wie *Apache Tomcat* oder *Eclipse Virgo*) als Laufzeitumgebung. Die Funktionalität solcher Container (z.B. Security oder Transaktionen) wird vom PIM Server selbst abgedeckt. Das Packen des Servers in ein JEE Archiv würde daher keinen Mehrwert bringen, da die Services und APIs der Container nicht genutzt würden.

– Der PIM Server nutzt *OSGi* und *Equinox* als Laufzeitumgebung. Zwar besteht der Server aus OSGi-Bundles, entspricht jedoch nicht strikt den OSGi-Designprinzipien (beispielsweise bei der Verwendung von Services, siehe Kapitel 2.3). Dies macht ein Deployment auf Basis individueller OSGi-Bundles schwierig.

– Bei einer Firmenanwendung wie dem PIM Server ist nicht davon auszugehen, dass im laufenden Betrieb Komponenten und Bundles deployt oder aktualisiert werden. Daher ist ein Deployment von einzelnen Bundles oder Quellcode nicht zielführend. Zwar könnte dieses Vorgehen in der Entwicklung genutzt werden, für ein Produktionssystem ist es aber unzureichend.

– OSGi und Equinox werden von *keinem* Cloud-Anbieter als PaaS unterstützt. Somit müsste ein OSGi-Framework und eine entsprechende Infrastruktur in einem zusätzlichen Schritt in der Cloud eingerichtet werden (wie etwa in Bakker and Ertman, 2013 beschrieben). Dies macht das Vorgehen unnötig aufwendig.

– Der PIM Server nutzt Windows-spezifische Bibliotheken, etwa für das Verarbeiten von Bildern. Dies führt zu Problemen, da der überwiegende Teil der Cloud-Anbieter Linux verwendet. Informatica plant jedoch eine Portierung auf Linux.

– Der PIM Server ist ein sehr umfangreiches Projekt (siehe Abbildung 2.4). Der entpackte Server ist größer als 500 MB (einschließlich Equinox und externer Bibliotheken). Dies hat weniger Einfluss auf die Deplyoment-Strategie, als auf die Zielplattform selbst, da einige Cloud-Anbieter (insbesondere PaaS-Anbieter) hinsichtlich der Anwendungsgröße ein Maximum vorsehen (*Heroku* erlaubt etwa nur Anwendungen bis zu 300 MB, siehe Heroku Dev Center, 2014a).

– Der Quellcode des PIM Servers ist vertraulich. Die Firmenpolitik verbietet, den Code auf Plattformen Dritter hochzuladen. Dies wäre jedoch ein nötiger Schritt, um den Code via *Git* (oder *SVN*) zu deployen, wie es viele Cloud-Anbieter vorsehen.

Aufgrund dieser Kriterien grenzt sich die Auswahl an Deployment-Methoden ein:

- Ein Deployment auf Basis von Source Code entfällt aufgrund von Firmenrichtlinien.
- Ein Deployment aus Basis von OSGi-Bundles entfällt, da der Server nicht hinreichend modular ist, Aktualisierungen zur Laufzeit nicht gewollt sind und es keinen PaaS-Anbieter für OSGi gibt.
- Ein Deployment aus Basis von JEE-Archiven entfällt, da der Server keinen solchen Application Container nutzt.
- Ein manuelles Deplyoment entfällt, da es keinen Mehrwert bietet.

Aufgrund dessen wird der PIM Server im Folgenden mit *Docker* verteilt. Alternativ wäre der Einsatz von VM-Images denkbar, jedoch bietet Docker diesem Ansatz gegenüber Vorteile: Neben kleineren und ressourcenschonenderen Images (siehe Abbildung 9.1) kann mit Docker auf ein Set von vorgefertigten Images aus dem *Docker Hub*[21] zurückgegriffen werden. Docker stellt außerdem eine REST API bereit, um Images und Container zu managen. So kann auf einer *PaaS-ähnlichen* Abstraktion aufgesetzt werden.

Mit Docker lassen sich außerdem heterogene Anwendungen deployen. So wird der Informatica PIM Server häufig in Verbindung mit dem hauseigenen *Media Manager* oder *Supplier Portal* verwendet. Diese können ebenfalls in Docker Images gepackt und analog verteilt werden. Das Zielsystem kann so nach einem Baukastenprinzip zusammengesetzt werden. Abbildung 5.8 stellt schematisch die Deployment-Strategie mit Docker dar (mehr dazu in Kapitel 9).

Zu bemängeln an dieser Lösung ist die Abhängigkeit von Linux. Zwar läuft der PIM Server unter Linux, wird jedoch nicht unterstützt oder getestet. Einige Features (wie das Bearbeiten von Bildern) nutzen Windows-spezifische Bibliotheken und sind daher nicht verfügbar. Informatica sieht jedoch eine Portierung auf Linux vor, weshalb diese Probleme an dieser Stelle ignoriert werden.

Am 9. Juni 2014 wurde auf der *Docker Con* in San Francisco die erste stabile Version von Docker (v. 1.0) vorgestellt (vgl. Docker Inc., 2014b). Aufgrund des Veröffentlichungsdatums wurden für diese Arbeit jedoch hauptsächlich Entwicklungsversionen (<= 1.0) genutzt. Mit dem Erscheinen der Version 1.0 wird Docker nun aber auch offiziell für Produktionssysteme empfohlen.

[21] *Docker Hub* (ehemals *Docker Index*) ist ein offizielles Repository in dem Docker Images veröffentlicht werden können. Er besteht aus einer Registry die Images speichert und einer Weboberfläche in der registrierte Nutzer Anwendungen suchen und kommentieren können.

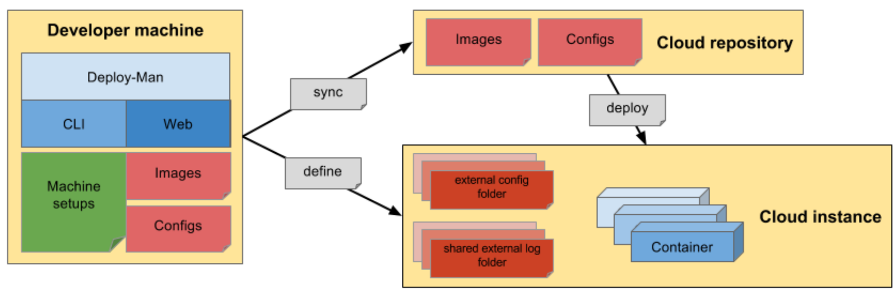

Abbildung 5.8.: Deployment-Strategie für den PIM Server mit Docker

5.3. Alternativen für modulare OSGi-Anwendungen

Das modulare Programmiermodel von OSGi eignet sich zur Umsetzung eines dynamischen Deployments in die Cloud. OSGi-Anwendungen bestehen aus unabhängigen Komponenten (sogenannten Bundles, siehe Kapitel 2.1), die zur Laufzeit deployt und ausgetauscht werden können. Aufbauend auf diesem Mechanismus beschreiben Bakker and Ertman [2013][22] ein Vorgehen für das Deployment solcher Anwendungen in die Cloud. Dieses wird im Folgenden verallgemeinert vorgestellt. Das Vorgehen entspricht der in Kapitel 5.1.5 diskutierten Deployment-Methode.

Festzuhalten ist, dass sich diese Art des Deployments *nicht* für den Informatica PIM Server eignet. Wie in Kapitel 2.3 beschrieben, ist der PIM Server nicht modular im Sinne von OSGi. Es ist nicht möglich, Komponenten zur Laufzeit zu deployen. Zwar werden Extension Points genutzt, jedoch keine OSGi-Services, wodurch Komponenten stark voneinander abhängen. Die eigene Kommunikationsschicht und die Abhängigkeit der Clients vom Server verhindern zusätzlich Aktualisierungen zur Laufzeit. Dieses Verhalten ist gewollt, da es das zugrundeliegende Programmiermodell vereinfacht. Aktualisierungen zur Laufzeit stellen außerdem (insbesondere für Produktionssysteme) ein hohes Risiko dar und sind in der Praxis meist unerwünscht. Das im Folgenden beschrieben Szenario geht daher von einer idealtypischen OSGi-Anwendung aus, die am Beispiel von *Apache ACE* verteilt werden soll.

Apache ACE ist ein quelloffenes Werkzeug zum Deployment auf verteilten Systemen. Es erlaubt das Installieren und Aktualisieren einzelner OSGi-Bundles über eine Weboberfläche. Neben Apache ACE gibt es weitere Lösungen, die vergleichbare Funktionen bieten. Tabelle 5.2 zeigt eine Gegenüberstellung dieser Optionen.

Während Apache ACE lediglich dem Deployment dient, stehen bei den verglichenen Lösungen anderen Funktionen im Vordergrund: *Eclipse Virgo* ist ein Application Server[23], *Eclipse Gyrex* eine Cluster-Lösung und *Apache Karaf* eine erweiterte OSGi-Laufzeitumgebung. Jedoch bieten alle Lösungen ein ähnliches Vorgehen für das Deployment

[22] Paul Bakker arbeitet für die niederländische Firma *Luminis* (http://luminis-technologies.com). Diese entwickelte eine Lernplattform auf Basis von OSGi in der Cloud (siehe Bakker and Offermans, 2013). Aus dieser Arbeit ging *Apache ACE* sowie die Cloud-Bibliothek *Amdatu* hervor. Die im Folgenden zitierten Autoren *Marcel Offermans* und *Jago de Vreede* sind ebenfalls Angestellte der Firma Luminis.

[23] Virgo ging aus dem *SpringSource dm Server* hervor, einem Application Container für OSGi und Spring.

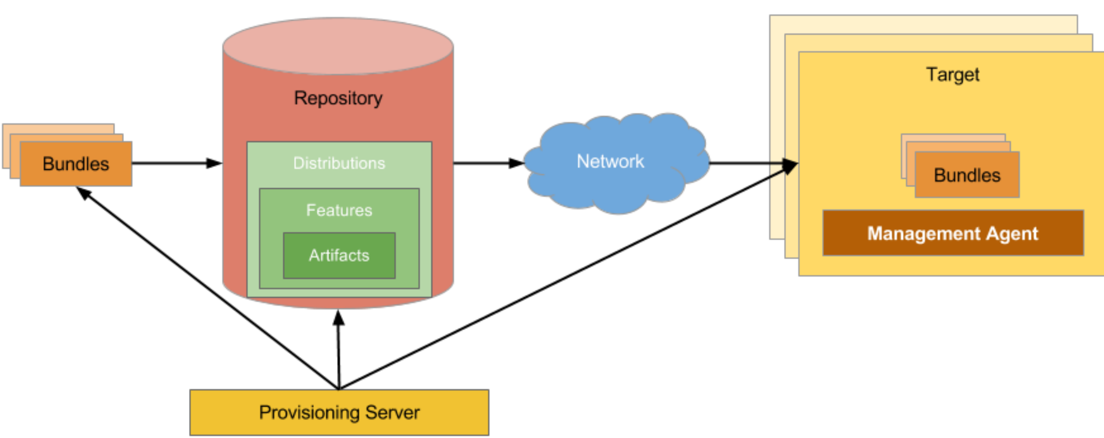

Abbildung 5.9.: OSGi Provisioning (nach de Vreede and Offermans, 2013, Folie 22)

von OSGi-Bundles. Apache ACE, Eclipse Virgo und Eclipse Gyrex bieten eine Weboberfläche, Apache Karaf ein Kommandozeileninterface, mit dem Bundles auf eine (entfernte) Maschine verteilt und gestartet werden können.

OSGi-Bundles werden hierzu zunächst in einem Repository hinterlegt (auch als *OSGi Bundle Repository*, kurz OBR bezeichnet). Bei Apache ACE können Bundles über die Weboberfläche in das OBR hochgeladen werden. Andere Tools (wie Eclipse Virgo) erlauben das Ablegen der Bundles in einem Ordner. Die Bundles im OBR können dann zu logischen Einheiten gruppiert werden. Apache ACE nennt diese Einheiten *Features*[24], Eclipse Virgo bezeichnet sie als *Plan*. Je nach Werkzeug können diese logischen Einheiten wiederum zusammenfasst und beispielsweise mit Konfigurationen versehen werden. Apache ACE bezeichnet dies als *Distribution*. Letztendlich können Bundles auf ein Zielsystem deployt werden. Apache ACE und Eclipse Gyrex können mehrere solcher Systeme verwalten, Apache Karaf und Eclipse Virgo nur das eigene. Abbildung 5.10 zeigt die Deployment-Oberfläche von Apache ACE (aus Apache ACE, 2014).

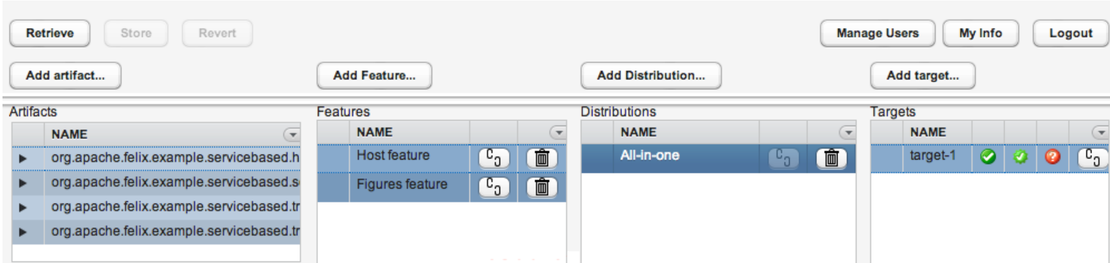

Abbildung 5.10.: Deployment-Oberfläche von Apache ACE (aus Apache ACE, 2014)

[24]Ein *Apache ACE Feature* ist kein *Eclipse Equinox Feature*. Beide Formate teilen sich zwar den Namen, sind technisch aber nicht kompatibel.

Werkzeuge dieser Art können für ein Deployment in die Cloud genutzt werden. Offermans and Bakker [2013] beschreiben hierzu ein Beispielszenario mit Apache ACE und Amazon Web Services (ab Minute 43): Eine Cloud-Instanz wird mit einem Standard-Image gestartet, welches lediglich ein OSGi-Framework (hier *Apache Felix*) und einen Managementagenten von Apache ACE enthält. Dieser verbindet sich mit einem zentralen Apache ACE Server und bietet sich als Zielplattform an. Die Cloud-Instanz kann dann von diesem Server mit OSGi-Bundles eingerichtet werden. Insofern die Anwendung hinreichend modular ist, können auch Aktualisierungen vom Server auf die Instanzen verteilt werden.

Dieses Vorgehen hat den Vorteil, dass Anwendungen feingranular verteilt werden können. Es lassen sich so bestimmte Versionen oder einzelne logische Teile der Anwendung deployen. Auch lassen sich Bundles zur Laufzeit aktualisieren.

An diesem Vorgehen kritisch zu bewerten sind folgende Punkte:

- Eine Aktualisierung zur Laufzeit ist (für Produktionssysteme) riskant und daher oft nicht gewollt. Werden Bundles aktualisiert, so werden sie zunächst gestoppt und deinstalliert (siehe Abbildung 2.2). Erst danach wird das neue Bundle installiert, aufgelöst und gestartet. Dadurch fällt die Funktionalität des Bundles aus. Abhängige (und transitiv abhängige) Bundles werden ebenfalls gestoppt, bis das neue Bundle verfügbar ist. Dadurch können große Teile der Anwendung ausfallen. Vor einem Update muss daher bekannt sein, welche Bundles voneinander abhängen (und ausfallen) und ob (und auf welche Weise) verbundene Clients betroffen sind. Folglich ist viel Wissen über die Interna der Anwendung nötig.

- Alle vorgestellten Lösungen arbeiten zwar auf Basis von OSGi, nutzen aber proprietäre Lösungen. Features von Apache ACE sind nicht kompatibel mit Features in Equinox. Während Equinox Bundles als Ordner anstatt als JAR-Archive erlaubt, unterstützt dies Apache ACE nicht (obwohl Apache ACE Equinox als Laufzeitumgebung nutzen kann). Eclipse Virgo schlägt mit PAR-Archiven gar ein eigenes Dateiformat vor. Bestehende Anwendungen sind so nicht ohne Weiteres kompatibel[25].

- Es lassen sich nur OSGi-Anwendungen (teilweise noch JAR- und WAR-Archive) über diese Methode verteilen. Die meisten Anwendungsszenarien benötigen jedoch weitere Softwarekomponenten, etwa einen Web Server, Caches oder Logging-Werkzeuge.

Das Fazit zum Deployment von OSGi in die Cloud lässt sich daher wie folgt ziehen: Anwendungen könne auf Grundlage von OSGi in die Cloud deployt werden. Die Modularität zur Laufzeit und die Unterteilung von Komponenten in einzelne Bundles, lassen granuläre Installationen und Aktualisierungen zu. Dieses Vorgehen wird von verschiedenen Werkzeugen unterstützt. Jedoch bietet OSGi keinen entscheidenden Vorteil gegenüber

[25] Es war beispielsweise nicht möglich, den PIM Server in Virgo zu deployen, da dieser dasselbe Package mehrmals exportiert, was Virgo nicht erlaubt. Beide Anwendungen basieren jedoch auf Equinox als Laufzeitumgebung.

anderen Technologien in der Cloud. Anwendungen müssen weiterhin mit Hinblick auf Cloud-Services und Plattformen entwickelt werden. Die Verwendung der dargestellten Werkzeuge ist teils mit erheblichem Aufwand verbunden, was nicht zuletzt am uneinheitlichen OSGi-Standrad und proprietären Features liegt. Zudem wird OSGi von keinem Cloud-Anbieter als Plattform offeriert, wohl auch deshalb, weil OSGi keine Cloud-Technologie ist.

	Apache ACE	Karaf	VIRGO from eclipseRT	GYREX
Deployment-Einheit	Artifact	Bundle	Bundle	Artifact
Logische Einteilung 1	Feature		Plan	Installable Units
Logische Einteilung 2	Distribution	Feature	PAR-Datei	Profile
Domäne	Provisioning	Runtime (Application Container, Provisioning)	Application Container (Provisioning)	OSGi-Clusters (Provisioning, Web Container)
Runtime	Eclipse Equinox, Apache Felix, Knopflerfish	Eclipse Equinox, Apache Felix	Eclipse Equinox	Eclipse Equinox
Interface	Web	Terminal	Web	Web
Deployment	eigene Lösung	eigene Lösung	p2, eigene Lösung	p2
Features	Provisioning	Logging, Provisioning, Administration, Security	Logging, Web Container, Application Server, Provisioning, Administration, Enterprise APIs	Logging, Web Container, Provisioning, Configuration Management, Monitoring
Upload	Weboberfläche	Ordner	Ordner, Weboberfläche	p2
Cloud Support	ja		(ja)	ja
URL	https://ace.apache.org	http://karaf.apache.org	http://eclipse.org/virgo	http://eclipse.org/gyrex
Kommentar	junges Projekt; etliche Teile der Projektseite sind leer; speziell für Cloud-Deployment;	Links zu Downloads führen teilweise ins Leere; veraltet;	umfangreiches Projekt; teilweise Cloud-Unterstützung; seit 2012 stark sinkende Aktivität;	junges Projekt; Support Forum hat bislang 29 Posts, etliche von Projektmitgliedern selbst;

Tabelle 5.2.: Vergleich von Deployment-Werkzeugen für OSGi

6. Skalierung

Ein zentrales Charakteristikum der Cloud (egal ob *IaaS*, *PaaS* oder *SaaS*) ist die Skalierbarkeit von Applikationen. Im Folgenden wird die vertikale und horizontale Skalierung vorgestellt und der Begriff der *Elastizität* erklärt.

6.1. Vertikale Skalierung (*scale up*)

Vertikale Skalierung (auch als *scale up* bezeichnet) erhöht die Leistungsfähigkeit eines Systems durch das Hinzufügen von Ressourcen (vgl. Smith, 2012). Dies bedeutet, dass beispielsweise durch das Aufrüsten von CPU oder RAM ein Server vertikal skaliert wird.. Diese Art der Skalierung ist sehr einfach und kann auf jede Applikation angewandt werden (da sie sich nur auf die Hardware bezieht). Jedoch stößt sie schnell an (physikalische) Grenzen wie etwa Obergrenzen für RAM oder I/O-Operationen.

6.2. Horizontale Skalierung (*scale out*)

Horizontale Skalierung (auch als *scale out* bezeichnet) erhöht die Leistungsfähigkeit eines Systems durch das Hinzufügen von Instanzen (vgl. Smith, 2012). Eine Anwendung wird also durch das Hochfahren neuer (physischer) Server skaliert. Diese Art der Skalierung setzt eine entsprechende Architektur der Applikation voraus, um diese mit mehreren Instanzen betreiben zu können. Ein Beispiel für eine solche Architektur ist das sogenannte *Shared-Nothing*-Prinzip (siehe Stonebraker, 1986). Instanzen einer Anwendung teilen demnach keinen gemeinsamen Zustand wie etwa HTTP-Sessions oder Caches (vgl. Steinacker, 2013). Dies macht horizontale Skalierung deutlich aufwändiger. Jedoch ist diese Art der Skalierung weitaus leistungsfähiger, da sie kaum Begrenzungen unterliegt[1].

[1] Begrenzungen sind etwa der steigende Aufwand, um die Instanzen zu verbinden oder Last zwischen ihnen zu verteilen.

6.3. Elastizität

Während Skalierung (nur) den Leistungsausbau einer Anwendung adressiert, geht der Begriff der *Elastizität* einen Schritt weiter: Elastizität meint das (dynamische) Vergrößern *und* Verkleinern der Leistungsfähigkeit einer Anwendung (vgl. Bakker and Ertman, 2013, Seite 102-104). Elastische Anwendungen bzw. Plattformen können also unter Last (etwa durch das Starten weiterer Server) skaliert und bei Wegfall der Last wieder verkleinert werden. So kann gezielt auf die Auslastung des Systems reagiert werden[2].

6.4. Skalierung des Informatica PIM Servers

Informatica arbeitet aktuell (Stand *März 2014*) an einer horizontalen Skalierung für den PIM Server. Diese soll Teil einer kommenden Version werden. Dabei sollen mehrere Instanzen des Servers parallel betrieben und die Last (gemessen anhand eingehender Client-Verbindungen) im Round-Robin Verfahren zwischen ihnen verteilt werden können. Die Server besitzen dabei eine oder mehrere Rollen (z.B. Job Server oder Web Server) und können so exklusiv für Background-Jobs oder Client-Verbindungen zur Verfügung stehen. Einer der Server übernimmt eine Master-Rolle (und damit auch das Load Balancing), die anderen sind Slaves. Die Server sollen untereinander kommunizieren, etwa um Locks auszutauschen oder neue Server in das Cluster aufzunehmen.

Der geplante Multi-Server-Betrieb ist ein ideales Szenario für die Cloud und die in dieser Arbeit entwickelte Deployment-Strategie. Da alle Server-Instanzen auf derselben Code-Basis aufbauen und sich nur in ihrer Konfiguration unterschieden, können sie einfach mittels Docker verteilt werden. Die Überwachung des Clusters (beispielsweise welche Container laufen) kann über die REST API von Docker und Amazon Web Services[3] erfolgen (siehe Kapitel 10.4.2). Log-Informationen könne wie in Kapitel 10.4.1 beschrieben zentral aggregiert werden.

Da die horizontale Skalierung des PIM Servers momentan von einem Team von Informatica umgesetzt wird, wird dieses Thema an dieser Stelle nicht näher betrachtet.

[2] Offermans and Bakker [2013] beschreiben dazu folgenden Anwendungsfall: Eine Software für Schulen wird während der Unterrichtszeiten stark genutzt. Auf diese Last wird mit dem Zuschalten neuer Instanzen reagiert. Nach Schulende wird die Software kaum genutzt, die Last fällt bis zum nächsten Morgen deutlich ab. Um Kosten zu sparen, werden während dieser Zeit alle Instanzen bis auf eine (um den Service verfügbar zu halten) abgeschaltet.

[3] *Amazon Web Services* wurde für die Implementierung in dieser Arbeit verwendet. Die Plattform wird in Kapitel 8.1 vorgestellt.

7. Cloud-Services

Cloud-Services sind Softwarekomponenten, die von Anbietern als Dienstleistung betrieben und für Anwendungen zur Verfügung gestellt werden. Hierzu gehören etwa Datenbanken (z.B. *MSSQL*), Caches (z.B. *Memcached*) oder Suchmaschinen (z.B. *Elasticsearch*). Nutzer können die Services beliebig starten, konfigurieren und als Bestandteil der eigenen Anwendung verwenden. Der Anbieter gewährleistet dabei die Verfügbarkeit, Skalierbarkeit und Performance der Dienste.

Services in der Cloud stellen den wesentlichen Unterschied zwischen IaaS- und PaaS-Angeboten dar. Während IaaS-Angebote lediglich Infrastruktur wie Maschinen, Speicher und Netzwerke anbieten, offerieren PaaS-Angebote komplette Softwarekomponenten, die auf Wunsch des Nutzers bereitstehen. Dies verringert den Aufwand in der Entwicklung und Administration, hilft aber auch, Anwendungen (einfach) zu skalieren.

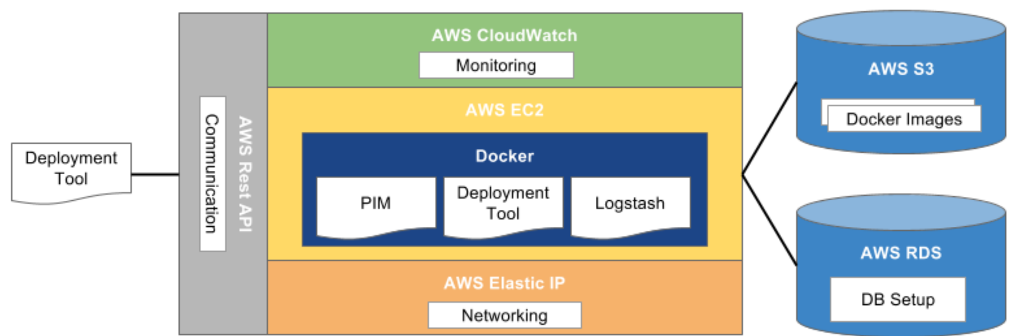

Abbildung 7.1.: AWS Cloud-Dienste für den Informatica PIM Server

In Abbildung 7.1 sind die Cloud-Dienste von Amazon dargestellt, die in dieser Arbeit verwendet werden. Die Dienste werden in Kapitel 8.1 vorgestellt.

Für den Informatica PIM Server als datenintensive Anwendung ist vor allem eine *Database as a Service* interessant. Da der PIM Server nur *MSSQL* und *Oracle* unterstützt, kamen als Anbieter lediglich *Microsoft Azure* (mit einer MSSQL Datenbank) und *Amazon Web Services* (mit einer MSSQL und einer Oracle Datenbank) in Frage (siehe Kapitel 8.2). Wie in Kapitel 8 dargelegt, wurde Amazon Web Services als Plattform für die Implementierung in dieser Arbeit gewählt. Aus diesem Grund wurde Amazons Datenbankservice (*AWS RDS*) evaluiert.

RDS bietet MySQL-, PostgreSQL-, MSSQL- und Oracle-Datenbanken als Cloud-Dienst von Amazon. Je nach Datenbank sind diese mit unterschiedlichen Lizenzmodellen verbunden: für MSSQL und Oracle besteht die Möglichkeit, existierende Lizenzen wiederzuverwenden oder neue zu erwerben, MySQL und PostgreSQL sind lizenzfrei. Unabhängig von der Datenbank fallen Kosten für Betrieb und Speicher an[1].

Amazon stellt für alle Datenbanken eine Web-basierte Administrationsoberfläche sowie eine REST API zur Verfügung. Es können manuelle oder automatische *Snapshots* (also Abbilder der Datenbank zu einem bestimmten Zeitpunkt) erstellt werden und diese als neue Instanz gestartet werden. Über *CloudWatch*[2] steht ein detailliertes Monitoring der Datenbanken bereit. Instanzen könne zur Laufzeit skaliert werden und lassen sich in mehreren Datenzentren (sogenannten *Availability Zones*) betreiben. Eine Übersicht der Funktionen von RDS findet sich in Amazon Web Services [2014].

Bei der Evaluation von RDS stellte sich in erster Linie das handgeschriebene Datenbank Setup aus SQL- und ANT-Skripten als problematisch heraus:

– Das Setup für die Oracle-Datenbank benötigt `SYSDBA`*-Rechte* bzw. einen Zugriff als `SYSTEM`*-User*. Beides steht auf RDS nicht zur Verfügung (vgl. Amazon Web Services, 2013). Es ist zu bezweifeln, dass diese Rechte bei anderen Anbietern verfügbar sind, da mit ihnen Interna der Datenbank konfiguriert werden können[3].

– Die Verbindung läuft nicht über JDBC, sondern über die Console der jeweiligen Datenbank (also die *MSSQL sqlcmd* bzw. die *Oracle SQLPlus Console*). Die lokale Installation dieser Werkzeuge ist Voraussetzung für das Ausführen des Setups. Die eigentlichen SQL-Skripte werden erst während des Setups mittels ANT generiert. Dies macht es schwierig, das Setup selbst in die Cloud oder einen anderen Prozess auszulagern.

– Der Server unterstützt nur wenige Major-Minor-Build-Versionen der Datenbanken. Diese sind auf AWS RDS nicht verfügbar. Nicht verifizierte Versionen können jedoch in einer Konfigurationsdatei freigegeben werden. Welche Versionen aus welchem Grund unterstützt sind (bzw. warum nicht), ist nicht dokumentiert. Das hinzufügen einer weiteren Datenbankversion führte im Lauf dieser Arbeit jedoch zu keinen Komplikationen.

– Dem Datenbankschema liegt ein insgesamt großes Setup zugrunde. Je Datenbank (*MSSQL* oder *Oracle*) werden ca. 350 SQL-Scripte benötigt, von denen etliche mehrere hundert Zeilen umfassen (einige Scripte enthalten mehr als 40.000 generierte Statements). Neben dem Grundschema werden mehr als 300 Updates eingespielt. Dies macht das Setup zeitintensiv: über eine Internetverbindung mit etwa 6500 kBit/s Upstream dauert das Setup ca. 35 Minuten.

[1]Die Preise können unter http://aws.amazon.com/de/rds/pricing eingesehen werden.
[2]*CloudWatch* ist Amazons Monitoring-Dienst. Mit ihm lassen sich virtuelle Maschinen, Datenbanken und Netzwerke überwachen. Mehr unter http://aws.amazon.com/de/cloudwatch.
[3]Hierzu gehören etwa Speicher und Prozesse, welche als Angebotsbestandteil berechnet werden.

Es ist jedoch festzuhalten, dass das Setup *nicht* für eine Installation auf entfernten Rechnern vorgesehen ist. Es wird lediglich für das Einrichten einer lokalen Datenbank verwendet. Trotz dessen konnte eine MSSQL-Instanz in der Cloud erfolgreich aufgesetzt werden. Bis auf die oben genannten Kritikpunkte, funktioniert das Setup zuverlässig.

Teil IV.

Prototypische Umsetzung

8. Cloud-Plattform

8.1. Amazon Web Services

Layer 6	SaaS	Applications			*End-users*
Layer 5	App Services	Compute App Services	Communicate App Services	Store App Services	*Citizen developers*
Layer 4	Model-Driven PaaS	bpmPaaS, Model-Driven aPaaS	Model-Driven iPaaS	baPaaS	*Business engineers*
Layer 3	PaaS	AWS Elastic Beanstalk	iPaaS	AWS RDS	*Professional developers*
Layer 2	Foundational PaaS	Application containers	AWS Elastic IP	AWS S3	*DevOps*
Layer 1	Software Defined Datacenter	AWS EC2	Software Defined Networking (SDN)	AWS EBS	*Infrastructure engineers*
		Compute	Communicate	Store	

Abbildung 8.1.: Einordnung von AWS in das Cloud-Modell von Haan, 2013

Amazon Web Services (kurz *AWS*) ist der Marktführer im Cloud-Geschäft (siehe Bort, 2013) und steht in Konkurrenz zu Angeboten von Microsoft (*Azure*), Google (*App Engine*), Red Hat (*OpenShift*) und zahlreichen weiteren Unternehmen. Das Angebot existiert seit 2006 und umfasst Cloud-Services für Computing (z.B. *EC2*), Storage (z.B. *S3*), Networking (z.B. *Elastic IP*) und vielem mehr.

AWS wurde aufgrund seiner umfangreichen Plattform gewählt: Es deckt das gesamte Cloud-Spektrum (*IaaS*, *PaaS* und *SaaS*) ab und unterstützt verschiedenste Technologien (unter Anderem *Linux*, *Windows*, *Java* und *Docker*) für seine Dienste[1]. AWS ist außerdem der einzige Anbieter, der sowohl MSSQL als auch Oracle als Datenbanksysteme anbietet (die einzigen Datenbanken, die vom Informatica PIM Server unterstützt werden).

Im Folgenden werden die wichtigsten AWS-Dienste eingeführt, die in dieser Arbeit verwendet werden. Anschließend wird ein Überblick zu vergleichbaren Angeboten gegeben.

[1] AWS wird aufgrund seines Umfangs bisweilen auch als *EaaS*-Plattform (*Everything-as-a-Service*) bezeichnet (vgl. Lipinski et al., 2013).

8.1.1. AWS REST API

AWS bietet eine *REST API* zur Steuerung seiner Services. Durch die API können unter Anderem virtuelle Maschinen gestartet, IP-Adressen zugewiesen oder Rechtegruppen vergeben werden. Zu dieser REST API stellt Amazon Bibliotheken in verschiedenen Sprachen bereit, unter Anderem *Java*, *Python* und *JavaScript*. Dadurch kann die API nahtlos in Anwendungen eingebunden werden. Zudem existieren Implementierungen von Drittherstellern, etwa *Apache Libcloud*[2] oder *Apache JClouds*[3]. Diese versprechen eine einheitliche API für Dienste verschiedener Cloud-Anbieter, haben jedoch einen geringeren Funktionsumfang. Sie stellen die Schnittmenge der unterstützten Anbieter dar. In dieser Arbeit wurde daher das offizielle AWS Java SDK verwendet, da dieses den voll Funktionsumfang der Amazon Web Services implementiert.

8.1.2. AWS EC2

Amazon Elastic Compute Cloud (kurz *EC2*) ist eine IaaS. EC2 bietet virtuelle Linux- oder Windows-Maschinen in der Cloud, die über die AWS REST API konfiguriert und gestartet werden können. Die Konfiguration umfasst z.B. den Instanztyp[4], das installierte Image[5] oder die Rechtegruppen. Die Maschinen gewähren vollen Root-Zugriff über SSH bzw. RDP (Microsofts *Remote Desktop Protokoll*).

8.1.3. AWS S3

Amazon Simple Storage Service (kurz *S3*) ist ein Key-Value-Store für beliebig große Objekte. Objekte werden dabei in Buckets organisiert und über einen eindeutigen Schlüssel identifiziert. S3 kennt keine *Ordner*, jedoch können Ordnerstrukturen über entsprechende Schlüssel abgebildet werden. Auf Dateien in S3 kann über einen öffentlichen Link oder der AWS REST API zugegriffen werden.

8.1.4. AWS RDS

Amazon Relational Database Service (kurz *RDS*) ist eine „Datenbank as a Service“ in der Cloud und unterstützt *MySQL*, *MSSQL*, *PostgreSQL* sowie *Oracle*. RDS ermöglicht die Konfiguration einer Datenbank (etwa Instancetyp, Nutzer oder Lizenz) und das Starten dieser auf Amazons Infrastruktur. AWS übernimmt dabei die Skalierung, das Backup, Snapshots und mehr (siehe hierzu Kapitel 7).

[2] *Apache Libcloud* ist eine Python-Bibliothek, die AWS, OpenStack, Google Compute Cloud und viele weitere Anbieter unterstützt. Mehr unter https://libcloud.apache.org.

[3] *Apache JClouds* ist eine Java-Bibliothek. Sie unterstützt ebenfalls AWS, OpenStack und Google Compute Cloud sowie weiter Angebote. Mehr unter http://jclouds.apache.org.

[4] Instanztypen definieren die virtuelle Hardware einer VM, also z.B. die Anzahl der CPUs und die Größe des RAM. Es kann zwischen ca. 30 Instanztypen gewählt werden. Mehr unter https://aws.amazon.com/de/ec2/instance-types.

[5] AWS stellt Images für alle gängigen Betriebssysteme bereit und erlaubt das Verwenden (und Veröffentlichen) von eigenen Images.

8.1.5. AWS Elastic IP

AWS Elastic IP ist ein Netzwerk-Service, der es erlaubt, EC2-Instanzen feste IP-Adressen zuzuweisen. IP-Adressen werden dazu reserviert und dann einer beliebigen EC2-Instanz zugewiesen. Dies eignet sich etwa zur Umsetzung eines einfachen Blue-Green-Deployments (siehe hierzu Sato, 2013)[6] oder dazu zu gewährleisten, dass ein bestimmter Service immer unter einer festen Adresse verfügbar ist.

8.2. Vergleichbare Angebote

AWS und insbesondere einzelne seiner Services stehen in Konkurrenz zu verschiedenen Anbietern. Ein ähnliches Gesamtangebot aus Computing, Storage und Networking bieten *Microsoft Azure* und *OpenStack*. Dabei ist vor allem OpenStack hervorzuheben, da es eine Cloud-Infrastruktur als quelloffenes Projekt zum Betrieb auf eigener Hardware bereitstellt. Services wie *AWS Elastic Beanstalk* stehen in Konkurrenz zu PaaS-Anbietern wie Heroku. *AWS S3* steht in Konkurrenz zu Storage-Anbietern wie etwa Google Cloud Storage. Andere Services von AWS finden ebenfalls ihre Entsprechung in Anbietern, die sich auf den entsprechenden Dienst spezialisiert haben. Jedoch ist das Gesamtangebot von Amazon in der Cloud marktführend.

Wie bereits in Kapitel 8.1 erwähnt, ist die Datenbankunterstützung von AWS hervorzuheben. AWS ist der einzige Anbieter, der sowohl eine MSSQL Datenbank als auch eine Oracle Datenbank als Service offeriert. Beide Datenbanken werden vom Informatica PIM Server unterstützt.

Tabelle 8.1 zeigt einen Gegenüberstellung einiger Cloud-Anbieter. Dabei werden sowohl IaaS- als auch PaaS-Angebote aufgelistet. Die verglichenen Anbieter wurden aufgrund ihrer Polarität gewählt. Der Vergleich ist also weder repräsentativ noch erhebt er Anspruch auf Vollständigkeit. Der Markt an Cloud-Angeboten umfasst mittlerweile viele Dutzend Angebote und ist in ständigem Wandel. Ein ausführlicher Vergleich ist daher kaum möglich. Letztendlich bieten die meisten Angebote ähnliche Konzepte und unterscheiden sich lediglich in Details.

[6]Bei einem Blue-Green-Deployment wird eine aktuelle Instanz einer Anwendung betrieben (und „*blau*“ genannt). Parallel wird eine aktualisierte Instanz der Anwendung einrichtet (und als „*grün*“ bezeichnet). Zu einem bestimmten Zeitpunkt wird zwischen der alten und neuen Instanz umgeschaltet (also von *blau* auf *grün*).

	OS	Hosting	Plattform	API	RDBS als Service	Docker
amazon webservices	Linux, Windows	public	IaaS, PaaS	yes	MySQL, MSSQL, Oracle	nativer Support
Windows Azure	Linux, Windows	public	IaaS, PaaS	yes	MSSQL	
Google Compute Engine	Linux, Windows	public	IaaS	yes	MySQL	nativer Support
openstack	Linux	private (quelloffen)	IaaS	yes		nativer Support
CLOUD FOUNDRY	Linux	public/private (quelloffen)	PaaS	no	MySQL, PostgreSQL	geplanter Support
heroku	Linux	public	PaaS	yes	MySQL, PostgreSQL, SQLite	
Jelastic	Linux	public	PaaS	no	MariaDB, MySQL, PostgreSQL	
OPENSHIFT	Linux	public/private (quelloffen)	PaaS	yes	MySQL, PostgreSQL	

Tabelle 8.1.: Vergleich von Cloud-Anbietern

9. Docker als Application-Container

9.1. Docker

Wie bereits in Kapitel 5.1.3 eingeführt, ist Docker ein Application-Container zur isolierten Ausführung von Anwendungen auf Linux. Mit Docker können sogenannte *Images* gepackt werden, die eine oder mehrere Anwendungen und deren Abhängigkeiten enthalten. Die Images sind *self-contained*, können also auf ein anderes System portiert und dort durch Docker gestartet werden. Ein gestartetes Image wird als *Container* bezeichnet und kann mehrere Prozesse beinhalten. Jeder Container hat ein *eigenes* Dateisystem und ist isoliert von anderen Containern. Zur Kommunikation können Container externe Ordner einbinden oder Ports öffnen.

Docker Images können (durch inkrementelles Einrichten des Containers) von Hand erstellt oder durch ein sogenanntes *Dockerfile*[1] gebaut werden. Listing 9.1 zeigt das Dockerfile zum Bau des PIM Server Images. Ausgehend von Ubuntu installiert das Dockerfile Java, fügt Plugins, Features und Konfigurationen ein, öffnet die nötigen Ports und gibt ein Shell-Skript als Startkommando an.

Jedes Image basiert auf einem sogenannten *Base Image* (z.B. *Ubuntu*). Im Dockerfile können beliebige (Shell-) Befehle ausgeführt werden, etwa um Pakete (z.B. *Java*) zu installieren. Außerdem lassen sich Dateien bzw. Ordner vom Host in das Image kopieren und es können Ports geöffnet werden. In aller Regel enden Dockerfiles mit einem Befehl, welcher beim Start im Container ausgeführt wird (jedoch ist dies nicht zwingend, vgl. Docker Inc., 2014c).

Algorithmus 9.1 Dockerfile für den PIM Server

```
FROM ubuntu
RUN apt-get update && apt-get install -y openjdk-7-jre-headless
ADD /server/plugins/ /var/www/pim/plugins/
ADD /server/features/ /var/www/pim/features/
ADD /server/configuration/ /var/www/pim/configuration/
ADD /settings/run.sh /var/www/pim/run.sh
EXPOSE 1501 1712
CMD sh /var/www/pim/run.sh
```

Jeder Schritt beim Bau wird als eigener Layer im Image hinterlegt. Daher werden Images auch als *Repositories* bezeichnet, da sie eine komplette Historie enthalten. Dieser Mechanismus ermöglicht es, ein Image inkrementell zu aktualisieren oder einen Rollback durchzuführen (um zu einem früheren Stand zurückzukehren).

[1] *Dockerfiles* sind Textdateien mit einer Bash-ähnlichen Syntax. Sie haben keine Dateiendung.

Das Image, welches mit dem Dockerfile in Listing 9.1 erstellt wird, beinhaltet zu Beginn beispielsweise zehn Layer (also eines für jede Zeile, sprich Anweisung). Wird das Image erneut gebaut, so wird nur dann ein neuer Layer hinzugefügt, wenn sich Änderungen ergeben. Zudem teilen sich unterschiedliche Images einzelne Layer. Nutzt ein weiteres Image etwa ebenfalls *Ubuntu* als Base Image (wie in Listing 9.1), so teilen sich die Images diesen Layer. Images können so wesentlich schneller gebaut werden (da vorhandene Layer wiederverwendet werden) und sind kleiner (da Layer von mehreren Images geteilt werden). Abbildung 9.1 stellt das geschichtete Model von Docker schematisch einer virtuellen Maschine gegenüber.

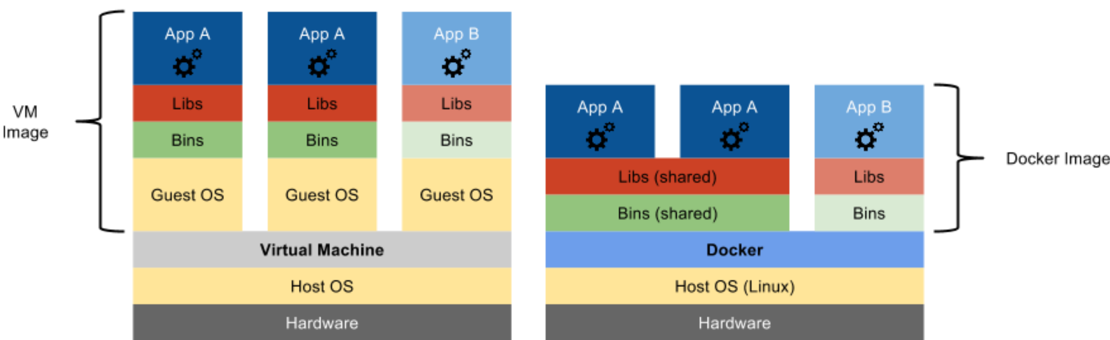

Abbildung 9.1.: Vergleich von virtuellen Maschinen und Docker (nach Docker Inc., 2014a)

Es gibt zwei Möglichkeiten, um fertige Images zu verteilen: (1) Images können als TAR-Archive exportiert werden oder (2) in ein spezielles Repository übertragen werden. Der Export in ein TAR-Archiv stellt eine bestechend einfache Lösung dar, hat jedoch den Nachteil, dass jedes Archiv alle Layer des jeweiligen Images enthält, also keine Layer mehr geteilt werden. Die Archive werden dadurch entsprechend groß. Das Hochladen der Images in die sogenannte *Docker Registry* ermöglicht das Teilen von Layern, bereits hochgeladene Layer werden so nicht erneut übertragen. Es ist dazu jedoch nötig, entweder die öffentliche Registry zu nutzen oder eine private aufzusetzen. Das Aufsetzen und Sichern einer privaten Registry ist dabei nicht trivial, da z.B. SSL-Zertifikate generiert werden müssen und ein Speicher für die Images nötig ist.

Für das in dieser Arbeit entwickelte Deployment-Werkzeug wurde Möglichkeit eins, der Export als TAR-Archiv gewählt. Zusätzlich wurde auch eine private Docker Registry eingerichtet (gesichert mit einem *Nginx-Server*, *Basic Auth* und *SSL*), die mit dem Tool selbst als Docker Image deployt werden kann.

Das Erstellen von Docker Images kann zwei Prinzipien folgen: (1) Anwendungen, die zusammen deployt werden sollen, werden gemeinsam in ein großes Image gepackt oder (2) es wird ein separates Image für jede Anwendung erstellt und mehrere Images deployt. Während die erste Variante das Deployment erleichtert, fördert die zweite Variante die Wiederverwendung und Separierung von Komponenten. Im Zuge dieser Arbeit wurde letztere Variante gewählt, da so aus einem Repository von Images diejenigen ausgewählt (und deployt) werden können, die zusammen den gewünschten Anwendungsstack ergeben.

Unabhängig von der gepackten Anwendung ist der Deployment-Prozess mit Docker immer derselbe: das Image wird gebaut oder geladen (aus einem TAR-Archiv oder einer Registry) und durch einen einzigen Befehl gestartet. So können auch unterschiedliche Anwendungen (z.B. Java- und Ruby-Programme) einheitlich verteilt werden.

Docker wird in erster Linie über ein *Command Line Interface* bedient, bietet jedoch auch eine *Remote API* über eine Web-Schnittstelle. Diese Remote API ermöglicht zum Beispiel, Container über einen HTTP-Aufruf zu stoppen oder alle laufenden Container als JSON-Objekt abzufragen. Für die API existieren *Bindings* für verschiedene Sprachen (beispielsweise Java) als 3rd-Party-Bibliotheken.

9.2. Einordnung von Docker in die Cloud-Landschaft

Docker ist *keine* dedizierte Cloud-Technologie und kann unabhängig von der jeweiligen Plattform auf Linux-Maschinen betrieben werden. Die Eigenschaft, Anwendungen portabel zu machen und einen einheitlichen Deployment-Prozess zu gewährleisten, prädestinieren Docker jedoch für die Cloud. Da Docker alle Abhängigkeiten zur eigentlichen Plattform beseitigt, setzt es das „*Write once, run everywhere*“-Prinzip um, das Sun Microsystems seinerseits für Java ausrief (vgl. Paolini, 1996).

In das in Kapitel 3.1.4 vorgestellte Modell zur Kategorisierung von Cloud-Services lässt sich Docker - wie in Abbildung 9.2 zusehen - einordnen. Die Haupteigenschaft von Docker ist das Packen von Anwendungen in Application Container. Darüber hinaus bietet Docker grundlegende Funktionen für das Verwalten von Ports und IP-Adressen. Die öffentliche Docker Registry (auch *Docker Hub* genannt), bietet außerdem die Möglichkeit, vorgefertigte Anwendungen herunterzuladen und als Service zu nutzen. So können Anwendungsstacks aus einer Art „Baukasten“ zusammengesetzt werden.

Layer 6	SaaS	Applications			*End-users*
Layer 5	App Services	Compute App Services	Communicate App Services	Store App Services	*Citizen developers*
Layer 4	Model-Driven PaaS	bpmPaaS, Model-Driven aPaaS	Model-Driven iPaaS	baPaaS	*Business engineers*
Layer 3	PaaS	Docker Index	iPaaS	dbPaaS	*Professional developers*
Layer 2	Foundational PaaS	Application Container	Ports	Object storage	*DevOps*
Layer 1	Software Defined Datacenter	Virtual Machines	Software Defined Networking (SDN)	Software Defined Storage (SDS)	*Infrastructure engineers*
		Compute	Communicate	Store	

Abbildung 9.2.: Einordnung von Docker in das Cloud-Modell von Haan [2013]

Docker wird unter Anderem[2] von Amazons PaaS *Elastic Beanstalk* nativ unterstützt. Hier kann ein Docker Image über eine Weboberfläche hochgeladen werden und direkt auf Amazons Infrastruktur deployt werden. Docker stellt sich damit in eine Reihe mit Plattformen wie *Node.js*, *IIS* oder *Tomcat* ein (mehr hierzu in Kapitel 9.5). Jedoch lässt sich Docker auf jeder kompatiblen Linux-Plattform installieren und bietet dann ähnliche Eigenschaften wie PaaS-Anbieter wie etwa *Heroku* (eine Gegenüberstellung von Docker und Heroku findet sich in Tabelle 9.1).

docker	heroku	
Dockerfile	Build Pack	Images werden in Docker über *Dockerfiles* gebaut. Diese enthalten Anweisungen, die das Image definieren (etwa welche Packages installiert werden sollen). Heroku sieht hierfür sogenannte *Build Packs* vor. Build Packs sind Shell-Skripte, die beim Deployment einer Anwendung auf Heroku den sogenannten *Slug* bauen (das Pendant zu Images in Docker).
Image	Slug	Die Deployment-Einheit von Docker sind Images. Einmal erstellt, können sie beliebig verteilt und gestartet werden. Heroku nennt diese Einheiten Slugs. Wie Images in Docker sind dies komprimierte und gepackte Kopien der Applikation (vgl. Heroku Dev Center, 2013). Sie sind *self-contained* und können isoliert gestartet werden.
Container	Dyno	Ein gestartetes Docker Image heißt Container. Heroku kennt dieses Konzept als *Dyno*. Ein Dyno ist ein leichtgewichtiger und isolierter Prozess auf Basis von LXC (vgl. Heroku Dev Center, 2014b), also der gleichen Technologie, die auch Docker nutzt.
Hub	Add-ons	Über das öffentlichen Repository bietet Docker die Möglichkeit, vorgefertigte Anwendungen als Bausteine herunterzuladen. Ein vergleichbares Angebot hat Heroku mit seinem *Add-on Market*. Hier können Nutzer vorgefertigte Services kostenlos oder gegen Gebühr beziehen.
CLI	CLI	Docker und Heroku verfügen jeweils über ein ähnliches Command Line Interface zum Managen von Containern. Beide können laufende Container listen, Logs einsehen oder Anwendungen stoppen.

Tabelle 9.1.: Vergleich von Docker und Heroku

[2] Die Unterstützung von Docker bei verschiedenen Anbietern ist in Tabelle 8.1 dargestellt.

9.3. Vergleichbare Angebote

Für Docker als Gesamttechnologie (Application Container, Virtualisierung, Deployment-Werkzeug, usw.) gibt es derzeit keine äquivalenten Angebote. Jedoch gibt es Alternativen zu den einzelnen Bestandteile von Docker.

Docker kann als *Application Container* Anwendungen packen und portabel machen. Ähnliches ermöglichen im Java-Umfeld JAR-, WAR- oder EAR-Archive.

Linux Containers (kurz *LXC*) werden von Docker intern als Virtualisierungstechnologie genutzt und stellen isoliert betrachtet eine Alternative zur Virtualisierung mit Docker dar. Des Weiteren können Betriebssystemvirtualisierungen wie OpenVZ und Linux-V Server oder virtuelle Maschinen wie VirtualBox genutzt werden.

Der Einsatz von Docker steht in erster Linie in Konkurrenz zu PaaS-Angeboten wie Heroku, da mit Docker ein vergleichbarer Arbeitsablauf realisiert werden kann. Wie in Tabelle 9.1 zu sehen, haben Docker und Heroku beispielsweise viele ähnliche oder gar gleiche Konzepte.

9.4. Entwicklungsabläufe mit Docker

Docker kann nicht nur als eigenständiges Deployment-Werkzeug dienen, sondern auch in einzelne Entwicklungsabläufe eingebunden werden. Im Folgenden werden Szenarien beschrieben, in denen Docker abseits des Deployments genutzt werden kann.

9.4.1. Tests

Docker eignet sich als Baustein in einem *Continuous Delivery* Prozess, da es einheitliche und isolierte Bedingungen in unterschiedlichen Umgebungen sicherstellt. Ein Docker Image kann etwa im ersten Schritt eines Build-Prozesses gebaut werden und als Grundlage für alle weiteren Schritte dienen. So können beispielsweise Unit- oder Integration-Tests auf dem Image ausgeführt werden. Da sich das Image immer gleich verhält, kann dieser Schritt leicht auf mehrere Maschinen verteilt werden. Sollten die Tests fehlschlagen, kann das Image verworfen werden und es bleiben keine Artefakte (z.B. Logs oder Umgebungsvariablen) auf dem Testsystem zurück. Waren die Tests erfolgreich, kann *dasselbe* Image als Grundlage für manuelle Tests in einer Qualitätssicherung (kurz *QA*, vom Englischen *Quality Assurance*) dienen. Da das Image auch hier die immer gleiche Umgebung gewährleistet, können mehrere Personen auf demselben Stand arbeiten. Werden Fehler gefunden, ist deren Rekonstruktion leicht, da das Image (und somit die Umgebung) einfach kopiert werden kann. Ein Entwickler, der den Fehler bearbeitet, kann so ohne großen Aufwand das Szenario, in dem der Fehler auftrat, wiederherstellen. Ist ein Image schließlich in allen Instanzen getestet und verifiziert, kann es direkt deployt werden. So können Tests nicht nur die Integrität der Software gewährleisten, sondern auch die der Installationsumgebung. Da das Image zudem nur einmal gebaut und aufgesetzt wird, entfällt dieser Aufwand in allen weiteren Schritten (wie etwa bei Unit-Tests oder QA). Dies macht einen Continuous Delivery Prozess verlässlicher und schneller.

9.4.2. Entwicklung

In Entwicklungsprozessen wird häufig eine Anwendung gegen die API einer anderen entwickelt. So wird z.B. das *Informatica Supplier Portal* gegen die REST API des PIM Servers programmiert. Hierfür ist es nötig, eine lokale Installation des PIM Servers (oder generell der Zielplattform) einzurichten. Wird ein Docker Image der Zielplattform bereitgestellt, kann dieses als *Blackbox*[3] verwendet werden. So kann eine gewünschte Zielplattform schnell (durch das Starten eines Images) bereitgestellt und bei Bedarf wieder gelöscht werden. Auch lässt sich einfach zwischen Zielplattformen wechseln, etwa zwischen verschiedenen Versionen. Dies kann Entwicklungsprozesse deutlich vereinfachen.

9.4.3. Dokumentation

Ähnlich wie ein ANT- oder Maven-Skript die Build-Schritte eines Projekts formalisiert und beschreibt, dokumentiert ein *Dockerfile* die Schritte zum Aufsetzen der Anwendungsumgebung. Dadurch wird etwa an zentraler Stelle hinterlegt welche Ordnerstruktur das Projekt vorschreibt oder welche Umgebungsvariablen benötigt werden.

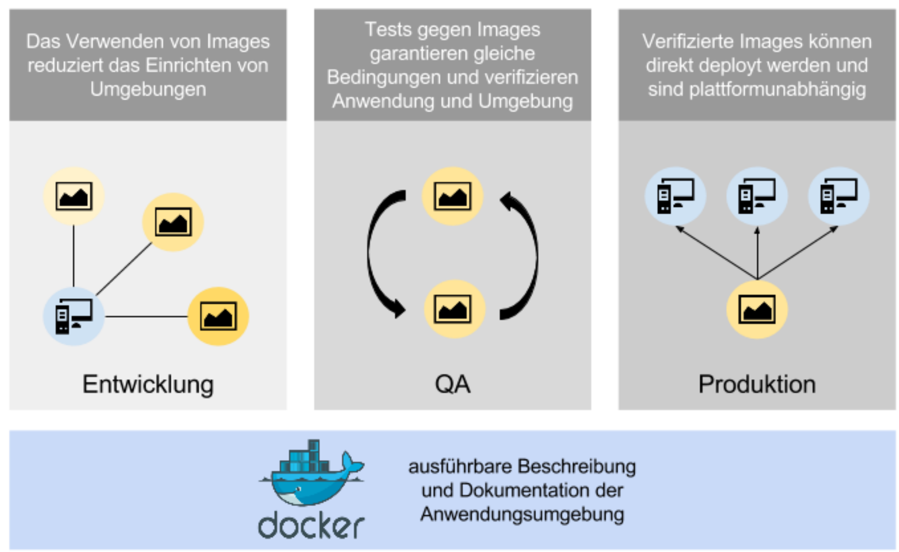

Abbildung 9.3.: Docker in Entwicklung, Tests und Produktion

[3]APIs selbst sind ebenfalls Blackboxen, da sie konkrete Implementierungen verstecken.

9.5. Elastic Beanstalk und Docker

Elastic Beanstalk ist das PaaS-Angebot von Amazon. Es existiert seit Januar 2011 und steht in Konkurrenz zu PaaS-Angeboten von *Heroku*, *Jelastic* oder *OpenShift*.

Mit Elastic Beanstalk lassen sich Anwendungen über verschiedene (grafische) Oberflächen[4] auf EC2 deployen. Zunächst wird hierzu eine Laufzeitumgebung gewählt[5] und anschließend das entsprechende Artefakt hochgeladen[6]. Anhand weiterer Konfigurationen (z.B. Einstellungen für einen Load Balancer) wird eine entsprechende virtuelle Maschine erzeugt und die Anwendung auf diese deployt.

Seit dem 23. April 2014 unterstützt Elastic Beanstalk *Docker* als Laufzeitumgebung (vgl. Barr, 2014). Damit können Docker Images (oder ZIP-Archive mit Ressourcen und einem Dockerfile) innerhalb weniger Schritte direkt deployt werden. Amazon übernimmt dabei unter Anderem die Erstellung der virtuellen Maschine, die Installation von Docker und das Starten der Anwendung. Dies ist das erste PaaS-Angebot dieser Art für Docker[7].

Elastic Beanstalk für Docker wurde in dieser Arbeit aus zweierlei Gründen nicht verwendet:

1. Die Unterstützung für Docker wurde erst nach Beginn der Arbeit eingeführt. Zu diesem Zeitpunkt war das entwickelte Deployment-Werkzeug weitestgehend fertiggestellt.
2. Bisher ermöglicht Elastic Beanstalk lediglich, ein einziges Docker Image je Instanz zu deployen. Dies entspricht nicht dem angestrebten Baukasten-Prinzip von Docker, in dem für jede Anwendung ein Image verwendet wird.

Daher macht die Einführung von Docker für Elastic Beanstalk das entwickelte Deployment-Tool nicht überflüssig, sondern bietet eine zusätzliche Möglichkeit, bereits bestehende Images zu verwenden. Es ist in Zukunft mit weiteren Angeboten dieser Art zu rechnen[8].

[4]Es existiert unter anderem eine Weboberfläche und ein Eclipse-Plugin.

[5]Momentan stehen *IIS*, *Node.js*, *PHP*, *Python*, *Ruby* und *Tomcat*, sowie *Docker* zur Verfügung.

[6]Bei einem Tomcat-Server wird etwa ein WAR-File hochgeladen.

[7]Zwar existieren mit *Deis* und *Flynn* (siehe Kapitel 10.1) PaaS-Implementierung für Docker, jedoch müssen diese selbst gehostet werden.

[8]Wie bereits erwähnt gibt es von Google bereits es eine testweise Unterstützung von Docker (mehr unter https://developers.google.com/compute/docs/containers). Außerdem ist Docker jüngst der Cloud Foundry Foundation beigetreten (vgl. Docker Inc., 2014d). Daher ist auch hier mit einer zukünftigen Unterstützung zu rechnen.

10. Deployment

10.1. Deployment-Werkzeuge

Für das Deployment und das Einrichten von Cloud-Instanzen gibt es eine wachsende Zahl von Softwarelösungen. Ähnlich wie Canonical/Ubuntu mit *Cloud-Init* oder Amazon mit *CloudFormation* (beide werden neben anderen Angeboten im Folgenden vorgestellt) platzieren viele Hersteller ihre eigenen Werkzeuge. Doch auch Drittanbieter bieten Umgebungen und Vorgehen zum sogenannten *Orchestrieren* an.

Tabelle 10.1 zeigt eine Gegenüberstellung von Werkzeugen für das Deployment. Wichtig ist, dass diese Werkzeuge teils völlig unterschiedliche Konzepte verfolgen: *Puppet* und *Chef* sind etwa Werkzeuge aus der IT-Administration, mit denen Nutzer, Anwendungen, Rechte und Einstellungen von Maschinen permanent überwacht und geändert werden können; *Deis* und *Flynn* sind PaaS-Angebote für Docker, die selbst gehostet werden können; *Vagrant* dient dazu, virtuelle Maschinen anhand von Skripten zu erzeugen; *CloudFormation* und *OpsWorks* sind AWS-spezifische Angebote, die wiederum eine Integration für Cloud-Init, Chef oder Puppet bieten. Alle diese unterschiedlichen Angebote haben jedoch gemein, dass mit ihnen eine Cloud-Instanz (mit Docker Images) eingerichtet werden kann. Sie unterscheiden sich jedoch substantiell in Ihrem Vorgehen.

Die umfangreichsten Lösungen stellen Puppet und Chef dar. Mit ihnen ist es möglich, eine komplette IT-Infrastruktur (unabhängig von Cloud-Plattformen) zu managen. Sie bieten Methoden für das Erstellen von Maschinen, Einspielen von Updates und Monitoring. Außerdem ermöglichen sie eine Integration zahlreicher anderen Tools wie *Vagrant* oder *Icinga* (einer Monitoring-Lösung). Diese Lösungen decken jedoch weit mehr ab, als im Rahmen dieser Arbeit benötigt wird.

Deis und *Flynn* sind junge Werkzeuge, die die Idee einer PaaS auf Basis von Docker umsetzen. Sie ermöglichen Code via *Git* automatisch in einem Container zu bauen und zu deployen. Das Angebot richtet sich an Entwickler, die neuen Code innerhalb weniger Minuten auf einer Cloud-Instanz sehen möchten (mehr zu diesem Vorgehen in Kapitel 5.1.6).

CloudFormation, OpsWorks und Beanstalk sind Amazon-spezifische Werkzeuge. Sie bieten zwar eine hohe Integration in AWS, sind jedoch nicht auf andere Plattformen übertragbar.

Im Zuge dieser Arbeit wurde *Cloud-Init* als Deployment-Tool gewählt. Es wird im folgenden Kapitel erklärt.

	Plattform	Agent	Domäne	Installation	Komplexität	Umfang	Syntax/Interface
Cloud-Init	Ubuntu, (Linux)	Nein	Boot-Script	Nein	gering	gering	Bash, Python, DSL
Puppet	Linux, (Windows)	Ja	Orchestration	Ja	hoch	sehr hoch	DSL, Web UI
Chef	Linux, AWS, Windows	Ja	Orchestration	Ja	hoch	sehr hoch	DSL, Console, Web UI
Ansible	Linux, AWS, OpenStack	Nein	Orchestration	Ja	hoch	hoch	DSL
AWS CloudFormation	AWS	Nein	Management	Nein	mittel	spezifisch	DSL, (Web UI)
AWS OpsWorks	AWS	Nein	Management	Nein	hoch	spezifisch	DSL, Web UI
Ubuntu Juju	AWS, Azure	Nein	Management	Ja	hoch	hoch	DSL, Web UI
AWS Elastic Beanstalk	AWS	Nein	PaaS	Nein	gering	spezifisch	Web UI
Deis	Docker (Linux)	Ja	PaaS	Ja	mittel	spezifische	Console, Git
Flynn	Docker (Linux)	Ja	PaaS	Ja	mittel	spezifisch	Console, Git
Vagrant	Linux, AWS	Nein	VMs	Ja	gering	gering	Ruby
Fabric	Linux	Nein	SSH	Ja	gering	mittel	Bash, Python
Capistrano	Linux	Nein	SSH	Ja	gering	mittel	Bash, Ruby

Tabelle 10.1.: Vergleich von Provisioning-Werkzeugen

10.2. Cloud-Init

Cloud-Init ist eine Linux-Software zur Initialisierung von Maschinen. Sie wird von *Scott Moser* als quelloffenes Projekt (GNU GPL v3) auf Launchpad[1] entwickelt und ist in Python implementiert[2]. Standardmäßig wird Cloud-Init auf allen *Ubuntu Cloud Images* auf AWS eingesetzt, ist jedoch auch auf Fedore, RHEL (Red Hat Enterprise Linux) und weiteren Linux-Distributionen verfügbar (siehe Canonical).

Amazon (aber auch Anbieter wie *OpenStack*) bieten die Möglichkeit, beim Erzeugen einer neuen Instanz sogenannte *User Data* zu übergeben. Dies sind Skripte, die während bzw. unmittelbar nach dem Hochfahren der Instanz von Cloud-Init ausgeführt werden. Cloud-Init unterstützt Skripte in Python, Bash oder einer eigenen DSL, wobei Skripte mit verschiedener Syntax gemischt werden können (ein sogenannter *Multipart Input*). Mittels dieser Skripte können z.B. Programme installiert, Nutzer angelegt oder Dateien kopiert werden, um die Instanz zu initialisieren.

Das anschließend vorgestellte Deployment-Werkzeug generiert Cloud-Init Skripte anhand von Konfigurationsdateien aus Vorlagen. Dies bedeutet, dass der Nutzer eine Konfigurationsdatei (im JSON-Format) an das Werkzeug übergibt und dieses das entsprechende Cloud-Init Skripte erzeugt. Das Skript wird an eine Maschine übergeben und initialisiert diese.

10.3. Entwicklung eines Deployment-Tools

Zur Automatisierung des Deployments wurde im Zuge dieser Arbeit ein Deployment-Werkzeug entwickelt. Das Werkzeug ermöglicht es anhand einer Konfigurationsdatei (im JSON-Format) neue EC2-Instanzen mit einer Reihe von Docker Images zu starten. Auch können Datenbanken erstellt und das Setup des PIM Servers darauf ausgeführt werden.

10.3.1. Implementierung

Das Deployment-Werkzeug wurde in *Java 8* implementiert. Zur Kommunikation mit AWS nutzt es die offizielle *AWS Java-API*, etwa um neue EC2-Instanzen zu erzeugen. Zum Monitoring des Deployments wird die SSH-Bibliothek *JSch* und die *Docker Java-API* eingesetzt. Das Einrichten der Maschinen wird über generierte *Cloud-Init* Skripte bewerkstelligt. Diese laden Images und Konfigurationen aus dem Cloud-Speicher S3.

Das Tool wird über ein Command Line Interface auf Basis von *Apache Commons CLI* und *Java ASCII Table* gesteuert. Außerdem steht mit *AngularJS* oder *Bootstrap CSS* (auf dem Client) sowie Java *Spark* (als REST-Server) ein eingeschränktes Webinterface zur Verfügung. Das Projekt wird mit *Maven* gebaut.

[1] Launchpad ist ein öffentliches Repository der Firma Canonical, vergleichbar mit *GitHub*, *Bitbucket* oder *SourceForge*.

[2] Das Projekt findet sich unter https://launchpad.net/cloud-init.

Konfigurationen für Instanzen und Datenbanken werden lokal als JSON-Dateien bzw. ANT-Skripte hinterlegt. Docker Images und Konfigurationen werden im Cloud-Speicher S3 gespeichert und aus diesem auf die Maschinen verteilt. Das Werkzeug selbst wird über Properties-Dateien konfiguriert.

10.3.2. Ablauf des Deployments

Der Deployment-Prozess des Tools gestaltet sich wie folgt:

1. Docker Images und Konfigurationen werden in Amazons Cloud-Speicher S3 abgelegt. Das Deployment-Tool biete hierfür eine Upload-Funktion sowie einen Befehl zur Synchronisation lokaler Dateien mit S3.
2. In einem Konfigurationsfile (`*.json`) werden die EC2-Instanz, die Docker Images und deren Konfiguration beschrieben. Jedes Konfigurationsfile (*Formation* genannt) dient zur Definition einer Maschine.
3. Das Deployment-Werkzeug erstellt mit Hilfe der AWS Java-API eine entsprechende EC2-Instanz. Aus den Informationen über die Docker Images und deren Konfiguration wird ein Cloud-Init Skript generiert. Dieses wird der Instanz als *User Data* übergeben und richtet die Maschine ein.
4. Nach dem Hochfahren führt Cloud-Init das mitgegebene Skript aus:
   a) Das Skript installiert das AWS Command Line Interface (AWSCLI) zum Zugriff auf S3, womit die Images und Konfiguration heruntergeladen werden.
   b) Das Skript installiert Docker zur Ausführung der Images und öffnet die REST-API von Docker für den Zugriff von außen.
   c) Das Skript lädt die angegebenen Docker Images aus S3.
   d) Das Skript lädt die angegebenen Konfigurationen aus S3.
   e) Das Skript startet die Container und bindet die zugehörige Konfiguration ein.

Das Ausführen dieses Setups dauert ca. 8 bis 15 Minuten für das Deployment von drei Images mit insgesamt ca. 1,7 GB. Danach ist die Maschine mit der Anwendung online. Auf die Installation von Docker und dem AWS CLI kann verzichtet werden, wenn ein entsprechend vorkonfiguriertes VM Image verwendet wird.

Der Status des Deployments kann mit dem Tool nachvollzogen werden. So lässt sich einsehen, welche Schritte gerade ausgeführt werden, wann die Maschine „fertig“ ist und welche Container auf welchen Ports laufen.

10.4. Logging, Monitoring und Konfigurationsmanagement

Bei der Arbeit mit mehreren verteilten Maschinen ist der Umgang mit Log- und Konfigurationsdateien wichtig. Schon bei einem einfachen Setup in der Cloud fallen unterschiedlichste Logdateien und Konfigurationen an, die sich auf mehrere Rechner verteilen. Diese effektiv zu handhaben ist essentiell für einen erfolgreichen Betrieb in der Cloud.

Step	Status
Booting	DONE
Initialization	DONE
Download images/kibana.tar	DONE
Download configs/Logstash_Config	DONE
Load images/kibana.tar	INPROGRESS
Start images/kibana.tar	PENDING

Abbildung 10.1.: Übersicht der Deployment-Schritt im Web-Interface

10.4.1. Logging

Der Informatica PIM Server schreibt je nach Konfiguration mehrere Logdateien, standardmäßig drei. Hinzukommen Logdateien des Betriebssystems und angegliederter Services (etwa des optionalen Zuliefererportals von Informatica). All diese Dateien haben ein unterschiedliches Format und werden in anderen Ordnern hinterlegt. Um den Betrieb in der Cloud zu überwachen, wurde daher eine Möglichkeit gesucht, die anfallenden Logs zentral zu aggregieren.

Als Lösung dieses Problems wird *Logstash*[3] vorgeschlagen. Logstash ist eine Anwendung zum Sammeln und Aggregieren von Logdateien. Es wird auf einem zentralen Server aufgesetzt und kann Log-Informationen empfangen (*Inputs*), verarbeiten (mittels *Codecs* und *Filter*) und weiterleiten (*Outputs*). Logstash unterstützt dabei eine Vielzahl von Inputs (z.B. *Files*, *Log4J* oder *Websockets*) und Outputs (z.B. *Files*, *S3* oder *E-Mail*) sowie verschiedene Filter und Codecs.

Ein häufig verwendeter Output ist *Elasticsearch*, eine Suchmaschine für Dokumente[4]. Ein typisches Setup aggregiert Logs in Logstash und reicht sie zur Indizierung an Elasticsearch weiter. Elasticsearch ermöglicht dann eine umfangreiche Volltextsuche auf diesen aggregierten Logs.

Als Interface für Elasticsearch kommt Kibana, eine HTML/JavaScript Oberfläche, zum Einsatz. Dieses bietet verschiedene Sichten auf die Logs und ermöglicht, diese zu durchsuchen. Abbildung 10.2 zeigt Kibana mit Logs des PIM Servers.

Um Logdateien an Logstash zu senden, gibt es mehrere Möglichkeiten. So kann ein Logstash etwa Logs an anderen Server weiterleiten oder es kann ein spezieller Agent genutzt werden. Für diese Arbeit wurde der offizielle *Logstash-Forwarder*[5], ein etwa 5 MB großes *Go*[6] Programm, verwendet. Dieses wird auf den Maschinen installiert und kann mehrere Dateien oder Ordner überwachen. Ändert sich eine Datei, so wird diese an den hinterlegten Logstash-Server gesendet.

[3] Logstash findet sich auf GitHub unter https://github.com/elasticsearch/logstash.

[4] Logstash, Elasticsearch und Kibana werden als quelloffene Projekte von der Firma *Elasticsearch* entwickelt und finden sich auf GitHub.

[5] Er findet sich ebenfalls im GitHub-Repository von Elasticsearch unter https://github.com/elasticsearch/logstash-forwarder.

[6] *Go* ist eines systemnahe, typisierte Sprache von Google. Sie wird auch in Docker verwendet.

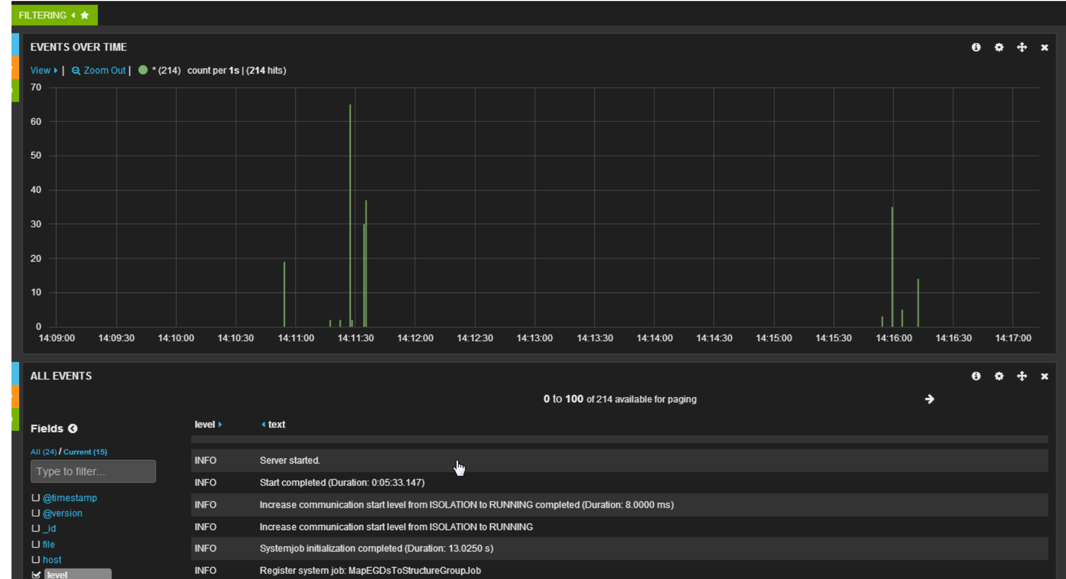

Abbildung 10.2.: Kibana mit Log-Information des PIM Servers

Durch die Kombination von Logstash, Elasticsearch, Kibana und dem Logstash-Forwarder werden alle Logs zentral gesammelt und aufbereitet. Logs von verschiedenen Maschinen sind über eine Weboberfläche zugänglich und können durchsucht und gefiltert werden. Außerdem können Logs auch dann noch abgerufen werden, wenn die eigentliche Maschine, von denen die Logs stammen, abgeschaltet wurde (was in der Cloud häufiger der Fall sein kann). Die Lösung ist generisch und bedarf keiner Änderung an bestehendem Code oder Logging-Einstellungen.

Die Umsetzung mit Docker funktioniert wie folgt: Logstash, Elasticsearch und Kibana werden als Docker Images gepackt und auf einen zentralen Server deployt. Mit jedem Node wird ein Docker Image des Logstash-Forwarders deployt. Beim Start des Logstash-Forwarders werden die zu überwachenden Ordner in den Container eingebunden. Umgekehrt exportieren andere Docker Container ihre Ordner mit Logdateien nach außen.

Auf der beiliegenden CD findet sich eine detaillierte Beschreibung zum Aufsetzen von Logstash und zum Parsen der Logdateien.

10.4.2. Monitoring

Neben Logdateien müssen auch Systemmetriken der Cloud-Instanzen überwacht werden. Zu solchen Metriken zählen etwa die CPU-Auslastung oder der verfügbare Festplattenspeicher.

Zum Überwachen solcher Metriken sind Docker Images nur bedingt geeignet. Da Docker Anwendungen in isolierten Sandboxen laufen, haben sie keine Zugriff auf globale Metriken des Systems. Daher können keine Systemmonitore als Docker Images deployt

werden - sie könnten lediglich ihren eigenen Container überwachen. Docker bietet jedoch grundlegende Metriken über seine Remote API an. So können alle laufenden Container und deren Prozesse gelistet werden sowie statische Metriken (wie der maximale CPU-Anteil eines Containers) abgefragt werden. Diese Metriken eignen sich für ein rudimentäres Monitoring, etwa um zu sehen, welche Anwendungen gerade laufen.

Für weitere Metriken kann auf Services des Cloud-Anbieters (hier Amazon) zurückgegriffen werden. Diese bieten meist ein umfangreiches Monitoring für ihre Instanzen über Weboberflächen an. Amazon offeriert hierzu *CloudWatch*, einen Service zur Überwachung von virtuellen Maschinen, Datenbanken und Netzwerken auf Amazons Cloud-Infrastruktur. Über CloudWatch lassen sich Metriken wie CPU- oder Speicherauslastung leicht nachvollziehen. Der Service kann während des Betriebs für beliebige Maschinen ein- oder ausgeschaltet werden.

Algorithmus 10.1 Metriken über die Docker API

```
# zeigt alle laufenden Container
docker ps

# zeigt alle laufenden Prozesse eines Containers
docker top <container id>

# zeigt statische Metriken des Containers
docker inspect <container id>
```

10.4.3. Konfigurationsmanagement

Anders als beim Logging und Monitoring wurde das Konfigurationsmanagement aufgrund der bestehenden Software und ihrer Voraussetzungen weitgehend selbst implementiert.

Die Konfiguration von Docker Containern kann auf drei Arten erfolgen: (1) Das Image kann eine (Default-) Konfiguration enthalten, (2) dem Container können beim Start Umgebungsvariablen übergeben werden oder (3) Konfigurationsdateien können als externe Ordner in den Container eingebunden werden. Die erste Variante macht Images einfach zu deployen, gestaltet sich jedoch im weiteren Verlauf unflexibel, da die Konfiguration statisch ist. Das Mitgeben von Umgebungsvariablen bietet dagegen wesentlich mehr Flexibilität. Jedoch besteht die Einschränkung, dass die Anwendung im Container, Einstellungen über diesen Weg erwarten muss und dass aufwendige Konfigurationen (etwa über XML oder JSON) nicht möglich sind. In der letzten Variante wird die Konfiguration komplett vom Docker Image getrennt. Alle Konfigurationen liegen in externen Ordnern und werden beim Start des Containers eingebunden.

Im Zuge dieser Arbeit wurde letztere Variante gewählt. Alle Konfigurationen werden in einem Cloud-Speicher (Amazons S3) gehalten. Wird eine neue Instanz gestartet, so wird die entsprechende Konfiguration auf die Maschine kopiert und beim Start in den Docker Container eingebunden.

Um die Konfigurationen zu aktualisieren, synchronisieren die Instanzen die heruntergeladenen Ordner regelmäßig mit dem Cloud-Speicher. Dies wird über das *AWS Command*

Line Interface und dessen *sync*-Befehl sowie einen *Cronjob* erreicht. So kann ein Update von einem lokalen Rechner in den Cloud-Speicher geladen und von dort von den einzelnen Maschinen „gepullt“ werden[7].

Der wesentliche Unterschied dieser Lösung zu Werkzeugen wie *Puppet* oder *Chef* liegt darin, dass Maschinen Aktualisierungen *pullen*, anstatt dass diese *gepusht* werden. Dank der Amazon-Plattform lässt sich dies ohne weitere Werkzeuge bewerkstelligen. Dieses Konfigurationsmanagement ist sehr einfach, lässt es aber beispielsweise nicht zu, Aktualisierungen gezielt auf bestimmten Maschinen einzuspielen. Aufgrund der Eigenschaft von Cloud-Plattformen, Instanzen bei Bedarf erzeugen zu können, kann stattdessen eine neue Instanz gestartet werden, anstatt bestehende Maschinen zu aktualisieren.

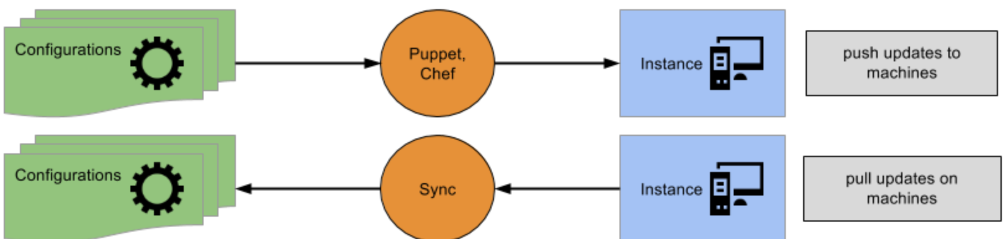

Abbildung 10.3.: Push Updates vs. Pull Updates

[7]Die Konfiguration liegt also in einer Art „Dropbox“, was durchaus wörtlich verstanden werden kann, da Dropbox für seinen Dienst selbst Amazons S3 nutzt.

Teil V.

Diskussion und Fazit

11. Bewertung der Ergebnisse

Die im Rahmen dieser Arbeit entwickelte Strategie hat gezeigt, dass der Informatica PIM Server (überwiegend) automatisiert in die Cloud deployt werden kann. Cloud-Services wie Datenbanken können mit wenigen Anpassungen genutzt werden.

Der Einsatz von Docker als Anwendungscontainer vereinfacht das Deplyoment deutlich. Heterogene Anwendungen können mit einem einheitlichen Vorgehen gepackt, verteilt und gestartet werden. Dies ermöglicht ein Baukastenprinzip, mithilfe dessen Anwendungsstacks aus vorgefertigten Containern zusammengesetzt werden. Das Einbinden von Ordnern in die Container schafft eine Trennung von Anwendung und Konfiguration. Da Docker nicht an eine Plattform gebunden ist, werden Abhängigkeiten vermieden und Images können auch lokal für Entwicklung und Tests genutzt werden (siehe Kapitel 9.4).

Abbildung 11.1 zeigt die Einordnung der entwickelten Lösung in das Cloud-Modell von Haan [2013].

Layer 6	SaaS	Applications			*End-users*
Layer 5	App Services	Compute App Services	Communicate App Services	Store App Services	*Citizen developers*
Layer 4	Model-Driven PaaS	bpmPaaS, Model-Driven aPaaS	Model-Driven iPaaS	baPaaS	*Business engineers*
Layer 3	PaaS	Docker Index	iPaaS	AWS RDS	*Professional developers*
Layer 2	Foundational PaaS	Docker	AWS Elastic	AWS S3	*DevOps*
Layer 1	Software Defined Datacenter	AWS EC2	Software Defined Networking (SDN)	Software Defined Storage (SDS)	*Infrastructure engineers*
		Compute	Communicate	Store	

Abbildung 11.1.: Einordnung der entwickelten Lösung in das Modell von Haan, 2013

Aufgrund der Ausgangslage des Informatica PIM Servers waren andere Deployment-Methoden hingegen nicht umsetzbar. Die vielversprechende Methode, den Server via OSGi-Bundles zu verteilen, ist bedingt durch Architektur und Abhängigkeiten nicht möglich. Bestehende PaaS-Angebote scheiden sowohl durch fehlende Unterstützung für OSGi, als auch durch geringen Einsatz von Standardtechnologien im PIM Server aus.

12. Hindernisse und Probleme

Bei der Portierung des Informatica PIM Servers in die Cloud haben sich einige Hindernisse bzw. Probleme herauskristallisiert. Diese sollten in Zukunft für eine erfolgreiche Portierung in die Cloud gelöst werden. Angefangen mit den gravierendsten, sind die Probleme und ihre Auswirkungen im Folgenden beschrieben.

- Zwar ist der Informatica PIM Server (wie die meisten ihm angegliederten Applikationen) eine reine Java-Anwendung, jedoch wird er ausschließlich für Windows entwickelt und getestet. Cloud-Plattformen (und Werkzeuge wie *Docker*) basieren jedoch hauptsächlich auf Linux-Distributionen[1].
- Das eigens entwickelte Kommunikationsprotokoll zwischen Client und Server ist bislang unverschlüsselt. Wird der Server in der Cloud betrieben, muss dieses Protokoll unbedingt gesichert werden.
- Dem Datenbankschema liegt ein großes und anfälliges Setup zugrunde (siehe Kapitel 7). Das Aufsetzen des Datenbankschemas sollte überdacht werden.
- Der PIM Server hat eine umfangreiche und monolithische Code-Basis. Zwar besteht der Server aus vielen logischen Komponenten (Import, Export, Load-Balancer, User-Verwaltung, Data-Quality, etc.), die in OSGi-Bundles gepackt sind, jedoch ist er nicht modular (siehe Kapitel 2.3). Dies erschwert die Portierung, da der Server nicht in einzelne Pakete heruntergebrochen werden kann. Eine losere Kopplung der Komponenten würde die Portierung in die Cloud oder auf ein anderes Betriebssystem erleichtern.
- Viele Komponenten im PIM Server sind selbst entwickelt. Um die Anbindung an Cloud-Services zu vereinfachen, sollte auf Standardlösungen gesetzt werden.

[1] Nach meinem aktuellen Kenntnisstand bieten lediglich Amazon und Microsoft Cloud-Lösungen auf Basis von Windows an. Dies sind nur zwei von mehreren Dutzend Anbietern. Leider gibt es hierzu bislang keine Erhebungen.

13. Ausblicke und nächste Schritte

Neben der Adressierung der in Kapitel 12 aufgeführten Probleme gibt es weitere Anknüpfungspunkte zur Fortsetzung des Themas. Diese werden im Folgenden beschrieben. Es wird davon ausgegangen, dass Docker längerfristig als Plattform genutzt werden soll.

- Für das Erstellen der Docker Images sollte ein Build-Job erstellt werden. So könnten Docker Images analog zu anderen Build-Artefakten (wie etwa ZIP-Files) automatisch gebaut werden. Zudem sollte eine Strategie zur zentralen Verwaltung der Images und Konfigurationen gefunden werden. Insbesondere sollte eine gesicherte Docker-Registry anstelle eines S3-Buckets aufgesetzt werden.

- Amazon und Docker bilden eine ideale Grundlage für eine automatische horizontale Skalierung des PIM Servers. Es sollte evaluiert werden, inwieweit der Server in der Cloud automatisch skaliert werden kann. Dies könnte im Zuge der aktuellen Entwicklung des Multi Server Supports geschehen.

- Neben AWS RDS sollte die Integration weiterer Cloud-Services evaluiert werden. Insbesondere ist S3 als Cloud-Speicher für Dateien und SES (Amazon Simple Email Service) als Service für das Verschicken von E-Mails interessant.

- Ein Vorteil von Docker Images ist deren gute Integration in Entwicklungsprozesse. Sie eignen sich als Grundlage für Team-übergreifende Entwicklung und als Testeinheit in der Qualitätssicherung (siehe Kapitel 9.4). Daher sollten Entwicklungs- und Testprozesse etabliert werden, in denen Docker Images genutzt werden.

- Ende April 2014 erweiterte Amazon sein Cloud-Angebot *Elastic Beanstalk* um die Möglichkeit, Docker Images zu deployen (vgl. Barr, 2014). Diese ist aufgrund des Veröffentlichungsdatums nicht mehr in diese Arbeit eingeflossen. Jedoch könnte diese Methode eine einfache Möglichkeit darstellen, einzelne Images zu deployen und sollte daher evaluiert werden (sieh Kapitel 9.5).

- Bislang wurde keine zufriedenstellende Methode für das Konfigurationsmanagement gefunden. Im Zuge dieser Arbeit wurden alle Konfigurationsdateien in einem Cloud-Speicher von Hand verwaltet. Diese Lösung ist zwar praktikabel, für eine dauerhafte Nutzung jedoch unbefriedigend.

- Für das Erstellen von Cloud-Formationen ist ein visueller (Web-basierter) Editor denkbar. Er könnte Images und Konfiguration auflisten und die Freigabe von Ports sowie das Einbinden von Ordnern in die Container ermöglichen. Durch den Einsatz eines Editors würde das Anlegen von Anwendungsstacks so vereinfacht.

- Das entwickelte Deployment-Werkzeug könnte für weitere Plattformen (einschließlich lokaler Linux VMs) erweitert werden und so auch für das interne Aufsetzen von Systemen dienen. Der bestehende Code müsste hierzu nur wenig verändert werden, da sowohl SSH-Schnittstellen als auch Shell-Skripte bereits existieren.

Teil VI.

Anhang

Abhängigkeit von OSGi Bundles im PIM Server

Der Informatica PIM Server baut auf OSGi, einem Framework für modulare Java-Anwendungen, auf. OSGi separiert Komponenten in *Bundles*, die zu *Features* und *Anwendungen* zusammengefasst werden können. Jedes Bundle definiert dabei seine Abhängigkeiten, was es möglich macht Bundles zur Laufzeit aufzulösen und zu installieren. Jedoch garantiert der Einsatz von OSGi allein keine Modularität, was im folgenden Beispiel gezeigt wird.

Fallstudie: Die Data-Quality-Komponente

Die Data-Quality-Komponente (kurz *DQ*) ist ein neues Feature des Informatica PIM Servers, welches im vergangenen Jahr implementiert wurde. Sie verbindet den Server mit einer bestehenden Datenaufbereitungsumgebung von Informatica und teilt sich in zwei serverseitige OSGi-Bundles auf[1]. Dies allein macht die Komponente jedoch nicht modular: Werden die zwei Bundles der Komponente entfernt, so wirkt sich dies auf mehr als 90 andere Bundles aus - entweder direkt oder über transitive Abhängigkeiten. Dies entspricht rund einem Viertel der insgesamt 378 Bundles. Diese ca. 90 Bundles können durch das OSGi-Framework nicht mehr gestartet werden, da ihre Abhängigkeiten nicht erfüllt sind. Trotz allem lässt sich der Server weiterhin starten. Beim Hochfahren (also zur *Laufzeit*) führen die fehlenden Bundles allerdings zu Fehlern, durch die der Server letztlich „abstürzt“, sich also ohne Fehlerbehandlung beendet (eine Log-Datei eines solchen Vorgangs findet sich auf der beigefügten CD).

OSGi soll solche „Abstürze zur Laufzeit“ eigentlich durch das Auflösen von Abhängigkeiten verhindern. Wären die OSGi-Bundles (im Sinne von OSGi) richtig geschrieben, dürfte der Server nicht versuchen hochzufahren, da nicht alle Abhängigkeiten erfüllt sind.

OSGi versucht so `ClassNotFoundExceptions` (zur Laufzeit) zu verhindern. Jedoch führen genau solche Ausnahmen zum Absturz des Servers. Dies ist auf eine handgeschriebene Klassen-Registry[2] zurückzuführen, in der Klassen registriert und zur Laufzeit mit ihrem Namen instantiiert werden können. Solche Techniken umgehen die Möglichkeiten von OSGi (und die des Java-Compilers), Abhängigkeiten aufzulösen. Meta-Programmierung dieser Art soll die Flexibilität der Anwendung erhöhen, verhindert in diesem Fall aber Modularität.

[1] `com.heiler.ppm.dataquality.server` und `com.heiler.ppm.dataquality.core`
[2] `com.heiler.ppm.classregistry.core.ClassRegistry`

Bewertung von Deployment-Methoden für die Cloud

Im Folgenden finden sich die detaillierten Bewertungen der fünf Deployment-Methoden *VM Images*, *manuelle Installation*, *Docker*, *JEE Archive*, *OSGi Bundles* und *Source Code* aus Kapitel 5. Zur besseren Übersicht sind die Bewertungskriterien noch einmal in Tabelle 13.1 dargestellt. In Tabelle 13.2 findet sich eine Zusammenfassung aller fünf Deployment-Methoden. *Docker* ist als gewählte Methode in dieser Arbeit hervorgehoben.

Kriterium	Beschreibung
Granularität	Eine Methode oder Plattform gilt als granulär, wenn sie es erlaubt, einzelne Teile einer Anwendung (z.B. OSGi-Bundles) zur Laufzeit zu deployen und zu aktualisieren.
Unabhängigkeit	Eine Plattform oder Methode ist unabhängig, wenn sie nicht an einen konkreten Anbieter oder eine Implementierung gebunden ist.
Kapselung	Eine Methode wird als gekapselt betrachtet, wenn für das Deployment keine detaillierten Kenntnisse über die Anwendung oder Plattform notwendig sind.
Komplexität	Ein Vorgehen wird als komplex betrachtet, wenn es sehr aufwändig in seiner Umsetzung ist, etwa durch das Einrichten von Servern oder Schreiben von Skripten.
Leistungsfähigkeit	Eine Plattform/Methode ist leistungsfähig, wenn mit ihr heterogene Anwendungen deployt werden können und sie nicht an eine spezielle Technologie (etwa Java oder OSGi) gebunden ist.
Level	Eine Plattform/Methode wird einem oder mehreren der drei Cloud-Level (IaaS, PaaS oder SaaS) zugeordnet.

Tabelle 13.1.: Kriterien zur Bewertung von Deployment-Strategien

	Granularität	Unabhängigkeit	Kapselung	Komplexität	Lstg.-fähigkeit	Level
VM Images	*	***	**	***	***	IaaS
Manuelle Installation	*	***	*	***	***	IaaS
Docker	**	***	**	**	***	**IaaS/PaaS**
J2EE Archive	*	**	**	**	**	IaaS/PaaS
OSGi-Bundles	**	**	*	***	**	IaaS/PaaS
Source Code	***	*	***	**	*	PaaS

Tabelle 13.2.: Bewertung von Deployment-Methoden

Deployment von VM Images

Granularität	Unabhängigkeit	Kapselung	Komplexität	Lstg.-fähigkeit	Level
*	***	*	**	***	IaaS

Granularität - Die Methode ist nicht granulär, da die Applikation und ihre Abhängigkeiten in ein abgeschlossenes Image gepackt werden. Um Teile der Anwendung zu aktualisieren, muss ein neues Image erstellt oder die Applikation manuell auf der virtuellen Maschine geändert werden.

Unabhängigkeit - Da VM Images die gesamte Applikation und alle Abhängigkeiten enthalten, sind sie unabhängig von der Zielplattform. Sie können auf verschiedene Cloud-Anbieter, eigenen Server oder Entwicklermaschinen deployt werden, solange ein kompatibler Supervisor verfügbar ist.

Kapselung - Ein VM Image kapselt zwar alle Abhängigkeiten und Konfigurationen einer Anwendung, jedoch ist zur Erstellung detailliertes Wissen über die Applikation, deren Abhängigkeiten und Konfigurationen notwendig. Auch kapselt ein Image unter Umständen nicht die Konfiguration der Anwendung, da diese auf der jeweiligen Maschine vorgenommen werden muss.

Komplexität - Das Aufsetzen eines VM Images ist nicht trivial - Betriebssystem, Anwendung und Abhängigkeiten müssen installiert und konfiguriert werden. Zudem muss das Image kompatibel zu den Vorgaben/Restriktionen der Zielplattform sein. Das Aufsetzen eines VM Images kann jedoch automatisiert werden und unterscheidet sich nicht vom Aufsetzen eines normalen Produktionssystems.

Leistungsfähigkeit - Images für virtuelle Maschinen können für alle Arten von Anwendungen genutzt werden, sowohl auf Windows als auch auf Linux.

Level - Um VM Images zu betreiben, wird lediglich eine kompatible Infrastruktur in der Cloud benötigt. VM Images lassen sich daher mit minimaler Anbieterunterstützung auf einer IaaS verwenden.

Deployment mit einer manuellen Installation

Granularität	Unabhängigkeit	Kapselung	Komplexität	Lstg.-fähigkeit	Level
*	***	*	***	***	IaaS

Granularität - Die Methode ist nicht granulär. Die Anwendung wird mit ihren Abhängigkeiten auf einer Maschine installiert. Sollen Teile der Anwendung aktualisiert werden, bietet die Methode selbst keine Unterstützung. Im schlimmsten Fall muss die Anwendung (oder Maschine) neu aufgesetzt werden.

Unabhängigkeit - Jede Anwendung bietet eine eigene manuelle Installation auf dem Zielsystem. Insofern die Anwendung mit dem Zielsystem kompatibel ist, kann sie dort auch von Hand eingerichtet werden.

Kapselung - Die manuelle Installation kapselt den Deployment-Prozess nicht. Es ist notwendig, sich mit einer (Cloud-) Maschine (etwa über SSH unter Linux oder RDP unter Windows) zu verbinden, um dort die Installation durchzuführen. Je nach Anwendung ist dieser Schritt mehr oder weniger komplex. Die manuelle Installation ist per definitionem nicht automatisiert.

Komplexität - Die manuelle Installation einer Anwendung gestaltet sich auf einer Cloud-Maschine analog zu einer lokalen (physischen) Maschine. Sie hat nur wenig mehr Komplexität als ohnehin nötig wäre, die Anwendung zu starten (etwa für die Verbindung zur entfernten Maschine). Die Methode bietet jedoch keine Reduktion der Komplexität.

Leistungsfähigkeit - Ist die Cloud-Plattform kompatibel zur Anwendung, so kann diese auf der Plattform manuell (mit mehr oder minder großem Aufwand) eingerichtet werden.

Level - Eine manuelle Installation setzt immer auf einer IaaS auf. Solange der Anbieter eine hinreichende Plattform zur Verfügung stellt, kann sich zu dieser verbunden und dort die Anwendung eingerichtet werden. Hierzu ist keine besondere Unterstützung des Anbieters (etwa durch Deployment-Werkzeuge) nötig.

Deployment mit Docker

Granularität	Unabhängigkeit	Kapselung	Komplexität	Lstg.-fähigkeit	Level
**	***	**	**	***	IaaS/PaaS

Granularität - Ähnlich wie VM Images enthalten Docker Images die komplette Applikation und deren Abhängigkeiten. Um Teile der Anwendung zu aktualisieren, muss ein neues Image erstellt oder der laufende Container manuell geändert werden. Anwendungsstacks könne jedoch in separate Images heruntergebrochen werden (etwa für Backend, Webserver, Cache, Webinterface, usw.). Durch das Schichtmodel von Docker können Aktualisierungen und Rollbacks schnell an bestehen Images durchgeführt werden. Daher ist diese Lösung granulärer als VM Images.

Unabhängigkeit - Docker Images sind abgeschlossen und portabel. Sie können überall gestartet werden, wo Docker verfügbar ist. Sie sind damit nicht an einen speziellen Anbieter oder die Cloud im Allgemeinen gebunden. Es ist außerdem nicht nötig, die Anwendungen speziell für Docker anzupassen, weshalb ein Lock-in-Effekt ausgeschlossen ist.

Kapselung - Alle Abhängigkeiten und Konfigurationen einer Anwendung werden in einem Docker Image gekapselt. Zur Erstellung der Images ist also Wissen über die Applikation sowie deren Abhängigkeiten und Konfigurationen notwendig. Jedoch bietet Docker mit den sogenannten *Dockerfiles* ein Vorgehen, um den Bau von Images zu automatisieren.

Komplexität - Das Einrichten von Docker Images ist vergleichbar mit dem Einrichten einer virtuellen Maschine. Da Docker jedoch auf einer bestehenden Linux-Maschine aufsetzt, entfallen viele Schritte (wie etwa die Auswahl des Betriebssystems oder das Einrichten von Netzwerken). Außerdem bietet Docker Werkzeugunterstützung (*Dockerfiles)* zum Bau von Images.

Leistungsfähigkeit - Mit Docker können alle Anwendungen gepackt und deployt werden, die auf Linux laufen.

Level - Das Einrichten von Docker Images bewegt sich auf einem IaaS-Level. Durch die hohe Portabilität, die REST-Schnittstelle und Angebote wie dem Docker Hub (aus dem fertige Images geladen werden können) hat Docker selbst jedoch auch PaaS-Komponenten. Zudem gibt es erste PaaS-Angebote für Docker, wie etwa Amazon Elastic Beanstalk, mit denen Docker Images direkt deployt werden können.

Deployment von JEE Archiven

Granularität	Unabhängigkeit	Kapselung	Komplexität	Lstg.-fähigkeit	Level
*	**	**	**	**	IaaS/PaaS

Granularität - Die Methode ist nicht granulär, da die Applikation und ihre Abhängigkeiten in ein abgeschlossenes Archiv gepackt werden. Um Teile der Applikation zu aktualisieren, muss ein neues Archiv erstellt und dieses in einen JEE-Container deployt werden.

Unabhängigkeit - Ein JEE-Archiv ist teilweise unabhängig, da es die gesamte Applikation und deren Abhängigkeiten enthält. Es ist jedoch auf einen JEE-Container angewiesen, meist auf eine konkrete Implementierung. Ein JEE-Archiv kann außerdem weitere Abhängigkeiten haben (z.B. zum Betriebssystem oder zu einem Cache), die nicht gewährleistet werden können.

Kapselung - Ein JEE-Archiv kapselt alle Abhängigkeiten zu Bibliotheken einer Anwendung. Abhängigkeiten zu Middleware oder Betriebssystemen können jedoch nicht abgedeckt werden, wie dies etwa mit VM Images oder Docker möglich wäre.

Komplexität - Ein JEE-Archiv ist ein ZIP-File, welches Konventionen entsprechen muss, um von einem JEE-Container ausgeführt werden zu können. Im Zuge dieser Arbeit hat sich gezeigt, dass es nicht trivial ist, eine bestehende Anwendung in einen JEE-Container zu portieren. Es muss auf zahlreiche Details der konkreten Anwendung und des Containers geachtet werden, da sich diese Art von Containern auf dem Level von Programmiersprachen und Bibliotheken bewegen (VM Images und Docker sind hingegen unabhängig von konkreten Sprachen).

Leistungsfähigkeit - Beinahe jede Java-Anwendung kann in ein JEE-Archiv gepackt werden. Für (Legacy-) Anwendungen, die jedoch nicht für die Ausführung in einem JEE-Container konzipiert sind, können Anpassungen (z.B. um eine Verbindung zum Container herzustellen) notwendig sein. Verglichen mit generischen Containern wie Docker ist diese Lösung daher weniger leistungsfähig.

Level - Das Bereitstellen von Containern bewegt sich auf der Infrastrukturebene. Einige Anbieter offerieren jedoch PaaS-Lösungen mit vorinstallierten Containern.

Deplyoment von OSGi Bundles

Granularität	Unabhängigkeit	Kapselung	Komplexität	Lstg.-fähigkeit	Level
**	**	*	***	**	IaaS/PaaS

Granularität - Die Methode ist auf dem Level von Bundles granulär, d.h. es ist möglich, einzelne Bundles zur Laufzeit auszutauschen. Es können jedoch keine kleineren Einheiten als Bundles getauscht werden. Außerdem führt ein Austauschen von Bundles zur Laufzeit immer zu einem temporären Ausfall von Funktionalität (das Bundle wird gestoppt und ist kurzzeitig nicht verfügbar). Durch Abhängigkeiten zwischen Bundles kann dieser Ausfall zahlreiche weitere Bundles betreffen[3].

Unabhängigkeit - Die Methode hängt von der Verwendung von OSGi und der Applikation selbst ab. Nur wenn die Applikation *modular* ist (siehe hierzu Kapitel 2.2), können Bundles dynamisch getauscht werden. Einzelne Bundles können jedoch externe Abhängigkeiten haben, die durch diese Methode nicht garantiert werden können.

Kapselung - Die Methode kapselt Schritte wie das Hochladen und Starten von Bundles. Es muss jedoch ein Bundle-Repository mit (semantisch-versionierten) Bundles gepflegt werden und initial ein OSGi-Framework mit einer Managementkomponente aufgesetzt werden, in welches die Bundles installiert werden. Außerdem sollten die Abhängigkeiten von Bundles und die Funktionsweise von OSGi bekannt sein.

Komplexität - Die Methode setzt ein tiefes Verständnis für OSGi und die Applikation selbst voraus. Es muss bekannt sein, welche Bundles benötigt werden und wie diese voneinander abhängen. Ein fehlerhaftes (oder fehlendes) Bundle kann Auswirkungen auf zahlreiche andere haben und selbst wenn einzelne Bundles gestartet werden können, gibt dies keine Garantie für die Funktion des Gesamtsystems.

Leistungsfähigkeit - Mit der Methode kann jede *modulare* OSGi-Anwendung verteilt werden. Jedoch setzt sie Deployment-Werkzeuge (z.B. Apache ACE oder Virgo, siehe Kapitel 5.3) voraus, die in der Praxis Einschränkungen aufweisen (Apache ACE und Virgo kennen etwa keine Equinox-spezifischen Techniken wie *Features* oder *Products*). Ist die Anwendung nicht für ein modulares Deployment konzipiert, scheidet diese Methode aus oder reduziert sich auf ein einmaliges Laden und Starten von Bundles.

Level - Die Bereitstellung und Administration einer entsprechenden Umgebung läuft auf dem IaaS-Level. Ist eine Umgebung eingerichtet oder von einem Anbieter bereitgestellt, können Bundles etwa über Weboberflächen deployt werden. Dies entspricht einer PaaS.

[3] Wird ein Bundle gestoppt, so werden auch transitiv alle anderen Bundles gestoppt, die von diesem Bundle abhängen.

Deployment von Source Code

Granularität	Unabhängigkeit	Kapselung	Komplexität	Lstg.-fähigkeit	Level
***	*	***	**	**	PaaS

Granularität - Die Methode erlaubt die Aktualisierung von Anwendungen auf der Ebene von Code. Dies bedeutet, dass etwa nur einzelne Änderungen in Klassen oder Methoden in die Cloud deployt werden können. Die eigentliche Aktualisierung wird hierbei vom Anbieter verborgen.

Unabhängigkeit - Die Methode hängt stark vom jeweiligen Cloud-Anbieter ab. Bei einigen Anbietern (wie z.B. Red Hat*s OpenShift*) ist die Applikation während des Build-Prozesses offline, was zu Down-Times führt. Andere Anbieter (wie Google App Engine) setzen eigene SDKs voraus. Dadurch kann es bei dieser Methode zu erheblichen Lock-In-Effekten kommen.

Kapselung - Die Methode kapselt den kompletten Build- und Deployment-Prozess, welcher vom Cloud-Anbieter übernommen wird. Der Entwickler muss lediglich wissen, welchen Konventionen sein Projekt folgen muss um erfolgreich gebaut zu werden. Der Rest ist aus Sicht des Entwicklers eine *Blackbox*.

Komplexität - Die Methode ist für Anwendungen, die sich an bestimmten Konventionen orientieren (und z.B. ein Maven `pom.xml` besitzen) sehr einfach (ein `git push` genügt, um die Anwendung zu bauen und zu deployen). Für Anwendungen, die dies nicht tun (also insbesondere Legacy-Projekte), ist die Methode jedoch kompliziert zu übernehmen, da diese Projekte nicht für ein solches Deployment ausgelegt sind. Es muss sich in jedem Fall mit der jeweiligen Plattform des Anbieters vertraut gemacht werden.

Leistungsfähigkeit - Die Methode ist nur für Applikationen geeignet, die sich an verbreitete Standards (etwa *Maven* oder *Spring*) halten. (Legacy-) Applikationen, die von diesen Standards abweichen können, sind ungeeignet für diese Art des Deployments. Außerdem gibt es teilweise anbieterspezifische Beschränkungen wie eine maximale Größe von Anwendungen oder eine Deckelung der Build-Dauer.

Level - Dieses vorgehen ist eine PaaS. Der Anbieter übernimmt den kompletten Build- und Deployment-Prozess der Anwendung.

Inhalt der beiliegenden CD

Die beiliegende CD enthält Quellcode, Skripte und Dokumente zu dieser Arbeit. Sie ist wie folgt aufgebaut:

- `CD`
  - `Demo Videos/`
    Mit Kommentaren unterlegte Screencasts. Sie zeigen Beispielszenarien zum Bau von Docker Images, Logging, Monitoring und Deployment, sowie Anleitungen für die Entwicklung. Die Videos sind sinnvoll durchnummeriert, beginnend mit Einführungen und Grundlagen.
  - `DeployMan/`
    Das Java-Projekt des Deployment-Werkzeugs, dass im Zuge dieser Arbeit entstanden ist. Das Projekt kann mit Maven gebaut werden.
  - `Docker Images/`
    Bereits gebaute Images von quelloffenen Anwendungen wie Logstash, wie sie im Rahmen dieser Arbeit genutzt und deployt wurden.
  - `Dockerfiles/`
    Build-Dateien für Docker zum Bau von Images.
  - `Formations/`
    Konfigurationsdateien im JSON-Format für das Deployment-Werkzeug. Die Dateien beschreiben Cloud-Instanzen und können durch das Deployment-Werkzeug gestartet werden.
  - `HTML Version/`
    Enthält eine HTML Version dieser Thesis zur Betrachtung im Browser.
  - `Logs/`
  - `Road Map Meetings/`
    PDFs der Zwischenpräsentationen bei Informatica. Diese fanden einmal monatlich statt und zeigen Entwicklungsfortschritte und diskutierte Themen.
  - `Vagrant Setups/`
    Ausführbare Skripte für Vagrant (und Puppet) zum automatisierten Aufsetzen von Entwicklungsumgebungen (inklusive der Umgebung, die während dieser Arbeit zum Bau der Docker Images verwendet wurde).
  - `Dokumentation.pdf`
  - `Thesis.pdf`
    Diese Arbeit als PDF.

Literaturverzeichnis

Wikimedia Commons. Cessna ec-2. 2013. URL https://en.wikipedia.org/wiki/File:Cessna_EC-2.jpg.

Informatica. Product information management (pim). 2014. URL http://www.informatica.com/de/products/master-data-management/product-information-management/.

OSGi Alliance. The osgi architecture. 2014a. URL http://www.osgi.org/Technology/WhatIsOSGi.

Jeff McAffer, Paul VanderLei, and Simon Archer. *OSGi and Equinox - Creating Highly Modular Java Systems*. Addison-Wesley Professional, Boston, 1. aufl. edition, 2010. ISBN 978-0-321-60943-4.

Johan Den Haan. The cloud landscape described, categorized, and compared. *The Enterprise Architect*, October 2013. URL http://www.theenterprisearchitect.eu/blog/2013/10/12/the-cloud-landscape-described-categorized-and-compared/.

Docker Inc. The whole story. *Docker.io*, 2014a. URL https://www.docker.io/the_whole_story/.

Jago de Vreede and Marcel Offermans. Continuous automated deployment with apache ace. page 43, Oktober 2013. URL http://goo.gl/R6oGKf.

Apache ACE. 2014. URL http://ace.apache.org/user-doc/user-guide.html.

OSGi Alliance. Members. 2014b. URL http://www.osgi.org/About/Members.

Paul Bakker and Bert Ertman. *Building Modular Cloud Apps with OSGi*. Ö'Reilly Media, Inc.", Sebastopol, 2013. ISBN 978-1-449-34513-6.

Neil Bartlett. A comparison of eclipse extensions and osgi services. *EclipseZone*, February 2007. URL http://www.eclipsezone.com/articles/extensions-vs-services/.

Arif Mohamed. A history of cloud computing. *ComputerWeekly.com*, März 2009. URL http://www.computerweekly.com/feature/A-history-of-cloud-computing.

Christian Metzger, Thorsten Reitz, and Juan Villar. *Cloud Computing - Chancen und Risiken aus technischer und unternehmerischer Sicht*. Hanser Fachbuchverlag, München, 2011. ISBN 978-3-446-42454-8.

Brian J. S. Chee and Curtis Jr. Franklin. *Cloud Computing - Technologies and Strategies of the Ubiquitous Data Center*. CRC PressINC, Boca Raton, 2010. ISBN 978-1-439-80612-8.

Philip Wik. Thunder clouds: Managing soa-cloud risk - part i. *ServiceTechMag.com*, Oktober 2011. URL http://servicetechmag.com/I55/1011-1.

Ian Foster. What is the grid? a three point checklist. Juli 2002. URL http://www.mcs.anl.gov/~itf/Articles/WhatIsTheGrid.pdf.

Peter Mell and Timothy Grance. The nist definition of cloud computing. September 2011. URL http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf.

Nick Martin. Virtualization vs. cloud: Let's get this straight. *SearchServerVirtualization*, June 2013. URL http://searchservervirtualization.techtarget.com/feature/Virtualization-vs-cloud-Lets-get-this-straight.

Christoph Nölke. *Betrachtung und prototypische Implementierung einer Installations- und Aktualisierungsstrategie von modularen dreistufigen Enterprise Softwaresystemen*. PhD thesis, Universität Augsburg, November 2013.

Golo Roden. Anwendungen mit docker transportabel machen. *Heise Developer*, Februar 2014. URL http://www.heise.de/developer/artikel/Anwendungen-mit-Docker-transportabel-machen-2127220.html.

Ubuntu. Release notes. Technical report, November 2013. URL https://wiki.ubuntu.com/SaucySalamander/ReleaseNotes.

Heroku Dev Center. *Limits*. Heroku, Februar 2014a. URL https://devcenter.heroku.com/articles/limits.

Docker Inc. Dockercon day 1: Keynote highlights. *Docker.io*, Juni 2014b. URL http://blog.docker.com/2014/06/dockercon-day-1-keynote-highlights/.

Paul Bakker and Marcel Offermans. 2013. URL https://www.youtube.com/watch?v=oN3jYKOQ1Tk.

Marcel Offermans and Paul Bakker. Osgi in the cloud: A case study. JavaOne, 2013. URL https://www.youtube.com/watch?v=oN3jYKOQ1Tk.

Blake Smith. Understanding horizontal and vertical scaling. *BlakeSmith.me*, Dezember 2012. URL http://blakesmith.me/2012/12/08/understanding-horizontal-and-vertical-scaling.html. Abgrufen am 19.03.2014.

Michael Stonebraker. The case for shared nothing. page 5, 1986. URL http://db.cs.berkeley.edu/papers/hpts85-nothing.pdf.

Guido Steinacker. Architekturprinzipien. April 2013. URL http://dev.otto.de/2013/04/14/architekturprinzipien-2/.

Amazon Web Services. Amazon rds produktdetails. 2014. URL http://aws.amazon.com/de/rds.

Amazon Web Services. Oracle on amazon rds. 2013. URL http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/CHAP_Oracle.html.

Julie Bort. Amazon is crushing ibm, microsoft, and google in cloud computing, says report. *Business Insider*, Novmeber 2013. URL http://www.businessinsider.com/amazon-cloud-beats-ibm-microsoft-google-2013-11.

Klaus Lipinski, Hans Lackner, Oliver P. Laué, and Gerhard Kafka. Xaas (anything as a service). *ITWissen.info*, Februar 2013. URL http://www.itwissen.info/definition/lexikon/XaaS-anything-as-a-service-Anything-as-a-Service.html.

Danilo Sato. Implementing blue-green deployments with aws. *ThoughtWorks.com*, 2013. URL http://www.thoughtworks.com/insights/blog/implementing-blue-green-deployments-aws.

Docker Inc. Dockerfile reference. *Docker.io*, 2014c. URL http://docs.docker.io/reference/builder/.

George Paolini. Javasoft ships java 1.0. *Sun Microsystems, Inc*, Januar 1996. URL http://www.thefreelibrary.com/JavaSoft+Ships+Java+1.0%3B+Programming+environment+available+free+for...-a017853565.

Heroku Dev Center. *Slug Compiler*. Heroku, December 2013. URL https://devcenter.heroku.com/articles/slug-compiler.

Heroku Dev Center. *Dynos and the Dyno Manager*, 2014b. URL https://devcenter.heroku.com/articles/dynos.

Jeff Barr. Aws elastic beanstalk for docker. *Amazon Web Services Blog*, April 2014. URL http://aws.typepad.com/aws/2014/04/aws-elastic-beanstalk-for-docker.html.

Docker Inc. Docker joins cloud foundry foundation. *Docker.io*, Mai 2014d. URL http://blog.docker.com/2014/05/docker-joins-cloud-foundry-foundation/.

Canonical. Cloud-init - availability. 2014. URL http://cloudinit.readthedocs.org/en/latest/topics/availability.html.

Alle Grafiken, die in dieser Arbeit verwendet werden, sind - sofern nicht anders gekennzeichnet - mit *Google Drive* selbst erstellt. Alle Icons stammen von http://simpleicon.com und sind zur freien Verwendung freigegeben.