Bachelorarbeit am Institut für Informatik der Freien Universität Berlin,

ImmobilienScout24

Migration einer Legacy-Anwendung auf RPM-basierte Inbetriebnahme im Rahmen der Einführung von *Continuous Delivery*

Maximilien Riehl
Matrikelnummer: 4424658
maximilien.riehl@gmail.com

Betreuer: Prof. Dr. Lutz Prechelt
Eingereicht bei: Arbeitsgruppe Software Engineering

Berlin, 19. September 2012

Zusammenfassung

Im Rahmen eines Projektes zur Einführung von *Continuous Delivery*, für eine über 14 Jahre gewachsene Webapplikation, soll das bisherige Inbetriebnahmeverfahren der Applikation auf ein Red Hat Package Manager (RPM) basiertes Verfahren umgestellt werden. Eine besondere Herausforderung stellt hierbei die schrittweise Migration der insgesamt ca. 300 Server, die sich in 19 Servertypen klassifizieren lassen. Die Server sind zusätzlich über drei unterschiedliche Umgebungen verteilt. Eine weitere Herausforderung bei *Continuous Delivery* ist, dass die Inbetriebnahme mitgetestet werden muss. Zurzeit wird die Applikation im Wochenrythmus veröffentlicht. Ziel wäre also, dass eine Version der Applikation, welche in den Entwicklungs- und Testumgebungen per RPM in Betrieb genommen wurde, dann auch auf Produktion per RPM in Betrieb genommen wird. Da aus Gründen der Risikominimierung nicht alle Server eines Typs auf einmal umgestellt werden können, muss ein Parallelbetrieb beider Inbetriebnahmeverfahren gewährleistet werden.

Eidesstattliche Erklärung

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

19. September 2012

Maximilien Riehl

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Die vollautomatische Inbetriebnahme von Applikationen, speziell von Webapplikationen, ist kein neues Problem. Dies schließt zahlreiche komplexe Probleme mit ein. Beispiele dafür sind die Trennung von Applikation und Konfiguration, Hochverfügbarkeit bei der Inbetriebnahme, sowie das Zusammenspiel der Prozesse, die bei der Erzeugung und Inbetriebnahme der Applikation involviert sind.
Allerdings beschäftigen sich existierende Arbeiten ausschließlich mit der Konzeption und Implementierung dieser Verfahren für eine neue Applikation, nicht etwa mit der Migration einer existierenden Applikation auf eines dieser Verfahren. Dabei ist die Migration oftmals eine größere Herausforderung als die Implementierung des Verfahrens selbst, denn sie muss parallel zur Weiterentwicklung der Applikation erfolgen, ohne den laufenden Betrieb zu stören. So muss, neben der Auswahl und Implementierung des Automatisierungsverfahrens, auch eine Untersuchung der Risiken und Strategien für die Migration durchgeführt werden.
Im Rahmen dieser Arbeit wurde die Konzeption und Implementierung eines Ansatzes unterstützt, welcher auf RPM-Softwarepaketen aufbaut. Eine vollständige Migration der kompletten Applikation war, angesichts des Umfangs der Applikation und des für die Arbeit vorgesehenen Zeitraumes, nicht möglich. Vielmehr sollte ein solides Grundgerüst zur Migration der Applikation aufgebaut werden. Aufgrund der Komplexität lag ein besonderes Augenmerk auf dem Parallelbetrieb des neuen und des alten Verfahrens.
Die Problemlösung ist an einigen Stellen auf Besonderheiten der untersuchten Applikation zugeschnitten, sodass diese nicht ohne Weiteres verallgemeinert werden kann, allerdings sind alle verwendeten Technologien quelloffen oder haben gleichwertige quelloffene Alternativen, was eine Verallgemeinerung stark erleichtert.

2 Grundlagen

2.1 Abkürzungen, Anglizismen

Einige englische Begriffe und Abkürzungen haben sich im Bereich von *Continuous Delivery* stark eingebürgert. Sofern diese nicht allzu oft auftauchen, wurde auf eine Übersetzung verzichtet, diese werden dann entweder im Text erläutert oder befinden sich im Glossar auf Seite 47.
Folgende Anglizismen werden verwendet und sind im Glossar zu finden :

- VCS
- Build/Buildagent
- CI-Server
- Stage, Staging
- Legacy
- Continuous Delivery (nicht im Glossar, aber im Abschnitt *Continuous Delivery* und *Continuous Deployment* auf Seite 3)

2.2 RPM, YUM

RPM Um Software auf Linux in Betrieb zu nehmen wird in vielen Fällen der *Red Hat Package Manager* (RPM) verwendet. Dieser unterstützt zahlreiche Operationen wie Installation, Abfrage, Überprüfung und Löschung von Softwarepaketen im RPM-Format. Ein RPM-Paket enthält üblicherweise binär ausführbare Dateien, deswegen gibt es eine implizite direkte Abhängigkeit zwischen dem RPM-Paket und dem Betriebssystem der Maschine auf der das Paket installiert werden soll. Diese Abhängigkeit wird explizit angegeben, indem die Pakete mit ihrer Zielarchitektur versehen werden. Ein RPM-Paket wird durch einen Namen, eine Version und eine sogenannte Releasenummer charakterisiert. Das Paket besteht aus mindestens einer Kopfzeilenstruktur, jede davon enthält Informationen zu einem bestimmten Schlüssel. Die Signatur enthält kryptographische Informationen, mit denen sich die Integrität des Paketes überprüfen lässt. RPM-Pakete können ebenfalls eine beliebige Anzahl an Skripten enthalten, die nach ihren Verantwortungen klassifiziert werden (Entpacken, Kompilieren, Installieren, Deinstallieren, Post-Installieren, Post-Deinstallieren). Mit dem Post-Installationsskript lässt sich erkennen ob die Installation gut verlaufen ist. (frei übersetzt nach [Dea07, S. 4-5, Abschnitt 'Linux']).

YUM Eine verbreitete Kritik am RPM-Paketformat ist die Komplexität beim Installieren neuer Software, denn die Suche nach Paketen (insbesondere um Abhängigkeiten eines gewünschten Pakets zu erfüllen) wird dem

Benutzer selbst überlassen [Har02]. Der Paketverwalter *Yellowdog Updater, Modified* (YUM) konsultiert eine Liste von Software-Repositorien und löst bei Installationen oder Aktualisierungen von Paketen selbstständig die Abhängigkeiten auf, sodass der Nutzer nur noch den Installations- oder Aktualisierungsbefehl erteilen muss [Vid12].

2.3 *Continuous Delivery* und *Continuous Deployment*

Continuous Delivery bedeutet übersetzt kontinuierliche Lieferung, implizit ist das Liefern von geschäftlichem Wert gemeint. Dabei handelt es sich um eine Mustersprache welche den Lieferprozess von Software verbessern soll. Klassischerweise besteht dieser aus der Ansammlung von einzelnen Liefereignissen, die manuell durchgeführt werden. Bei *Continuous Delivery* soll dieser Prozess stattdessen ein kontinuierlicher Fluss sein. Dies wird unter anderem durch die Automatisierung des kompletten Prozesses erreicht, sodass dieser zu einem Fließband wird (*Deployment Pipeline* oder *Delivery Chain*). Dabei ist das Ziel den Code zu einer Applikation zusammenzusetzen, in unterschiedlichen Testumgebungen in Betrieb zu nehmen und zu testen, und schließlich auf produktiven Systemen in Betrieb zu nehmen.
Im Gegensatz zu *Continuous Deployment* erfolgt bei *Continuous Delivery* die Lieferung von Software nicht ohne menschliche Interaktion, wie z. B. ein Knopfdruck, bevor die Applikation produktiv in Betrieb genommen wird [JH10, S. 266-267]. Allgemein ist es nicht für jede Applikation möglich bis zu *Continuous Deployment* zu gehen. Sofern gewisse Qualitätsgrenzen eingehalten werden müssen, ist es sogar unwirtschaftlich ohne menschliche Kontrolle Software zu veröffentlichen [JH10, S. 267].

2.4 ImmobilienScout24

2.4.1 Unternehmen

Kennzahlen Das Unternehmen besteht aus über 500 Mitarbeitern. Die Legacy-Anwendung wird mit mehr als 300 Servern betrieben und hat monatlich über 7 Millionen eindeutige Besucher und 2 Milliarden Seitenaufrufe [Imm12a].

2.4.2 Die Legacy-Webapplikation

Definition Die Legacy-Applikation existiert seit den Anfängen des Unternehmens als Start-Up in 1998 und wurde seitdem weiterentwickelt und erweitert. Es handelt sich dabei um die zentrale Komponente für das Veröffentlichen und Suchen von Immobilienangeboten auf der ImmobilienScout24-Plattform. Inzwischen ist die Anwendung ein sogenanntes Altsystem geworden, sodass gerade wegen der Komplexität und des hohen Funktionsumfangs keine einfache Ablösung möglich ist. Hinzu kommt dass die Anwendung einen

kritischen Wert für das Kerngeschäft des Betriebs darstellt und jegliche Änderungen mit hohem Risiko verbunden sind.

Module der Anwendung Die Legacy-Anwendung ist ein verteiltes System und besteht aus 19 Servertypen, die unterschiedlich untereinander gekoppelt sind. Diese Servertypen werden Funktionsgruppen oder auch Module genannt, und bestehen aus [Imm12b]:

- API : API server (Extern zugängliche Schnittstellen, darunter eine (alte) XML Fernaufruf-Schnittstelle und eine (neue) REST-Schnittstelle)
- APP : APPlication server (Applikationsserver, auch Middletier genannt)
- BAD : Genau wie WEB, dürfen aber nur ganz bestimmte riskante Operationen ausführen (daher der Name bad, Englisch für schlecht)
- BOT : Genau wie WEB, sind ausschließlich für das Beantworten von Anfragen der Googlebot Server zuständig
- DB : DataBase (Datenbankserver)
- FFS : FulFillment Server (diese Server waren früher für Immobiliengesuchsbenachrichtigungen zuständig, dies wird inzwischen von der Funktionsgruppe APP übernommen, die Funktionsgruppe wird aber weiterhin benötigt)
- FPS : Floorplan Polling Server (arbeiten Grundrissaufträge asynchron ab)
- GIS : Geographic Information Server (sorgen für den Schluss aus einer gegebenen Adresse auf eine Koordinate)
- IMP : IMPorter/Exporter (ermöglichen das Importieren oder Abgleichen von Datensätzen zu ImmobilienScout24)
- IMS : ImageMagick Server (ermöglicht das Editieren von Bildern)
- IMW : ImageMagick Web (liefert das JavaScript für das Editieren von Bildern aus)
- MMS : MultiMedia Scaling servers (skalieren Bilder auf die nötigen Größen)
- PDF : Exposé PDF generator (erzeugen PDF-Dateien aus Immobilien-Datenblättern (Exposés))
- RES : REporting Server (erzeugen Kennzahlen, z. B. die Anzahl an Aufrufe für ein bestimmtes Exposé)

- RSS : RSS-Feedservice (RSS-Dienst für Immobiliengesuche)
- SCD : SCheDuler (führen Aufgaben periodisch oder wenn bestimmte Bedingungen erfüllt sind aus, z. B. das Anlegen von neuen Verträgen wenn das Anfangsdatum erreicht wurde)
- TAT : Terms Admin Tool (Internes Verwaltungswerkzeug)
- VID : VIDeo server
- WEB : WEBserver

2.4.3 Zustand vor dieser Arbeit

Altes Inbetriebnahmeverfahren und Veröffentlichungszyklus Bisher erfolgt die Inbetriebnahme semi-automatisch mit einer netzlaufwerkbasierten Lösung. Dabei werden die von der Applikation benötigten Dateien nach der Kompilierung erst auf das Netzlaufwerk kopiert und bei der Inbetriebnahme eines Servers schließlich auf den Server selbst. Die Verwaltung und Verteilung der Konfiguration erfolgt auf den gleichen Weg. Die Applikation wird im Wochenrythmus in Betrieb genommen. Ferner wird zwischen Inbetriebnahmen der Applikation mit (*Downtime Deployment*) und ohne (*Zero Downtime Deployment*) geplanter Ausfallzeit unterschieden.
Bei der Variante mit geplanter Ausfallzeit werden alle Server heruntergefahren, in Betrieb genommen und in der richtigen Reihenfolge wieder hochgefahren.
Die Variante ohne Ausfallzeit wird dadurch möglich, dass die ImmobilienScout24-Server über zwei Standorte (Hamburg/Berlin) verteilt und dupliziert sind. Um Hochverfügbarkeit herzustellen, wird bei der Inbetriebnahme die Applikation sequenziell über einen Standort nach dem anderen ausgerollt. Zu jedem Zeitpunkt sind die Server von mindestens einem Standort verfügbar, sodass die Wartung keinerlei Auswirkungen auf die Nutzer hat.

Umgebungen Es existieren drei Umgebungsklassen zum *Staging* bei ImmobilienScout24. Das *Staging* ist ein Testprozess um Änderungen an einer komplexen und teuren Umgebung zu validieren (vgl. Stage, Staging auf Seite 48).

- zur Entwicklung (*DEV*-Umgebungen)
- zur Qualitätssicherung (*TUV*-Umgebungen, *TUV* steht für *Test Und Verifikation*)
- zum produktiven Einsatz (*PRO*-Umgebung), wobei zwischen dem Standort Berlin (*BER*) und Hamburg (*HAM*) unterschieden wird

Erzeugungsprozess der Applikation Aufgrund der hohen Anzahl an Entwicklern verlaufen die Kompilationen parallel. Dabei werden Quelldateien kompiliert, Modultests ausgeführt, die Anwendung in Betrieb genommen und die resultierende *DEV*-Umgebung wird automatisch mit Integrations- und Rauchtests geprüft. Dies ermöglicht den Betrieb von kontinuierlicher Integration und ist eine nötige Vorbedingung zur Implementierung von *Continuous Delivery*. Bisher wird bei der Kompilation der Anwendung eine Ansammlung von Dateien erstellt (*Deployment Unit*) und auf das Netzlaufwerk kopiert. Ein Mal wöchentlich wird die aktuellste Version, welche die Tests in *DEV* bestanden hat, ausgewählt und in *TUV* in Betrieb genommen. Falls die automatischen und manuellen Tests dort gut verlaufen, wird diese Version auch in *PRO* in Betrieb genommen.

2.4.4 ImmobilienScout24-Technologie

Konfigurationsverwaltung Bei der Implementierung von Continuous Delivery ist es besonders wichtig dass nicht nur Quellcode, sondern auch Konfiguration versionsverwaltet sind [JH10, S. 31-54]. Für die Konfiguration von Systemen die mit einem RPM-Verfahren in Betrieb genommen werden, wurde von ImmobilienScout24 der *config-rpm-maker* entwickelt. Dieser ermöglicht das hierarchische Verwalten von Konfigurationen mit einem zentralen VCS. Im VCS werden die hierarchischen Ebenen mithilfe von Unterverzeich-

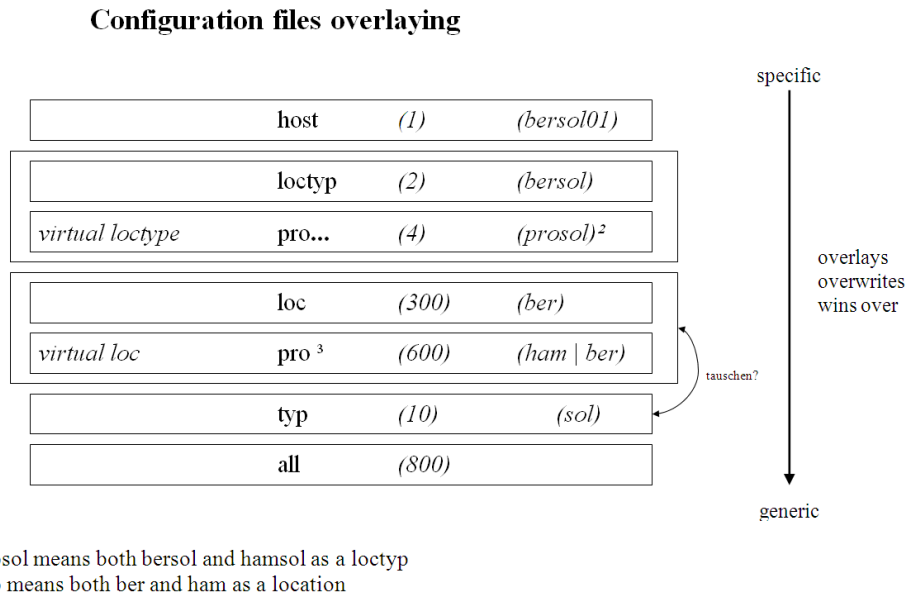

Abbildung 1: config-rpm-maker [Imm12b]

nissen abgebildet (config-rpm-maker [Imm12b] auf Seite 6). Diese Ebenen sind :

- eine globale Ebene (Verzeichnis *all*)
- Typ des Systems (Verzeichnis *typ*)
- Position des Systems im Staging (Verzeichnis *loc*, z. B. *DEV*- oder *TUV*-Umgebung)
- Kombination aus Typ und Position (Verzeichnis *loctyp*, z. B. tuvapp für Applikationsserver in *TUV*)
- Eine maschinenspezifische Ebene (Verzeichnis *host*)

Für jede Maschine kann mithilfe dieser Hierarchie eine Konfiguration erstellt werden, indem spezifische Konfigurationsebenen (wie *host*) generischere Konfigurationsebenen überschreiben. Damit können Wiederholungen vermieden werden, indem globale Konfigurationen auf höheren Hierarchieebenen vorgenommen werden.
Die Konfiguration wird vom *config-rpm-maker* zu einem RPM-Softwarepaket zusammengesetzt, welches dann auf der Maschine installiert werden kann. Der Prozess muss nicht manuell vorgenommen werden da der *config-rpm-maker* nach Änderungen im VCS automatisch neue RPM-Pakete erstellt. Die neue Konfiguration wird Maschinen, die mit RPM in Betrieb genommen werden, somit bei der nächsten Paketaktualisierung zugesteuert. Die Paketabhängigkeiten des RPM Konfigurationspaketes sind auch über das VCS editierbar. Mehr Details befinden sich im Anhang (*config-rpm-maker* auf Seite 46).

YADT *YADT* ist eine eigens von ImmobilienScout24 entwickelte Lösung zur Orchestrierung der Inbetriebnahme von Diensten, die als Softwarepakete (RPM, DEB, ...) vorliegen. *YADT* nimmt ein Dienstmodell entgegen und koordiniert die Aktualisierung der Dienste unter Berücksichtigung der einzelnen Abhängigkeiten. Die Kommunikation mit den Maschinen findet über *SSH* statt, deswegen ist kein klientseitiger Dienst außer *SSH* erforderlich. Eine ausführlichere Beschreibung befindet sich im Anhang Funktionale Übersicht von *YADT* auf Seite 45.

2.5 Lösungen zur Inbetriebnahme von Webapplikationen

2.5.1 Inbetriebnahmeverfahren

Es existieren unterschiedliche Ansätze um Software automatisiert auf viele Maschinen in Betrieb zu nehmen, welche in drei wesentliche Kategorien unterteilt werden können [TZG+] : softwarepaketbasierte Inbetriebnahmeverfahren, systemabbildbasierte Inbetriebnahmeverfahren, und verhaltensbasierte Inbetriebnahmeverfahren. Mit softwarepaketbasierten Ansätzen werden nur die nötigen Dateien für die Applikation mit den Paketen ausgeliefert.

Im Gegensatz dazu wird bei systemabbildbasierten Verfahren statt Softwarepaketen ein Systemabbild der Server verteilt [CXC+]. Ein Nachteil von Systemabbildern ist deren eingeschränkte Eignung für bestimmte Systeme, denn das Abbild beinhaltet das komplette Betriebssystem. Mit verhaltensbasiertem Inbetriebnahmeverfahren [Zha] existiert ein Lösungsansatz um diese Einschränkungen aufzuheben. Dabei wird die Software auf eine Maschine installiert und die Betriebssystemoperationen werden während der Installation gespeichert. Anschließend werden die Operationen zu einem Systemabbild zusammengefügt, welches dann auf beliebige Maschinen installiert werden kann.

2.5.2 Dienstkoordination und Konfigurationsverwaltung

Um viele unterschiedliche Server zu verwalten existieren zwei grundlegende Gedankenrichtungen.

1. Das Abbilden von Komplexität.
   Dafür wird eine domänenspezifische Sprache benutzt. Mit dieser lassen sich alle möglichen Bedingungen ausdrücken. *Puppet* ist ein bekanntes quelloffenes System zur Konfigurationsverwaltung und verfolgt einen ähnlichen Ansatz wie die Alternativprodukte *Chef* und *CfEngine*. Daher kann *Puppet* als Beispiel zur Erläuterung genutzt werden - diese ist aus [JH10, S. 292-293] übernommen. Bei *Puppet* wird die Konfiguration von einem Dienst (*puppetmasterd*) auf einem zentralen Server verwaltet. Dieser pflegt eine Liste von überwachten Maschinen. Auf diesen läuft ein Hintergrunddienst (*puppetd*), der mit dem zentralen Server kommuniziert und die Konfiguration regelmäßig synchronisiert.

2. Die Reduktion von Komplexität durch Einschränkungen.
   Mit *YADT* und dem *config-rpm-maker* gilt es die Konfigurationswerkzeuge bewusst eingeschränkt zu halten. Dadurch kann nicht jede beliebige Komplexität abgebildet werden. Dies erzwingt eine Vereinfachung der Konfiguration.

2.5.3 Migrationsstrategie der Inbetriebnahme

Es kommen zwei Migrationsstrategien in Frage :

1. Höchstes Risiko zuerst

2. Niedrigstes Risiko zuerst (ein Problem nach dem anderen)

2.5.4 Testen der Inbetriebnahme

Für *Continuous Delivery* ist wichtig, dass das Inbetriebnahmeverfahren für alle Umgebungen gleich ist, so Humble in [JH10, S. 153-154]. Dann kann die

Inbetriebnahme in jeder Umgebung mitgetestet werden. Das Testen erhöht die Zuversicht, dass das Verfahren dann auch in Produktion funktionniert. Für das automatische Testen stehen folgende Ansätze zur Diskussion :

1. Das Verwenden von allen bereits vorhandenen Integrations- und Rauchtests der Applikation
2. Das Erstellen von wenigen separaten Tests für die Inbetriebnahme; eventuell können schon vorhandene Tests wiederverwendet werden

3 Problemlösung

3.1 Ansatz, Alternativen und Diskussion

3.1.1 Zielsetzung

Ist-Zustand Das aktuelle Inbetriebnahmeverfahren ist fehleranfällig und nicht ohne manueller Nachhilfe in der Lage, Hochverfügbarkeit herzustellen. Es werden nur simple Operationen wie *fahre die Server von Funktionsgruppe X herunter* verstanden - die Abhängigkeiten der Dienste werden dabei nicht berücksichtigt. Es ist nicht sichergestellt dass die ausgeführten Befehle erfolgreich sind, sodass ein hoher manueller Aufwand bei der Inbetriebnahme entsteht.

Soll-Zustand Das aktuelle Inbetriebnahmeverfahren soll durch ein neues ersetzt werden, sodass der manuelle Aufwand abnimmt und die Zuverlässigkeit der Automation zunimmt. Hochverfügbarkeit soll nicht manuell erzwungen werden, sondern von der Automation realisierbar sein. Durch den verringerten Aufwand und der vereinfachten Bedienung soll die Zykluszeit zwischen zwei veröffentlichten Versionen der Legacy-Applikation reduziert werden, ideal wären 48 Stunden.

3.1.2 Diskussion der möglichen Ansätze

Inbetriebnahmeverfahren Der unmittelbare Vorteil bei systemabbildbasierten Verfahren ist, dass die Inbetriebnahme perfekt reproduzierbar ist, denn das komplette Betriebssystem wird auf allen Servern in dem gleichen Zustand ausgeliefert. Obwohl die Menge an zu transferierenden Daten natürlich größer ist, lassen sich beispielsweise mittels Peer-To-Peer-Verfahren wie bei Project Murder [Gad10], die Transferraten stark erhöhen, sodass die Größe der transferierten Dateien nur noch eine unwesentliche Rolle spielt. Die problematische Betriebssystemabhängigkeit lässt sich durch verhaltensbasierte Inbetriebnahmeverfahren lösen, dieser Ansatz ist allerdings außerordentlich kompliziert, da tiefe Eingriffe in das Betriebssystem notwendig sind um alle Operationen aufnehmen zu können. Generell ist bei abbildbasierten Verfahren das Übernehmen von Betriebssystemaktualisierungen problematisch. Bei der Verwendung von Systemabbildern muss dann ein neues Abbild erstellt werden.
Bei der softwarepaketbasierten Lösung wird nur eine Installation/Aktualisierung der Pakete durchgeführt, sodass bei jeder Inbetriebnahme n die Überreste der Inbetriebnahmen $0..(n-1)$ auffindbar sein können. Dies ist nicht wünschenswert denn dadurch erhöht sich die Anzahl der unbekannten Änderungen (*Delta*) nach jeder Inbetriebnahme. Die Softwarepaketverwaltung muss also dafür sorgen, dass bei Aktualisierungen keine Überreste hinterlassen werden. Dies wird von RPM sichergestellt, indem bei einer

Aktualisierung erst das neue Paket installiert wird, danach werden alle alten Versionen gelöscht [Bai00]. Der Vorteil von Softwarepaketbasierten Inbetriebnahmeverfahren ist dass ein Vorwärtsrollen der Software erzwungen wird - Pakete in der gleichen Version müssen identisch sein. Somit wird für jede Änderung ein neues Paket in einer höheren Version erstellt. Das Erzwingen des Vorwärtsrollens ist für Continuous Delivery wünschenswert, da manuelle Eingriffe vermieden werden sollten [JH10, S. 5-7]. Zusätzlich können Betriebssystemaktualisierungen und Aktualisierungen der Applikation gleichberechtigt durchgeführt und getestet werden. Der einzige Unterschied ist dann die Herkunft des Paketes. Eine Herausforderung mit Softwarepaketbasierter Inbetriebnahme wäre somit die Verwaltung der unterschiedlichen Softwarepaketquellen.

Dienstkoordination und Konfigurationsverwaltung *YADT* modelliert das komplette Datenzentrum inklusive der redundanten Standorte (*chunks*) und Dienstabhängigkeiten. So kann bei der Inbetriebnahme Hochverfügbarkeit hergestellt werden (vgl. Funktionale Übersicht von *YADT* auf Seite 45). Mit dem *config-rpm-maker* ist die Konfiguration sehr einfach vorzunehmen, denn die Konfiguration kann nur in *all, loc, typ, loctyp* und *host* geändert werden. Zusätzlich ist diese dateibasiert sodass auch Menschen ohne Programmiererfahrung die Konfiguration verstehen und verändern können. Allerdings können einige Komplexitäten nicht abgebildet werden. Dies kann auch zum Nachteil werden, denn diese Komplexitäten müssen dann zwingend aufgelöst werden. Die Auflösung der Komplexität ist mit erhöhtem Aufwand und Eingriffen in das zu migrierende System verbunden und somit kurzfristig ein Nachteil. Auf lange Sicht entsteht aber ein besseres, einfach verständliches System. In Kombination mit dem *config-rpm-maker* wird zusätzlich aus Sicht des Nutzers atomar (aber nicht instantan) auf einem neuen Server die komplette Applikation samt Konfiguration ausgeliefert (Abbildung Konfigurationsverwaltung mit YADT / mit Puppet für neue Maschinen [Rie12] auf Seite 13). Ein Vorteil, da neu erstellte Maschinen nach der Installation sofort einsatzbereit sind.
Mit den Alternativen wie *Puppet* ist dieser Vorgang aus Sicht des Nutzers nicht atomar. Ausgehend von einem neuen Server wird erst der Hintergrunddienst *puppetd* installiert. Dieser verändert solange die Konfiguration bis der Zielzustand erreicht ist. Vorteil ist dass *Puppet* die Konfiguration synchronisiert und so keine äußeren Änderungen erfolgen können. Diese werden sofort erkannt und beseitigt. Mit Systemen wie *Puppet* kann beliebige Komplexität abgebildet werden. Während dies die Umsetzung einfacher gestaltet, kann es längerfristig eine schlechte Lösung für das Unternehmen sein. Sollte die Konfiguration selbst irgendwann zum Altsystem werden, so verstehen nur noch wenige Mitarbeiter die komplexen Skripte und Einstellungen. Insbesondere beim ähnlichen System *Chef* können arbiträre Skripte in der Pro-

grammiersprache *Ruby* hinzugefügt werden, sodass die Konfiguration nur von Programmierern verstanden werden kann. Folglich kommt die Mächtigkeit dieser Konfiguration mit der Gefahr, diese zu missbrauchen und unnötig komplex zu gestalten.

Migrationsstrategie Zur besseren Anschaulichkeit kann aus den Softwareabhängigkeiten ein gerichteter Graph erzeugt werden, der als Baum interpretiert wird. Dabei sind die Funktionsgruppen die Knoten, die Abhängigkeiten der Funktionsgruppe aufeinander sind gerichtete Kanten (Abbildung Abhängigkeiten der Legacy-Applikation [Rie12] auf Seite 21). Die Abhängigkeiten können sowohl Laufzeit als auch Startabhängigkeiten sein. Dabei entspricht das höchste Risiko zuerst der Priorisierung von Wurzeln - dies sind die Knoten mit den meisten eingehenden Kanten. Es hängen sehr viele Funktionsgruppen von diesen Knoten ab, sodass das Risiko bei der Migration dieser Knoten sehr hoch ist.
Der Ansatz, ein Problem nach dem Anderen zu lösen, entspricht einem minimal zunehmenden Risiko, wobei im Abhängigkeitsbaum die Blätter priorisiert werden. Die Blätter haben die wenigsten eingehenden und ausgehenden Kanten, sodass die Migration unkompliziert ist. Für die bei einer Migration notwendigen Koexistenz beider Inbetriebnahmeverfahren bedeutet dies, dass die Reihenfolge der Verfahren bei der Inbetriebnahme umgedreht wird. Mit Blattpriorisierung muss bei der Inbetriebnahme der kompletten Anwendung erst das alte Inbetriebnahmeverfahren angewandt werden, damit die Abhängigkeiten der Blatt-Funktionsgruppen vorhanden sind. Erst danach können die migrierten Maschinen mit dem neuen Verfahren in Betrieb genommen werden (siehe Migrationsstrategie : Blätter zuerst [Rie12] auf Seite 13).
Mit Wurzelpriorisierung muss stattdessen erst das neue Inbetriebnahmeverfahren angewandt werden, um die Abhängigkeiten zu erfüllen, und danach das alte Inbetriebnahmeverfahren (siehe Migrationsstrategie : Wurzel(n) zuerst [Rie12] auf Seite 14). Die Reihenfolge der Inbetriebnahmeverfahren beim Parallelbetrieb ist für die Risikoverwaltung kritisch, denn wenn die automatische Inbetriebnahme der kompletten Applikation fehlschlägt, gibt es beim wurzelbasierten Ansatz keine Möglichkeit mehr weiterzumachen. Wenn sich das Problem nicht beheben lässt, müssten die bereits migrierten Maschinen wieder auf das alte Inbetriebnahmeverfahren zurückgerollt werden. Dies lässt sich anschaulich mithilfe von Migrationsstrategie : Wurzel(n) zuerst [Rie12] (Seite 14) begründen : während der Migration ist das neu eingeführte Verfahren (automatische Inbetriebnahme) fehleranfälliger als das alte, bewährte Verfahren. Wenn in Abbildung Migrationsstrategie : Wurzel(n) zuerst [Rie12] das neue Verfahren fehlschlägt (Schritt 3), dann kann Schritt 4. aufgrund der Softwareabhängigkeiten auch nicht ausgeführt werden. Folglich ist auf keiner einzelnen Maschine die Applikation in Betrieb genommen worden. In Abbildung Migrationsstrategie : Blätter zuerst [Rie12] (Seite 13)

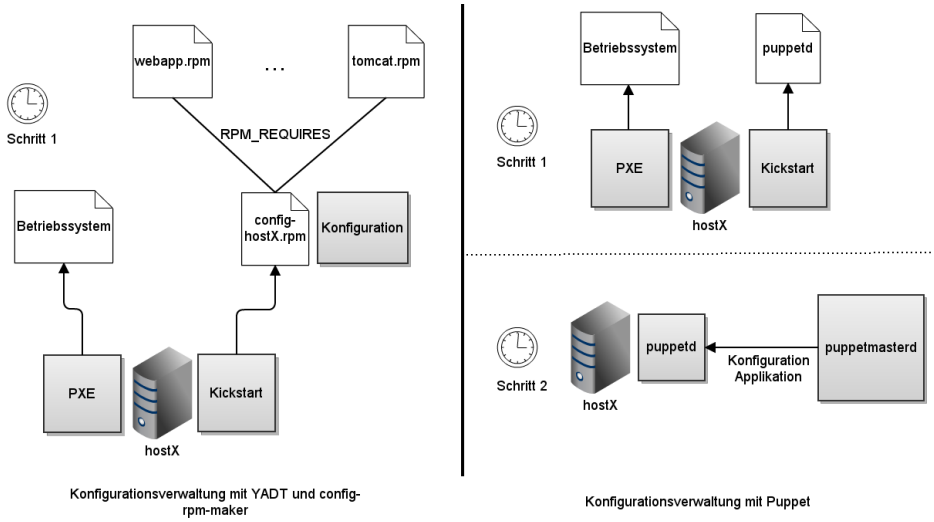

Abbildung 2: Konfigurationsverwaltung mit YADT / mit Puppet für neue Maschinen [Rie12]

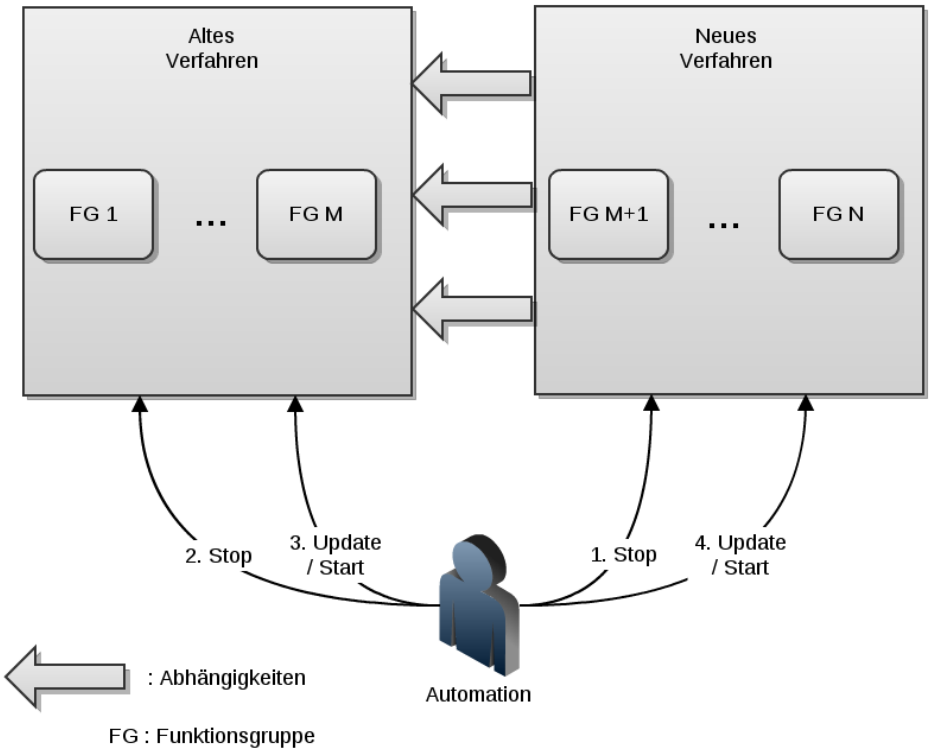

Abbildung 3: Migrationsstrategie : Blätter zuerst [Rie12]

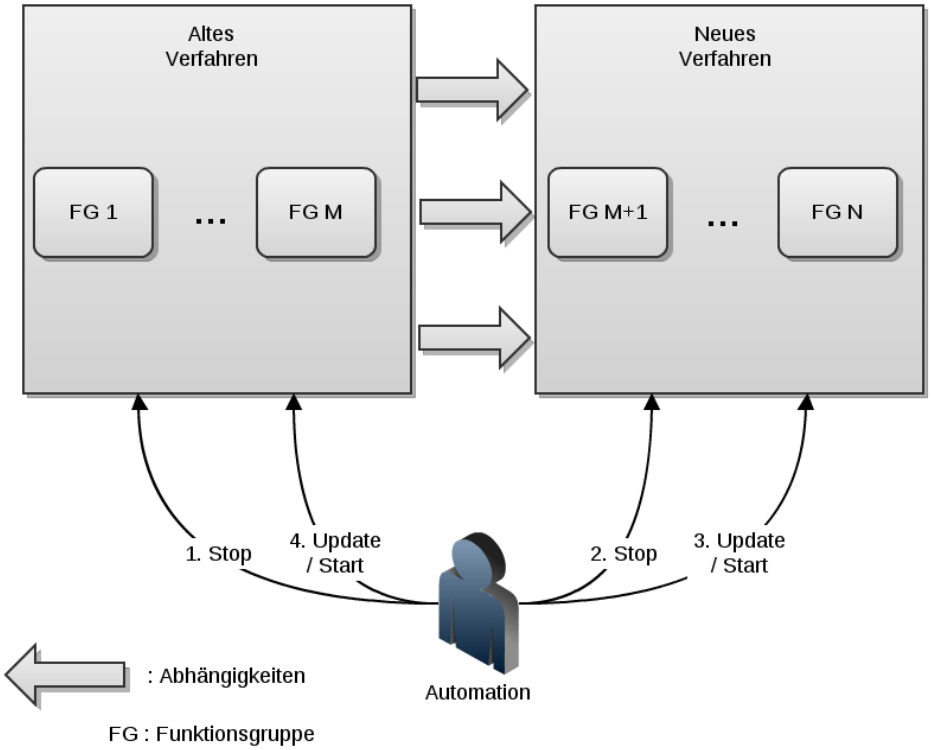

Abbildung 4: Migrationsstrategie : Wurzel(n) zuerst [Rie12]

hingegen ist das Aktualisieren der Softwarepakete mit dem neuen Verfahren der letzte Schritt. Das Scheitern dieses Schritts hat zur Folge dass lediglich die migrierten Maschinen nicht in Betrieb genommen werden. Ein Vorteil zur wurzelbasierten Alternative, wo keine einzige Maschine funktionnieren würde.
Obwohl das blätterbasierte Verfahren viele Vorteile hat ist das wurzelbasierte Verfahren eine Überlegung wert. Mit dem höchsten Risiko zuerst anzufangen ist ein starker Machbarkeitsnachweis : wenn ein Inbetriebnahmeverfahren für die komplexesten Funktionsgruppen erfolgreich benutzt wird, ist es sicherlich auch für die einfacheren Funktionsgruppen geeignet. Zusätzlich wird schneller Wissen angesammelt, da viele Probleme gleichzeitig und zum ersten Mal gelöst werden müssen.

Testen der Inbetriebnahme Das Verwenden der bereits vorhandenen Tests für die Applikation ist die einfachste Lösung. Diese decken den Großteil der Funktionalität ab, sodass Probleme bei der Inbetriebnahme durch funktionale Fehler erkannt werden können. Allerdings ergibt sich aus der hohen funktionalen Abdeckung eine sehr hohe Laufzeit. Wenn Fehler bei der Inbetriebnahme auftauchen, kann keine Aussage mehr über die Qualität der Applikation getroffen werden - ob der Fehler in der Applikation oder in dem Inbetriebnahmeverfahren liegt ist zunächst unwichtig. Dies spricht für eine sehr geringe Menge an Tests, die speziell für die Ausführung kurz nach der Inbetriebnahme gedacht sind. Wenn diese fehlschlagen, kann der Vorgang sofort abgebrochen werden, damit die Fehlersuche sofort beginnen kann. Ein Nachteil dieses Ansatzes ist dass sich die Gesamtzeit durch die zusätzlichen Tests erhöht. Vorteilhaft ist die schnellere Rückmeldung, wenn Fehler auftauchen. Je nachdem wie groß die Quellcodebasis ist und wie lange die Integrations- oder Rauchtests benötigen, kann also abgewogen werden, welcher Ansatz praktischer ist.

3.1.3 Gewählter Ansatz für ImmobilienScout24

Inbetriebnahmeverfahren Der gewählte Ansatz soll ein softwarepaketbasiertes Inbetriebnahmeverfahren verwenden. Aufgrund der Tatsache, dass fast alle Server das Betriebssystem Red Hat Enterprise Linux benutzen, bietet sich im Falle von ImmobilienScout24 das RPM-Softwarepaketformat an.

Koordination der Dienste, Konfigurationsverwaltung *YADT* (siehe YADT auf Seite 7) wird benutzt um die Dienste bei der Inbetriebnahme untereinander zu koordinieren. Die Konfiguration wird mit dem *config-rpm-maker* verwaltet und ebenfalls durch *YADT* verteilt. Die Motivation für diese Entscheidung ist dass durch die eigene Entwicklung internes Wissen entsteht. Mit den Alternativen wäre ein hoher Schulungsaufwand notwendig. Zusätzlich soll die Einschränkung der Komplexität im Vergleich zu *Puppet*

oder *Chef* für eine klare, einfache Konfiguration sorgen an der sich sowohl Programmierer als auch Administratoren mit wenigen Programmierkenntnissen beteiligen können.

Migrationsstrategie und Risikoverwaltung Um von dem aktuellem zum RPM-basierten Inbetriebnahmeverfahren der Software zu migrieren, wurde eine blätterbasierte Strategie gewählt. Es wird mit einem geeignetem Auswahlverfahren eine Funktionsgruppe nach der anderen selektiert und auf das RPM-Inbetriebnahmeverfahren umgestellt. Die Umstellung soll durch *DEV,TUV* bis *PRO* propagiert werden, da bei Continuous Delivery insbesondere die Inbetriebnahme mitgetestet werden muss. Das heißt, dass zunächst die Entwicklungsserver der Funktionsgruppe auf RPM-Inbetriebnahme umgestellt werden. Wenn keine Fehler identifiziert werden, können die Qualitätssicherungsserver der Funktionsgruppe umgestellt werden. Letztlich werden die Produktionsserver der Funktionsgruppe auf das RPM-Verfahren umgestellt.
Das gewählte Auswahlverfahren ist wie folgt :
In jedem Schritt wird die Funktionsgruppe gewählt welche die wenigsten eingehenden Kanten hat. Von dieser Funktionsgruppe hängen die wenigsten anderen Funktionsgruppen ab, bei Fehlfunktionen ist das Risiko also minimal. Falls dies nicht eindeutig ist, wird die Funktionsgruppe mit den wenigsten ausgehenden Kanten ausgesucht. Dies bedeutet nämlich, dass die Konfiguration der neu aufgesetzten Maschinen einfacher ist, denn sie hängen nur von wenigen anderen Diensten ab. Es wird auch versucht, möglichst nur Funktionsgruppen zu wählen bei denen die eingehenden Kanten von bereits automatisch in Betrieb genommenen Funktionsgruppen stammen.
Es soll allerdings nicht strikt nach diesem Verfahren vorgegangen werden da viele Faktoren nicht durch den Abhängigkeitsbaum abgebildet werden können. Die Ausnahmen werden in der Durchführung begründet. Der Grund für diese Strategie ist sowohl konzeptionell als auch psychologisch motiviert. Der konzeptionelle Grund wird in der Diskussion von Alternativen (Migrationsstrategie auf Seite 12) geklärt. Der psychologische Grund leitet sich daraus ab, dass sich die aktuelle Infrastruktur über die Jahre bewährt und stabilisiert hat. Die Einführung des neuen Verfahrens ist mit einer tiefgreifender Änderung an dieser Infrastruktur verbunden. Es ist also wichtig, dass das neue Verfahren bei allen Betroffenen Akzeptanz findet. Dank der iterativen Migration mit progressivem Risiko und dadurch, dass in jedem Schritt auf die bereits erreichten Erfolge aufgesetzt wird, fällt es allen einfacher dem neuen Verfahren Vertrauen zu schenken.

- Der temporären Organisationseinheit, die für das Vorantreiben der Migration zuständig ist (*Delivery Chain Taskforce*), denn es muss nur ein Problem auf einmal gelöst werden
- Dem Betrieb, denn es werden keine unnötigen Risiken gewählt

- Dem Produktionsteam, denn die Komplexität des Ausrollens der Applikation erhöht sich nicht sondern nimmt langsam ab
- Den Entwicklern, denn sie können sich langsam an den Infrastrukturwechsel gewöhnen, da kritische Module erst später auf das neue Verfahren umgestellt werden. Durch die nur langsam steigende Komplexität bei der Migration bleibt mehr Zeit für Kommunikation und Absprache übrig, sodass die Änderungen nicht intrusiv sind

Testen der Inbetriebnahme Die Integrations- und Rauchtests der Legacy-Applikation benötigen im Durchschnitt eine Stunde und sind nach Funktionsgruppen organisiert. Aufgrund von Limitationen des Testprozesses wird immer eine Funktionsgruppe nach der anderen getestet. Die Reihenfolge ist festgelegt und lässt sich leider nicht dynamisch ändern. Mit dem ersten Ansatz, bei dem nur die vorhandenen Tests benutzt werden, kann dies zu einem problematischen Fall führen, wenn einem Entwickler ein Fehler unterläuft und eines der Module folglich gar nicht mehr startet oder schwerwiegende funktionale Fehler aufweist. Wenn dieses Modul zusätzlich eines der letzten getesteten ist, wird der fehlerhafte Code erst nach ein bis zwei Stunden entdeckt werden. Mit dem anderen Ansatz würde das Problem innerhalb von wenigen Minuten nach der Inbetriebnahme erkannt werden. Aus diesem Grund wurde der zweite Ansatz gewählt.

3.2 Durchführung

3.2.1 Planung, Vorgehensweise

Arbeitsweise Innerhalb des Unternehmens ist die *Delivery Chain Taskforce* für das Vorantreiben von *Continuous Delivery* und somit auch für die Migration der Inbetriebnahme verantwortlich. Im Rahmen dieser Arbeit habe ich mit der *Taskforce* gearbeitet. Diese besteht aus Entwicklern, die jeweils vier Wochen in der Taskforce mitwirken und dann verantwortlich für die Auswahl eines Nachfolgers sind. Die Arbeitsweise ist an die agilen Methoden *Kanban* und *Scrum* angelehnt, sodass Aufgaben nicht verteilt sondern als Aufgabenzettel täglich von einer Aufgabentafel gezogen werden. Die groben Ziele werden ausformuliert (*Epic*) und in kleinere Ziele (*Story*) aufgeteilt. Diese werden dann im Team in einzelne Aufgaben (*Tasks*) konkretisiert (*Storyplanning*). Sind alle *Tasks* erledigt worden, wird die *Story* abgeschlossen. Die Granularität für *Tasks* kann stark variieren, sodass eine Schätzung des tatsächlichen Aufwandes schwierig ist. Üblich sind ein bis zwei Zettel am Tag pro Person. Grundsätzlich werden alle Aufgaben immer mindestens zu zweit bearbeitet, denn die Änderungen an der Inbetriebnahme sind kritisch und konzeptionell schwierig. Die Problemlösung als Team ist eine Voraussetzung für den Erfolg des Projektes. Bei einem einzigen Beteiligten hätte, nicht

zuletzt aufgrund des Zeitaufwands, höchstens ein Prototyp als Machbarkeitsnachweis entwickelt werden können. Ein weiterer Vorteil der Durchführung des Projektes als Team ist die Möglichkeit, mit Paarprogrammierung zu arbeiten. Die Vorteile von Paarprogrammierung wurden in [CW00] erforscht. Insbesondere verbessert sich dadurch die Qualität der Ergebnisse. Zusätzlich ergibt sich durch die Arbeit im Team auch das Verteilen des Wissens über die Problemlösung. Dazu trägt zusätzlich ein tägliches kurzes Zusammentreffen (*Stand-up*) bei. Dabei fasst jeder Beteiligte zusammen, was er am vergangenen Tag getan hat, was er danach tun wird, und was ihn derzeit aufhält.

Planung, Auswahl der Funktionsgruppen Als erstes Ziel für die Migration wurde die Funktionsgruppe *RSS* ausgewählt - diese hat so gut wie keine Abhängigkeiten und erfüllt keine kritische Funktion (vgl. Abhängigkeiten der Legacy-Applikation [Rie12] auf Seite 21). Die Funktionsgruppen *VID, IMW, IMS* sind keine Kandidaten da sie gar keine Abhängigkeiten haben und somit nicht an den Versionen anderer Funktionsgruppen gebunden sind. Es ist einfacher diese Dienste aus der Legacy-Webapplikation zu extrahieren als sie während der Migration zu berücksichtigen. *BOT, BAD* sind auch keine Alternative, da diese Funktionsgruppen der komplexen Funktionsgruppe *WEB* sehr nah sind. Schließlich sind die Funktionsgruppen *SCD, FPS* zu kritisch um zuerst ausgewählt zu werden und *FFS* soll komplett abgeschafft werden da *APP* das Versenden der Gesuchsbenachrichtigungen übernommen hat. Es bleiben folglich *TAT, RSS, PDF* als Kandidaten übrig. Da *RSS* die am wenigsten genutzte Funktionsgruppe von diesen ist, wurde sie als erstes Migrationsziel ausgewählt. Danach sollen *PDF* und *TAT* migriert werden. Im Anschluß kann *API* migriert werden, die Abhängigkeit von *API* auf *PDF* ist unproblematisch da *PDF* zu diesem Zeitpunkt schon das neue Verfahren verwenden wird.

3.2.2 Abhängigkeitsgraph / Abhängigkeitsbaum

3.2.3 Modellierung der Delivery Chain, erste Iteration

Isolieren der Versionen Aufgrund der Tatsache dass eine softwarepaketbasierte Inbetriebnahme im Grunde eine Aktualisierung/Installation der Softwarepakete ist, muss die Versionskonsistenz zwischen den in Betrieb genommenen Funktionsgruppen der Applikation gewährleistet werden. Zu einer bestimmten Version der Applikation gehören genauso viele Softwarepakete, wie es Funktionsgruppen gibt. Da es nicht nur lose Kopplungen sondern auch z. B. Abhängigkeiten auf das unterliegende Datenmodell gibt (z. B. *Corba*-Schnittstellen), müssen die Softwarepakete für eine Serverumgebung alle in der gleichen Version vorliegen. Deswegen müssen die Maschinen bei einer Inbetriebnahme ihre installierten Softwarepakete, welche zur Webapplikati-

on gehören, auf die gleiche Version aktualisieren. Dies ist nicht gewährleistet wenn die Softwarepakete der Webapplikation in unterschiedlichen Versionen in dem gleichen Repositorium abgelegt werden, denn bei einer Aktualisierung werden die neuesten Versionen installiert, ohne dass Rücksicht auf die Konsistenz zwischen den Versionen genommen wird.
Zusätzlich hat das neue Inbetriebnahmeverfahren den Anspruch, dass ein Aktualisierungsbefehl jederzeit erteilt werden kann, ohne dass eine ungewünschte Version installiert wird.
In diesem Sinne wurde sich für sogenannte Versionsrepositorien entschieden, d. h. es gibt für jede Version der Applikation ein unabhängiges danach benanntes Repositorium. In diesem sind alle Softwarepakete der Applikation in der gleichen Version vorhanden.
Dieses Repositorium wird allen in Betrieb zu nehmenden Maschinen zugewiesen, sodass ein Aktualisierungs- oder Installationsbefehl zur Folge hat, dass alle Softwarepakete in der gleichen Version installiert werden. Das liegt daran, dass keine anderen Pakete vorliegen. Ein späterer Aktualisierungsbefehl hat keinerlei Auswirkungen, da die installierten Versionen identisch zu den Versionen der Pakete im Repositorium sind.

Testen und Staging der Inbetriebnahme In der DEV-Umgebung kann die Migration ohne Weiteres erfolgen. Es ist keine Absprache mit Testern nötig da nur automatische Tests ausgeführt werden. Die automatisierten Integrations- und Rauchtests sind ausreichend um die Korrektheit des Inbetriebnahmeverfahrens für diese Umgebung zu gewährleisten. Eine Untermenge dieser Tests wird nach der Inbetriebnahme ausgeführt mit dem Ziel zu prüfen, ob die Dienste richtig funktionnieren.
Die TUV-Umgebung wird zunächst in Betrieb genommen und von der Qualitätssicherungsabteilung getestet und freigegeben. Im laufenden Betrieb werden Maschinen aus dem Lastverteiler entfernt, automatisch in Betrieb genommen und dem Lastverteiler wieder hinzugefügt. Nun kann die Qualitätssicherung erneut testen und freigeben. Zuletzt werden nur automatisch in Betrieb genommene Maschinen dem Lastverteiler hinzugefügt, von der Qualitätssicherungsabteilung getestet und freigegeben.
Für die Migration der produktiv eingesetzten Server wird ebenfalls der Lastverteiler verwendet, um eine Beeinträchtigung des Dienstes zu verhindern. Die Maschinen werden einzeln ausgetauscht und auf das neue Inbetriebnahmeverfahren umgestellt. Das vorzeitige Entfernen der betroffenen Maschinen aus dem Lastverteiler versichert, dass die Anfragen von Kunden nicht während der Umstellung zu diesem Server gelangen. Zur nächsten Veröffentlichung der Applikation können die umgestellten Server dann sofort mit dem neuen Verfahren in Betrieb genommen werden.
Somit wurde das neue Inbetriebnahmeverfahren durch alle drei Umgebungen propagiert und zu jedem Zeitpunkt getestet.

Erster Ansatz für die Delivery Chain Die Delivery Chain ist der Zusammenhang zwischen den Prozessen, die verwendet werden, um Quellcode an den Business auszuliefern. Bei der Modellierung soll zunächst nur das neue Inbetriebnahmeverfahren berücksichtigt werden, denn der Parallelbetrieb beider Verfahren wird nur solange wie notwendig erfolgen. Folglich soll der Fokus beim Erstellen der Prozesse eher auf dem neuen Verfahren liegen, als auf dem alten Verfahren bzw. auf die Koexistenz beider Verfahren. In Abbildung 6 auf Seite 21 ist der erste Ansatz für die Delivery Chain zu sehen. Dieser beinhaltet lediglich die Inbetriebnahme der DEV-Umgebung. Dabei veranlasst der CI-Server, *Teamcity*, bei Änderungen im VCS einen Buildagent RPM-Pakete für die Version zu bauen. Diese werden in einem lokalen YUM-Repositorium hochgeladen. Anschließend wird eine ausgewählte DEV-Umgebung mithilfe von $YADT$ in Betrieb genommen. Danach führt der Server automatische Tests gegen die aufgesetzte Umgebung aus.

Problem : Repositorien müssen eindeutig sein Die Lösung für Konsistenz zwischen Versionen ist ein RPM-Repositorium für jedes Kompilat der Applikation zu erzeugen (Isolieren der Versionen auf Seite 18). Dafür muss jedes erstellte Kompilat eindeutig sein - das Repositorium soll so benannt werden, dass sofort ersichtlich ist, welche Version der Applikation darin vorliegt.
Zum Erstellen der RPM-Pakete wird ein Plugin für das benutzte Kompilations- und Verwaltungswerkzeug *Maven* verwendet : das *Maven RPM-Plugin*. Zum eindeutigen Benennen der Versionsrepositorien sollen die RPM-Versionsnummer und die RPM-Releasenummer der RPM-Pakete verwendet werden. Mit dem *Maven* RPM-Plugin ist allerdings nicht gewährleistet, dass eine sinnvolle RPM Releasenummer vorhanden ist : bei sogenannten Snapshotbuilds setzt das Plugin die RPM-Releasenummer auf den aktuellen Zeitstempel, wie in der offiziellen Dokumentation des Plugins [Cod10] nachzulesen ist. Grund dafür ist dass Maven zwischen Versionen während der Entwicklung (*Snapshots*) und veröffentlichte Versionen (*Releases*) unterscheidet [Fou12b]. Dies ist nicht wünschenswert da ein Zeitstempel keine Aussagekraft über die Version der Applikation hat. Beispielsweise könnte ein RPM aus einem älteren Stand der Applikation erstellt worden sein, der Zeitstempel wäre trotzdem höher als ein neuerer Stand der Applikation, der davor erstellt wurde. Darüber hinaus ist der Snapshotmechanismus nicht mit Continuous Delivery verträglich. Mit der klassischen Maven-Philosophie werden Snapshots gebaut, getestet, und wenn diese für gut befunden werden oder ein Meilenstein erreicht wird, dann wird ein Release erstellt und veröffentlicht.
Anders soll mit Continuous Delivery nicht bei Kompilieren entschieden werden ob die Version freigegeben wird oder nur zur Entwicklung benutzt wird. Stattdessen soll für jede erzeugte Version ein Grund gefunden werden, warum diese Version nicht freigegeben werden sollte. Es darf also keine Snapshots

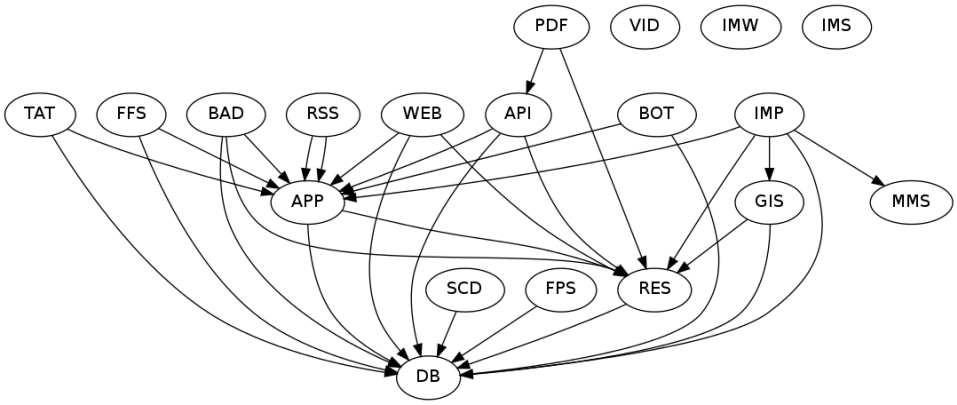

Abbildung 5: Abhängigkeiten der Legacy-Applikation [Rie12]

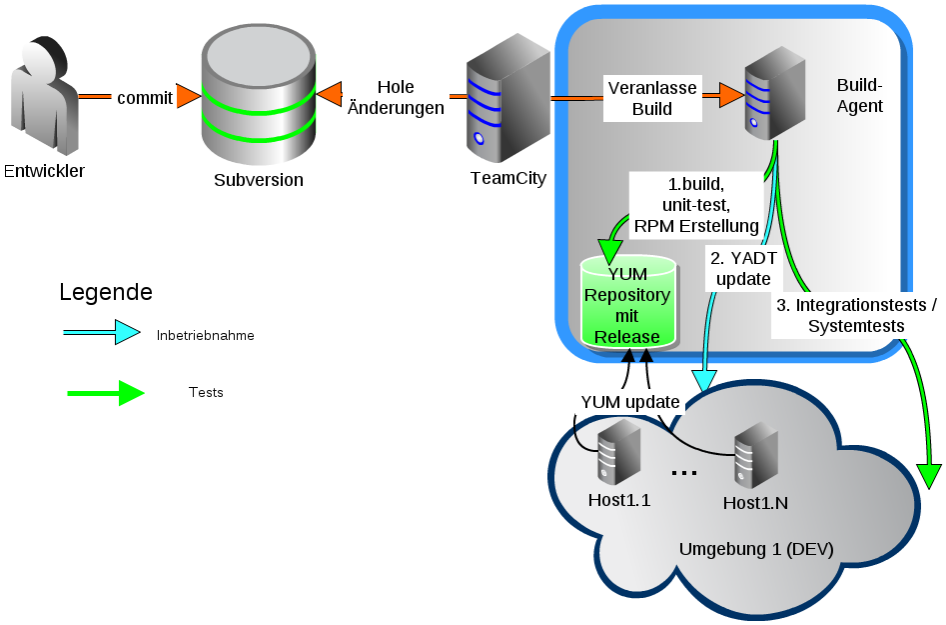

Abbildung 6: DEV Stage für die Delivery Chain [Rie12]

geben, da jede Version ein Kandidat zur produktiven Inbetriebnahme ist.

Zuordnung zwischen Repositorien und in Betrieb zu nehmenden Maschinen Die Notwendigkeit, ein Repositorium pro Release zu erstellen (Isolieren der Versionen auf Seite 18), bringt ebenfalls das Problem mit sich, dass die einzelnen in Betrieb genommenen Maschinen wissen müssen, welche Repositorien sie abfragen müssen. Dies geschieht für *YUM* üblicherweise durch eine Textdatei in */etc/yum/repos.d/* die Teil der Konfiguration der Maschine ist. Dies hat ungewünschte Folgen, denn die der Maschine bekannten Repositorien müssen mit jeder Version ersetzt werden. Die Konfiguration bei jeder Inbetriebnahme zu ändern ist aufwändig und konzeptionell falsch. Zusätzlich muss jeder Buildagent beim Kompilieren und Testen einer Version ein global erreichbares Repositorium erstellen können. Dies dient der Weitergabe an spätere Stages und der Inbetriebnahme der zugeordneten Umgebung.

Sicherheitsprobleme Anhand des Bildes Abbildung 6 auf Seite 21 wird klar, dass der aktuelle Ansatz aus Sicherheitssicht problematisch ist. Dies folgt aus der Tatsache, dass sich *YADT* mittels SSH (YADT auf Seite 7) auf die Maschinen einloggt und somit die Buildagents einen passwortlosen SSH-Zugang zu den Maschinen benötigen. Für die Entwicklungsumgebung ist dies noch annehmbar, für die Qualitätssicherungs- und Produktionsumgebungen allerdings nicht mehr, denn es würde bedeuten dass ein Angreifer, der sich Zugang zu einem Buildagent verschafft, sich auf alle anderen Maschinen ebenfalls ohne Passwort anmelden kann.

Lösungen Die Lösung für Problem : Repositorien müssen eindeutig sein auf Seite 20 war das Maven-RPM-Plugin anzupassen.

1. Die RPM-Versionsnummer wird auf *release*.0.*VCS-Revision* (wobei *release* die intern vergebene Nummer für die Veröffentlichungsversion der Applikation ist)

2. Die RPM-Releasenummer wird auf die sequenzielle Kompilierungsnummer des Buildagent gesetzt

Damit sind die Namen der RPMs immer eindeutig. Die Repositorien können auch nach diesem Schema benannt werden. Selbst wenn ein Buildagent mehrmals genau die gleiche VCS-Revision der Applikation herunterlädt, kompiliert und daraus ein Repositorium erstellt, ändert sich mindestens die (inkrementierte) Kompilierungsnummer, sodass ein neues, eindeutiges Repositorium erstellt werden kann.
Als Lösung für Sicherheitsprobleme auf Seite 22 wurde durch ein anderes Team eine dienstbasierte Lösung entwickelt. Damit ruft der Buildagent statt

YADT nur noch einen Intranetdienst zur Inbetriebnahme auf. Eine Beschreibung der Funktionsweise dieses Dienstes befindet sich im Anhang unter dem Abschnitt *YADT*-Dienst auf einem Bastionserver auf Seite 46.
Um Zuordnung zwischen Repositorien und in Betrieb zu nehmenden Maschinen auf Seite 22 zu lösen, ist ein Verwaltungsdienst für YUM-Repositorien nötig, der die notwendige Abstraktion liefert, um die Kopplungen zwischen Repositorien und in Betrieb zu nehmende Maschinen aufzulösen, und die Möglichkeit mitbringt, dynamisch Repositorien zu erzeugen und zu verwalten.

3.2.4 Implementierung des *yum-repo-server*

Absicht Mit der Implementierung des *yum-repo-server* wurden folgende Absichten verfolgt :

1. Die Verwaltung von YUM-Repositorien sollte global, einheitlich und nicht konfigurativ sein
2. Die Zuweisung von Repositorien zu einzelnen Maschinen ist einmalig konfigurativ, die Konfiguration soll nicht bei der Inbetriebnahme einer neuen Version geändert werden

Funktion und Implementierung Der *yum-repo-server* wurde von mir und der Delivery Chain Taskforce entwickelt. Der Code ist quelloffen und steht auf der entsprechenden *github*-Seite zur Verfügung [CT12]. Laut der Seite *Masterbranch* waren mir am 06.09.2012 $57,7\%$ der Änderungen (*commits*) zuzuschreiben [S.L12], allerdings berücksichtigt diese Statistik weder die Granularität der Änderungen noch die Änderungen vor der Veröffentlichung auf *github*, sodass sie mit Vorsicht zu genießen ist. Alle Funktionen des *yum-repo-server* sind durch eine REST-konforme HTTP-Anwendungsschnittstelle zugänglich und es steht ein Kommandozeilenprogramm (*yum-repo-client* oder kurz *repoclient*) zur Verfügung um die Funktionsaufrufe einfacher und expressiver zu gestalten. Die wichtigsten Funktionen des *yum-repo-server* sind

- Repositorien erstellen und löschen
- RPM-Pakete hochladen, verschieben (Propagation) und löschen
- Metadaten manuell oder periodisch erzeugen
- Repositorien mit Markierungen (Tags) versehen
- Repositorien nach Markierungen und anderen Eigenschaften filtern und suchen
- Virtuelle Repositorien erstellen, ändern und löschen

Das Konzept von virtuellen Repositorien ist eine wichtige Neuerung und baut auf der Idee auf, die Zuweisung von Repositorien nicht auf der Maschinenebene zu lösen, sondern auf der Repositoriumsebene - der Maschine wird einmalig ein Repositorium zugewiesen, und dieses *zeigt* dann auf ein beliebiges anderes Repositorium. Dabei wird zwischen *statischen* und *virtuellen* Repositorien unterschieden. Ein statisches Repositorium ist ein herkömmliches *YUM*-Repositorium. Im Gegensatz dazu ist ein virtuelles Repositorium leer und leitet jegliche Aufrufe an ein statisches Repositorium weiter. Das Ziel eines virtuellen Repositoriums kann jederzeit verändert werden (Virtuelle Repositorien [Rie12] auf Seite 33), indem es auf ein anderes statisches oder virtuelles Repositorium gesetzt wird. Beim Zuweisen auf ein virtuelles Repositorium wirkt die Operation transitiv, sodass das virtuelle Repositorium dann nicht mehr auf das virtuelle Repositorium, sondern auf das statische zeigt. Damit lassen sich Abhängigkeitsketten verhindern.

3.2.5 Inbetriebnahme mit *yum-repo-server*

Mit dem *yum-repo-server* konnte DEV Stage für die Delivery Chain [Rie12] (auf Seite 21) so geändert werden, dass Maschinen aus einer Staging-Umgebung ein virtuelles Repositorium benutzen, welches auf ein statisches Repositorium zeigt, welches die Pakete für ein bestimmtes Release der Applikation enthält (Abbildung 8 auf Seite 34). Dafür habe ich ein RPM-Paket erstellt welches bei der Installation das geeignete virtuelle Repositorium ermittelt und installiert. Diese Auswahl hängt in *DEV* von der Strangnummer der Maschine ab. Folglich muss nur die Strangnummer der Maschine ermittelt werden. Ein Platzhalter in der URL des Repositoriums kann damit dann ersetzt werden. Dieses RPM-Paket wird konfigurativ über *loctyp* dev$X$ für alle migrierten Funktionsgruppen X ausgeliefert.
Zusätzlich kann nun jeder Buildagent mithilfe des *repoclient* ein Versionsrepositorium erstellen nachdem er RPM-Pakete erstellt hat, indem er diesen aufruft. Repositorien, die nicht auf dem *yum-repo-server* liegen (sondern z. B. im ersten Stage auf dem Buildagent selbst), können mithilfe einer HTTP-Umleitung auch auf dem *yum-repo-server* zur Verfügung gestellt werden. Anfragen von YUM an das Repositorium werden dann an ein fiktives Repositorium auf dem *yum-repo-server* gerichtet, werden aber aufgrund der Umleitung von dem Buildagent beantwortet.

Weitergabe von Versionen Nachdem eine Version der Applikation in einer Umgebung N erfolgreich in Betrieb genommen und getestet wurde, muss sie an die nächste Umgebung weitergegeben werden. Dies geschieht durch ein Verweis des virtuellen Repositoriums von Umgebung $N + 1$ auf das virtuelle Repositorium der aktuellen Umgebung N (Abbildung 9 auf Seite 35). Diese Weitergabe erfolgt durch den Buildagent, welcher hierfür per Kommandozeile den *repoclient* aufruft.

Sperren von Repositorien In Abbildung 9 auf Seite 35 wird ein Problem erkennbar : was passiert wenn Umgebung $N+1$ gerade in Betrieb genommen wird? Sicherlich hätte das Modifizieren vom gerade verwendeten virtuellen Repositorium gravierende Auswirkungen auf die laufende Inbetriebnahme.
Als Lösung dafür müssen Repositorien während einer Inbetriebnahme gesperrt werden oder die Weitergabe muss getrennt erfolgen. Letzteres wird durch die Einführung eines Pre-Repositoriums für jede Umgebung realisiert (Abbildung 10 auf Seite 36). Dieses Pre-Repositorium dient als Wartepositon für Versionen, die durch den letzten Stage erfolgreich gelaufen sind. Dadurch ist die Automation von Umgebung N nur noch dafür verantwortlich, die neueste Version an die Automation von Umgebung $N+1$ zu vermerken. Die Entscheidung, die neueste vermerkte Version tatsächlich zu verwenden, liegt nun bei Umgebung $N+1$ mit dem Setzen des Repositoriums der Umgebung auf das Pre-Repositorium. Die Weitergabe erfolgt wie folgt :
Die Automation, die Version X auf Umgebung N in Betrieb nimmt und testet, setzt das Pre-Repositorium von Umgebung $N+1$ auf X um, wenn sie erfolgreich war.
Wenn die Automation aus Umgebung $N+1$ startet, setzt sie als erstes das Repositorium von Umgebung $N+1$ auf das Pre-Repositorium von Umgebung $N+1$ und rollt anschließend die Applikation in Version X auf Umgebung $N+1$ aus. Da das Verweisen eines virtuellen Repositoriums auf ein anderes virtuelles Repositorium transitiv wirkt, zeigt im Anschluß das virtuelle Repositorium von Umgebung $N+1$ nicht mehr auf das virtuelle Repositorium von Umgebung N, sondern auf das Versionsrepositorium mit X.

Nebenläufigkeitsprobleme Aus der Problemlösung in Sperren von Repositorien auf Seite 25 und der Tatsache, dass bei ImmobilienScout24 die Erstellungsprozesse der Legacy-Applikation parallel laufen, kann ein Konsistenzproblem entstehen, wenn ein Buildagent den anderen überholt.
Dies wird im Anhang durch ein problematisches Szenario verdeutlicht : Problemszenario bei parallelen Instanzen der *Delivery Chain* auf Seite 50

Abzusehende Sicherheitsprobleme Die Einführung eines zentralen Repositoriumsystems für Programmcode bringt einige Sicherheitsrisiken mit sich. Ein gravierender Angriffsvektor wäre beispielsweise das Einschleusen von gefälschten Softwarepaketen in den *yum-repo-server*, z. B. könnte in das RSS-Paket eine Hintertür eingebaut werden, sodass bei der Inbetriebnahme der RSS-Funktionsgruppe statt nur dem RSS-Dienst auch eine extern zugängliche Administratorkonsole in Betrieb genommen wird.
Ein anderer diskutabel gefährlicher Angriffsvektor ist das Löschen von Repositorien (z. B. das Löschen des Konfigurationsrepositorium - siehe Konfigu-

rationsverwaltung auf Seite 6). Da der Erstellungsprozess von RPM-Paketen eindeutig reproduzierbar ist, hat eine Löschung keine unwiderrufliche Konsequenzen, allerdings kann die Neuerstellung einige Zeit in Anspruch nehmen und den Betrieb somit über einen längeren Zeitraum komplett lähmen.

3.2.6 Absprache mit externen Dienstleistern

Zu Beginn dieser Arbeit wurden alle zu migrierenden Server von einem externen Dienstleister zur Verfügung gestellt. Die Umstellung auf ein RPM-basiertes Verfahren benötigt einige tiefgreifende Änderungen. Beispielsweise ist die Installation von arbiträren Softwarepaketen nicht möglich - genau dies ist aber eine Voraussetzung für das neue Verfahren. Um eine Maschine auf das neue Verfahren umzustellen, muss diese vom externen Dienstleister also mit einigen Änderungen neu aufgesetzt werden. Die Herausforderungen dabei sind Folgende :

1. Kostenminimierung - der Dienstleister rechnet pro Änderung (Ticket) ab
2. Überzeugung - es ist abzusehen, dass der Dienstleister sich weigert, Maschinen ohne einen genauen Plan umzustellen. Grund dafür ist dass das neue Verfahren einen Großteil der Administration der Maschinen an ImmobilienScout24 übergibt

Kostenminimierung Es gilt, die Risiken gegen die Kosten abzuwägen. Eine Möglichkeit wäre, bei der Migration einer Funktionsgruppe alle Maschinen einer Umgebung in einem Ticket aufzulisten, sodass der Dienstleister diese umstellt. Somit wurden die Kosten minimiert. Damit ist allerdings ein großes Risiko verbunden, und insbesondere für die produktive Umgebung bedeutet dies ein Ausfall des Dienstes. Besser ist die iterative Migration von jeweils einem oder zwei Servern der Funktionsgruppe. Damit bleibt der Dienst erreichbar und kann die vorhandenen Anfragen ohne Lastspitzen abarbeiten. Natürlich ist die Zahl der auf einmal zu migrierenden Maschinen von der Gesamtanzahl der Maschinen in der Funktionsgruppe abhängig. Für die kritischste Umgebung, Produktion, existieren allerdings immer mindestens vier Server eines Typs, unter anderem aufgrund der Replikation der Server über zwei Standorte (Umgebungen auf Seite 5). Folglich können immer mindestens zwei Server auf einmal migriert werden.

Überzeugung Wider Erwartungen waren keine besondere Überzeugungsmaßnahmen nötig. Diese hätten sicherlich auf den Vorteilen des neuen Verfahrens basiert.

3.2.7 Modellierung der Delivery Chain mit *yum-repo-server*

Mit dem *yum-repo-server* lässt sich die Delivery Chain auf das komplette Staging DEV, TUV, PRO erweitern (Delivery Chain mit *yum-repo-server* [Rie12] auf Seite 37). Im ersten Schritt erzeugt ein Buildagent eine Version sowie ein Versionsrepositorium. Eine freie DEV-Umgebung wird ausgesucht, ihr virtuelles Repositorium DEV wird auf das Versionsrepositorium gesetzt und die Umgebung wird in Betrieb genommen (indem die Softwarepakete mit *YADT* aktualisiert werden) und getestet. Wurde die Version erfolgreich getestet, wird es mittels Pre-Repositorium (Weitergabe eines getesteten Version (X) von Umgebung N an Umgebung $N + 1$ mit Pre-Repositorium [Rie12] auf Seite 36) an den nächsten Stage (TUV) weitergegeben. Die TUV-Umgebung wird in Betrieb genommen, indem das virtuelle Repositorium TUV auf das Pre-Repositorium TUV gesetzt wird, und die Softwarepakete aktualisiert werden. Danach kann manuell und automatisch getestet werden. Wenn eine Freigabe durch die Qualitätssicherung erfolgt, kann das Pre-Repositorium für PRO auf das getestete Versionsrepositorium gesetzt werden. Um die Produktionsumgebung in Betrieb zu nehmen muss nunmehr das virtuelle Repositorium PRO auf das Pre-Repositorium PRO gesetzt werden. Eine Aktualisierung der Softwarepakete der Maschinen in PRO hat zur Folge dass die Version, welche zwingend durch alle Stages propagiert wurde, auf die Maschinen installiert wird.

3.3 Migration *RSS, PDF, TAT, RST, API*

Dieser Teil war der zeitintensivste, da er sich mit der eigentlichen Migration der Funktionsgruppen befasst.

3.3.1 Vorgehensweise

Die Migration verläuft theoretisch für eine Funktionsgruppe wie folgt.

Anpassung der Funktionsgruppe sodass in der *package*-Phase von *Maven* ein RPM erstellt wird Zurzeit werden bei der Kompilation Webarchive (*war*) erstellt. Diese sollen anschließend in einem RPM verpackt werden, welches die benötigten zusätzlichen Pakete mit *RPM_REQUIRES* erfordert.

Erstellen der Konfigurationen für die Maschinen der Funktionsgruppe in *DEV, TUV, PRO* In den *DEV*-Umgebungen werden zurzeit aus Kostengründen alle Funktionsgruppen der Applikation auf einen Server in Betrieb genommen. Diese Systeme werden *devbas*-Systeme genannt. Das neue RPM-Verfahren bietet die Möglichkeit (und den Zwang), die Funktionsgruppen nun endlich auch in *DEV* auf separate Server zu verlagern. Damit wird die *DEV*-Umgebung den höheren Staging-Umgebungen näher, sodass

speziellere Fehler einfacher gefunden werden können. Die Konfiguration aller RPM-Maschinen muss erstellt werden. Dafür wird die bestehende Konfiguration als Vorlage genommen. In *DEV* müssen dabei viele Anpassungen vorgenommen werden, da die Dienste nun nicht mehr alle auf dem gleichen Server laufen. Für *TUV, PRO* ist die Konfiguration einfacher, denn dort sind bereits dedizierte Server für die Funktionsgruppen vorhanden. Damit kann die Konfiguration ohne allzuviele Anpassungen übernommen werden. Eine besondere Herausforderung ist dabei, die Konfiguration so einfach, generisch und minimal wie möglich zu halten.

Migration der Maschinen in *DEV, TUV, PRO* Um dies zu realisieren muss bei der Migration der Funktionsgruppe für jeden *devbas* eine virtuelle Maschine erstellt werden. Im Gegensatz dazu müssen in *TUV, PRO* lediglich die schon vorhandenen Maschinen als RPM-fähige Systeme neu aufgesetzt werden.

Entfernen der Überreste des alten Verfahrens Sobald das neue Verfahren für die Funktionsgruppe bis *PRO* propagiert worden ist, gibt es keinen Grund mehr, die *deployment units* zu erstellen.

Abweichungen Natürlich treten je nach Funktionsgruppe unterschiedliche zusätzliche Probleme auf, die gelöst werden müssen, allerdings ist die Vorgehensweise grundsätzlich für alle Funktionsgruppen gleich. Folglich wird lediglich die Migration von *RSS*, welche die erste migrierte Funktionsgruppe war, ausführlich vorgestellt. Für die restlichen Funktionsgruppen werden lediglich die besonderen Schwierigkeiten erläutert.

3.3.2 Migration der Funktionsgruppe *RSS*

Erstellung eines RPMs für *RSS* Mit *Maven* wird in der *package*-Phase ein Webarchiv erstellt. Dieses muss zu einem RPM-Paket zusammengesetzt werden. Besonders zu beachten ist, dass das Ausliefern eines herkömmlichen Webarchivs mit RPM keine elegante Lösung ist, viel besser ist es ein sogenanntes explodiertes Webarchiv in das RPM zu legen. Damit befasst sich ein ausführlicher Abschnitt im Anhang, vgl. Von Webarchiven und Explosionen auf Seite 49. Die Kurzversion ist, dass bei einem nicht-explodiertem Webarchiv der *tomcat*-Container die Dateien selbst entkomprimiert, sodass die ausgepackten Dateien nicht mehr von RPM verfolgt werden können.
Eine *ANT*-Direktive entpackt in der *package*-Phase das erstellte Webarchiv, um ein explodiertes Webarchiv zu erstellen. Es gibt bereits eine *Maven*-Direktive die ein solches Archiv erstellt (*mvn war:exploded*), allerdings fehlen dort Dateien (vgl. Inkonsistente explodierte Webarchive von *Maven* auf Seite 50).

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-antrun-plugin</artifactId>
  <version>1.7</version>
  <executions>
    <execution>
    <id>unpack-war</id>
    <goals>
      <goal>run</goal>
    </goals>
    <phase>package</phase>
    <configuration>
      <target>
        <unzip
          src="${project.build.directory}/${project.
              build.finalName}.war"
          dest="${project.build.directory}/${project.
              build.finalName}"
          overwrite="true"/>
        </target>
    </configuration>
    </execution>
```

Nun kann das explodierte Webarchiv in das RPM abgelegt werden.

```
<plugin>
  <groupId>de.is24.common</groupId>
  <artifactId>rpm-deploy-webapp-plugin</artifactId>
  <executions>
    <execution>
    <id>create-rpm</id>
    <goals>
      <goal>raw-rpm</goal>
    </goals>
    <phase>package</phase>
    </execution>
  </executions>
  <configuration>
    <skip>${skipRpmBuild}</skip>
    <name>is24-${project.build.finalName}</name>
    <exploded>true</exploded>
    <additionalDependencies>${tomcat_rpm_name}</
        additionalDependencies>
    <overrideNameVersionRelease>true</
        overrideNameVersionRelease>
    <version>${rpm.final.version}</version>
    <release>${rpm.release}</release>
    <mappings>
```

```
      <mapping>
      <targetDir>${tomcat_path}/tomcat/webapps/
         feedService</targetDir>
      <source>${project.build.directory}/${project.
         build.finalName}</source>
      </mapping>
    </mappings>
    <postInstallScript>if service tomcat6 status ;
       then service tomcat6 restart ; fi</
       postInstallScript>
  </configuration>
</plugin>
```

Als zusätzliche Änderung an der Applikation ist noch das Hinzufügen eines *tomcat*-Start- und Statusskript nötig. Damit kann nach der Inbetriebnahme geprüft werden ob die Applikation richtig geladen wurde.

Konfiguration Zur Konfiguration der *RSS*-Server kann hauptsächlich die Ebene *typ* verwendet werden. In die *RPM_REQUIRES*-Variable wird neben der *RSS*-Applikation auch das von mir erstellte Paket mit virtuellen Repositorien erfordert (vgl. Inbetriebnahme mit *yum-repo-server* auf Seite 24). Ohne diese Abhängigkeit wäre das Paket mit der *RSS*-Applikation nicht auflösbar, da das entsprechende Repositorium fehlen würde.

Konfiguration für *YADT* Für *YADT* muss eine Beschreibung der benötigten Dienste und Abhängigkeiten auf jedem verwalteten Server vorliegen. Des geschieht über die Datei *yadt.services*. Diese ist für *RSS* einfach und sieht in *DEV* folgendermaßen aus :

```
- httpd:
    needs_services: [tomcat]
    is_frontservice: true
- tomcat:
```

Der *HTTP*-Hintergrunddienst *httpd* erfordert den *tomcat*-Dienst, um gestartet zu werden. Der *tomcat*-Dienst hat selbst keine Abhängigkeiten, sodass *YADT* bei Starten der Dienste erst den *tomcat*-Dienst, dann den *httpd*-Dienst starten wird. Beim Stoppen wird erst *httpd*, dann *tomcat* gestoppt. In *TUV* und *PRO* hingegen ist *httpd* aufgrund des Lastverteilers kein Vordergrunddienst (*is_frontservice*). Somit verändert sich die Konfiguration zu

```
- loadbalancer:
    needs_services: [nagios]
    is_frontservice: true
    class: is24yadtshell.services.LoadbalancerService
    loadbalancer_clusters: [$LB_CLUSTER$]
    status_max_tries: $STATUS_MAX_TRIES_LB$
```

```
- nagios:
    needs_services: [httpd]
    class: is24yadtshell.services.
       NagiosNotificationsService
    status_max_tries: $STATUS_MAX_TRIES_NAGIOS$

- httpd:
    needs_services: [tomcat]

- tomcat:
```

Durch die Abhängigkeiten wird *YADT* bei der Inbetriebnahme des Servers zuerst den *loadbalancer*-Dienst stoppen. Dabei wird natürlich nicht der Lastverteiler selbst gestoppt, sondern der *RSS*-Server wird aus dem Lastverteiler-Pool genommen. Damit werden Anfragen nicht mehr an diesen Server geschickt, sondern an andere *RSS*-Server aus dem Lastverteiler-Pool. Danach wird der *nagios*-Dienst abgeschaltet - dabei handelt es sich um ein Überwachungsserver, der regelmäßig die ihm bekannten Server auf Funktion überprüft und gegebenenfalls einen Notruf tätigt oder Warnungen verschickt. Dies soll vermieden werden denn der Server ist nicht ausgefallen, die Dienste werden lediglich zur Inbetriebnahme einer neuen Applikationsversion neu gestartet. Danach können die Dienste *httpd* und *tomcat* auch heruntergefahren werden. Das Starten nach der Paketaktualisierung erfolgt dann auch in umgekehrter Reihenfolge.

Staging in *DEV, TUV, PRO* In *DEV* existieren bisher keine *RSS*-Server. Diese werden folglich als virtuelle Maschinen erstellt. Dadurch dass die Konfiguration bereits vorgenommen wurde, sind die neuen Server nach dem Starten sofort einsatzbereit - die Applikation mitsamt Konfiguration und allen Abhängigkeiten ist bereits installiert (vgl. Konfigurationsverwaltung mit YADT / mit Puppet für neue Maschinen [Rie12] auf Seite 13). Zur automatisierten Inbetriebnahme muss noch als letzter Schritt ein *YADT target* angelegt werden (vgl. Konfiguration von *YADT* auf Seite 45). Die *YADT-targets* enthalten jeweils nur einen Servernamen, denn die *RSS*-Server sind zurzeit die einzigen von *YADT* überwachten Server. Es gibt pro *devbas* ein neues *YADT-target*. Die Inbetriebnahme ist nun für *DEV* automatisch möglich. Für *TUV,PRO* ist noch die Erstellung eines *YADT-target* nötig - ein einfacher Schritt. Der Automatismus wird in den CI-Server, *Teamcity*, eingepflegt. Dabei entsteht ein neuer Schritt in allen betroffenen Konfigurationen, der *deploy rpm-based systems with yadt* heißt. Davor wird mithilfe des *repoclient* das virtuelle Repositorium des betroffenen *YADT-target* geändert, sodass es auf die neu erstellten und hochgeladenen *RPM*-Pakete zeigt. Im neuen Schritt wird der *yadtshell-controller* (vgl. *YADT*-Dienst auf einem Bastionserver auf Seite 46) mit dem entsprechendem *YADT-target* aufgerufen. Dies bewirkt dass die Applikation auf den Systemen, welche im

YADT-target eingetragen sind, automatisch in Betrieb genommen wird. Für die *TUV*-Umgebungen werden vorhandene Server als RPM-fähige Maschinen neu aufgesetzt, indem ein *ticket* an den Dienstleister geschickt wird. Danach werden die vorhandenen automatischen Systemtests gegen die neuen Server ausgeführt. Bei *RSS* gibt es einen einzigen automatischen Test. Dies ist auf die Simplizität des Dienstes zurückzuführen. Zu diesem Zeitpunkt sind wir zuversichtlich, dass die Konfiguration korrekt ist, sodass die Server von der Qualitätssicherung getestet werden können. Für *PRO* erfolgt die Umstellung der Server außerhalb des wöchentlichen Veröffentlichungsprozesses (vgl. Testen und Staging der Inbetriebnahme auf Seite 19).

Entfernen der Überreste des alten Verfahrens In *DEV* wird noch der *RSS*-Dienst auf den *devbas*-Systemen in Betrieb genommen. Dies wird konfigurativ entfernt.

3.3.3 Migration *PDF*

Keine Besonderheiten.

3.3.4 Migration *TAT*

Die *TAT*-Funktionsgruppe ist komplexer als die vorherigen, da sie CORBA-Abhängigkeiten auf andere Funktionsgruppen hat. Ein Symptom dafür ist dass die *TAT*-Server nicht mehr richtig funktionnieren wenn die *APP*-Server kurzzeitig ausfallen, da die CORBA-Verbindung nicht automatisch wiederhergestellt wird. Dann ist ein Neustart des *TAT*-Dienstes nötig. Dieses Symptom tritt dann auf, wenn die Reihenfolge bei der Inbetriebnahme nicht beachtet wurde (vgl. Migrationsstrategie : Blätter zuerst [Rie12] auf Seite 13).

3.3.5 Migration *RST,API*

Auf den *API*-Servern werden zurzeit zwei Komponenten betrieben : eine REST-basierte Programmierschnittstelle, und eine auf XML-Fernaufrufen basierende Programmierschnittstelle. Im Zuge der Migration wurden beide Komponenten in separate Funktionsgruppen verlagert : *RST* und *API*. Neben einem erhöhten Konfigurationsaufwand tauchte noch ein Pfadproblem auf - die *RST/API*-Applikationen benutzen Pfade, die auf RPM-fähigen Systemen anders als auf nicht RPM-fähigen Systemen sind. Grund dafür ist dass die Einhängepunkte auf RPM-Systemen anders bedient werden. Als Übergangslösung wurde ein RPM-Paket erstellt welches symbolische Verknüpfungen erstellt. Sobald alle Systeme RPM-fähig sind kann das Paket dann verworfen werden.

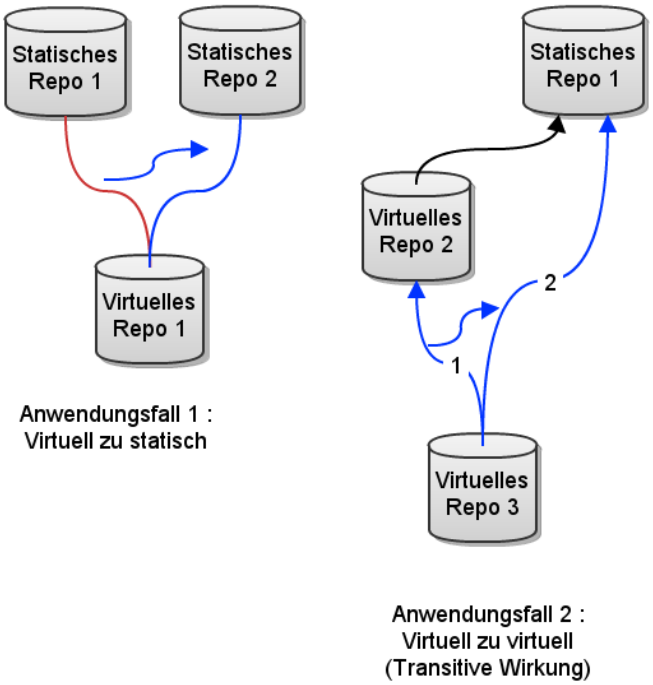

Abbildung 7: Virtuelle Repositorien [Rie12]

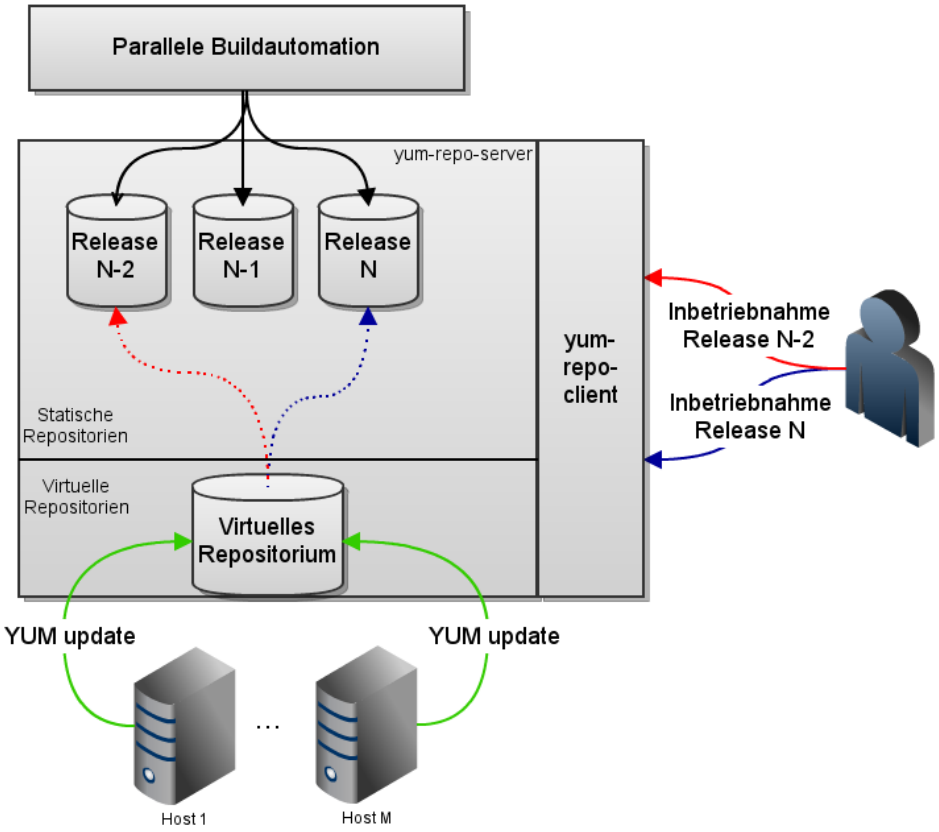

Abbildung 8: Virtuelle Repositorien für Stages [Rie12]

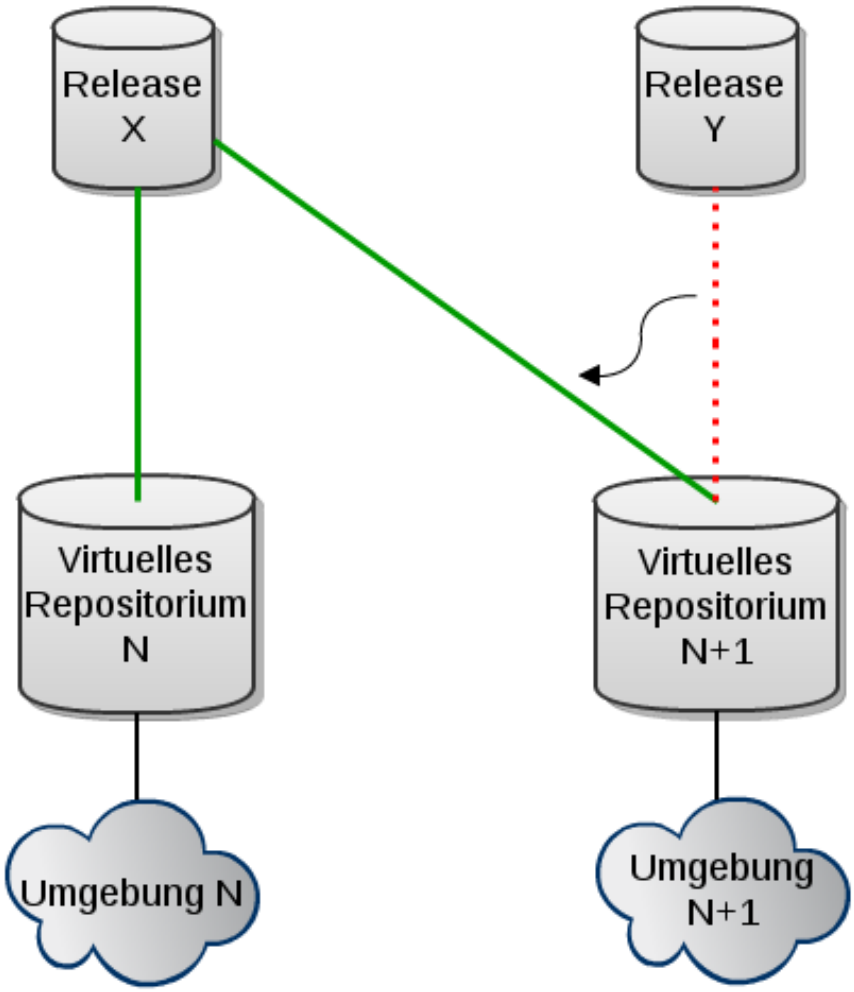

Abbildung 9: Weitergabe einer getesteten Version (X) von Umgebung N an Umgebung N+1 [Rie12]

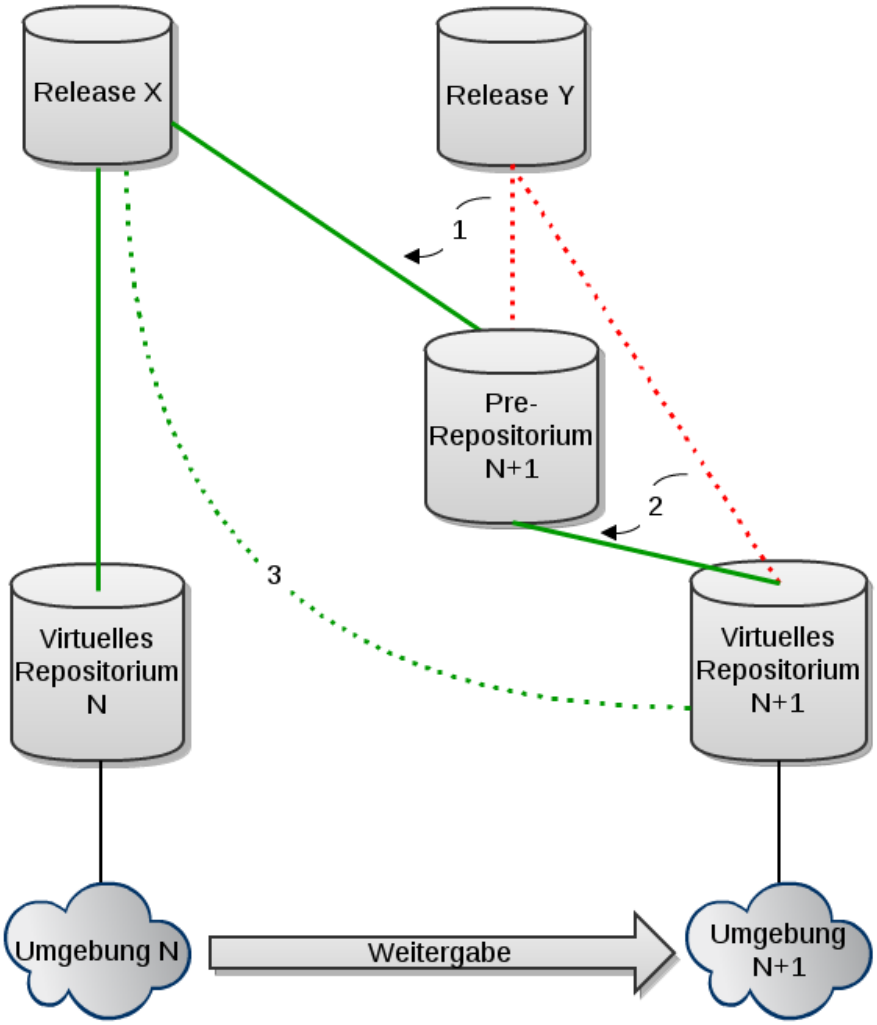

Abbildung 10: Weitergabe eines getesteten Version (X) von Umgebung N an Umgebung $N + 1$ mit Pre-Repositorium [Rie12]

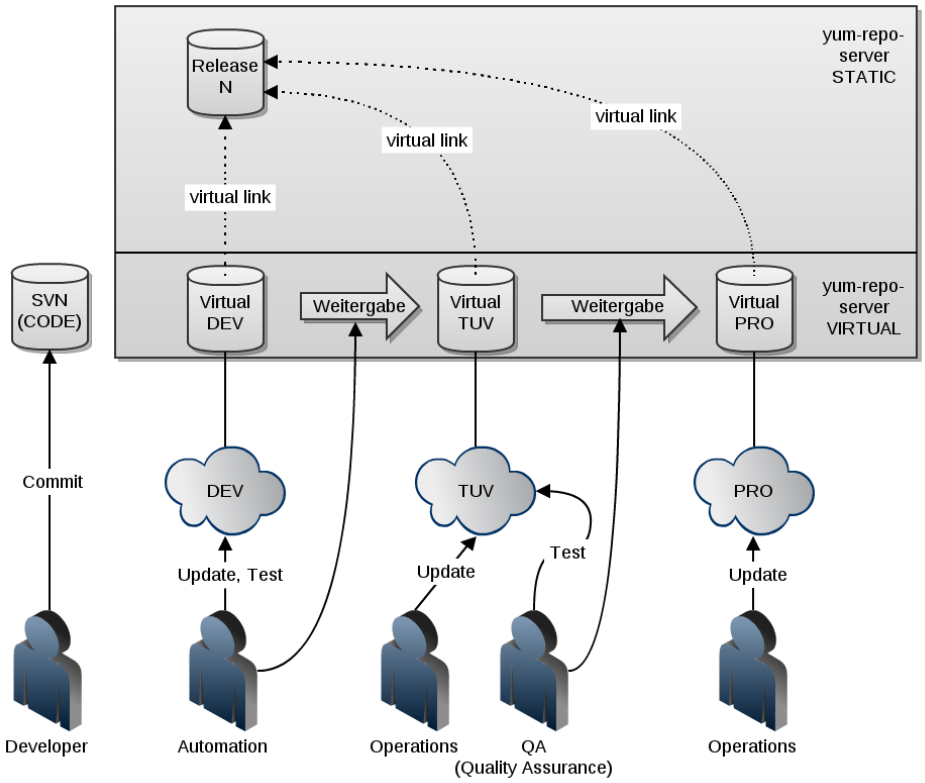

Abbildung 11: Delivery Chain mit *yum-repo-server* [Rie12]

4 Ergebnisse

4.1 Migrierte Funktionsgruppen

4.1.1 Erreichte Ziele

Folgende Funktionsgruppen wurden im Zuge dieser Arbeit vollständig auf das neue Verfahren umgestellt :

1. *RSS*
2. *PDF*
3. *TAT*
4. *API, RST*

Somit sind vier von neunzehn Funktionsgruppen umgestellt worden. Dies entspricht ca. einem Anteil von 20% der Applikation. Anhand von RPM/nicht-RPM (alle) [Imm12b] (Seite 39) ist zu sehen dass die Zahl der nicht-RPM-Server insgesamt stetig abnimmt.
Werden nur die Server der Legacy-Webapplikation betrachtet (RPM/nicht-RPM (legacy) [Imm12b] auf Seite 39), wird ersichtlich dass es dennoch viele zu migrierende Server gibt. Dies ist größtenteils auf die Funktionsgruppe *WEB* zurückzuführen, denn diese besteht aus über 80 Servern. Weiterhin fällt eine Diskrepanz zwischen beiden Kurven auf - die RPM-Kurve steigt viel schneller, als die nicht-RPM-Kurve sinkt. Dies ist vor allem den neuen virtuellen Servern in *DEV* zuzuschreiben (Erstellen der Konfigurationen für die Maschinen der Funktionsgruppe in *DEV, TUV, PRO* auf Seite 27).
In der Abbildung RPM/nicht-RPM (*RSS*) [Imm12b] (Seite 40) ist zu sehen dass die Funktionsgruppe vollständig migriert wurde - es gibt keinen einzigen nicht-RPM-Server mehr.
Abbildung RPM/nicht-RPM (*API*) [Imm12b] (Seite 40) ist etwas komplexer, da die Funktionsgruppe *API* in die Funktionsgruppen *API* und *RST* aufgeteilt wurde. Die Migration war zum 12. September noch nicht komplett abgeschlossen, sodass im Diagramm noch einige *API*-Server zu sehen sind.
TAT und *PDF* ergeben Kurven, die identisch zu *RSS* sind und wurden deswegen ausgelassen.

4.2 Probleme

Aufgetretene Probleme Es sind immer wieder kleinere Probleme mit der *Delivery Chain* aufgetreten. Beispielsweise führte eine Inkonsistenz zwischen den gepufferten Paketen von YUM und von *YADT* dazu, dass keine Aktualisierungen mehr stattfanden und die *Delivery Chain* stillstand. Keines dieser Probleme hatte Auswirkungen auf die Verfügbarkeit der ImmobilienScout24-Plattform.

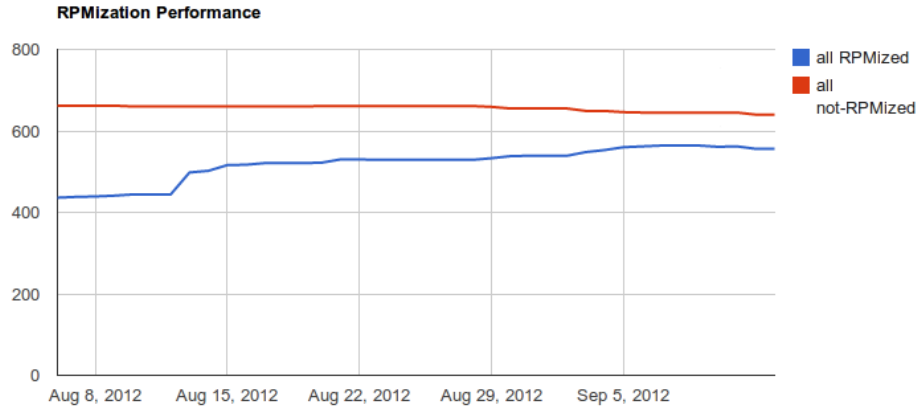

Abbildung 12: RPM/nicht-RPM (alle) [Imm12b]

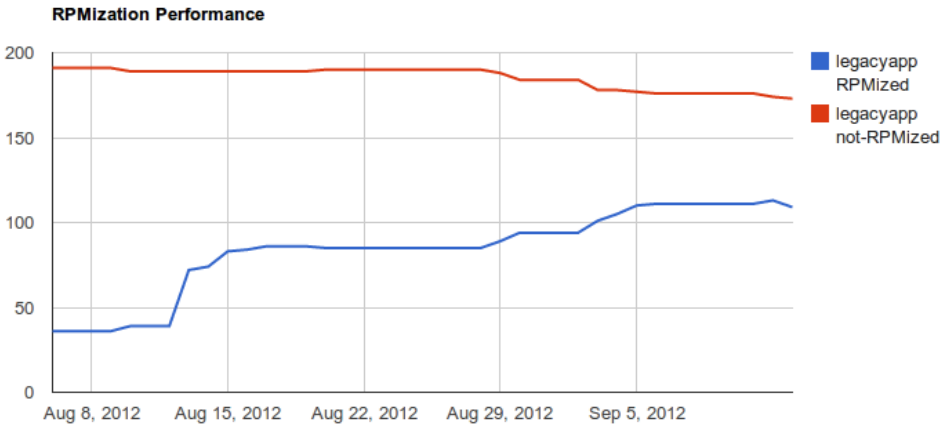

Abbildung 13: RPM/nicht-RPM (legacy) [Imm12b]

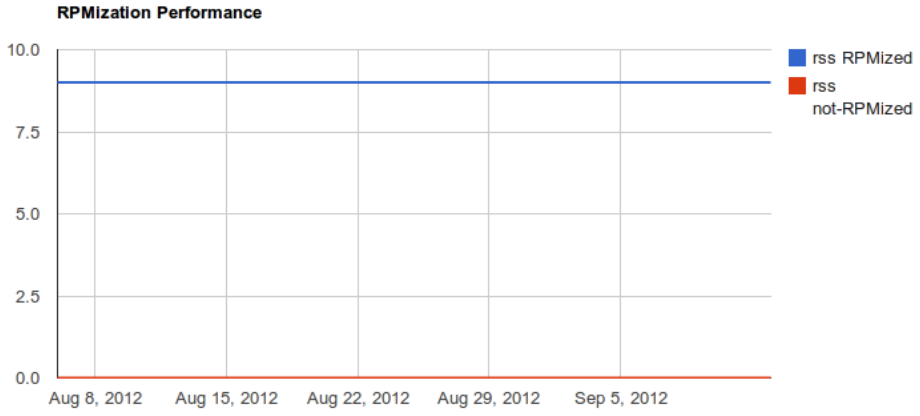

Abbildung 14: RPM/nicht-RPM (*RSS*) [Imm12b]

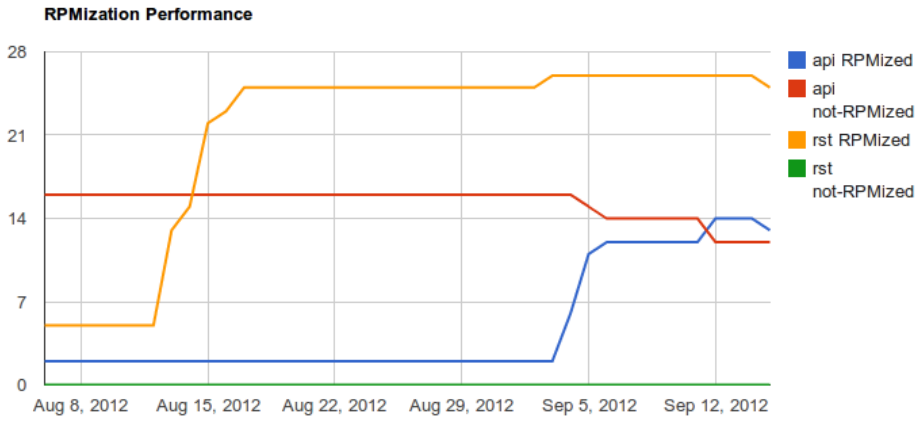

Abbildung 15: RPM/nicht-RPM (*API*) [Imm12b]

4.3 Zukunft

Mit der Automatisierung der Inbetriebnahme ist die Abschaffung der Veröffentlichungszyklen möglich. Es ist wahrscheinlich, dass die Qualitätssicherung von ImmobilienScout24 in absehbarer Zeit selbst auswählt, wann eine neue Version manuell getestet und freigegeben wird. Mit den bereits migrierten Funktionsgruppen hat sich ein routinierter, pragmatischer Prozess herauskristallisiert, sodass selbst die komplexe Funktionsgruppe *WEB* als nächstes Migrationsziel möglich ist. Ein mit RPM aufgesetzter *WEB*-Server existiert sogar bereits. Durch die Arbeit im Team und mit rotierenden Entwicklern hat sich das Wissen innerhalb der Firma verteilt, sodass die Akzeptanz des neuen Verfahrens groß ist.

5 Ausblick

5.1 Grenzen der Automation

5.1.1 Vorteile von RPM-basierter Inbetriebnahme

Trennung der Paketquellen Mit RPM lassen sich alle Änderungen am Zielsystem durch Paketnamen zusammenfassen. Der einzige Unterschied zwischen einer Betriebssystemaktualisierung, einer neuen Konfiguration und einer neuen Version der Applikation ist die Herkunft des Paketes. Dies kann z. B. ein Repositorium mit Konfigurationspaketen, ein offizielles Linux-Distributionsrepositorium, oder ein Versionsrepositorium sein.
Es sind also keine zusätzlichen Schritte mehr nötig um Betriebssystemaktualisierungen durchzuführen, da die Installation von Paketen Teil des Inbetriebnahmeverfahrens ist. Trotzdem sind die Betriebssystempakete von den Konfigurations- und Applikationspaketen getrennt.

Verwaltung der Abhängigkeiten Mit dem *RPM_PROVIDES* und *RPM_REQUIRES* von RPM lassen sich die Abhängigkeiten einer Applikation sehr gut modellieren. Durch die Metainformationen der Pakete sind diese Abhängigkeiten persistiert. Sofern alle Änderungen am System mit RPM durchgeführt werden ist ein Konfigurations-RPM hinreichend, um den Zustand des kompletten Systems reproduzierbar wiederherzustellen.

Einfachheit, Zentralität Durch die im RPM-Paket enthaltenen Skripte sind alle Schritte bei der Inbetriebnahme zentral verwaltet und Teil der Installation und Aktualisierung des Pakets. Mit der RPM-Paketverwaltung sieht der Aktualisierungsprozess für eine *tomcat*-Webapplikation wie folgt aus :

```
yum upgrade myapp
```

Ohne Paketverwaltung würde der Prozess wie folgt aussehen :

```
service tomcat6 stop
rm -rf /foo/bar/tomcat/work/ /foo/bar/tomcat/temp /foo
   /bar/tomcat/webapps/myappv1
cp /share/myappv2 /foo/bar/tomcat/webapps
service tomcat6 start
```

Nachvollziehbarkeit Mit

```
rpm -qi
```

ist es einfach zu überprüfen ob die Applikation installiert ist oder nicht, bzw. wann sie installiert oder aktualisiert wurde oder welche Version gerade vorhanden ist. Gerade mit Werkzeugen wie multiplextes SSH kann dann innerhalb von wenigen Sekunden festgestellt werden ob alle Server auf dem gleichen Stand sind. Falls nicht, liefert der Befehl Auskunft darüber wer für Änderungen verantwortlich ist. Der Befehl

```
rpm -qf FILE
```

liefert Auskunft darüber, welches Paket für die Datei *FILE* verantwortlich ist. Dies ist besonders in Kombination mit *exploded war*-Applikationen nützlich (Von Webarchiven und Explosionen auf Seite 49). Damit sind Daten nicht nur Artefakte von unbekannten Transaktionen, sondern es kann genau nachvollzogen werden wo eine Datei herkommt.

Erkennen von unerwünschten Änderungen Mit dem Befehl

```
rpm --verify PACKET
```

werden die von *PACKET* installierten Dateien mithilfe ihrer Prüfsumme, Größe und Berechtigungen, mit den Metadaten des Paketes *PACKET* verglichen [ME00]. So können manuelle Änderungen an den Applikationen sehr einfach verfolgt werden.

Schneller Überblick über die verpackte Applikation Mit dem Befehl

```
rpm -ql PACKET
```

kann überprüft werden, welche Dateien das installierte RPM enthält. Für ein nicht installiertes RPM kann

```
rpm -qpl PACKET
```

verwendet werden.

Zurückrollen bei Problemen Funktionniert die Applikation nicht wie erwartet kann schnell eine frühere Version installiert werden :

```
yum downgrade PACKET[optional:version angeben]
```

Einfache, zugängliche Prozesse für Entwickler und Administratoren Mit Softwarepaketen ist ersichtlich wo die Applikation installiert wird. Damit können Entwickler sehen, was ihre Applikation auf einem Server bewirkt, ohne Zugang zum Server zu haben. Dies ist ein wichtiger Schritt zum besseren Verständnis zwischen Entwicklern und Administratoren. Das Installieren der Applikation auf einem beliebigen eigenen Rechner ist auch einfach, denn aufgrund der RPM-Abhängigkeiten ist die Installation des Paketes ausreichend. Die von *YADT* und dem *config-rpm-maker* vereinfachten Konfigurationen sind zugänglicher und dienen auch der besseren Zusammenarbeit zwischen allen involvierten Menschen.

Standardisierung bei mehreren Applikationen Das RPM-Verfahren ermöglicht es viele Standards festzulegen. Beispielsweise kann sichergestellt werden, dass unterschiedliche Webapplikationen immer unter ähnlichen Pfaden abgelegt werden z. B. $/foo/bar/tomcat/webapps/X$ für eine Webapplikation X.

5.1.2 Nachteile von Inbetriebnahme mit RPM

Bei einigen Änderungen ist es nicht intuitiv, RPM zu benutzen. In einigen Fällen kann die Änderung so gestaltet werden, dass sie durch RPM ausgeliefert werden kann. Beispielsweise musste auf einem Server ein Sicherheitszertifikat zum *Java keystore* hinzugefügt werden. Das Zertifikat ist zwar eine Datei, allerdings soll es nicht abgelegt werden, sondern mit dem Kommandozeilentool *keytool* importiert werden. Danach ist das Zertifikat nicht mehr wichtig, sodass es nicht sinnvoll ist das Zertifikat in das RPM zu legen, und mit einem Post-Installationsskript dann zu importieren. RPM wäre dann für die Zertifikatsdatei verantwortlich, obwohl die Datei nach dem Importieren auch gelöscht werden könnte. Das Ausliefern der kompletten *keystore*-Datei ist auch nicht sinnvoll. Dann würden alle anderen Zertifikate überschrieben werden. Als Lösung habe ich ein RPM erstellt, welches statt dem Zertifikat ein Skript ausliefert. Das Skript unterstützt die drei Operationen *Installation, Überprüfung* und *Deinstallation*. Dabei wird das Zertifikat heruntergeladen und importiert (*install*), überprüft ob das Zertifikat importiert (*check*) oder aus dem *keystore* gelöscht (*uninstall*) wurde. Das RPM liefert lediglich dieses Skript aus, und prüft nach der Installation mit *check* ob das Zertifikat schon vorhanden ist. Falls ja, wird es erst gelöscht (*uninstall*) und dann installiert (*install*). RPM übernimmt die Verantwortung für das Skript, sodass die Änderung persistiert und sichtbar ist. In anderen Fällen

lässt sich allerdings keine elegante Lösung mit RPM umsetzen. Ferner ist die Betriebssystemabhängigkeit problematisch. Auch Pakete ohne Zielarchitektur (*noarch*), die beispielsweise nur plattformunabhängige *Java*-Programme enthalten, können nicht automatisch auf *Windows*-Betriebssysteme installiert werden. Dabei sind die vielen konfliktierenden Paketverwaltungswerkzeuge (*RPM, DEB, Homebrew, MSI, ...*) sicherlich ein Teil des Problems. Dafür existiert bisher keine Lösung.
Das neue Verfahren und *Continuous Delivery* allgemein werfen einige Sicherheitsrelevante Fragen auf. Da Paketverwaltungswerkzeuge standardmäßig immer nur das aktuellste Paket installieren, kann ein eingeschleustes Softwarepaket mit einer sehr hohen Versionsnummer schnell verbreitet werden. Die Möglichkeit, *Staging* für Pakete von Drittanbietern, wie offizielle Linux-Pakete, durchzuführen, ist sicherlich ein großer Vorteil, allerdings können Sicherheitslücken bei einem verkürzten Veröffentlichungszyklus nicht zuverlässig ausgeschlossen werden. Eine bessere Automation von Sicherheitstests wäre ein großer Schritt, um dies zu gewährleisten.
Zuletzt führen Aktualisierungen von Softwarepaketen zu Problemen, wenn es keinen linearen Aktualisierungspfad gibt, beispielsweise von *Java*-6 auf *Java*-7. Es ist nicht möglich im gleichen Schritt das Paket zu löschen und das neuere Paket zu installieren.

6 Anhang

6.1 Technologien

6.1.1 *tomcat*

Tomcat ist ein quelloffener Applikationsserver, welcher die *Java Servlet* und *JavaServer Pages*-Technologien implementiert [Fou12a].

6.1.2 *YADT*

Funktionale Übersicht von *YADT* 'Mittels YADT ist es möglich die komplette Landschaft eines Rechenzentrums in einem simplen und verständlichen YAML-Format abzubilden. Das YADT-Modell besteht aus Softwarepaketen und ihrer Konfiguration, sowie aus den Abhängigkeiten der Softwarepakete und aus einfachen Konventionen. Mit diesen Informationen kann YADT beliebige Software und Konfigurationen auf einer beliebigen Anzahl an Maschinen verteilen. Selbstverständlich kann YADT auch betriebssystemaktualisierungen durchführen sodass betriebssystemspezifische Aktualisierungen und Softwareaktualisierungen getrennt stattfinden können. YADT garantiert dass sowohl Aktualisierungen als auch Neustarts von Diensten innerhalb des kompletten Rechenzentrums in der korrekten Reihenfolge geschehen. Um dies zu erreichen analysiert YADT die Abhängigkeiten von Diensten und Softwarepaketen, und gruppiert abhängige Dienste und Maschinen als sogenannte 'chunks'. Im Anschluß werden Aktualisierungen sequenziell über alle chunks ausgerollt, sodass sich Hochverfügbarkeit auch bei der Inbetriebnahme von neuen Applikationsversionen herstellen lässt.' (freie Übersetzung nach [Imm12c]).

Konfiguration von *YADT* Bei der Verwendung von *YADT* werden die verwalteten Server in zusammengehörige Gruppen zusammengefasst (*targets*). Ein *target* besteht aus einer Datei, in der die Namen aller relevanten Server aufgezählt sind, wie beispielsweise [MG12]:

```
name: spameggs
log-dir: logs

hosts:
  - hostname1.spam.eggs
  - hostname2.spam.eggs
```

Dies definiert das *target* mit Namen spameggs. Dazu gehören die Server *hostname1.spam.eggs* und *hostname2.spam.eggs*. Alle von *YADT* durchgeführten Aktionen sind auf ein bestimmtes *target* bezogen. Wird z. B. eine Aktualisierung auf dem *target* spameggs durchgeführt, so sind die Server *hostname1* und *hostname2* davon betroffen. Zusätzlich wird für jeden Server eine Datei mit Namen *yadt.services* erstellt. Diese beschreibt den Zusammenhang

zwischen den Diensten auf dem Server. Diese kann beispielsweise wie folgt aussehen :

```
- dienst_1:
  needs_services: [dienst_2, dienst_3]
  is_frontservice: true
- dienst_2:
- dienst_3:
  needs_services: [dienst_4]
- dienst_4:
```

Dabei werden vier unterschiedliche Dienste deklariert (dienst_1 bis 4). Der Ausdruck *needs_services[x,y,..,z]* bedeutet dass der Dienst nur dann gestartet werden darf, wenn die Dienste *x,y,..,z* bereits aktiv sind. Umgekehrt darf der Dienst nur dann gestoppt werden, wenn die Dienste *x,y,..,z* bereits gestoppt sind. Der Ausdruck *is_frontservice: true* kennzeichnet den äußersten Dienst, der als erstes gestoppt und als letztes aktiviert wird. Im Kontext von Webapplikationen ist dieser äußere Dienst üblicherweise die Anwesenheit des Servers im Pool eines Lastverteilers.

***YADT*-Dienst auf einem Bastionserver** Mit dem *yadtshell-controller* und *yadtshell-receiver* wird den durch *YADT* induzierten Sicherheitsproblemen (Sicherheitsprobleme auf Seite 22) entgegengewirkt. Dabei wird *YADT* zu einem Dienst (*yadtshell-receiver*), der auf einem Bastionhost aktiv ist. Mit dem *yadtshell-controller* können Klienten *YADT*-Befehle ausführen, indem sie ein *target* und einen Befehl angeben. Das Kommando wird an einen Dienst (*yadt-broadcaster*) geschickt, der das Kommando an alle registrierten *yadtshell-receiver* weiterleitet. Falls einer der *yadtshell-receiver* für das *target* zuständig ist, antwortet dieser mit Ereignismeldungen an den *yadt-broadcaster*. Dieser leitet die Ereignisse an den Klienten weiter. Somit können Maschinen ohne passwortlosen SSH-Zugang *YADT* aufrufen. Lediglich der Bastionhost darf sich passwortlos mit SSH auf die betroffenen Server anmelden. Dies ist sicherheitstechnisch gesehen kein Problem, da die Kaperung des Bastionhosts bedeutet dass die höchste Sicherheit bereits durchbrochen wurde.

6.1.3 *config-rpm-maker*

Mit dem *config-rpm-maker* wird die Konfiguration auf den Ebenen *all, loc, typ, loctyp, host* abgebildet. Jede dieser Ebenen ist spezifischer als die vorherige. So ist z. B. *loctyp* spezifischer als *loc*, denn wenn sowohl die Position im Staging als auch der Servertyp bekannt sind, lassen sich mehr Parameter festlegen als wenn nur bekannt ist was die Position im Staging ist. Als konkretes Beispiel könnte für *loctyp/devfoo* der Datenbankserver für die *foo*-Server eingetragen werden. Ein weiteres Werkzeug zur Vereinfachung der Konfiguration sind die Variablen. Es gibt vordefinierte Variablen wie z. B.

HOSTNAME welche den Namen des Servers speichert. Es können auch eigene Variablen erstellt werden (auf jeder beliebigen Ebene). Die Variablen werden auch von spezifischeren Ebenen überschrieben, sodass eine globale Konfigurationsdatei aus *all* Variablen enthalten kann, die dann z. B. erst in *loctyp* definiert werden und für einige Server in *host* überschrieben werden. Variablen werden durch das Schema

```
@@@VARIABLE_NAME@@@
```

gekennzeichnet. Beispielsweise ist Folgendes möglich :

```
### Beispiel-konfigurationsdatei : loctyp/devfoo/etc/
    sysconfig/foo.txt ###
foo.datenbank=@@@FOO_DB_URL@@@
foo.greeting=hello foo!
```

```
### loctyp/devfoo/etc/VARIABLES/FOO_DB_URL ###
@@@HOSTNAME@@@.dbcluster.intranet
```

```
### host/devfoo01/etc/VARIABLES/FOO_DB_URL ###
mal_was_anderes.dbcluster.intranet
```

Daraus folgt für *devfoo01* die Datei */etc/sysconfig/foo.txt*

```
foo.datenbank=mal_was_anderes.dbcluster.intranet
foo.greeting=hello foo!
```

und für alle anderen *devfoo*-Server (wie z. B. *devfoo02*)

```
foo.datenbank=devfoo02.dbcluster.intranet
foo.greeting=hello foo!
```

6.2 Glossar

6.2.1 Inbetriebnahme

Die Inbetriebnahme einer Applikation besteht aus folgenden Schritten :

- Stoppen der Applikation, wenn diese nicht im laufendem Betrieb aktualisiert werden kann
- Verteilen und Installieren/Aktualisieren der Applikation auf den/die relevanten Server
- Konfiguration der Applikation
- Starten der Dienste in der richtigen Reihenfolge damit sich die Applikation in einem konsistenten Zustand befindet

6.2.2 VCS

VCS steht für den englischen Begriff Version Control System bzw. Versionsverwaltungssystem. Dabei handelt es sich um ein System welches die Versionierung von Dateien (üblicherweise Quellcode) ermöglicht. Bekannte Lösungen sind *Subversion, Git* und *Mercurial*.

6.2.3 Repositorium

Im Rahmen dieser Arbeit wird Repositorium stets für Softwarerepositorium verwendet. Es handelt sich hierbei um ein Verzeichnis welches durch das Netzwerk zugänglich ist, in dem Softwarepakete abgelegt werden können. Das Installieren bzw. Aktualisieren der Software aus dem Repositorium übernimmt ein Paketverwaltungsprogramm. Ein konkretes Beispiel für ein Repositorium ist ein *Yellowdog Updater, modified* (YUM) Repositorium, welches RPM-Softwarepakete enthält. Mit dem YUM-Paketmanager lässt sich Software aus einem YUM-Repositorium installieren bzw. aktualisieren (frei nach [Wik12b]).

6.2.4 CI-Server, Build/Buildagent

Ein Continuous Integration Server (CI-Server) ist ein Server der die kontinuierliche Integration von Quellcodeänderungen ermöglicht. Er erkennt Änderungen der Entwickler am Quellcode und überprüft diese um binäre Aussagen über die Korrektheit der Änderungen (richtig (grün), falsch (rot)) zu treffen. Dafür wird bei jeder Änderung ein untergeordneter Server (Buildagent) ausgewählt und vom CI-Server beauftragt, den geänderten Quellcode zu kompilieren und automatisch zu testen (Build). Je größer die Zahl der Entwickler (d. h. je größer die Anzahl an Änderungen pro Zeiteinheit), desto mehr Buildagents müssen zur Verfügung stehen um zu gewährleisten dass Änderungen möglichst einzeln überprüft werden können.

6.2.5 Stage, Staging

Um die Qualität von Software zu sichern, muss sich diese in unterschiedlichen Prüfungsumgebungen (*Stages*) in einer festgelegten Reihenfolge bestehen. In jeder dieser Prüfungsumgebungen wird die Applikation in Betrieb genommen und getestet. Üblicherweise ist jede Prüfungsumgebung feiner und näher an der produktiven Umgebung der Applikation als die vorherige. *Staging* ist das Propagieren von Änderungen durch alle Prüfungsumgebungen. Dies ist besonders vorteilhaft da grobe Fehler in den ersten Prüfungsumgebungen gefunden werden können, auch wenn sich diese aus Kostengründen stark von der Produktivumgebung unterscheiden. Die produktionsnahen Umgebungen sind teurer, können dafür spezifischere Fehler aufdecken.

6.2.6 Legacy

'Der Begriff Altsystem (engl. legacy system) bezeichnet in der Informatik eine etablierte, historisch gewachsene Anwendung im Bereich Unternehmenssoftware. Legacy ist hierbei das englische Wort für Vermächtnis, Hinterlassenschaft, Erbschaft, auch Altlast. Innerhalb der Anwendungslandschaft eines Unternehmens sind es zumeist großrechnerbasierte Individualentwicklungen, die sich oft durch unzureichende Dokumentation, veraltete Betriebs- und Entwicklungsumgebungen, zahlreiche Schnittstellen und hohe Komplexität auszeichnen. Die dort anzutreffende zentrale Daten- und Funktionshaltung galt seit der Client/Server-Euphorie als überholt. Diese Merkmale sind der Grund dafür, dass sich die Ablösung solcher Systeme oft deutlich über ein erwünschtes Lebensende hinauszieht. Sowohl in wirtschaftlichen Aufschwung- wie in Abschwungphasen wird oft repriorisiert, um die mit einer Ablösung verbundenen hohen Ausfallrisiken bzw. Umstellkosten zu umgehen, zumal der bloße Ersatz eines Legacy-Systems nicht mit einem direkten Mehrwert, sondern meist nur mit der Einsparung von kalkulatorischen Kosten (Kosten für temporären oder dauerhaften Ausfall) oder Opportunitätskosten (entgangene Umsätze wegen begrenzter Leistungsfähigkeit des Legacy-Systems) verbunden ist. Grundsätzliches Problem bei der Ablösung von Legacy-Systemen ist der gewachsene Funktionsumfang. Auch wenn recht häufig ein weiträumiger Ersatz durch mächtige Standardsoftware stattfindet, verbleiben meist nicht abgedeckte Zusatzfunktionen und Schnittstellen. Das sind manchmal goldene Aschenbecher, öfter jedoch Alleinstellungsmerkmale der gewachsenen und über Jahrzehnte entwickelten Software, über die Standardsoftware nicht unbedingt verfügt. Oft ist eine Runderneuerung der Systeme schon deshalb schwierig, weil sie über die Historie hinweg nicht konsistent beschrieben wurden, zum Beispiel mit Anforderungen, Anwendungsfällen etc.'([Wik12a])

6.3 Zusätzliche Informationen

6.3.1 Von Webarchiven und Explosionen

Motivation Ein Webarchiv (*foo.war*-Datei) ist ein *ZIP*-komprimiertes Verzeichnis mit den nötigen Dateien, um eine Webapplikation zu betreiben. Bei Verwendung des *tomcat*-Container wird das Webarchiv in das *webapps*-Verzeichnis abgelegt. Dieses wird vom *tomcat* bemerkt und er entkomprimiert das Webarchiv nach *webapps/foo* wobei *foo* der Name der *war*-Datei ist. Der Vorgang wird *autodeploy* genannt. Wenn das Webarchiv aus einem RPM-Paket kommt, so kennt RPM die entkomprimierten Dateien nicht. Führt man beispielsweise den Befehl

```
rpm -qf webapps/foo
```

aus, lautet die Antwort von RPM dass diese Datei keinem RPM-Paket gehört. Das heißt auch, dass der Ordner *webapps/foo* bei der Deinstallation des RPMs nicht gelöscht wird.
Um dieses Problem elegant zu lösen, kann ein explodiertes Webarchiv verwendet werden. Dabei handelt es sich um das entkomprimierte Webarchiv. Dies kann beispielsweise mit dem *Ant*-Befehl *unzip* erreicht werden. Wenn dieser Ordner nun von einem RPM in das *webapps*-Verzeichnis von *Tomcat* abgelegt wird, so

- gehören alle Dateien ab *webapps/foo* dem RPM-Paket sodass mit

  ```
  rpm -qf
  ```

  sofort ersichtlich wird, welches Paket für die Dateien verantwortlich ist
- werden die Dateien beim Deinstallieren des Pakets weggeräumt

Schlussendlich ist es besser, *tomcat*-Webapplikationen in der explodierten Form mit RPM zu verpacken.

Inkonsistente explodierte Webarchive von *Maven* Ein mit

```
mvn war:exploded
```

explodiertes Webarchiv ist nicht identisch zu einem selbst entkomprimierten, mit

```
mvn war
```

erstelltem Webarchiv. Es fehlen im *META_INF* Ordner der von *Maven* explodierten Version die Dateien *pom.xml* und *pom.properties*. Diese sind für die Applikation nicht erforderlich. Allerdings ist es praktisch diese Dateien mitzuliefern. Damit kann z. B. schnell die Version der Applikation ermittelt werden. Um die Konsistenz zu erzwingen wird also erst ein herkömmliches Webarchiv erstellt. Dieses wird dann zu einem explodiertem Webarchiv entkomprimiert.

6.3.2 Problemszenario bei parallelen Instanzen der *Delivery Chain*

Zum Zeitpunkt t_A Minuten checkt Entwickler A Code in das VCS ein. Erstellungsprozess A beginnt sofort und dauert d_A Minuten.
Zum Zeitpunkt t_B Minuten checkt Entwickler B Code in das VCS ein, wobei $t_B > t_A$. Erstellungsprozess B beginnt sofort und dauert d_B Minuten.
Wenn zusätzlich noch $t_B - t_A < d_A - d_B$ gilt, dann folgt $t_A + d_A > t_B + d_B$, d. h. Erstellungsprozess B wird als erstes fertig und setzt das nächste Pre-Repositorium auf den Codestand B. Erst danach wird Erstellungsprozess A fertig, und setzt eben genanntes Pre-Repositorium auf den Codestand A. Das ist falsch, denn der Codestand A ist älter. Als Lösung dafür muss vor

jedem Umhängen geprüft werden, ob die VCS-Revision tatsächlich höher ist. Ist das nicht der Fall, muss der Vorgang abgebrochen werden.

Literatur

[Bai00] Edward C. Bailey. Using RPM to Upgrade Packages. http://www.rpm.org/max-rpm/ch-rpm-upgrade.html, 2000. [Internet-Quelle; abgerufen am 30-August-2012].

[Cod10] Codehaus. RPM Maven Plugin. http://mojo.codehaus.org/rpm-maven-plugin/rpm-mojo.html, May 2010. [Internet-Quelle; abgerufen am 05-August-2012].

[CT12] ImmobilienScout24 CLD-Taskforce. ImmobilienScout24/yum-repo-server - GitHub. https://github.com/ImmobilienScout24/yum-repo-server, July 2012.

[CW00] Alistair Cockburn and Laurie Williams. The Costs and Benefits of Pair Programming. In *eXtreme Programming and Flexible Processes in Software Engineering XP2000*, pages 223–247. Addison-Wesley, 2000.

[CXC+] Bin Chen, Nong Xiao, Zhiping Cai, Zhiying Wang, and Ji Wang. Fast, On-demand Software Deployment with Lightweight, Independent Virtual Disk Images.

[Dea07] Alan Dearle. Software deployment, past, present and future. In *2007 Future of Software Engineering*, FOSE '07, pages 269–284, Washington, DC, USA, 2007. IEEE Computer Society.

[Fou12a] The Apache Software Foundation. Apache Tomcat - Welcome! http://tomcat.apache.org/, 2012. [Internet-Quelle; abgerufen am 29-August-2012].

[Fou12b] The Apache Software Foundation. Guide to using the release plugin. http://maven.apache.org/guides/mini/guide-releasing.html, August 2012. [Internet-Quelle; abgerufen am 29-August-2012].

[Gad10] Larry Gadea. Murder: Fast datacenter code deploys using BitTorrent. http://engineering.twitter.com/2010/07/murder-fast-datacenter-code-deploys.html, July 2010. [Internet-Quelle; abgerufen am 01-September-2012].

[Har02] John Hart. An analysis of rpm validation drift. In *In Proceedings of the USENIX LISA Conference*, pages 155–166. USENIX Association, 2002.

[Imm12a] ImmobilienScout24. Informationen zum Unternehmen ImmobilienScout24. http://www.immobilienscout24.de/de/ueberuns/

presseservice/unternehmensinformationen/index.jsp, February 2012. [Internet-Quelle; abgerufen am 29-August-2012].

[Imm12b] ImmobilienScout24. Interne ImmobilienScout24 Dokumentation, 2012.

[Imm12c] ImmobilienScout24. YADT - an Augmented Deployment Tool. http://code.google.com/p/yadt/w/list, July 2012. [Internet-Quelle; abgerufen am 28-July-2012].

[JH10] D. Farley J. Humble. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* Addison-Wesley, 2010.

[ME00] Erik Troan Marc Ewing, Jeff Johnson. rpm. http://www.rpm.org/max-rpm-snapshot/rpm.8.html, 2000. [Internet-Quelle; abgerufen am 14-September-2012].

[MG12] Arne Hilmann Michael Gruber. yadt/yadtshell Wiki. https://github.com/yadt/yadtshell/wiki, September 2012. [Internet-Quelle; abgerufen am 10-September-2012].

[Rie12] Maximilien Riehl. Interne ImmobilienScout24 Dokumentation, die von mir im Rahmen dieser Arbeit erstellt wurde, 2012.

[S.L12] DEBUGMODEON S.L. Masterbranch, Maximilien Riehl. https://masterbranch.com/mriehl, September 2012. [Internet-Quelle; abgerufen am 06-September-2012].

[TZG+] Hongbo Tian, Xiaoyi Zhao, Zhixing Gao, Taiqiang Lv, and Xiaoshe Dong. A Novel Software Deployment Method based on Installation Packages.

[Vid12] Seth Vidal. yum. http://yum.baseurl.org/, August 2012. [Internet-Quelle; abgerufen am 29-August-2012].

[Wik12a] Wikipedia. Altsystem. http://de.wikipedia.org/wiki/Altsystem, May 2012. [Internet-Quelle; abgerufen am 05-August-2012].

[Wik12b] Wikipedia. Repository. http://de.wikipedia.org/wiki/Repository, July 2012. [Internet-Quelle; abgerufen am 05-August-2012].

[Zha] Xu Zhang. Research and Application of Automated Software Delivery System. Master's thesis, Zhejiang University.