Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Virtuelle Maschinen und Umgebungen als Hilfmittel zum Testen web-basierter Python Projekte

Bachelor-Thesis
zur Erlangung des akademischen Grades B.Sc.

Virak Oum
1901577

Hochschule für Angewandte Wissenschaften Hamburg
Fakultät Design, Medien und Information
Departement Medientechnik

Erstprüfer: Prof. Nils Martini
Zweitprüfer: Prof. Andreas Plaß
Hamburg, 31.08.2015

Inhaltsverzeichnis

Zusammenfassung

Diese Bachelorarbeit befasst sich mit virtuellen Maschinen und Umgebungen als Hilfsmittel zum Testen von web-basierten Python Projekten. Hierzu werden die Grundlagen der Servervirtualisierung näher gebracht, Vor- und Nachteile der Technologie herausgestellt und Vagrant, ein Tool zur Erstellung und Verwaltung von virtuellen Maschinen, näher beschrieben. Im Anschluss wird die pythonspezifische Problematik der Paketabhängigkeiten erklärt und Tools zur Vermeidung vorgestellt. Danach soll anhand eines beispielhaften Szenarios, gezeigt werden wie ein Problem mit den Abhängigkeiten entsteht und mit Hilfe von isolierten Umgebungen gelöst werden kann. Abschließend werden die Ergebnisse der Arbeit zusammenfassend erklärt und ein Fazit abgegeben.

Abstract

This bachelor thesis deals with virtual machines and environments as a tool for testing web-based Python projects. To this end, the basics of server virtualization will be explained, advantages and disadvantages of the virtualization technology will be pointed out and Vagrant, a tool for creating and managing virtual machines, will be described in more detail. Following, the python specific problems of package dependencies will be explained and tools for prevention will be presented. Hereafter an exemplary scenario will be used to show how dependency problems can arise and how they can be solved by using isolated environments. Finally, a summary will show the results and a conclusion will be made.

1 Einleitung

1.1 Motivation

Die IT-Branche wächst und gedeiht noch immer in einem rasanten Tempo. Laut einem Prognose der Bitkom wird der deutsche Markt im Jahr 2015 im Software und IT-Dienstleistungsbereich voraussichtlich um 2,4 % wachsen (vgl. Bitkom 2014).
Mit dem Marktwachstum entstehen auch neue Technologien, Arbeitsprozesse und Methoden, an die man sich als Entwickler oder Entwicklering anpassen kann, um somit ein effizienteres, aber auch schöneres Arbeiten zu gewährleisten. Gerade als Entwickler oder Entwicklerin von webbasierten Anwendungen ist es möglich sich diesen Faktoren zu bedienen und einen Nutzen daraus zu schöpfen. Wenn man sich in die Lage der ersten Webseiten-EntwicklerInnen hinein versetzt, so war es damals noch relativ einfach den Arbeitsprozess zu planen und auch das Ergebnis der Arbeit war relativ schnell kontrollierbar, denn statische Webseiten wurden hauptsächlich in der Auszeichnungssprache HTML geschrieben und konnten auch lokal mit Hilfe eines Browsers dargestellt, auf die Funktionsfähigkeit geprüft und anschließend auf den Webserver übertragen werden. Jedoch entwickeln sich die IT-Branche und deren Technologien stetig weiter und damit verbunden steigen auch die Anforderungen an EntwicklerInnen sich dem anzupassen. Frameworks und Webserver mit mehr Darstellungsformen und Funktionen machen es möglich webbasierte Applikationen in Programmiersprachen wie Java, Ruby oder Python zu programmieren und BenutzerInnen komplexere Anwendungen anzubieten. Aus unternehmerischer Sicht werden solche webbasierten Projekte oft auch im Team entwickelt, da viele Prozesse der Entwicklung in kleine Teilaufgaben aufgeteilt werden, um so eine höhere Effizienz durch paralleles Arbeiten zu erzielen, aber auch die Wartbarkeit von Funktionen modularer zu gestalten.
Jedoch bringen neue Technologien und Arbeitsprozesse nicht nur Vorteile mit sich. So können unterschiedliche Konfigurationen eines Webservers dazu führen, dass programmierte Funktionalitäten nicht einwandfrei und im schlimmsten Fall gar nicht funktionieren. Es kann aber auch schon bei der generellen Programmierung der webbasierten Anwendungen zu Konflikten kommen. Gründe hierfür können zum Beispiel die Entwicklung auf unterschiedlichen Betriebssystemen der Teammitglieder und eine damit verbundene nicht gewährleistete Homogenität der Entwicklungsumgebung, aber auch durch die Installation oder das Updaten von Bibliotheken entstehende Konflikte in der bereitgestellten Systemumgebung sein.

1.2 Ziel der Arbeit

Im Rahmen der vorliegenden Arbeit möchte ich mich hauptsächlich auf die Verwendung von virtuellen Maschinen und Umgebungen als Hilfsmittel zum Testen von webbasierten Python Projekten fokussieren, und so versuchen, die vorher genannten Probleme zu lösen oder von vornherein zu vermeiden. Die Arbeit soll außerdem die Realisation von webbasierten Python Projekten erleichtern, indem die grundlegende Funktionsweise virtueller Maschinen und die Nutzung von isolierten Umgebungen vermittelt wird.

1.3 Aufbau der Arbeit

Dieses Kapitel soll einen Einblick in das Thema, die Motivation und das Ziel der Arbeit geben.

Im zweiten Kapitel wird dem Leser bzw. der Leserin eine Einführung in die Virtualisierung gegeben, um die Funktionsweise von virtuellen Maschinen und den damit verbundenen Vor- bzw. Nachteilen zu verstehen.

Das dritte Kapitel soll zeigen, wie man mit Hilfe von Vagrant Möglichkeiten schafft, schnell und einfach virtualisierte Entwicklungs- und Produktionsumgebungen zu erstellen und zu nutzen.

Das vierte Kapitel stellt einen Überblick und Vergleich von pythonspezifischen Paketen und Werkzeugen zur Realisierung von webbasierten Applikationen dar.

Im fünften Kapitel wird Anhand eines kleinen Szenarios der praktische Nutzen einer virtuellen Umgebung vermittelt werden.

Kapitel sechs stellt eine Zusammenfassung der in der Arbeit herausgestellten Ergebnisse zusammen und abschließend wird ein Fazit gezogen.

2 Virtuelle Maschinen

Als virtuelle Maschinen werden Software-Container bezeichnet, die mit Hilfe eines Virtual Machine Monitors (VMM) oder Hypervisors, physische Ressourcen eines Servers oder Hostsytems in einem isoliertem System nutzen können und somit als eigenständige Systeme arbeiten. Auf einem Hostsystem können mehrere virtuelle Maschinen parallel laufen, ohne dass sie voneinander wissen.
Zur besseren Verständlichkeit zeigt Abbildung 1 den schematischen Aufbau eines Systems mit einer virtuellen Maschine. Die unterste Schicht sind die physischen Ressourcen. Sie stellt das Fundament dar. Darüber liegt eine Kontrollschicht zur Steuerung und Verwaltung der Ressourcen. Ganz oben auf liegt der logische Software-Container.

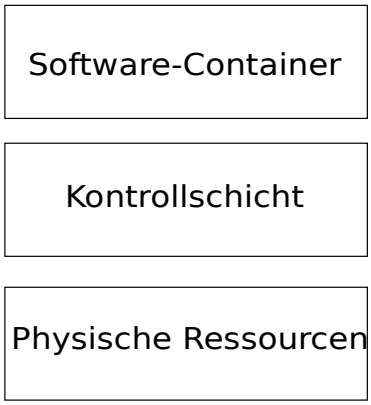

Abbildung 1: Schematischer Aufbau

Um ein grundlegendes Verständnis für virtuelle Maschinen zu bekommen und die damit verbundenen Einsatzmöglichkeiten im Rahmen der Arbeit zu vermitteln, soll in diesem Kapitel ein kurzer Einblick in die Grundlagen der Virtualisierung gegeben werden. Aufgrund der Komplexität des Gebiets der Virtualisierung soll hierbei hauptsächlich die Servervirtualisierung betrachtet werden.

2.1 Grundlagen der Virtualisierung

Virtualisierung ist eine Technologie, um physische Komponenten eines Computersystems in ein logisches Software-Objekt zu abstrahieren (vgl. Portnoy 2012: 20). Durch die Abstraktion der Komponente als solch ein logisches Objekt, bietet es die Möglichkeit die Ressource besser zu

nutzen, da diese von der physischen Hardware getrennt ist. So lassen sich beispielsweise Speicherressourcen in einem Storage Area Network (SAN) effizienter nutzen. SANs gewährleisten eine höhere Verfügbarkeit und die Ressourcen können flexibler gestaltet werden, da sich logische Objekte leichter und schneller manipulieren lassen. Weitere Komponenten eines Computersystems (wie z.B. Netzwerk – vLAN, Prozessor – vCPU, Arbeitsspeicher – vRAM) lassen sich ebenso abstrahieren und können somit von den Anpassungsmöglichkeiten logischer Objekte profitieren.
Im Rahmen der Arbeit soll – wie bereits vorher erwähnt – aber nur auf die Servervirtualisierung, sprich die Abstraktion ganzer Computer, die virtuellen Maschinen, eingegangen werden. (Vgl. Portnoy 2012: 20)

2.1.1 Kriterien eines Virtual Machine Monitors / Hypervisors

Der Virtual Machine Monitor, im späteren Kontext auch als Hypervisor bezeichnet und im folgenden als VMM abgekürzt, ist eine Softwareschicht mit der Aufgabe, die physischen Ressourcen dem logischen Objekt zur Verfügung zu stellen. Hierzu wurde von Goldberg und Popek (1974) (zitiert nach Portnoy 2012: 21) herausgestellt, dass der VMM drei essentielle Eigenschaften aufweisen muss, damit ein System virtualisierbar ist und auch eine virtuelle Maschine alle Ressourcen so nutzen kann, als wären diese physisch in dem System eingebaut.
Demnach muss ein VMM folgende Eigenschaften aufweisen, um sich als solcher zu definieren:

> „- Treue (Fidelity): Die Umgebung, die der VMM für die virtuelle Maschine erstellt, ist im Wesentlichen identisch mit der ursprünglichen Hardware (der physischen Maschine).
> - Isolation oder Sicherheit (Isolation or Safety): Der VMM muss die komplette Kontrolle über die System-Ressourcen verfügen.
> - Performance (Performance): Zwischen der Performance der virtuellen Maschine und ihrem physischen Gegenstück sollte es nur einen geringen oder gar keinen Unterschied geben.“ (Portnoy 2012: 21f.)

Die meisten VMMs verfügen über die ersten beiden Eigenschaften und gelten somit laut Definition als VMM. Wenn ein VMM zusätzlich auch das Performance-Kriterium erfüllt, so gilt dieser als effizienter VMM (vgl. Portnoy 2012: 22). In Abbildung 2 wird veranschaulicht, wann die Softwareschicht die Eigenschaften aufweist, um Virtualisierung zu gewährleisten, sprich, wann die beiden Kriterien als VMM erfüllt sind und wann sie als ein effizienter VMM aufgefasst werden

kann. Am linken Rand der Darstellung werden die verfügbaren physischen Ressourcen, wie Arbeitsspeicher, CPU, Festplatten und Netzwerkkarte des Hostsystems dargestellt. Rechts daneben liegt der VMM/Hypervisor. Dieser ist weiter unterteilt in die drei Kriterien Treue, Sicherheit und Performance. Am rechten Rand ist eine virtuelle Maschine dargestellt mit den äquivalenten virtuellen Ressourcen, wie vRAM, vCPU, vDisk und vLAN. Pfeile stellen den Fluss der durchgereichten Ressourcen dar. Man sieht anhand der Pfeile und der dazwischen liegenden Schicht, dass keine direkte Verbindung von den physischen Ressourcen zu der virtuellen Maschine besteht, d.h. der VMM erfüllt das Kriterium der Isolation/Sicherheit, da dieser über die komplette Kontrolle über die Systemressourcen verfügt. Da die virtuelle Maschine die gleichen Ressourcen wie auf dem physischen Host durchgereicht bekommen hat, ist zudem das Kriterium der Treue erfüllt und somit ist auch eine Virtualisierung möglich.

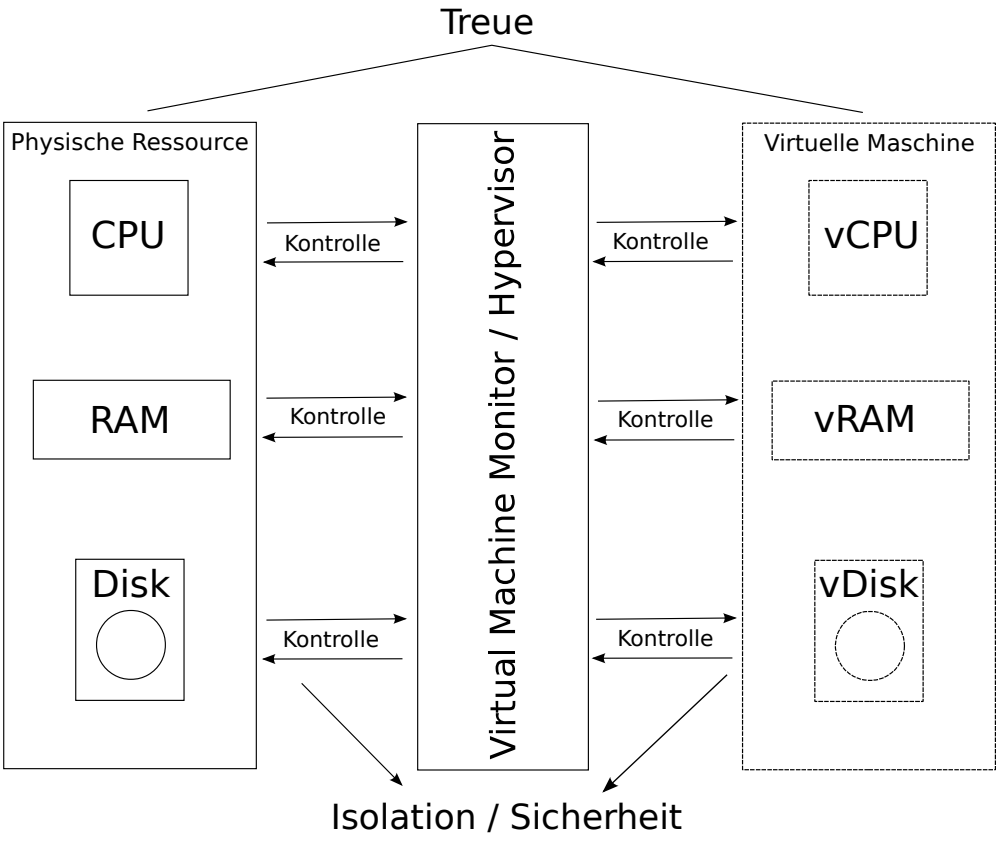

Abbildung 2: Aufbau und Kriteriumserfüllung für die Virtualisierung

2.1.2 Arten von Hypervisoren

Typ-1 Hypervisor

Ein Typ-1 Hypervisor kommt ohne Betriebssystem auf dem Server aus und wird direkt auf die Hardware ohne weitere Vermittlungsschicht aufgesetzt. Daher bezeichnet man diese Implementierung auch als Bare-Metal-Implementierung. Durch das direkte Kommunizieren mit den darunterliegenden Hardware-Ressourcen erreicht dieser Typ Hypervisor eine bessere Performance und kann daher auch als effizienter angesehen werden. Alle Operationen auf dem Gast-System werden direkt an die physische Ressource weitergereicht und alle Ereignisse betreffen nur den jeweiligen VM-Container. In Abbildung 3 sieht man, dass der Typ-1 Hypervisor direkt auf den Hardware-Ressourcen liegt und somit eine direkte Weiterleitung der I/O-Operationen der Hardware-Schicht (z.B. Lese- und Schreibzugriff auf die Festplatte) und der virtuellen Maschine durchgeführt werden kann.

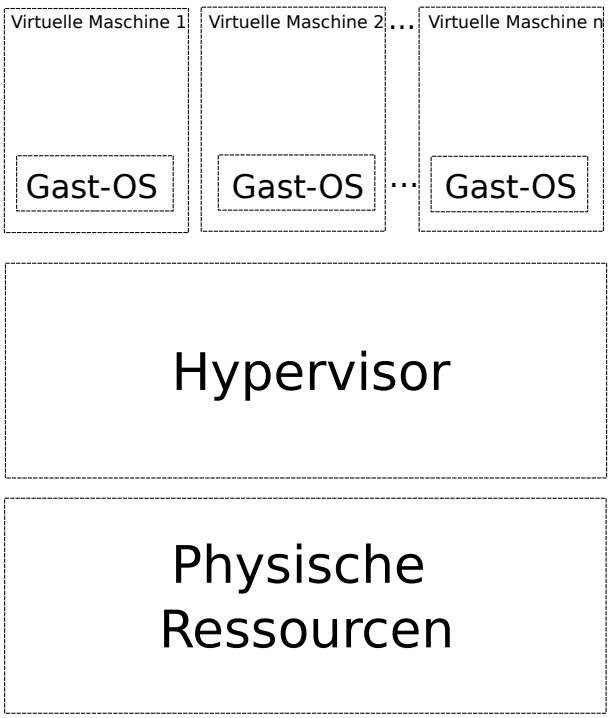

Abbildung 3: Aufbau eines Typ-1 Hypervisor

Typ-2 Hypervisor

Bei einem Typ-2 Hypervisor handelt es sich um eine Applikation, die auf einem bereits aufgesetzten Betriebssystem läuft. Hierbei werden die bereits vom Betriebssystem gehandhabten Hardware-Ressourcen an den Hypervisor herangereicht und dieser vermittelt dann die weiteren Operationen der VM. Der Vorteil hierbei ist, dass der Hypervisor nicht speziell auf die darunterliegende Hardware angepasst werden muss, da die meiste Konfigurationsarbeit bereits vom Betriebssystem übernommen wird. Auch durch diese zusätzliche Schicht gilt ein Typ-2 Hypervisor im Gegensatz zum Typ-1 nicht als so effizient was die Performance betrifft. Abbildung 4 zeigt ein Virtualisierungsmodell mit einem Typ-2 Hypervisor. Es ist klar erkennbar, dass zwischen der ganz unten liegenden Hardwareschicht des Hostsystems und der virtuellen Maschine noch zwei Schichten liegen, und zwar das Hostbetriebssystem und der Hypervisor selbst. Für I/O-Operationen bedeutet das, dass bei der Weiterreichung Daten- und Befehlsströme zunächst an den Hypervisor gehen, diese der richtigen Ressource des Betriebssystems zugeordnet werden müssen und dann erst die Operationen an die entsprechende Ressource weitergegeben werden. Der selbe Vorgang gilt auch umgekehrt, also wenn von der Hardware Datenströme in Richtung virtuelle Maschine geschickt werden.

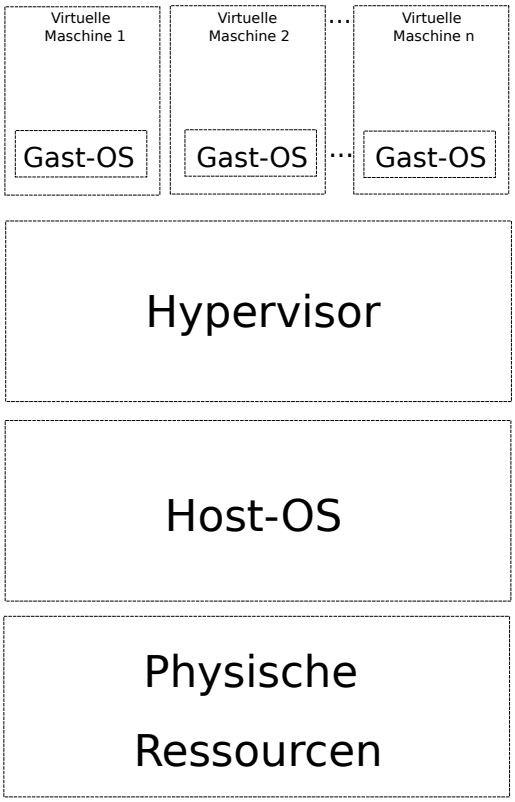

Abbildung 4: Aufbau eines Typ-2 Hypervisor

2.2 Anwendungsmöglichkeiten

Virtualisierung bzw. virtuelle Maschinen sind vielseitig einsetzbar. Im folgenden soll erörtert werden, welche Varianten der Servervirtualisierung es ermöglichen, virtuelle Maschinen in Zusammenhang mit der Entwicklung von webbasierten Python-Projekten und dem Testen auf produktionsnahen Umgebungen einzusetzen. Die folgenden Erläuterungen sollen dazu beitragen, die Frage zu klären, welche Variante der Virtualisierung am besten passt und welche Virtualisierungslösung daher verwendet werden kann. Dabei muss auch in Betracht gezogen werden, welche Anforderungen ein Software-Entwickler bzw. eine Software-Entwicklerin für webbasierte Projekte an solch eine virtuelle Maschine haben könnte und inwiefern sich diese in den Varianten finden lassen.

Anforderungen

Wenn es darum geht, webbasierte Applikationen bzw. größere Projekte zu realisieren, ist eine Sicherheit bezüglich der Systemwiederherstellung wichtig. Beim Testen noch nicht vollständiger Software kann es durchaus vorkommen, dass die Systemkonfigurationen neu angepasst werden müssen, wenn etwas schief geht. So ist es gut, wenn es eine Möglichkeit gibt, ähnlich wie in einem Versionskontrollsystem wie Git oder SVN, bei denen man auf letzte funktionierende Versionen zugreifen kann, um einen letzten stabilen Ausgangspunkt zu haben. Solch ein Versionskontrollsystem ist allerdings so nicht auf ganze virtuelle Maschinen anwendbar, aber inzwischen bieten Virtualisierungslösungen sogenannte Snapshots an und auch andere Sofwarebasiertelösungen bieten gute Ansätze.
Robuste, aber auch ohne Leistungseinbußen verwendbare Entwicklungsumgebung sind eine gute Grundlage, um angenehm zu arbeiten. Niemand möchte an einem instabilen System arbeiten, dass Programm- oder sogar Systemausfälle mit sich bringt. Zusätzlich wäre es von Vorteil, unterschiedliche Betriebssysteme verwenden zu können. Da verschiedene Benutzer bzw. Benutzerinnen unterschiedliche Betriebssysteme verwenden, könnte sich dieses von dem Entwicklungssystem unterscheiden und schlimmstenfalls könnte die Applikation nicht funktionieren. Aber auch aus Entwickler- bzw. Entwicklerinnensicht kann man durch die Auswahl an Betriebssystemen Code von Entwicklern und Entwicklerinnen testen, die auf einem anderen System gearbeitet haben, vorausgesetzt man hat die gleichen Systemumgebungsvariablen gesetzt und weitere Abhängigkeiten sind vorhanden. Außerdem sollten auch die Hilfswerkzeuge wie ein

Texteditor, Netzwerkschnittstellen und Systembibliotheken ohne weiteres nutzbar sein.
Die virtuelle Maschine sollte nicht zu viel Aufwand erfordern, da man sich mehr auf die Applikationsentwicklung fokussieren möchte. Je mehr Zeit es braucht, eine virtuelle Maschine lauffähig zu machen, desto weniger bleibt für die eigentliche Programmierarbeit.

2.3 Varianten

2.3.1 Vollständige Virtualisierung

Wenn man an die Servervirtualisierung denkt, so meint man eigentlich eine vollständige Virtualisierung. Dabei wird einer virtuellen Maschine eine eigene Hardwareumgebung vorgetäuscht. Aus Sicht der virtuellen Maschine wirkt dies, als wären diese Hardwarekomponenten in dem virtuellen System vorhanden. Eigentlich verwendet die virtuelle Maschine hierbei aber nur ein Teil der Hardware-Ressourcen. Die Kontrolle der Ressourcen übernimmt ein Typ-2 Hypervisor in Form einer Applikation, die auf einem Hostbetriebssystem läuft. Dadurch, dass zwischen der Hypervisor-Schicht und den Ressourcen ein Hostbetriebssystem liegt und der Hypervisor sich den Treibern zur Ansteuerung der Hardware bedienen kann, ist zwar die Einrichtung der virtuellen Umgebung einfacher gestaltet als bei einem Typ-1 Hypervisor, allerdings muss eine I/O-Operation zwei Schichten durchqueren, um einen Befehl an die addressierte Hardware zu schicken. Dies bringt den Nachteil mit sich, dass leichte Einbußen bezüglich der Performance entstehen können. (Vgl. ITWissen 2015a)
Ein weiterer Nachteil könnte bei parallelem Betrieb mehrerer Gastsysteme die Überbelastung der verwendbaren Ressourcen sein, die dann auch das Hostsystem langsam und instabil machen können.
Ein Vorteil bei der vollständigen Virtualisierung ist, dass keine spezielle Anpassung der Gastbetriebssysteme notwendig ist. Aber auch eine flexible Zusammenstellung der virtuellen Komponenten ist möglich, da aufgrund der vollständigen Virtualisierung alle Komponenten der physischen Hardware virtualisiert werden können.

Softwarelösungen für die vollständige Virtualisierung sind beispielsweise VMware Player, VMware Workstation, Oracle VirtualBox, Microsoft Virtual Server.

2.3.2 Prozessorunterstützte vollständige Virtualisierung

Es gibt spezielle, für die Virtualisierungstechnologie entwickelte moderne Prozessoren, die durch Befehlssätze erweitert wurden, um Virtualisierungsoperationen zu optimieren und die vollständige Virtualisierung zu unterstützen. Dies hat zur Folge dass Teilaufgaben des Hypervisors im Prozessor gehandhabt werden und diesen somit entlasten. Dadurch sind die Hypervisoren schlanker und Performancezuwächse können durch den geringeren Overhead erzielt werden, was auch durch die Aufteilung von Aufgaben unterstützt wird. (Vgl. ITWissen 2015b).
Die Prozessoren von Intel mit Unterstützung von Virtualisierung werden Intel VT genannt und sollen bei der Virtualisierung von CPU, Arbeitsspeicher und den Ein- und Ausgabeoperationen unterstützen (vgl. Intel). AMD beschreibt seine Prozessoren mit Virtualisierungsunterstützung wie folgt: „AMD Virtualization (AMD-V™) technology is a set of on-chip features that help you make better use of and improve the performance of your virtualization resources.“ (AMD 2014) Bei AMD werden die gleichen, auch von Intel genannten Komponenten unterstützt.
Demnach lassen sich folgende Vorteile erkennen: Zum Einen sind die Virtualisierungslösungen aufgrund der Auslagerung von Aufgaben auf die Prozessoren schlanker als herkömmlich. Zum Anderen können Gastbetriebssysteme auf die proprietären Treiber des Betriebssystems zugreifen und benötigen daher keine Modifikation.
Ein Nachteile ist eine starke Hardwareabhängikeit bei Verwendung von ausschließlich hardwarebasierten Virtualisierungslösungen, d.h. für die Verwendung von Hardewarevirtualisierungslösungen benötigt man bestimmte Prozessoren (AMD-V oder IntelVT).
Softwarelösungen, die prozessorunterstütze vollständige Virtualisierung anbieten, sind z.B. KVM oder QEMU, aber auch viele der Anbieter, die bei der vollständigen Virtualisierung genannt wurden, bieten inzwischen eine hardwarebasierte Lösung an.

2.3.3 Betriebssystemvirtualisierung

Betriebssytemvirtualisierung unterscheidet sich von den bisher genannten Virtualisierungsvarianten vor allem darin, dass kein Hypervisor zwischen dem virtuellen Betriebssystem und der Hardware sitzt, sondern das Betriebssystem selbst die Virtualisierung vornimmt. Dabei werden Betriebssysteminstanzen wie z.B. Kernel als Laufzeitumgebung in sogenannte Container isoliert erstellt und für die Virtualisierung verwendet (vgl. ITWissen 2015c). Die verwendeten Systemressourcen sind hierbei die gleichen, die das Betriebssystem selbst besitzt und verwaltet. Da

es sich eigentlich um das gleiche Betriebssystem handelt und nur eine bestimmte Laufzeitumgebung isoliert wird, ist es auch nur möglich, Systeme zu virtualisieren, die auch die selbe Laufzeitumgebung verwenden. Verwendet beispielsweise das Host-Betriebssystem einen Linux Kernel 3.16.0.43, so ist es auch nur möglich ein Gast-Betriebssystem zu virtualisieren, das den selben Kernel verwendet. Das bedeutet, dass man keine Möglichkeit hat ein virtualisiertes Windows auf einem Linux Host laufen zu lassen. Ebenso steht das Hostsystem in direktem Zusammenhang mit dem Gastsystem. Dies hat zur Folge, dass Änderungen am Host- auch das Gastsystem betreffen (vgl. Hoffmann 2010:18). Dafür bietet die Betriebssystemvirtualisierung die geringste Leistungseinbuße aufgrund der Tatsache, dass es sich grundsätzlich um das gleiche System handelt und nur bestimmte Laufzeitumgebungen isoliert werden. Auch Ressourcen lassen sich recht flexibel verwalten, denn diese müssen nicht fest gebunden werden, sondern lassen sich je nach Auslastung verteilen (vgl. ebd).
Bekannte Vertreter von Software, die eine Betriebssystemvirtualisierung ermöglichen, sind Parallels Virtuozzo Container, OpenVZ, BSD Jails.

2.3.4 Paravirtualisierung

Die Paravirtualisierung liegt genau zwischen der vollständigen Virtualisierung und der Betriebssystemvirtualisierung. Bei der Paravirtualisierung müssen die verwendeten Gastsysteme zunächst modifiziert werden, bevor man sie als solches verwenden kann. Anders als bei der vollständigen Virtualisierung, „weiß“ das virtualisierte System, dass es nur virtualisiert ist und dass die Kommunikation mit der Hardware über eine API durch den Hypervisor geschieht (vgl. Fischer 2009: 144). Der modifizierte Kernel des Gastsystems wurde um sogenannte Hypercalls erweitert, um so an den Hypervisor Befehle zu senden, um direkt mit der Hardware zu sprechen und damit der Hypervisor keine Überprüfung der Gastbefehle durchführen muss (vgl. Hoffmann 2010: 14). Daraus folgt, dass diese Art der Kommunikation zwischen virtualisiertem System und der Hardware höchst effizient verläuft. Das Hostsystem selbst besteht aus einem speziell angepassten Kernel und einem privilegierten Betriebssystem, das der Verwaltung der virtuellen Maschinen dient und auch die Treiber der Hardware bereitstellt. Die Anpassung der Kernelbefehle setzt voraus, dass es sich sowohl bei dem Host als auch bei den Gastsystemen um freie und quelloffene Systeme handelt. Daher wird Paravirtualisierung oft auf Linux-Systemen angewendet.
Paravirtualisierung bietet zwar mit der relativ direkten und hardwarenahen Kommunikation zwischen Gastsystem und den Hardware-Ressourcen die höchste Effizienz und wird auch als

leistungsstark bezeichnet, allerdings erfordert die Anpassung des Kernels doch etwas mehr Erfahrung als eine einfache Anwendung der Virtualisierungssoftware bei den anderen Varianten.
Die bekannteste Software zur Realisierung von Paravirtualisierung ist XEN, allerdings bietet selbst Microsoft mit dem Microsoft Server 2008 und der „Hyper-V“-Technologie die Möglichkeit modifizierte Gastsysteme zu verwenden (Fischer 2009: 145).

Fasst man die Vor- und Nachteile der Varianten und der dazugehörigen Virtualisierungslösungen zusammen, so entsteht folgende Tabelle:

Tabelle 1: Überblick der Varianten, Virtualisierungslösungen, Vor- und Nachteile

Variante	**Virtualisierungslösung**	**Vorteile**	**Nachteile**
Vollständige Virtualisierung	VMware Player, VMware Workstation, Oracle VirtualBox, Microsoft Virtual Server	+ flexible Ausstattung der virtuellen Maschinen möglich + große Auswahl an Gastsystemen	- geringe Performance - zuviele parallel laufende virtuelle Maschinen können Host einschränken
Prozessorunterstützte vollständige Virtualisierung	KVM, QEMU, Oracle VirtualBox	+ keine Modifikation nötig + schlanke Virtu-alisierungslösung	- Hardwareabhängigkeit
Betriebssystem-virtualisierung	OpenVZ, Parallels Virtuozzo Container, BSD Jails	+ flexible Res-sourcenzuteilung + kaum Leistungs-einbuße	- nur Kernelverwandte Gäste verwendbar - starke Abhängigkeit zwischen Host und Gast
Paravirtualisierung	XEN, Microsoft Server 2008	+ hohe Effizienz	- Kernelmodifikationen nötig

Betrachtet man die Tabelle im Hinblick auf die Verwendung von virtuellen Maschinen als Hilfsmittel, um Entwickler und Entwicklerinnen beim Testen der webbasierten Applikation zu unterstützen, so wird man die Paravirtualisierung nicht als erste Wahl sehen, denn der Aufwand bzw. das Know-How eine Kernelmodifikation an den Gastbetriebssystemen anzuwenden ist ohne spezielle Vorkenntnisse kaum möglich und mit einem größerem Zeitaufwand verbunden.
Betriebssystemvirtualisierung ist durchaus eine Option, wenn man sich bei dem Projekt darauf

einschränken kann nur die kernelbedingten Systeme bei der Entwicklung einzusetzen und nicht breit aufgestellt sein möchte, was das Testen auf unterschiedlichen Betriebssystemen betrifft. Je nachdem wieviele Anwender bzw. Anwenderinnen die Applikation verwenden, kann man nicht davon ausgehen, dass alle das gleiche Betriebssystem verwenden. Die Möglichkeit, Ressourcen flexibel anzupassen ist attraktiv, jedoch nicht ausschlaggebend. Die Prozessorgestützte vollständige Virtualisierung, aber auch die vollständige Virtualisierung sind beides sehr gute Optionen, um Flexibilität für unterschiedliche Projekte bereit zu stellen. Das bedeutet für ein Projekt, dass es möglich ist, Test- und Entwicklungsumgebungen mit unterschiedlichen Konfigurationen herzustellen, je nachdem, was gerade gebraucht wird. Ob man nun aber die Lösung mit Hardwareunterstützung oder ohne Hardwareunterstützung verwenden soll, so sollte man sich für die prozessorunterstützte Variante entscheiden, sofern die Hardware dies zulässt. Der Performancegewinn und der schlankere Hypervisor ist ein weiteres Argument dafür.

Vollständige Virtualisierungslösungen, ob hardwarebasiert oder nicht, sind also die Virtualisierungslösungen, die man am besten für das Testen von webbasierten Python Projekten einsetzen kann. Doch trotzdem bleibt die Frage, wie man diese Virtualisierungslösungen nun praktisch verwendet. Man kann bei einem Team aus mehreren Personen nicht davon ausgehen, dass jede Person, die an dem Projekt arbeitet, auch weiß, wie man so eine virtuelle Maschine installiert. Auch kann man nicht davon ausgehen, dass jede virtuelle Maschine exakt so gebaut wurde wie bei einem anderen Teammitglied und somit besteht die Gefahr, dass bei Verteilung von Applikationscode dieser aufgrund von Unterschieden auf Systemebene auf einem anderen System nicht funktioniert. Dies sollte man vermeiden, denn die Fehlersuche kostet Zeit für die Entwicklung und auch die Kosten, die durch die Suche nach der Fehlerursache entstehen, sind nicht zu unterschätzen. Deshalb wäre es besser, die Entwicklungsmaschinen gleich auf einen homogenen Status zu bekommen. Das ist auch möglich durch Virtualisierung, denn die virtualisierten Komponenten einer virtuellen Maschine sind bei der vollständigen Virtualisierung flexibel anpassbar. Es gibt allerdings Softwarelösungen die dabei helfen können.

3 Vagrant

3.1 Was ist Vagrant?

Vagrant ist eine in Ruby geschriebene Open-Source Kommandozeilenanwendung, also eine Anwendung, die hauptsächlich über Textbefehle gesteuert wird, von HashiCorp. Entwickelt wurde Vagrant von Mitchell Hashimoto und John Bender und dient der Verwaltung und Erstellung virtueller Maschinen. Vor allem dient es dem Anwender oder der Anwenderin dazu, das zeitintensive Konfigurieren von virtuellen Maschinen zu vereinfachen, aber auch die Portierbarkeit, d.h. die Möglichkeit Anwendungen bzw. Dateien auf verschiedenen Betriebssystemen zu verwenden, und Wiederverwendung von erstellten VMs zu ermöglichen. (Vgl. Hashicorp 2013a) Vagrant steht für alle gängigen Betriebssysteme wie Linux, Mac OS X oder Windows als ausführbares Installationspaket in den Varianten 32-Bit und 64-Bit zum Download zur Verfügung (vgl. HashiCorp 2013b).

Die Grundlage, um Vagrant verwenden zu können, sind die sogenannten Provider. Im Kontext der Virtualisierung sind damit aber Typ-2-Hypervisor-Lösungen gemeint. Gerade weil dieser Typ von Hypervisoren für die hardwarebasierte vollständige Virtualisierung bzw. vollständige Virtualisierung verwendet wird, ist Vagrant als schlankes Kommandozeilenwerkzeug interessant. Die bekanntesten Virtualisierungslösungen (z.B. VMware, VirtualBox und Hyper-V) sind mit dem Tool verwendbar und bieten daher ein gewisse Wahlmöglichkeit. Das ist vor allem dann vorteilhaft, wenn nicht auf allen Systemen die gleichen Lösungen installiert sind. Jedoch muss dann eine neue virtuelle Maschine für die Virtualisierungslösung vorbereitet werden. Steht die virtuelle Maschine bereit, macht es zunächst keinen Unterschied auf welchem Betriebssystem man arbeitet oder welche Virtualisierungs-Software man verwendet. Durch die Anpassung der Vagrantfile, also der Konfigurationsdatei, mit deren Hilfe Vagrant weiß, welche providerspezifischen Einstellungen die VM benötigt und mit welchen weiteren Attributen die VM ausgestattet werden soll, ist es schnell und einfach möglich, eine homogene, portierbare Lösung für die EntwicklerInnen anzubieten. Um einen Einblick dafür zu bekommen, wie Vagrant funktioniert, werden im Folgendem die einzelnen Komponenten genauer betrachtet und erklärt. Im Rahmen der Arbeit beziehen sich die aufgeführten Beispiele hauptsächlich auf Kommandos in der Linux Shell und der Vagrant Version 1.7.4 . Befehle und Konfigurationen können daher auf anderen Systemen anders aussehen.

3.2 Vagrantfile

Damit Vagrant eine virtuelle Maschine starten kann, benötigt es eine Konfigurationsdatei, um nach den darin definierten Einstellungen eine virtuelle Maschine zu starten, zu konfigurieren und gegebenenfalls zu provisionieren, d.h. Dienste, BenutzerInnen, Ressourcen usw. bereitzustellen. Diese Datei heißt bei Vagrant „Vagrantfile“ und kann entweder über den Befehl 'vagrant init' im Root-Verzeichnis eines Projekts initialisiert werden oder man verwendet eine bereits vorhandene Vagrantfile. Denn gerade beim kollaborativen Arbeiten an Projekten, kann man eine Vagrantfile mit in die Versionskontrolle aufnehmen und hat so auch die Möglichkeit, testweise konfigurierte Boxen auszuprobieren. So ist gewährleistet, dass man im Falle, dass etwas schief geht, die virtuelle Maschine schnell reproduzieren kann.

Beispielkonfiguration

Listing 1 zeigt eine minimal definierte Vagrantfile um eine virtuelle Maschine zu starten:

```
Vagrant.configure("2") do |config|
  config.vm.box = "Name_einer_Box"
end
```

Listing 1: Beispiel einer Minimalkonfiguration

Das Konfigurationsbeispiel zeigt hier die Initiierung einer Konfigurationsversion mit der Versionsnummer „2“ in Zeile 1 und beschreibt einen Mechanismus zur Rückwärtskompatibilität der Konfigurationsdatei. Hierbei kann man sich zwischen zwei einsetzbaren Werten entscheiden: „1“ oder „2“. Version 1 bedeutet, dass die Konfigurationskommandos ab Vagrant Version 1.0.x gelten und Version 2 heißt ab Vagrant Version 1.1+ bis 2.0.x. Es ist durchaus möglich zwei Konfigurationsblöcke gleichzeitig in einer Vagrantfile zu haben, allerdings funktionieren auch nur die Kommandos für die jeweiligen Versionsnummern. (Vgl. Hashicorp 2013c)
Zwischen dem do...end Block, also zwischen Zeile 1 und 3 werden die Einstellungen gesetzt. Diese sind nach dem Schema config.namespace.setting aufgebaut. Es gibt dabei vier config.namespaces: config.vm, config.ssh, config.winrm und config.vagrant.
Bei dem minimalem Konfigurationsbeispiel ist zu erkennen, dass die Einstellung config.vm.box mit dem Wert „Name_einer_Box“ gesetzt wurde. Der Name kann beliebig heißen, allerdings findet man

oft Boxen mit der Namenskonvention:„Organisation/Betriebssystem“. So gibt der Name der Box schon Informationen darüber, welche Organisation bzw. Person die Box erstellt hat und um was für ein Betriebssystem es sich dabei handelt. Zusätzlich kann man auch den verwendeten Provider, installierte Pakete und ob es sich um ein 32-Bit oder 64-Bit System handelt mit angeben.

3.3 Boxes

Im Vagrant-Jargon wird eine vorkonfigurierte virtuelle Maschine als Box bezeichnet. Diese kann bereits für ein Projekt benötigte Dateien, Bibliotheken und Programme usw. konfiguriert haben. Im Sinne des Open-Source-Gedankens gibt es hierfür öffentlich zugängliche Plattformen[1][2], auf denen es möglich ist, sich eine Box auszusuchen oder aber auch eine selbst erstellte Box anderen Vagrant-BenutzerInnen zur Verfügung zu stellen. Somit ist allen Interessierten ein schneller Einstieg möglich, sich mit der Software vertraut zu machen. Hat man sich für eine Box entschieden oder sich auf eine Box geeinigt, mit der man arbeiten möchte, dann kann man die Box entweder erst mit den Befehl `vagrant box add [<argument>]` hinzufügen und danach ein `vagrant up` ausführen, um die virtuelle Maschine zu starten oder man schreibt direkt in die Vagrant-File, ähnlich wie es in der Minimalkonfiguration dargestellt war.

„Box“ vs. „base Box“

Man unterscheidet bei Boxes zwischen „Box“ und „base Box“. Eine „base Box“ wird dann als solche bezeichnet, wenn es sich um eine vorkonfigurierte Box handelt, die mit der nötigsten Software versehen ist, um Vagrant zu starten. Unterschiede für die minimalen Anforderungen entstehen auch durch die Wahl des Betriebssystem, aus dem eine „base Box“ erstellt werden soll.
In Tabelle 2 sind die Linux Derivate bzw. Windows Versionen aufgelistet, die als „base Box“ in Frage kommen. Zudem werden die benötigten Minimalanpassungen benannt, die Vagrant benötigt, um eine korrekte virtuelle Maschine zu starten.

1 https://atlas.hashicorp.com/boxes/search
2 http://www.vagrantbox.es/

Tabelle 2: Verwendbare Betriebssysteme zur Erstellung einer „base Box“

Plattform	**Linux**	**Windows**
Derivate/ Versionen	Debian, Arch, Red Hat, Slackware	Windows 7, Windows 8, Windows Server2008, Windows Server 2008 R2, Windows Server 2012, Windows Server 2012 R2
Minimale Softwarekonfiguration	Paketverwaltung (apt, dpkg, rpm uvm.), SSH-Server (mit UseDNS=no), SSH-Benutzer, optional: Provisioning-Software, providerspezifische Zusatzsoftware (VirtualBox: Guest Additions)	UAC abgeschaltet, komplexe Passwörter, deaktiviert, Shutdown Event Tracker deaktiviert, Server Manager deaktiviert

Quelle: HashiCorp (2013d); eigene Darstellung

Grundsätzlich ist die Erstellung einer „base Box“ sehr spezifisch in Anbetracht der Virtualisierungslösung für die die Box konzipiert wird. Außerdem spielt die Wahl des Betriebssystems auch eine entscheidende Rolle bezüglich der Vorgehensweise zur Erstellung einer Box. Dennoch sollte man für die Komponenten Festplattenspeicher, Arbeitsspeicher und Peripherie einige Regeln beachten damit die Leistung des Hostsystems nicht beeinträchtigt wird. (Vgl. HashiCorp 2013d)

Festplattenspeicher

HashiCorp rät dazu, dem Anwender bzw. der Anwenderin genügend Festplattenspeicher zu gewähren, so dass Platz genug ist, um die benötigten Bibliotheken, Frameworks Module und Entwicklerwerkzeuge verwenden zu können, ohne ihn/sie in anderer Weise zu beeinträchtigen und auch damit Projekte damit realisierbar sind (vgl. HashiCorp 2013d). Dies könnte beispielsweise dann der Fall sein, wenn bei Verwendung der Box ein großer Teil der Festplattenressourcen des Hostsystems blockiert werden, weil dieser von der virtuellen Maschine genutzt wird. Exemplarisch bei VirtualBox sollte man daher eine dynamische Anpassung des Laufwerks einstellen, dafür aber eine passende Maximalgröße festlegen.

Arbeitsspeicher

Für den Arbeitsspeicher gilt grundsätzlich das gleiche Prinzip wie bei der Wahl der Festplattenspeicher: Genügend freigeben, um die virtuelle Maschine projektfähig zu machen, allerdings die weitere Funktionsfähigkeit des Hostsystems gewährleisten. Der Arbeitsspeicher lässt sich auch nachträglich noch über die Vagrantfile anpassen (HashiCorp 2013d). Ein guter Defaultwert ist zum Beispiel 512 MB (ebd.). Je nach Projektbedarf kann die benötigte Ressource variieren.

Peripherie (Audio, USB, etc.)

Hierbei ist die Regel, alle unnötigen Ressourcen wie USB oder Audio zu deaktivieren. Grundsätzlich benötigt man mit Vagrant diese Ressourcen nicht. Allerdings lassen sich auch diese Ressourcen über die Vagrantfile einstellen, wenn sie benötigt werden.

Standardbenutzer

Ein Standardbenutzer ist ein bereits eingerichteter Account mit vorkonfigurierten Einstellungen. Die Entscheidung, ob ein Standardbenutzer mit der Erstellung der „base Box“ eingerichtet werden soll, ist abhängig davon, ob man diese nur für den privaten Gebrauch oder einen ausgewählten Personenkreis verwenden oder veröffentlichen möchte und somit gewisse Standards gesetzt sein müssen, um „out of the box“ zu funktionieren. Dadurch dass beim Standardbenutzer sicherheitsrelevante Faktoren bereits bekannt sind, erhöht sich das Risko, dass das Hostsystem angreifbar wird. Um sich dem Sicherheitsrisiko allerdings bewusst zu werden, ist es von Vorteil, auch die Gründe für die Verwendung eines Standardbenutzers zu kennen. (Vgl. ebd.)

Ein Grund weshalb es bei öffentlichen Boxen einen Benutzer „vagrant“ mit dem Login-Passwort „vagrant“ und einem unsicheren Schlüsselpaar gibt, ist, dass Vagrant diesen Benutzer für die SSH-Verbindung zur virtuellen Maschine erwartet. Grundsätzlich würde die Authentifizierung für die SSH-Verbindung auch über dieses Schlüsselpaar genügen, jedoch setzt man das Passwort aufgrund einer vagrantspezifischen Konvention trotzdem. Bei Bedarf soll so auch das manuelle Einloggen funktionieren z.B. um manuelle Konfigurationen an der Box vorzunehmen. Zusätzlich wird allerdings beim Booten einer Box überprüft, ob das unsichere Schlüsselpaar vorhanden ist und während sich eine Box im laufenden Betrieb befindet mit einem anderen zufällig generierten Schlüsselpaar ersetzt. Dies soll dem erhöhten Sicherheitsrisiko entgegen wirken. (Vgl. ebd.)

Root-Zugriffsrechte

Auf einem System privilegierte Zugriffsrechte über das allgemein bekannte Passwort „vagrant“ zu geben, ist normalerweise etwas, das nicht vorkommen darf. Allerdings ist es nur für öffentlich zur Verfügung gestellte „base Boxen“ vorgesehen, um damit auch bei Bedarf entsprechende Änderungen an der virtuellen Maschine vornehmen zu können. (Vgl. ebd.)

Sudo ohne Passwort

Ein sudo-Befehl ohne Passwort ausführen zu können stuft HashiCorp als wichtig ein, da es sich essentiell auf die Arbeit mit Vagrant auswirkt. Vagrant automatisiert beim Start zum Beispiel die Prozesse zum Netzwerkkonfigurieren, das Mounten von synchronisierten Ordnern, Software-Installationen usw. Um dies zu ermöglichen, ist es nötig sicherzustellen, dass sudo auf der Box installiert und so konfiguriert ist, dass keine Passworteingabe dabei notwendig ist. Es ist definitiv nicht sicher, aber gerade bei Automatisierungsprozessen möchte man manuelle Eingabeaufforderungen vermeiden.

Das Box-File Format

Das Box-File Format ist das für Vagrant zu verarbeitende Format, um eine virtuelle Maschine mit der vorbereiteten virtuellen Hardwareumgebung zu starten. Es handelt sich dabei um eine komprimierte Datei mit den providerspezifischen Dateien, die entstehen, wenn man eine virtuelle Maschine erstellt. Eine für Virtualbox komprimierte Datei kann beispielsweise eine Vagrantfile, eine Image-Datei der virtuellen Festplatte, eine Datei mit der virtuellen Hardware (z.B. *.ovf) und eine JSON-Datei mit Metadaten, mit der Vagrant arbeitet, um den Provider zu identifizieren, sein.

3.4 Provider

Provider bilden das Fundament, auf dem Vagrant steht, um es zu ermöglichen, virtuelle Maschinen mit Vagrant zu starten. Als Standard wird bereits VirtualBox von Oracle unterstützt. Für den Produktiveinsatz wird jedoch VMware empfohlen. Falls man eine andere Virtualisierungslösung als Provider verwenden möchte, muss man dafür erst ein entsprechendes Plugin installieren.
In der folgenden Tabelle sind die Provider aufgelistet, für die es notwendig ist, ein Plugin zu installieren und bekannte Probleme, die damit auftreten können.

Tabelle 3: Bekannte Probleme von Providern in Verwendung mit Vagrant

Provider	**Probleme**	**Plugin benötigt**
VirtualBox	- Konsolenbefehle werden nicht ausgeführt aufgrund von falsch vergebenen Benutzerrechten. - Virtualbox DNS funktioniert nicht und muss über das Host DNS aufgelöst werden.	- nein
VMware	- VMware Workstation (Windows) hat Probleme mit der Port-Weiterleitung. `vagrant reload` kann das Problem beseitigen .	- ja (kommerzielle Lizenz)
Docker	- keine Angaben.	- nein
Hyper-V	- bei Verwendung von Hyper-V kann es zu Kompikationen mit anderen Providern kommen.	- nein (über Windows Features)

Quelle: HashiCorp (2013e); eigene Darstellung

3.5 Provisioner

Vagrant unterstützt automatische Konfigurationswerkzeuge, wie Chef oder Puppet uvm. Mit Hilfe dieser Werkzeuge lassen sich Installations- und Konfigurationsprozesse reproduzieren und versionieren (z.B. mit Git oder SVN), da die Prozesse mit dem Kommando zum Starten der Box gleich zusätzliche Konfigurationsanweisungen automatisieren und Paketinstallationen, Netzwerkkonfigurationen oder Ordnerfreigaben an der virtuellen Maschine vorgenommen werden. Dies kann zum Beispiel die Installation von zusätzlich benötigter Software sein, um eine Entwicklungs- und Testumgebung bereit zu stellen. Außer den Konfigurations- und Automatisierungslösungen kann man auch ein einfaches Shell-Skript verwenden. Dies könnte wie folgt aussehen:

```
#!/bin/bash
apt-get install -y apache2
```

Listing 2: Bash-Skript zur Installation von Apache

Zu Beginn des Skripts wird anhand der Zeichenfolge #! (shebang) signalisiert, dass es sich um ausführbaren Code handelt, der vom Bash-Interpreter verarbeitet werden soll. Die zweite Zeile ist eine Anweisung für die Paketverwaltung „apt“, um das Paket „apache2“ zu installieren mit der Option „-y“. Dies hat zur Folge, dass alle folgenden Anfragen mit „Ja“ beantwortet werden und keine Eingabeaufforderung erfolgt.
Wenn man das Provisioning nicht verwendet, ist es trotzdem möglich, die Box noch im Nachhinein mit den benötigten Anwendungen und Umgebungsvariablen manuell auszustatten.

3.6 Kommandoübersicht

Da es sich bei Vagrant um eine Kommandozeilenanwendung handelt, gibt es neben den Konfigurationsbefehlen, die man in der Vagrantfile verwendet, auch die grundlegenden Befehle, um Vagrant überhaupt zu verwenden. Es gibt Befehle, um Boxen zu entfernen bzw. diese hinzuzufügen, um eine virtuelle Maschine zu starten, anzuhalten oder diese komplett zu löschen usw. Generell ist die Vagrant-Syntax wie folgt aufgebaut:

vagrant [Optionen] <Befehl> [<Argument>]
Listing 3: Schema eines Vagrant-Befehls

Laut HashiCorp brauche man sich, wenn man mit Vagrant arbeitet, nur einen Befehl zu merken und dieser lautet ` vagrant up `. Der Befehl `vagrant up` veranlasst, eine virtuelle Maschine mit den in der Vagrantfile festgelegten Einstellungen zu erstellen, zu konfigurieren und zu starten. Eine weiterere Option, die in jeder guten Kommandozeilenanwendung vorhanden sein sollte ist ` --help`oder einfach `-h`. Die Ausgabe des Befehls `vagrant –help`:

```
[redacted]:~$ vagrant -h
Usage: vagrant [options] <command> [<args>]

    -v, --version                    Print the version and exit.
    -h, --help                       Print this help.

Common commands:
     box             manages boxes: installation, removal, etc.
     connect         connect to a remotely shared Vagrant environment
     destroy         stops and deletes all traces of the vagrant machine
     global-status   outputs status Vagrant environments for this user
     halt            stops the vagrant machine
     help            shows the help for a subcommand
     init            initializes a new Vagrant environment by creating a Vagrantfile
     login           log in to HashiCorp's Atlas
     package         packages a running vagrant environment into a box
     plugin          manages plugins: install, uninstall, update, etc.
     provision       provisions the vagrant machine
     push            deploys code in this environment to a configured destination
     rdp             connects to machine via RDP
     reload          restarts vagrant machine, loads new Vagrantfile configuration
     resume          resume a suspended vagrant machine
     share           share your Vagrant environment with anyone in the world
     ssh             connects to machine via SSH
     ssh-config      outputs OpenSSH valid configuration to connect to the machine
     status          outputs status of the vagrant machine
     suspend         suspends the machine
     up              starts and provisions the vagrant environment
     version         prints current and latest Vagrant version

For help on any individual command run `vagrant COMMAND -h`

Additional subcommands are available, but are either more advanced
or not commonly used. To see all subcommands, run the command
`vagrant list-commands`.
```

Abbildung 5: Übersicht der Vagrant-Kommandos

3.7 Einsatzmöglichkeiten

Hashicorp unterscheidet die Einsatzmöglichkeiten anhand der AnwenderInnen „developer“, „operations engineer“ und „designer“. Im Hinblick auf das Thema dieser Arbeit möchte ich mich jedoch nur auf die Möglichkeiten für die AnwenderInnen developer und operations engineer fokussieren, da diese Personengruppen bei webbasierten Python Projekten am meisten mit dem Tool arbeiten werden, wobei sich dies nicht nur speziell auf Python bezieht, sondern analog zu unterschiedlichen Programmierprojekten angewendet werden kann.

Developer

> „If you're a **developer**, Vagrant will isolate dependencies and their configuration within a single disposable, consistent environment, without sacrificing any of the tools you're used to working with (editors, browsers, debuggers, etc.). Once you or someone else creates a single Vagrantfile, you just need to vagrant up and everything is installed and configured for you to work. Other members of your team create their development environments from the same configuration, so whether you're working on Linux, Mac OS X, or Windows, all your team members are running code in the same environment, against the same dependencies, all configured the same way. Say goodbye to "works on my machine" bugs.“ (HashiCorp 2013f; Hervorh. i. O.)

Aus EntwicklerInnen-Sicht ist Vagrant eine interessante Anwendung. Je nach Projekt kann man sich entweder zu Testzwecken bezüglich den Webserverkonfigurationen oder der Applikation unter Produktionsbedingungen eine vorkonfigurierte Testumgebung (z.B. mit bereits vorkonfigurierten Apache, Nginx usw.) schaffen und in einer isolierten Umgebung arbeiten. Dadurch, dass man in der Vagrantfile festlegen kann, welche Box man verwenden möchte und Vagrantfiles sich auch mit Git oder einem anderen Versionskontrollsystem verwalten und somit versionieren lassen, ist es möglich im Falle, dass etwas nicht funktioniert, auf eine ältere Version der Testumgebung zurückgreifen. Außerdem ist es möglich, sich nach und nach eine perfekte Entwicklungsumgebung mit den Anwendungen und Tools, die man benötigt, um effizient arbeiten zu können, zu konfigurieren. Aber den größten Vorteil bringt vermutlich Vagrant für die Arbeit im Team. Die Verteilung der virtuellen Maschinen kann auch durch die Vagrantfile über das Versionskontrollsystem organisiert werden. Dadurch gibt es keine abweichenden Entwicklungsumgebungen und geschriebener Applikationscode sollte auf jeder Maschine gleich laufen.

Operations engineer

> „If you're an **operations engineer**, Vagrant gives you a disposable environment and consistent workflow for developing and testing infrastructure management scripts. You can quickly test things like shell scripts, Chef cookbooks, Puppet modules, and more using local virtualization such as VirtualBox or VMware. Then, with the *same configuration*, you can test these scripts on remote clouds such as AWS or RackSpace with the *same workflow*. Ditch your custom scripts to recycle EC2 instances, stop juggling SSH prompts to various machines, and start using Vagrant to bring sanity to your life.“ (HashiCorp 2013g; Hervorh. i. O.)

Für AdministratorInnen der IT-Infrastrukturen bietet Vagrant mit Verwendung von Automatisierungs- und Konfigurationsanwendungen wie Ansible, Chef oder Puppet viel Spielraum zum Experimentieren mit Konfigurationskombinationen. Man könnte mit einer „base Box“ beginnen und diese soweit einrichten wie es gewünscht ist. Braucht man nun mehrere Instanzen einer Konfiguration, könnte man sich die Eigenschaft zu Nutze machen, dass Vagrant und damit auch die Vagrantfile eine Ruby Syntax verwendet. Deshalb können Schleifen verwendet werden, um mehrere virtuelle Maschinen nur mit dem `vagrant up`-Befehl zu starten. Natürlich sollte dafür entsprechend starke Hardware vorhanden sein. Die lokalen Tests lassen sich danach möglicherweise auch auf Cloud-Instanzen wie den Amazon Web Services oder Google Compute Engines übertragen. Da diese Instanzen nach Laufzeit und verbrauchten Ressourcen abgerechnet werden, spricht hier auch der Kostenfaktor dafür, mit Vagrant erst einmal lokal Testkonfigurationen durchzuführen. Vagrant ist weitgehend open-source und steht frei zur Verwendung, solange man keine speziellen Plugins benötigt oder andere Lizenzen (Betriebssysteme, Virtualisierungslösungen) zu zahlen hat.

4 Python

Der folgende Abschnitt soll keine komplette Beschreibung der Programmiersprache Python sein, sondern greift nur auf bestimmte Konzepte von Python zurück. Es sollen Probleme, die bei der Installation von Paketen (also installierbaren Sammlungen von Python Modulen) entstehen können, behandelt werden und gezeigt werden, wie man diese Probleme umgehen kann.

4.1 Kurzeinführung

Python ist eine Anfang der 1990er Jahre, als „Hobby-Projekt“ entwickelte Programmiersprache. Guido van Rossum startete das Projekt, um die Programmiersprache ABC weiterzuentwickeln. Die Namensgebung ist eine Anlehnung an die Sendung „Monty Python's Flying Circus“, von der er ein großer Fan war (vgl. Van Rossum 1996).
Die aktuelle Version ist 3.3.4, allerdings verwendet noch ein großer Teil der Python-Community Version 2.7.x. Der Wechsel auf die neueste Version geht schleppend voran, wenn man in Betracht zieht, dass Version 3.0 schon 2008 erschienen ist. Mögliche Gründe könnten z.B. Abhängigkeiten in Paketen sein, bei denen nicht gewährleistet ist, dass diese auch mit der neuesten Python-Version

kompatibel sind und es nicht zu ungewollten Bugs kommt. (Vgl. Python Wiki 2014).
Eine sehr gute Beschreibung über wichtige Merkmale von Python sind bereits auf der allgemeinen FAQ-Seite und der Einleitung des offiziellen Python-Tutorials zu finden:

> „Python is an **interpreted**, interactive, **object-oriented** programming language. It incorporates modules, exceptions, dynamic typing, very high level dynamic data types, and classes. Python combines remarkable power with very clear syntax. It has interfaces to many system calls and libraries, as well as to various window systems, and is extensible in C or C++. It is also usable as an extension language for applications that need a programmable interface. Finally, Python is **portable**: it runs on many Unix variants, on the Mac, and on PCs under MS-DOS, Windows, Windows NT, and OS/2.“ (Python Software Foundation 2015a, Hervorh. durch V. O.)

> „Python is an **easy to learn**, powerful programming language. It has efficient **high-level** data structures and a **simple** but effective approach to object-oriented programming. Python’s elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas **on most platforms**.“ (Python Software Foundation 2015b, Hervorh. durch V. O.)

Die hervorgehobenen Wörter beschreiben im Grunde genommen Python als Programmiersprache recht gut.
Die Beschreibung „interpreted“ soll bedeuten, dass man ein in Python geschriebenes Programm nicht wie in anderen Programmiersprachen wie z.B. C oder C++ durch einen Compiler kompilieren muss, um Quellcode in maschinenlesbaren Binärcode umzuwandeln, sondern das Python-Programm läuft direkt aus dem Quellcode heraus. Python wandelt den Quellcode zunächst in eine Zwischenform, die sich Bytecode nennt, um danach in Maschinencode übersetzt und ausgeführt zu werden. Das bringt den Vorteil mit sich, dass man sich nicht um den Kompiliervorgang kümmern muss, um die entsprechenden Bibliotheken zu laden, Quellcode in Binärcode umwandeln usw. (vgl. Swaroop C. H. 2014).
Auch objektorientierte Programmierung mit Klassen und Methoden, wie man es von anderen Programmiersprachen wie z.B. Java, C++ usw. kennt ist möglich. Wobei es in Python keine Methoden gibt, sondern Funktionen auch als Methoden verwendet werden können. Oft wird Python durch die simple minimalistische Syntax, als eine einfach zu lernende Sprache empfohlen. Bei Kontrollanweisungen hat eine Einrücktiefe aus einem Tab bestehend bzw. vier Leerzeichen eine syntaktische Bedeutung und ist etwa gleichzusetzen mit den geschweiften Klammern {…} in Java,

C oder C++. Demnach wird nach einer Kontrollanweisung wie z.B. if-Anweisung, for-Schleife oder while-Anweisung alles syntaktisch zusammengefasst, was innerhalb des Anweisungsblocks steht und auf der gleichen Einrückungstiefe liegt. Dadurch lässt sich gut geschriebener Python-Code relativ leicht lesen und man kann sich mehr der Lösung des Problems widmen als sich Gedanken über die richtige Syntax zu machen.
Python ist sehr gut, was die Portierbarkeit betrifft, denn Python läuft auf den meisten Betriebssystemen bereits. Hat man beispielsweise ein Python-Programm auf einem GNU/Linux-System geschrieben und keine systemspezifischen Abhängigkeiten verwendet, ist es auch möglich, das Python-Programm auf einem anderen Betriebssystem auszuführen. Jedoch sollte man sicher gehen, dass die gleiche Python-Version auf dem System läuft.
Zudem gibt es eine ganze Menge Pakete und Module, die von der Python-Community entwickelt wurden, um bereits entwickelte Funktionen anderen ProgrammiererInnen zur Weiterverwendung bereit zu stellen. Für die Verteilung und das Zusammenstellen von Modulen und Paketen gibt es die Python Distribution Utilities („Distutils“), genauer gesagt geht es um die *distutils.core.setup()* Funktion, die in jeder Python-Installation mitgeliefert wird und somit verwendbar ist, wenn man beabsichtigt das Paket zu veröffentlichen. Empfohlen wird aber die Verwendung von Drittanbietern (z.B. Wheels und Twine), da diese eine bessere Alternative bezüglich Sicherheit und Bedienung bieten. (Vgl. Python Software Foundation 2015c)

4.2 Das Pythonische Paketsystem

Wenn es um die Verteilung der Pakete geht, muss zunächst betrachtet werden, wo diese bei der Installation hin kopiert werden bzw. in welchen Pfaden Python nach den Paketen sucht. Dann soll kurz erklärt werden wie die Pakete bereitgestellt werden und welche Tools man verwendet, um Pakete zu installieren.

Python Installationspfade

Python unterscheidet zunächst, ob es sich um built-in Module handelt, also Module die mit der Standardinstallation installiert werden und einen globalen Gültigkeitsbereich besitzen oder ob es Third-Party-Module sind, d.h. nachinstallierte Module, die von Drittanbietern kommen. Third-Party-Module müssen erst mit einer *`import`*-Anweisung importiert werden. Wobei es sein kann, dass auch Module, die früher als Drittanbieter galten, so starken Anklang bei der Community

finden, dass diese möglicherweise in eine der zukünftigen Python Versionen integriert werden.
Nachdem eine `import`-Anweisung stattgefunden hat, sucht der Python-Interpreter zunächst in dem Ordner der built-in Module danach, ob sich das zu importierende Modul darin befindet. Ist das Modul darin nicht auffindbar, wird daraufhin eine Liste mit Pfaden durchsucht, die man sich auch manuell auflisten lassen kann. (Vgl. Python Software Foundation 2015d) Das Listing 4 zeigt die Ausgabe der `sys.path`-Liste:

```
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> print sys.path
['',
'/usr/lib/python2.7',
'/usr/lib/python2.7/plat-i386-linux-gnu',
'/usr/lib/python2.7/lib-tk',
'/usr/lib/python2.7/lib-old',
'/usr/lib/python2.7/lib-dynload',
'/usr/local/lib/python2.7/dist-packages',
'/usr/lib/python2.7/dist-packages',
'/usr/lib/python2.7/dist-packages/PILcompat',
'/usr/lib/python2.7/dist-packages/gtk-2.0',
'/usr/lib/python2.7/dist-packages/ubuntu-sso-client']

```

Listing 4: Liste der abgesuchten Installationspfade

Man sieht, dass es sich hierbei um eine einfache Python-Liste handelt. Das macht es zwar einerseits flexibler, lokale Module einzubinden, da man man dann die Liste um einen weiteren Pfad erweitern kann, andererseits könnte ein Angreifer beispielsweise so ungewollt Pfade einfügen, in denen sich ein modifiziertes schadhaftes Module befindet.
Es gibt noch einen weiteren Pfad, in dem die meisten Third-Party-Module installiert werden. Da sich auf dem System aber keine Third-Party-Module befinden bzw. nicht in dieser Umgebung insttalliert sind, wird dieser Pfad auch nicht aufgeführt. Der Pfad, in dem üblicherweise die Module von Drittanbietern installiert werden ist unter Linux: *`/usr/local/lib/pythonX.Y/site-packages`* (vgl. Python Software Foundation 2015e). Die Pfade können aber je nach Betriebssystem abweichen und auch alternative Installationspfade sind möglich. Mehr dazu findet sich in der Python Dokumentation[3].

3 Die Dokumentation ist zu finden unter: https://docs.python.org/2/

4.3 PyPi

PyPi ist ein zentralisiertes Software-Repository für die Programmiersprache Python. Das Akronym PyPi steht für „Python Package Index“ und erfasst bereits aktuell 65496 Pakete (Stand: August 2015). Mithilfe von PyPi kann man nach Paketen suchen und diese dann gegebenenfalls installieren. Die angebotenen Pakete sind meistens von Personen oder Gruppen aus der Python Community hinzugefügt worden, die so ihre Projekte anderen bereitstellen können. Grundsätzlich kann jeder Entwickler bzw. jede Entwicklerin sein/ihr Projekt dort zur Verfügung stellen. Aus EntwicklerInnen-Sicht braucht man lediglich einen registrierten Account und sollte sich möglichst an den Python Packaging Guide[4] halten, damit es für die InteressentInnen einfacher ist, diese auch herunterzuladen und bei der Installation die Module in die richtigen Pfade installiert werden. Wichtig ist zu wissen, dass bei den angebotenen Paketen oft auch wiederum andere Module verwendet wurden und diese Abhängigkeiten dann mit installiert werden.
Man kann sich bei der Menge an bereits hochgeladenen Paketen vorstellen, dass es schwer ist, sich von allen Paketen ein genaueres Bild zu machen und damit die Garantie zu haben, dass es nicht zu Problemen kommen kann, weil sich Abhängigkeiten innerhalb der Pakete geändert haben. Ob dies der Fall ist, ist aufgrund der nicht vorhandenen Überprüfung seitens der Plattform schwer. Der Autor bzw. die Autorin oder die Maintainer des Programms müssen sich selbstständig um die Pflege der Abhängigkeiten kümmern. Dadurch kann es durchaus vorkommen, dass sie ein veraltete Paket verwenden, aber gar nicht merken, dass es inzwischen erhebliche Änderungen gegeben hat, die auch ihr Paket betreffen. Dies hätte die Folge, dass auch ein weiteres Programm, das die veralteten Abhängigkeiten verwendet, nicht mehr so funktioniert, wie es eigentlich soll.

4.4 Pip

Es gibt in Python zwei viel genannte Paketmanager, mit denen man arbeiten kann, um Pakete zu installieren, wenn man ein interessantes Paket im PyPi gefunden hat: pip oder easy_install, das allerdings ein Teil von setuptools ist und sich in einigen Punkten von pip unterscheidet. Beide Tools haben ihre Daseinsberechtigung. Am besten wird der Unterschied anhand der folgenden Tabelle deutlich:

4 https://packaging.python.org/en/latest/distributing.html

Tabelle 4: Vergleich von pip und easy_install

	pip	**easy_install**
Wheels Pakete installieren	Ja	Nein
Pakete deinstallieren	Ja	Nein
Abhängigkeiten überschreiben	Ja	Nein
Pakete über Listen installierbar	Ja	Nein
PEP438 Support	Ja	Nein
Installationsformat	Quelldateien + egg-info metadaten	Egg-Format
Sys.path modifikation	Nein	Ja
Installationen von Eggs	Nein	Ja
pylauncher Unterstützung	Nein	Ja
Multi-version Unterstützung	Nein	Ja

Quelle: pyPA 2013

Pip is eine Weiterentwicklung des Projekts pyinstall das ursprünglich von Ian Bicking initiiert wurde.[5] Das Projekt hat er aber inzwischen an andere abgegeben. Pip ist auch ein Akronym, das für „pip installs packages“ steht. (Vgl. Bicking 2008) Aber auch an setuptools hat er mitgewirkt und einige Funktionen implementiert. (Vgl. Python Software Foundation 2015e) In der Tabelle ist zu erkennen, dass easy_install vor allem dann Verwendung findet, wenn es um die Installation von Paketen im Egg-Format geht. Ein Nachteil ist, dass man nicht mit einem Befehl wie bei pip Pakete deinstallieren kann und auch die nicht vorhandene Möglichkeit eine Installation über Listen auszuführen ist sehr schade.

Aufgrund der Features in der Tabelle möchte ich daher auch eher pip empfehlen, wenn es darum geht, Pakete zu Testen. Auch weil sich pip ähnlich wie ein Paketmanager für Linux bedienen lässt, z.B. apt oder yum usw., ist es vielleicht für diese AnwenderInnengruppe freundlicher. Es gibt die grundlegenden Kommandos wie *install*, *uninstall und search.* Ein Paket kann über folgenden Befehl installiert werden:

pip install <Paketname>

Listing 5: Installationsbefehl für pip

5*"pyinstall is dead, long live pip!“ (Bicking 2008)*

Dabei kann man zusätzlich die gewünschte Version mit angeben, da es sonst die letzte stabile Version installiert. Ansonsten kann man mit den Operatoren wie == *oder* >=, <= arbeiten um sich auf spezielle Versionsabhängigkeiten festzulegen oder aber auch einen Spielraum zu lassen was die Versionsfrage betrifft. Dies hat zum Vorteil, dass auch bei einer Versionsänderung der verwendeten Pakete eine Eingrenzung der Versionen geschaffen wird.
Die Deinstallation von Paketen funktioniert analog wie der *install*-Befehl nur mit den Kommando *uninstall.* Der Befehl sieht dann wie folgt aus:

pip uninstall <Paketname>
Listing 6: Deinstallationsbefehl für pip

Ein zusätzliches Feature, das man bei easy_install nicht vorfindet, ist eine Suchfunktion für Pakete. Pip bietet dies mit dem Kommando *search* an und lässt sich folgendermaßen ausführen:

pip search <Paketname>
Listing 7: Suchbefehl in pip

Dabei wird immer der Python Package Index nach Paketen mit ähnlichem Kontext abgesucht.

4.4.1 Requirements.txt

Eine sehr nützliche Funktion von Pip ist die Arbeit mit der requirements.txt, da diese einige Vorteile bringt, wenn man mit mehreren Personen an einem Projekt arbeitet oder einfach nur Pakete ausprobieren möchte. Es handelt sich bei der requirement.txt um eine Textdatei, die man entweder über den Befehl *pip freeze > requirements.txt* mit den aktuell installierten Paketen füllt oder man verwendet eine bereits gefüllte Datei zum Installieren der darin beschriebenen Pakete. So ist es möglich anderen Personen die gleichen Python Abhängigkeiten zur Verfügung zu stellen. Die Person, die eine vordefinierte requirements.txt erhalten hat, kann einfach mittels des Befehls *pip install -r requirements.txt* alle darin angegebenen Pakete installieren.
Ein anderer Anwendungsfall der requirements.txt ist, Abhängigkeiten zu schützen, indem es pip veranlasst die Abhängigkeiten richtig aufzulösen. Man könnte sich folgendes vorstellen: Paket A benötigt eine Version von Paket C >=1.0. Gleichzeitig aber benötigt Paket B eine Version von Paket C >=1.0,<=2.6. Paket A und Paket B benötigen also beide Paket C, jedoch in zwei unterschiedlichen

Versionsabhängigkeiten. Eine Regel um das Problem zu lösen ist, zuerst die Pakete A und B in die requirements.txt einzutragen und dann zusätzlich das Paket C mit der Einschränkung auf >=1.0,<=2.6. mit aufzunehmen. (Vgl. PyPA 2014)
In Verbindung mit isolierten Python Umgebungen hat man somit bereits ein sehr gutes Setup zum Testen von Python Paketen.

4.5 virtualenv

Virtualenv ist auch ein von Ian Bicking gestartetes Projekt, um isolierte Python Umgebungen zu erstellen und damit dem mit der Installation von Paketen entstehenden Problem der Abhängigkeitskonflikte entgegen zu treten. Wie bereits vorher erwähnt werden alle Module von Drittanbietern in dem plattformspezifischen Pfad */usr/lib/python2.x/site-packages* (in diesem Fall Ubuntu) gespeichert. Dabei kann es aufgrund der Abhängigkeiten zu anderen Modulen ungewollt ein bereits vorhandenes Modul updaten und damit die Abhängigkeit der Pakete, die das fälschlicherweise aktualisierte Modul verwendet hatten, stören. Eigentlich kann das Problem immer dann auftreten, wenn eine Applikation auf anderen Module aufgebaut ist und sich eine der Abhängigkeiten geändert hat. Dies hat die Folge, dass die Applikation nicht mehr funktioniert, auch wenn sich nichts am eigentlichen Applikationscode geändert hat. (Vgl. Virtualenv 2014)
Genau dieses Problem mit den Abhängigkeiten greift virtualenv auf, indem es die Abhängigkeiten isoliert und für jedes Projekt eine eigenständige Installation anlegt. Die Grundbedienung ist die, dass man mit dem Befehl `*virtualenv Name*` eine isolierte Umgebung anlegt, bei der in das aktuelle Verzeichnis ein neuer Ordner mit den ausführbaren Python Dateien und der pip Bibliothek erstellt wird. Danach muss mit dem Befehl `*source Name/bin/activate*` die isolierte Umgebung „scharf“ geschalten werden. Alles, was nun Python betrifft, ist von anderen Umgebungen dort isoliert und betrifft auch nur diese eine Umgebung. Würde man danach eine zweites Terminal aufmachen, sollte dies nicht von den isolierten Umgebung betroffen sein.

4.5.1 virtualenvwrapper

Wie der Name schon vermuten lässt, handelt es sich bei virtualenvwrapper um eine Anwendung, die zur Verbesserung des Workflows bei der Arbeit mit virtualenv dient, da man bei der blanken Verwendung von virtualenv auch schnell mal den Überblick verlieren kann, wenn man an zu vielen

Projekten gleichzeitig arbeitet. Für die Installation des Tools verwendet man am besten pip und wird über die Kommandozeile mit folgendem Befehl installiert:

```
pip install virtualenvwrapper
```

Listing 8: Befehl zur Installation von virtualenvwrapper

Jedoch muss man vorher noch ein paar kleine Vorbereitungen vornehmen, um es einwandfrei verwenden zu können. Da es sich bei den hier verwendeten Werkzeugen um Kommandozeilentools handelt, kann bzw. muss man auch manchmal die richtigen Umgebungsvariablen für diese setzen. Das kann man in der versteckten Datei .bashrc (unter Ubuntu), die sich im Home Verzeichnes des aktuellen Benutzers befindet, tun. Die .bashrc-Datei ist ein shell-Skript für die Sprache Bash, das bei jedem Start eines neuen Terminalfensters ausgeführt wird. In dem Skript sind oft Umgebungsvariable und weitere Konfigurationen gesetzt, z.B. um die automatische Vervollständigung von Befehlen zu ermöglichen. Da bash relativ mächtig ist und es unzählige Anwendungsmöglichkeiten dafür gibt, möchte ich im Rahmen der Arbeit mich nicht darauf fokussieren zu erklären, wie Bash funktioniert, jedoch auf die offizielle Dokumentation[6] hinweisen. Dieses Shell-Skript kann nach Belieben angepasst und erweitert werden. Im Fall von virtualenvwrapper sollte man das auch machen, da man sonst beim Öffnen eines neuen Terminals immer wieder erst die Umgebungsvariable für den virtualenvwrapper gesetzt werden muss und dann das aktuelle Terminal wieder neu laden muss, um die vorher gesetzte Umgebungsvariable zu laden. Ein Beispiel, wie man die .bashrc erweitern kann, sieht folgendermaßen aus:

```
if [ -f /usr/local/bin/virtualenvwrapper.sh ]; then
    export WORKON_HOME=$HOME/.virtualenvs
    source /usr/local/bin/virtualenvwrapper.sh
fi
```

Listing 9: Erweiterung der .bashrc zur Verwendung von virtualenvwrappper

Hier wird zunächst über eine if-Anweisung überprüft, ob es sich bei der Datei virtualenvwrapper.sh um eine Datei handelt und ob diese auch existiert. Denn falls kein virtualenvwrapper installiert wurde, ist dieser auch nicht vorhanden und man könnte sich das Setzen der Umgebungsvariable sparen. Trifft die Bedingung zu, so wird über das *export*-Kommando die Umgebungsvariable *WORKON_HOME* mit dem Wert *$HOME/.virtualenvs* gesetzt. Ruft man den Befehl *echo*

6 www.gnu.org/software/bash/manual

$WORKON_HOME auf, so sollte dann der Pfad */home/USER/.virtualenv* ausgegeben werden. Die Variable *WORKON_HOME* muss gesetzt werden, da der virtualenvwrapper diesen Pfad verwendet, um die erstellten isolierten Umgebungen dort zu speichern. Mit dem *source*-Befehl, kann man die Skript-Datei virtualenvwrapper.sh ausführen lassen als würde man jede einzelne Zeile Code darin in eine Kommandozeile eingeben.
Ist die Umgebungsvariable gesetzt, kann man nun auch die Befehle des virtualenvwrappers verwenden. Die wichtigsten Befehle sind *mkvirtualenv*, *workon* und *rmvirtualenv*.

5 Fallbeispiel

5.1 Beschreibung

Das Fallbeispiel soll ein Szenario darstellen, bei dem aufgrund einer Änderung in einem der verwendeten Pakete die Funktionalität eines Programms nicht mehr gewährleistet ist. Zudem soll aufgezeigt werden, wie man mit Hilfe der vorher genannten Werkzeuge, Vorkehrungen schließen kann, um das Problem zu lösen und damit die Funktionalität erhalten bleibt. Aufgeteilt wird das Beispiel in zwei Teile.
Der erste Teil beschreibt die Webapplikation, die ein JSON-Objekt erzeugt, das beispielsweise als Schnittstelle zu einem anderen Programm oder Skript verwendet werden könnte. Aus Gründen der einfacheren Darstellung wurde mit Hilfe von Jquery und Bootstrap eine dynamische Eingabemöglichkeit erstellt, die allerdings nicht zu dem genannten Problem beiträgt. Daher werde ich auf diesen Teil des Quellcodes nicht im Detail eingehen.
Bei der in Teil 1 beschriebenen Applikation handelt es sich um eine kleine webbasierte Anwendung, die mit Hilfe des Microframeworks Flask[7] umgesetzt wird. Das Microframework Flask verwendet zusäztliche Abhängigkeiten in Form des WSGI toolkits Werkzeug und Jinja2, einer template engine. Die Anwendung hat einerseits die Möglickeit einen String als Benutzereingabe zu erhalten und stellt diesen dann als invertierten String dar, ndererseits ist es auch möglich die Schnittstelle direkt anzusprechen. Der String wird mittels eines HTTP-Requests an eine URL gesendet. Die Anwendung verarbeitet bei der Anfrage den hinteren Teil der URL als invertierten String und wandelt das Resultat in ein JSON ähnliches Objekt. Dieses Objekt wird dann auch als HTTP-Response zurückgegeben.

7 http://flask.pocoo.org/

Bei Teil 2 handelt es sich um ein kleines Skript, das mit der Bibliothek Requests[8] realisiert wurde, um diese HTTP-Response auszulesen. Das Skript soll dazu dienen, das Szenario einer nicht mehr funktionierenden Funktion aufgrund einer kritischen Änderung an einer Komponente des Moduls zu zeigen. Die Applikation zeigt keine vollständige Web-Applikation, sondern ist nur zur Demonstration auf minimale Funktionsfähigkeit programmiert worden.

Testumgebung

Betriebssystem: Ubuntu 14.04
Pyton Version: 2.7.6
Python-Abhängigkeiten: Flask 0.9, Requests 0.14.1, Requests 2.7.1
Sonstiges: pip, virtualenv + virtualenvwrapper, git, atom, Bootstrap, jquery

Git initialisieren

Git ist ein schlankes, aber mächtiges dezentrales Versionskontrollsystem, das es erlaubt zum Einen eine gute Übersicht über den Projektverlauf zu erhalten, aber auch bei Komplikationen, die nicht lösbar sind, auf eine stabile funktionierende Version zurückzukehren. Daher ist es „good practice“ bevor man weitere projektkritische Änderungen vornimmt, ein lokales Git-Repository anzulegen. Deswegen ist der erste Befehl des Projekts im Root-Verzeichnis *git init.* Weitere Vorgehensweisen wie Commits, erstellen einer .gitignore-Datei usw. werde ich nicht weiter ausführen, da in dieser Arbeit der Fokus nicht darauf liegt.

Virtualenvs erstellen

Es wird davon ausgegangen, dass bereits virtualenv zusammen mit dem virtualenvwrapper auf dem System installiert ist. Der Grund, weswegen die isolierte Umgebung jetzt erstellt wird, ist, dass man das System auch nach dem Projekt möglichst sauber hinterlassen möchte. So entstehen keine Altlasten von ehemaligen Projekten, die eventuell nicht mehr gebraucht werden. Der Befehl *mkvirtualenv Name* legt so eine isolierte Umgebung an, die dadurch erkennbar ist, dass der Name der Umgebung, die aktiv ist, in runden Klammern am Anfang der Kommandozeile vor dem Benutzer angezeigt ist. In Abbildung 6 sieht man eine aktive isolierte Umgebung mit dem Namen

8 http://www.python-requests.org/en/latest/

„flask“.

```
(flask)claister@b0xTo60:~$
```

Abbildung 6: Aktive isolierte Umgebung

Da nun die isolierte Umgebung aktiv ist, kann man sich an die Installation der benötigten Python Pakete machen und die Applikation schreiben. Man sollte hierbei einen möglichst passenden Namen finden.

5.2 Applikationscode

5.2.1 Teil 1: Webapplikation

Die Applikation kann über das Terminal mit dem Befehl `*python app.py*` gestartet werden. Danach wird sie im Browser wie in Abbildung 7 dargestellt, wenn man die URL *localhost:5000/index* eingibt. Oberhalb des Eingabefelds wird der invertierte String angezeigt. Per Default wurde im Code der String „tratS“ bereits eingefügt, um das Wort „Start“ erkenntlich zu machen.

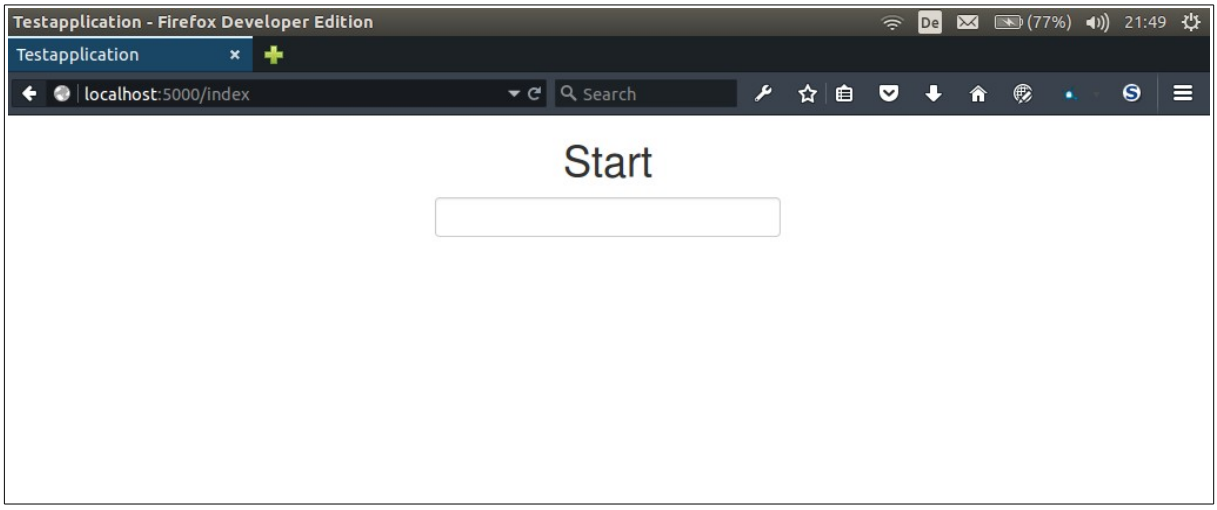

Abbildung 7: Die Demonstrations-Applikation im Browser

App.py

Im Folgenden soll anhand des Listings 10 der Programmcode für die Webapplikation „app.py“ erklärt werden.

```python
from flask import Flask, jsonify, abort, render_template

app = Flask(__name__)

@app.route('/')
@app.route('/index')
def index():
    return render_template('index.html')

@app.route('/<string:string>')
def reverse(string):
    result = {'result': string[::-1]}
    return jsonify(result)

if __name__ == '__main__':
    app.run(debug=True)

```

Listing 10: Applikationscode von app.py

In Zeile 1 wird das mit dem *import*-Statement aus dem Modul *flask*, die Klasse *Flask* und die Funktionen *jsonify, abort und render_template* importiert. Danach wird in Zeile 3 eine Instanz der Flask-Klasse mit dem Parameter *__name__* erstellt und in der Variable *app* gespeichert. Der Parameter dient dazu, Flask darüber zu informieren, welche Dateien zu der Applikation gehören. Da in dem Beispiel nur eine Datei verwendet wird, kann man einfach *__name__* verwenden. Handelt es sich allerdings um ein Paket, sollte man lieber hard-coded Parameter verwenden, um die Abhängigkeiten absoluter zu bestimmen.

Zeile 5-8 verwendet den decorator *route()* für das URL-Routing, d.h. allgemein formuliert bedeutet es, wenn die in den runden Klammern angegebenen URL-Strings einen HTTP-Requests erhalten, wird daraufhin ein Funktion ausgeführt und weiter verarbeitet. In diesem Fall gilt das für die URLs: „/“ und „/index“. Kommt also ein HTTP-Request an, wird die Funktion *index()* ausgeführt. In der darin verwendeten Funktion *render_template()* wird daraufhin die *index.html* aufgerufen.

In den Zeilen 10-13 wird wieder der *route()* decorator benutzt, um HTTP-Requests anzunehmen und darauffolgend eine Funktion ausführen zu lassen. Dieses Mal ist der hintere Teil der URL anhand der Variable *string* allerdings variabel. Der Funktion *reverse()* wird die Variable *string* als Parameter übergeben und dieser wird dann wiederum in Zeile 12 als invertierter String in der Variable *result* gespeichert. Zeile 13 wandelt mit Hilfe der *jsonify()*-Funktion den Wert in die qw Variable *result* und daraufhin in ein JSON ähnliches Objekt. Diese Implementierung dient auch als Schnittstelle für das Skript in Teil 2 des Beispiels.

In der Zeile 15-16 des Programms wird mit der *run()*-Funktion (debug-Modus) der lokale Server mit der Anwendung gestartet. Die If-Bedingung *__name__ == '__main__ '* stellt dabei sicher, dass das Skript nur dann ausgeführt wird, wenn es direkt aus dem Python Interpreter kommt.

Index.html

Die index.html ist das Template-Dokument zur Darstellung von HTML. Wie allerdings bereits in der Beschreibung erwähnt, dient das Template hauptsächlich der Darstellung des Eingabe-Strings als invertierter Wert. Da es allerdings nicht weiter zu der Lösung des Problems beiträgt, möchte ich auch nicht näher auf die darin verwendeten jQuery-Abfragen und HTML-Elemente eingehen.

5.2.2 Teil 2: Requests-Skripts

Im ersten Teil wurde bereits die Webapplikation beschrieben, die bei Anfrage einer URL mit dem Schema http://.../<string> ein invertiertes JSON-Objekt zurückgibt. Dieses könnte man als Simulation einer Schnittstelle sehen, mit der beispielsweise ein Programm, das die Requests Bibliothek verwendet, kommuniziert.
Es handelt sich bei Teil 2 um ein kleines Python-Skript mit der Requests Version 0.14.1, das eine HTTP-Anfrage stellt und als Antwort das JSON-Objekt erwartet. Listing 11 zeigt das Skript „Testskript_0.14.1.py“:

```
1  #!/usr/bin/env python
2
3  import requests
4
5
6  r = requests.get('http://localhost:5000/streawkceuR')
7
8  print r.json
9
```

Listing 11: Requests 0.14.1 Skript

In Zeile 1 wird das Skript mit einem „Shebang“, also der speziellen Zeichenfolge *#!* eingeleitet. Daraufhin ist es möglich, das Skript auch aus dem Terminal auszuführen ohne den Interpreter zusätzlich mit anzugeben. In Zeile 3 wird die Requests-Library anhand des *import*-Statements importiert. Ein Get-Request wird dann in Zeile 6 an die lokale URL der Webapplikation gesendet

und die Response in der Variable r gespeichert. Abschließend wird das in der Variable gespeicherte Objekt ausgegeben. Die Ausgabe des Skrips wird in Abbildung 8 dargestellt.

```
:~/Documents/uni/BA-Thesis/Assets/Dependency_fail$ python Testscript_0_14_1.py
{u'result': u'Rueckwaerts'}
```

Abbildung 8: Ausgabe nach Ausführung des Skripts

Das Skript funktioniert so, wie es soll. Der übermittelte String „straewkceuR“ wurde an die Schnittstelle der Webapplikation übermittelt und entsprechend der *route()*-Funktion invertiert. Der String wurde als JSON-Objekt gespeichert und ausgegeben. Abbildung 9 zeigt eine Ausgabe des gleichen Skripts nachdem das System weitere Projekte absolviert hat und im Zuge der Projekte weitere Pakete installiert wurden.

```
:~/Documents/uni/BA-Thesis/Assets/Dependency_fail$ python Testscript_0_14_1.py
<bound method Response.json of <Response [200]>>
```

Abbildung 9: Ausgabe nach Ausführung des Skripts einige Zeit später

Was ist passiert?

Das Debugging der Fehlermeldung lässt vermuten, dass eine Änderung am Requests-Modul im site-packages Ordner stattgefunden hat. Möglicherweise hat eines der Pakete bei einem anderen Projekt das Requests-Modul upgedated. Das wurde jedoch zunächst nicht bemerkt, sondern erst nachdem das Skript nochmals verwendet wurde. Nach Einsicht der Changelogs von Requests stellte sich heraus, dass sich die Funktion „r.json()“ von „property“ auf „callable“ geändert hatte.

Problemanalyse

Was das genau zu bedeuten hat, kann man gut mit Hilfe der Python Funktionen *type()* und *dir()* sehen. Zusätzlich werden zwei isolierte Umgebungen erstellt. Eine Umgebung stellt die Requests Version 0.14.1 dar, mit der das Skript noch funktioniert hatte, weil die Funktion *r.json()* noch „property“ war. Die zweite Umgebung verwendet die neueste Version von Requests, d.h. „2.7.0“. In den jeweiligen Umgebungen werden dann jeweils die Funktionen *type()* und *dir()* im interaktiven Python Interpreter mit der *r.json* Funktion als Parameter aufgerufen. Die Ausgaben können dann analysiert werden und danach kann der Code wieder angepasst werden, so dass das Skript auch mit der neuen Version funktioniert. Abbildung 10 zeigt die Ausgabe mit der Requests Version 0.14.1

und Abbildung 11 zeigt die Ausgabe mit Requests Version 2.7.0. Zur besseren Vergleichbarkeit, werden die Abbildungen direkt untereinander dargestellt.

```
(request014)[redacted]:~/Documents/uni/BA-Thesis/Assets/Dependency_fail$ python
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import requests
>>>
>>> r = requests.get('http://localhost:5000/streawkceuR')
>>>
>>> type (r.json)
<type 'dict'>
>>>
>>>
>>>
>>> dir (r.json)
['__class__', '__cmp__', '__contains__', '__delattr__', '__delitem__', '__doc__', '__eq__', '__format__', '__ge_
_', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__init__', '__iter__', '__le__', '__len__', '__lt_
_', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__setitem__', '__sizeof__',
'__str__', '__subclasshook__', 'clear', 'copy', 'fromkeys', 'get', 'has_key', 'items', 'iteritems', 'iterkeys',
'itervalues', 'keys', 'pop', 'popitem', 'setdefault', 'update', 'values', 'viewitems', 'viewkeys', 'viewvalues']
>>>
>>> print r.json
{u'result': u'Rueckwaerts'}
>>>
```

Abbildung 10: Analyse von r.json nach type() und dir() in Requests 0.14.1

```
(requests27)[redacted]:~/Documents/uni/BA-Thesis/Assets/Dependency_fail$ python
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import requests
>>>
>>> r = requests.get('http://localhost:5000/streawkceuR')
>>> type (r.json)
<type 'instancemethod'>
>>>
>>>
>>>
>>> dir (r.json)
['__call__', '__class__', '__cmp__', '__delattr__', '__doc__', '__format__', '__func__', '__get__', '__getattrib
ute__', '__hash__', '__init__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__self__', '__setattr__',
 '__sizeof__', '__str__', '__subclasshook__', 'im_class', 'im_func', 'im_self']
>>>
>>>
>>> r.json()
{u'result': u'Rueckwaerts'}
>>>
```

Abbildung 11: Analyse von r.json nach type() und dir() in Requests 2.7.0

Man erkennt bereits am Typ, dass es sich um gänzlich unterschiedliche Datentypen handelt. In Version 0.14.1 war das gespeicherte Response-Objekt noch ein Dictionary, wohingegen es sich in der Version 2.7.0 um eine Methode handelt. Zusätzliche Informationen gibt der Befehl *dir()* über das Objekt. Denn damit lassen sich die Attribute des r.json Objekts untersuchen.

5.2.3 Schlussfolgerung

Da es sich bei der Version 2.7.0 um eine aufrufbare Methode handelt und das zusätzlich mit dem Attribut __call__ untermalt wird, kann man daraus folgern, dass die *print*-Anweisung so nicht mehr

funktionieren wird. Etwas Aufrufbares, wie etwa eine Methode oder eine Funktion, benötigt die runden Klammern, um ein korrektes aufrufbares Objekt darzustellen.
Da es sich nur um ein kleines Skript handelt, muss glücklicherweise nur in Zeile 8 Code ausgebessert werden. Von `*print r.json*` zu `*r.json()*`. Jedoch hätte das Modul nicht verändert worden können, wenn zum Testen der Pakete oder auch um die Projekte zu verwalten isolierte Python Umgebungen erstellt worden wären. Zusätzlich hätte das Anlegen einer requirement.txt mit dem Eintrag `requests==0.14.1` die Abhängigkeit des Requests Moduls auf diese spezifische Version beschränkt und somit die Abhängigkeit vor einer Änderung geschützt.

6 Ergebnis und Fazit

Ziel der der Arbeit war, die Probleme, die bei der Entwicklung von webbasierten Python Projekten entstehen können, herauszustellen und zu benennen. Diese sollten dann mit zur Hilfenahme der Funktionalität, die virtuelle Maschinen und Umgebungen mit sich bringen, gelöst bzw. vermieden werden.
Dadurch dass man in Python Projekten relativ häufig Pakete aus dem PyPi verwendet und diese Pakete dann grundsätzlich in dem site-packages Ordner installiert werden, ist die Verwendung einer speziellen Umgebung zur Isolation und Organisation der einzelnen Projekte durchaus praktikabel. Auch das Testen bestimmter Pakete oder ausgesuchter Paketversionen kann damit leichter realisiert werden als diese direkt in die Standardsystemumgebung zu installieren und damit ungewollte Konflikte auszulösen. Stattdessen probiert man zunächst das Paket aus und verwendet die Möglichkeit des paketmanagerähnlichen Tools pip mit der requirements.txt, um die benötigten Paketinstallationen zu reproduzieren oder aber auch hinsichtlich der Möglichkeiten, die es beim teambasierten Arbeiten bringt. Probleme, die durch eine Installation ohne eine isolierte Umgebung entstehen könnten, sind dadurch schon getestet worden. Die Vorteile, die die vorgestellten Python spezifischen Tools im Hinblick auf das Thema bieten, sind sehr hoch einzuschätzen und definitiv Wert, sie zu verwenden.
Auch virtuelle Maschinen, die nicht abhängig von der Programmiersprache sind und daher in jedem anderen Programmierprojekt anwendbar wären, können dann nützlich sein, wenn es darum geht, serverähnliche Testumgebungen zu schaffen oder um ein betriebssystemspezifisches Verhalten der Applikation zu testen. Ein Faktor der relativ stark für die Verwendung von virtuellen Maschinen spricht, ist die der Homogenität der Entwicklersysteme, also dass beispielsweise für ein Projekt das komplette Team auf einer Basis die Entwicklung der Applikation beginnt und so die

Austauschbarkeit und Verwendung des Applikationscodes gewährleistet ist. Das ist auch ein Grund, weswegen die Verwendung von Vagrant durchaus Sinn macht, denn bei der Installation der virtuellen Maschinen können Probleme auftreten, da nicht alle die gleichen Vorkenntnisse bezüglich der Installation und Konfiguration von virtuellen Maschinen ohne Verwendung des Tools besitzen. Für länger andauernde oder sicherheitsbedenkliche Projekte sollte man allerdings bei der Verwendung darauf achten, eine eigene Box zu erstellen, bei der das Sicherheitsrisiko minimiert ist, da öffentlich bekannte Logindaten durchaus ein Risiko darstellen, wenn ein Angreifer bereits auf das Hostsystem vorgedrungen ist. Eine Einschätzung, wie gut der Einsatz virtueller Maschinen als Hilfsmittel zum Testen von webbasierten Python Projekten ist, ist abhängig von dem Projekt selbst. Aufgrund der vorher genannten Vorteile ist es empfehlenswert, aber je nach Projekt nicht zwingend notwendig.

Zusammenfassend kann man über die Verwendung von virtuellen Maschinen und isolierten Umgebungen durchaus sagen, dass beide ein gutes Mittel sind, um einerseits dem Problem, dass bei der Verwendung und Installation von Paketen entstehen kann, entgegenzuwirken, aber andererseits auch teambasiertes Arbeiten bei Projekten zu fördern.

Eine Kombination der virtuellen Maschine und einer darin verwendeten isolierten Umgebungen stellen eine starke Projektbasis dar, um damit das Testen und Ausprobieren von Paketen, Serverkonfigurationen und Applikationscode zu gewährleisten.

Tabellenverzeichnis

Abbildungsverzeichnis

Listingsverzeichnis

Literaturverzeichnis

AMD (2014): *AMD Virtualization*. URL: http://www.amd.com/en-us/solutions/servers/virtualization (aufgerufen am 25.08.2015).

Bicking, Ian, The Open Planning Project, PyPA: *Virtualenv – virtualenv 13.1.2 documentation*. URL: https://virtualenv.pypa.io/en/latest (aufgerufen am 28.08.2015).

Bicking, Ian (2008): *pyinstall is dead, long live pip!*. URL: http://www.ianbicking.org/blog/2008/10/pyinstall-is-dead-long-live-pip.html (aufgerufen am 28.08.2015)

Bitkom (2014): *Deutscher IT-Markt wächst 2015 um 2,4 Prozent.* URL: https://www.bitkom.org/Presse/Presseinformation/Pressemitteilung_1379.html (aufgerufen am 22.07.2015).

Fischer, Marcus (2009): *XEN*. Galileo-Press, Bonn.

HashiCorp (2013a): *Why Vagrant? - Vagrant Documentation*. URL: https://docs.vagrantup.com/v2/why-vagrant/index.html (aufgerufen am 12.08.2015).

HashiCorp (2013b): *Vagrant Downloads*. URL: https://www.vagrantup.com/downloads.html (aufgerufen am 14.08.2015).

HashiCorp (2013c): *Configuration Version - Vagrantfile - Vagrant Documentation*. URL: https://docs.vagrantup.com/v2/vagrantfile/version.html (aufgerufen am 12.08.2015).

HashiCorp (2013d): *Creating a Base Box - Vagrant Documentation.* URL: https://docs.vagrantup.com/v2/boxes/base.html (aufgerufen am 12.08.2015).

HashiCorp (2013e): *Providers - Vagrant Documentation*. URL: https://docs.vagrantup.com/v2/providers/index.html (aufgerufen am 24.08.2015).

HashiCorp (2013f): *Why Vagrant?*. URL: https://docs.vagrantup.com/v2/why-vagrant/index.html (aufgerufen am 24.08.2015).

HashiCorp (2013g): *Why Vagrant?*. URL: https://docs.vagrantup.com/v2/why-vagrant/index.html (aufgerufen am 24.08.2015).

Hoffman, Benjamin (2010): *Hostvirtualisierung – Vergleich der Konzepte und Produkte*. Seminararbeit, Universität der Bundeswehr, Neubiberg. Abrufbar unter:

https://www.unibw.de/rz/dokumente/public/getFILE?fid=5811473&tid=public

Intel: *Intel® Virtualization Technology (Intel® VT)*. URL: https://www-ssl.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html (aufgerufen am 25.08.2015).

ITWissen (2015a): *Vollständige Virtualisierung.* URL: http://www.itwissen.info/definition/lexikon/Vollstaendige-Virtualisierung-full-virtualization.html (aufgerufen am 25.08.2015).

ITWissen (2015b): *Prozessorunterstützte Virtualisierung.* URL: http://www.itwissen.info/definition/lexikon/Prozessor-unterstuetzte-vollstaendige-Virtualisierung-process-supported-full-virtualization.html (aufgerufen am 25.08.2015).

ITWissen (2015c): *Betriebssystemvirtualisierung.* URL: http://www.itwissen.info/definition/lexikon/Betriebssystemvirtualisierung-operating-system-virtualization.html (aufgerufen am 25.08.2015).

ITWissen (2015d): Para*virtualisierung.* URL: http://www.itwissen.info/definition/lexikon/Paravirtualisierung-para-virtualization.html (aufgerufen am 25.08.2015).

Portnoy, Matthew (2012): *Virtualisierung für Einsteiger*. Wiley-VCH, Weinheim.

Python Software Foundation (2015a): *General Python FAQ – Python 2.7.10 documentation*. URL: https://docs.python.org/2/faq/general.html#what-is-python-good-for (aufgerufen am 21.08.2015).

Python Software Foundation (2015b): *The Python Tutorial - Python 2.7.10 documentation*. URL: https://docs.python.org/2/tutorial/ (aufgerufen am 21.08.2015).

Python Software Foundation (2015c): *Distributing Python Modules – Python 2.7.10 documentaion*. URL: https://docs.python.org/2/distutils/ (aufgerufen am 21.08.2015).

Python Software Foundation (2015d): *Modules – Python 2.7.10 documentation*. URL: https://docs.python.org/2/tutorial/modules.html#the-module-search-path (aufgerufen am 28.08.2015).

Python Software Foundation (2015e): *setuptools 18.2: Python Package Index.* URL: https://pypi.python.org/pypi/setuptools (aufgerufen am 28.08.2015).

PyPA (2014a): *pip vs easy_install*. URL: https://packaging.python.org/en/latest/pip_easy_install.html (aufgerufen am 28.08.2015).

PyPA (2014b): *User Guide – pip 7.1.2 documentation*. URL:

https://pip.pypa.io/en/latest/user_guide.html (aufgerufen am 28.08.2015).

Python Wiki (2014): *Python2orPython3 - Python Wiki*. URL: https://wiki.python.org/moin/Python2orPython3 (aufgerufen am 22.08.2015)

Swaroop C H (2014): *A Byte of Python*. URL: http://www.swaroopch.com/notes/python/#interpreted (aufgerufen am 22.08.2015)

Van Rossum, Guido (1996): *Foreword for "Programming Python" (1st ed.)*. URL: http://legacy.python.org/doc/essays/foreword/ (aufgerufen am 22.08.2015)

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelor-Thesis mit dem Titel:

Virtuelle Maschinen und Umgebungen als Hilfsmittel zum Testen web-basierter Python Projekte

selbständig und nur mit den angegebenen Hilfsmitteln verfasst habe. Alle Passagen, die ich wörtlich aus der Literatur oder aus anderen Quellen wie z. B. Internetseiten übernommen habe, habe ich deutlich als Zitat mit Angabe der Quelle kenntlich gemacht.

(Unterschrift)