01-2017 | Frühling | www.ijug.eu

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

Aus der Community – für die Community

Java aktuell

D: 4,90 EUR A: 5,60 EUR CH: 9,80 CHF Benelux: 5,80 EUR ISSN 2191-6977

Java ist die beste Wahl

Einfacher programmieren
Erste Schritte mit Kotlin

Security
Automatisierte Überprüfung von Sicherheitslücken

Leichter testen
Last- und Performance-Test verteilter Systeme

4 191978 304903 01

iJUG Verbund

Der Unterschied von Java EE zu anderen Enterprise Frameworks

Kotlin ist eine ausdrucksstarke Programmiersprache, um die Lesbarkeit in den Vordergrund zu stellen und möglichst wenig Tipparbeit erledigen zu müssen

Innerhalb von Enterprise-Anwendungen spielen Sicherheits-Aspekte eine wichtige Rolle

Last- und Performance-Test verteilter Systeme mit Docker & Co.

Dr. Dehla Sokenou, GEBIT Solutions

Moderne Virtualisierungs-Umgebungen haben das Potenzial, die Software-Entwicklung zu revolutionieren. Neben der immer engeren Verzahnung von Entwicklung und Betrieb (Stichwort „DevOps“) unterstützen Lösungen wie Docker auch andere Phasen und Tätigkeiten im Software-Entwicklungsprozess, etwa den Test. Hierbei profitiert insbesondere der Last- und Performance-Test.

Auch wenn Testen inzwischen aus der Exotenecke in der Gegenwart angekommen ist, verursachen Tests immer noch einen nicht unerheblichen Anteil an den Kosten der Software-Entwicklung. Test-Automatisierung hilft, diese Kosten in den Griff zu bekommen, und sollte, wo immer möglich, den Vorzug vor manuellen Tests erhalten. Dabei kann Automatisierung sowohl bei der Vorbereitung, beim Aufsetzen der Test-Umgebung, beim Deployment der zu testenden Anwendung wie auch bei der eigentlichen Testausführung und -auswertung zum Einsatz kommen.

Neben den fachlichen Tests auf Unit-, Integrations- und System-Ebene müssen zusätzliche Tests das nichtfunktionale Verhalten eines Systems berücksichtigen. Ein Beispiel dafür – neben anderen – sind Last- und Performance-Tests, die das Verhalten des Systems in seinen Grenzbereichen und außerhalb seiner Grenzen überprüfen. Der erreichbare Automatisierungsgrad ist hier im Gegensatz zu anderen nichtfunktionalen Tests besonders hoch.

Verteilte Systeme stellen beim Test eine besondere Herausforderung dar, weil zusätzlich zum Verhalten eines einzelnen Systems die Kommunikation innerhalb des Gesamtsystems eine große Rolle spielt. Für den Test sollte eine möglichst realitätsnahe Umgebung zur Verfügung zu stehen. Allerdings ist es meist nicht möglich, Szenarien mit 10.000 oder mehr vollwertigen Knoten zu Testzwecken aufzusetzen. Wie testet man also die Performance solcher Systeme möglichst realistisch und im Idealfall automatisiert?

Moderne Virtualisierungs-Umgebungen wie Docker bieten sich hier als Lösung an. Aber auch die Verwendung von VMs als Test-Umgebung hat weiterhin ihre Daseinsberechtigung. Virtualisiert wird hierbei das zu testende System. Wenngleich Virtualisierungs-Umgebungen auch noch in anderer Hinsicht beim Testen hilfreich sein können, etwa bei der Virtualisierung der Build- und Test-Umgebung, ist dies nicht Gegenstand dieses Artikels. Nachfolgend

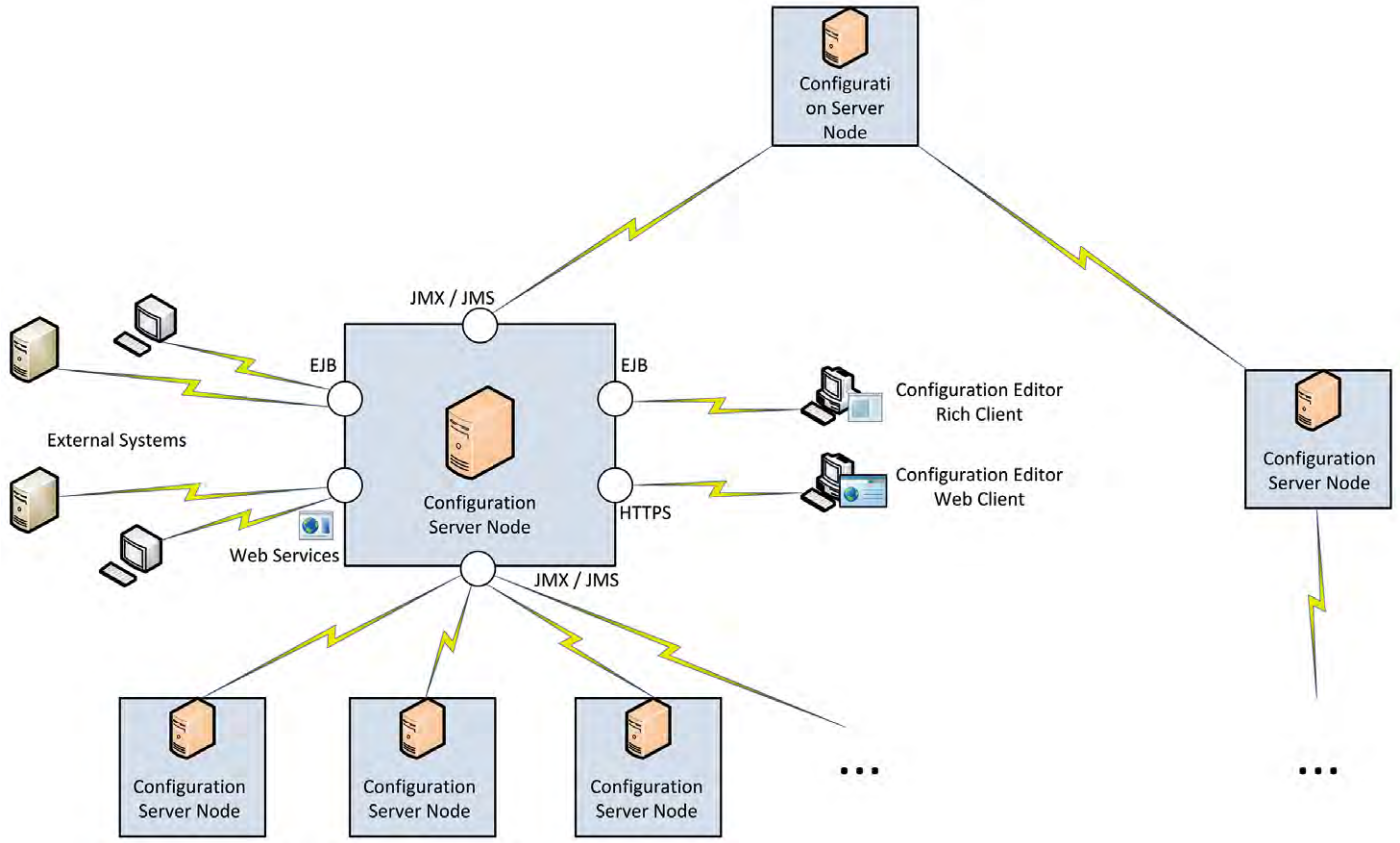

Abbildung 1: Komponenten des Konfigurationssystems

sind verschiedene Szenarien anhand eines Praxisbeispiels aufgezeigt und bewertet.

Ein Praxisbeispiel

Eines der Produkte aus dem Unternehmen des Autors ist ein weltweit verteilt eingesetztes System zur Erstellung und Verteilung von Anwendungskonfigurationen *(siehe Abbildung 1)*. Es besteht im Kern aus einer Server-Komponente mit diversen Schnittstellen, einem Web-Client und einem Rich-Client. Ein Anwendungsfall ist beispielsweise die Übertragung wichtiger Betriebsparameter für Kassen- oder Warenwirtschaftssysteme, die in verschiedenen Ländern und Regionen zu unterschiedlichen Zeiten in unterschiedlichen Ausprägungen definiert sein sollen.

Aus Gründen der Betriebssicherheit und Lastverteilung ist üblicherweise die Server-Komponente an verschiedenen Standorten verteilt im Einsatz. Die einzelnen Server-Instanzen kommunizieren miteinander, einerseits um Betriebsdaten auszutauschen, andererseits um Konfigurationsdaten weltweit zu replizieren und somit lokal zur Verfügung zu stellen. Die einzelnen Server-Instanzen bilden dabei eine Baumstruktur mit einem Top-Level- und Kind-Knoten über mehrere Ebenen; dabei kann ein Netzwerk leicht 10.000 und mehr Knoten umfassen. Für die Kommunikation der Knoten untereinander werden aktuell JMS- beziehungsweise JMX-Schnittstellen zur Verfügung gestellt, zukünftig alternativ auch weitere, etwa Webservice-Schnittstellen. Die Kommunikation erfolgt dabei primär zwischen Eltern und den direkten Kindern, für einzelne Funktionen aber auch zwischen Eltern und indirekten Kindern.

Es gibt verschiedene Arten von Clients des Systems. Die im Rahmen des Projekts entwickelten eigenen Clients dienen zur Erstellung und Änderung der Konfigurationen sowie zur Überwachung des Server-Netzwerks und nutzen dazu spezielle eigene Schnittstellen zur Kommunikation. Daneben gibt es auch eine Reihe bekannter sowie eine Reihe unbekannter externer Systeme. Diese nutzen zur Kommunikation entweder die öffentlichen EJB- oder die Web-Service-Schnittstellen. Dabei umfassen die bekannten externen Systeme sowohl projektbezogene Eigenentwicklungen als auch Systeme von Drittherstellern.

Ein Schwerpunkt liegt auf dem Last- und Performance-Test des gezeigten Systems. Es war unter anderem aufgefallen, dass unter bestimmten Bedingungen wie zu geringer Netzwerk-Bandbreite bei gleichzeitig geringem Speicherplatz die verwendete Messaging-Lösung (HornetQ) instabil wurde. Um jedoch das Szenario zu testen und Optimierungen an der Software und der HornetQ-Konfiguration vornehmen zu können sowie Empfehlungen für den Betrieb zu geben, war es notwendig, eine möglichst realistische Test-Umgebung aufzubauen. Im ersten Schritt wurden VMs eingesetzt, im zweiten Docker-Container.

Last- und Performance-Test auf VMs

Um die Umgebung, bei der die Probleme auftraten, möglichst realistisch nachbilden zu können, wurden VMs aufgesetzt, die die Realität soweit wie möglich abbildeten. Die Basis bildete ein spezieller, exklusiv für Last-und Performance-Test genutzter VMware-ESXi-Host. Auf diesem können Projekte, die eine entsprechende Testumgebung brauchen, VM-Templates anlegen, die anschließend für den eigentlichen Test vervielfältigt werden.

Als Betriebssystem für die VMs kam das beim entsprechenden Kunden verwendete Betriebssystem (Windows Server) zum Einsatz. Auf einer Template-VM wurde die gesamte notwendige Software inklusive Application-

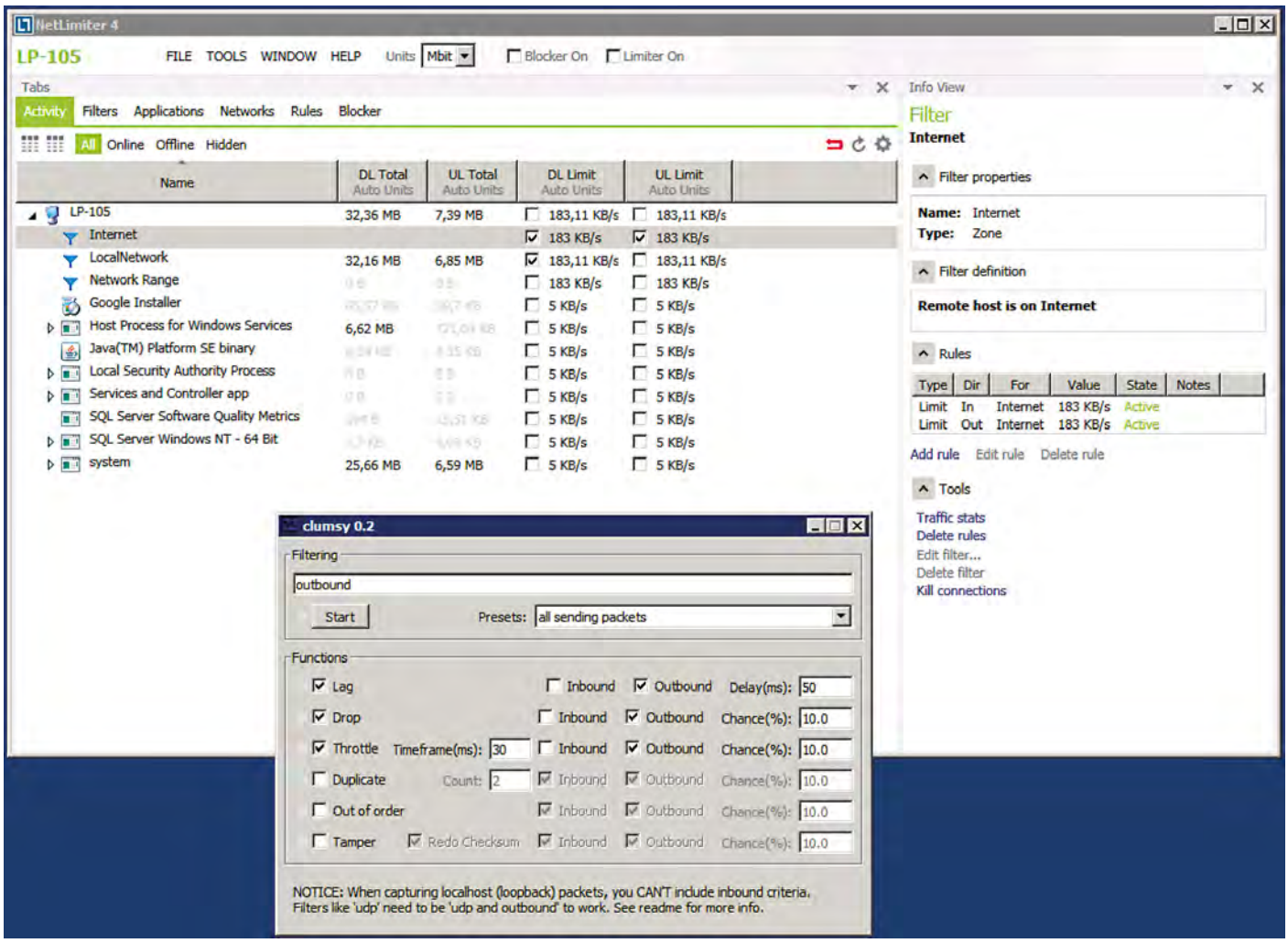

Abbildung 2: Simulation von Netzwerk-Problemen

Server und Datenbank installiert. Haupt- und Festplattenspeicher wurden soweit reduziert, dass sie dem realen System entsprachen. Um Netzwerk-Probleme wie geringe Bandbreite, Package-Loss und Verbindungsabbrüche zu simulieren, kamen zwei Werkzeuge zum Einsatz: NetLimiter *[1]* zur Begrenzung der verfügbaren Bandbreite und clumsy *[2]*, um wahrscheinlichkeitsgesteuert Netzwerk-Probleme zu erzeugen *(siehe Abbildung 2)*.

Da das Serversystem bereits die Möglichkeit einer Auto-Initialisierung bot, um das Aufsetzen der Server-Instanzen zu erleichtern, konnte diese für den Last- und Performance-Test in leicht abgewandelter Form wiederverwendet werden. Es reichte also aus, auf dem VM-Template einen Server so einzurichten, dass er mit dem System gestartet wird. Nach der Vervielfältigung ermittelten die gestarteten Server-Instanzen ihre eigene Identität und ihre Position im Netzwerk anschließend automatisch, sodass ein manuelles Aufsetzen des Netzwerks überflüssig wurde.

Auch das Monitoring der einzelnen Instanzen erfolgte teilweise über bereits vorhandene Funktionalität. Jeder Server meldet regelmäßig seinen Status an seine Eltern-Knoten; auch die im Augenblick zu verarbeitende Last ist Teil dieser Statusberechnung. Damit kann über den Top-Level-Knoten der Zustand des gesamten Netzwerks überwacht werden.

Stehen solche Funktionen nicht bereits zur Verfügung, müssen sie bereitgestellt werden. So ist beispielsweise ein manuelles Aufsetzen einer großen Anzahl von Knoten des verteilten Systems in der Regel mit viel Aufwand verbunden. Ein System, das beim Testen nicht überwacht werden kann, ist nicht testbar, da Testbarkeit neben der Steuerbarkeit der Eingaben auch die Überwachung der erwarteten Ergebnisse verlangt.

Der ESXi-Host wurde für den Test vollständig ausgelastet, es konnten knapp über sechzig Server erstellt und betrieben werden. Dies reichte für ein realistisches Szenario aus, sodass in einem ersten Schritt die HornetQ-Konfiguration optimiert werden konnte. So wurde zum Beispiel zur Reduzierung des benötigten Arbeitsspeichers das HornetQ-Paging eingeschaltet. Weitere Optimierungen wurden am System selbst vorgenommen, um die Datenreplikation bei größeren Datenmengen zu entlasten. So hat man auch eine weitere Stage zur Zwischenspeicherung von Nachrichten in einer Datenbank eingeführt, bevor sie an andere Instanzen weitergereicht werden. Als Nebeneffekt hat die Einführung der zusätzlichen Stage einen Austausch des Transport-Layers HornetQ durch eine andere Technik möglich gemacht, sodass nun alternativ unter anderem auch Web-Services genutzt werden könnten.

Eine Umgebung auf Docker

Der Einsatz von VMs für realitätsnahe Tests war aus unserer Sicht unabdingbar, allerdings konnte aufgrund der Limitierungen des ESXi-Hosts keine große Menge von Instanzen

betrieben werden. Eine Unterstützung von deutlich mehr Instanzen war jedoch eine der Anforderungen, die validiert werden mussten. Es war also eine Möglichkeit notwendig, deutlich mehr als die bisher im Test verwendeten sechzig Instanzen zu unterstützen. Es wäre natürlich möglich gewesen, die benötigten Ressourcen für den Test temporär zu mieten. Einer Nutzung von Docker *[3]* wurde allerdings der Vorzug gegeben.

Da das vorgestellte System in Java implementiert ist, auf einem Standard-Application-Server (JBoss, WildFly) läuft sowie unterschiedliche Datenbanken unterstützt, ist ein Betrieb auf einem Linux-Betriebssystem möglich, zumal Linux zu den explizit unterstützten Plattformen gehört.

Man hat daher entschieden, das System als Docker-Instanzen auf Debian-Basis aufzusetzen. Das offizielle Debian-Image für Docker ist sehr minimal gehalten und somit ein erster Schritt, um möglichst viele Instanzen betreiben zu können. Zudem wurden die Teile des Systems, die für den Last- und Performance-Test nicht relevant sind, durch Mock-Implementierungen ersetzt oder gar nicht erst eingerichtet. Die Blattknoten im Server-Netzwerk wurden beispielsweise als reine Datensenken implementiert, da insbesondere die Last auf dem Top-Level-Knoten sowie den Zwischen-Knoten von Interesse war, da es bisher nur dort bisher zu Auffälligkeiten gekommen war. Der Einsatz von Mocks und Minimal-Implementierungen sollte je nach eigenen Anforderungen kritisch hinterfragt werden, da gegebenenfalls die Ergebnisse des Tests nur bedingt oder gar nicht auf das real im Einsatz befindliche System übertragen werden können.

Docker bietet zwei unterschiedliche Methoden, ein System aufzusetzen und zu vervielfältigen. Beide basieren zunächst einmal auf einem Image. Die öffentliche Docker-Registry *[4]* stellt eine Menge von offiziellen und inoffiziellen Images zur Verfügung, sodass dies ein guter Startpunkt für das eigene System ist. Auf Basis eines Images lässt sich nun ein laufender Container starten.

Die erste Möglichkeit nimmt ein vorhandenes Image, also in unserem Fall ein Debian-Image, und startet es als Container. Anschließend kann dort die notwendige Software installiert, daraus wiederum ein Image erzeugt und dieses in einer Registry für die weitere Verwendung abgelegt werden. Aus Erfahrung des Autors ist dieses Vorgehen zwar das einfachere, allerdings kann es dazu kommen, dass gegebenenfalls auch Software installiert wird, die für den eigentlichen Zweck nicht notwendig ist, sondern lediglich zur Unterstützung des Administrators bei der Einrichtung des Containers dient, zum Beispiel die Installation eines Texteditors, um mal schnell den Inhalt einer Datei anzupassen. Dies kann den Container und damit das daraus erzeugte Image unnötig aufblähen. Es ist also die entsprechende Disziplin bei der Einrichtung des Containers geboten.

Die zweite Möglichkeit ist die Verwendung eines Docker-Files. Dieses basiert ebenso auf einem vorhandenen Image und beschreibt alle Befehle, um den Container mit notwendiger Software auszustatten und diese laufen zu lassen. Der Vorteil ist, dass man sich hier in der Regel auf die essenziell notwendigen Befehle beschränkt, um das Docker-File nicht unnötig anwachsen zu lassen. Die Installation eines Texteditors hätte hier zum Beispiel gar keinen Vorteil.

Ein weiterer Vorteil der Verwendung von Docker-Files ist die einfache Anpassung an sich ändernde Versionen von genutzter Software. Ändert sich etwa die Version des Application-Servers oder der Datenbank, so ist einfach das Docker-File entsprechend anzupassen und erneut auszuführen.

Zu beachten ist, dass jeder Befehl im Docker-File ein neues Image erzeugt und im Docker-Cache speichert. Wenn dieses bereits vorhanden ist, werden die entsprechenden Docker-Befehle nicht ausgeführt; es wird stattdessen auf das vorhandene Image im Cache zurückgegriffen. Dies sollte im Auge behalten werden, wenn beispielsweise der Download von Installationsdateien von einem Build-Server in einem Docker-File erfolgen soll. Dazu sollte einer der Docker-File-Befehle „ADD" oder „COPY" verwendet werden, die den Cache invalidieren, wenn sich der Inhalt und damit die Checksumme der hinzugefügten oder kopierten Dateien geändert haben.

Für Remote-Dateien sollte allerdings „wget" oder „curl" verwendet werden, denn „ADD" unterstützt zwar den Download von Remote-Dateien, aber keine Authentifizierung. Zudem können mit „ADD" heruntergeladene Archive nicht im gleichen Befehl wieder gelöscht werden, was wiederum die Imagegröße negativ beeinflusst.

Alternativ lässt sich der Cache deaktivieren, dann werden allerdings die Befehle im Docker-File immer vollständig abgearbeitet, wodurch sich das Aufsetzen wiederum verlangsamt. Um die Anzahl der erzeugten Images klein zu halten, sollte man Befehle – wo möglich – zusammenfassen; dazu werden Befehle einfach mit „&&" verknüpft. *Abbildung 3* zeigt einen Ausschnitt aus der Ausführung des Docker-Files, bei dem die PostGreSQL-Datenbank aufgesetzt und der Port 5432 containerübergreifend verfüg-

```
Step 9 : RUN apt-get update && apt-get -y install postgresql-${PGVERSION} postgresql-client-${PGVERSION}
 ---> Using cache
 ---> 7fd0c0260d4d
Step 10 : USER postgres
 ---> Using cache
 ---> 253bfc7b8b29
Step 11 : RUN /etc/init.d/postgresql start &&    psql --command "CREATE USER test WITH SUPERUSER PASSWORD 'test';" &&    createdb -O test test
 ---> Running in f708d2ca31a1
Starting PostgreSQL 9.4 database server: main.
CREATE ROLE
 ---> 221e15ae9f4d
Removing intermediate container f708d2ca31a1
Step 12 : RUN echo "host all  all    0.0.0.0/0  md5" >> /etc/postgresql/${PGVERSION}/main/pg_hba.conf
 ---> Running in 280966343f8f
 ---> 9d99b6240cbf
Removing intermediate container 280966343f8f
Step 13 : RUN echo "listen_addresses='*'" >> /etc/postgresql/${PGVERSION}/main/postgresql.conf
 ---> Running in 1c5367701898
 ---> ce4c41f34a8a
Removing intermediate container 1c5367701898
Step 14 : EXPOSE 5432
 ---> Running in 3b0cf5b2b657
 ---> a233038dad9e
```

Abbildung 3: Ausschnitt aus einem Docker-File-Run

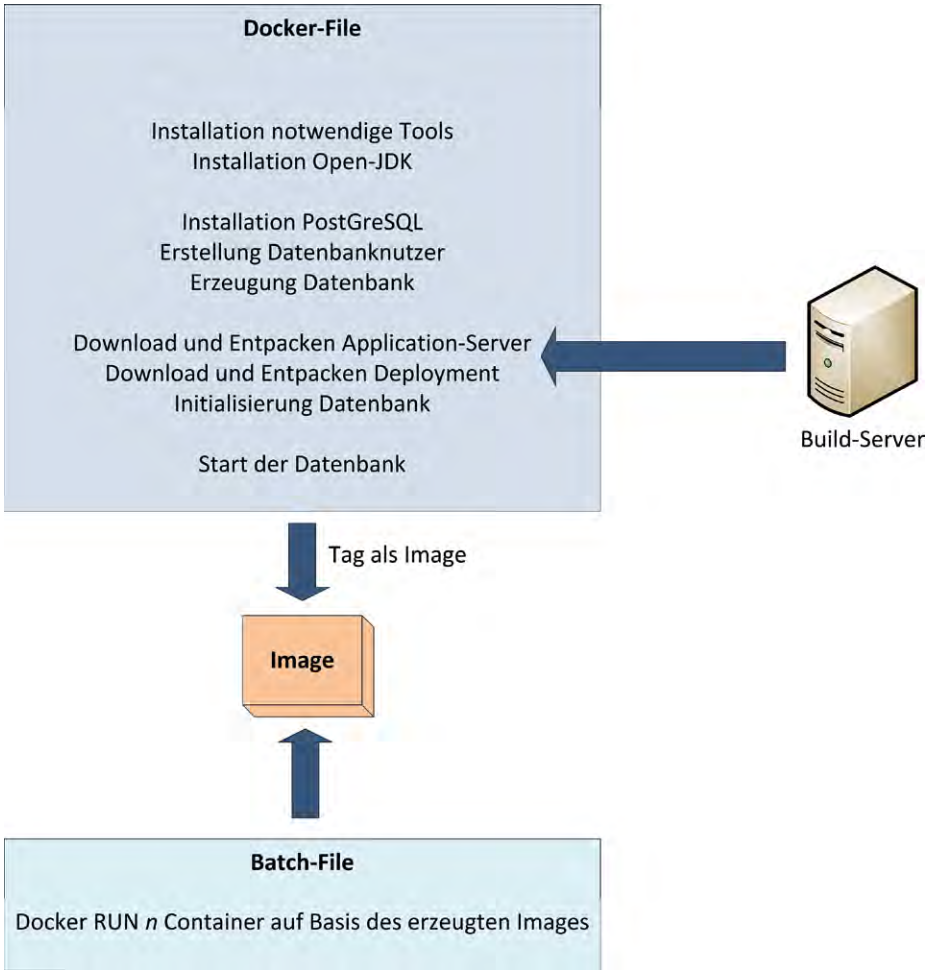

Abbildung 4: Aufsetzen der Umgebung mit Docker

bar gemacht wird („EXPOSE"). Wird auf den Cache zurückgegriffen, so ist dies bei der Ausführung des Docker-Files ersichtlich.

Der gesamte Prozess des Aufsetzens mithilfe eines Docker-Files ist schematisch in *Abbildung 4* dargestellt. Zunächst wird die benötigte Software installiert. Anschließend werden der Application-Server, eine angepasste Version mit allen benötigten Modulen und der passenden Konfiguration sowie das Deployment für den Application-Server vom Build-Server heruntergeladen. Das so erzeugte, entsprechend getaggte Image wird nun mit dem Befehl zum Starten des Application-Servers sowie der Identität des jeweiligen Containers gestartet.

Die Last- und Performance-Tests wurden auf demselben ESXi-Host durchgeführt wie die Tests mithilfe von VMs. Es konnten knapp dreihundert Docker-Container auf dem Host betrieben werden. Das Verhältnis von Instanzen auf VM zu Instanzen auf Docker beträgt also 1:5. Dabei wurde der ESXi-Host vom Docker-Host noch nicht einmal vollständig ausgelastet, sodass hier noch Spielraum nach oben ist.

Fazit

Tests unter Verwendung von VMs haben somit ebenso ihre Berechtigung wie Tests mithilfe von Docker. Beide bieten Vor- und Nachteile und eignen sich – je nach Testziel – beide gut für den Last- und Performance-Test.

Last- und Performance-Tests auf VM-Basis sind immer dann sinnvoll, wenn eine möglichst realistische Abbildung der Wirklichkeit gefordert ist und diese sich mithilfe von Docker nicht abbilden lässt. Gerade Betriebssysteme, die wie ältere Windows-Versionen keine Docker-Unterstützung bieten, lassen sich nur auf einer VM aufsetzen. So ist unter anderem das Verhalten bei vielen offenen Netzwerk-Verbindungen unter Windows grundlegend anders als unter einem Unix-basierten System. In diesem Fall hätte ein Test in Linux-basierten Docker-Containern keinerlei Aussagekraft für das Verhalten des realen Systems unter Windows.

Allerdings haben VMs den Nachteil, dass sie eher schwergewichtig sind. Es ist immer ein ganzes Betriebssystem notwendig, während Docker-Container sich Ressourcen teilen können. Zudem dauert das Aufsetzen der VMs relativ lange, da jeweils eine vollständige VM diverse Male geklont und angepasst werden muss.

Container sind sehr viel leichtgewichtiger, der Start eines Containers aus einem Image geht sehr schnell und auch das Aufsetzen der Images mithilfe von Docker-Files ist durch die Verwendung des Cache ein schneller Prozess. Zudem ermöglichen Docker-Files eine einfache Anpassung der erzeugten Images und damit der erzeugten Container bei sich ändernden Software-Versionen, während auf einer VM die Installation eines Updates notwendig ist.

Wird der Test wiederholt, startet jeder Test in einem Docker-Container „clean", ein Bereinigen oder eine Neuinstallation ist also nicht notwendig. Bei Verwendung von VMs müssen diese entweder neu geklont werden oder alternativ muss zumindest eine Herstellung des Initialzustands des zu testenden Systems erfolgen, beides eher aufwändige Prozesse. So sind in den meisten Fällen Docker-Container dem Einsatz von VMs beim Last- und Performance-Test vorzuziehen.

Referenzen

[1] clumsy: *https://jagt.github.io/clumsy*
[2] NetLimiter: *https://www.netlimiter.com*
[3] Docker: *https://www.docker.com*
[4] Offizielle Docker-Registry: *https://hub.docker.com*

Dr. Dehla Sokenou
dehla.sokenou@gebit.de

Dr. Dehla Sokenou promovierte im Jahr 2005 an der Technischen Universität Berlin über das Thema „UML-basiertes Testen objektorientierter Systeme". Seit Anfang 2006 ist sie als Senior Software Consultant bei GEBIT Solutions am Standort Berlin tätig. Neben Projektleitung, Konzeption und Entwicklung großer objektorientierter Softwaresysteme mit modellbasierten Methoden umfassen ihre Schwerpunkte modellgetriebenes Requirements Engineering und modellbasiertes Testen. Seit dem Jahr 2016 ist sie stellvertretende Sprecherin des Arbeitskreises „Testen objektorientierter Programme / Model-Based Testing" der GI-Fachgruppe „Test, Analyse und Verifikation von Software".