Testen und Docker
anhand eines Beispiels aus der Praxis

Dehla Sokenou
GEBIT Solutions
Koenigsallee 75b, 14193 Berlin, Germany
dehla.sokenou[at]gebit.de

1 Motivation

Beim Testen ist ein möglichst hoher Automatisierungsgrad anzustreben, um einerseits die Wiederholbarkeit, aber auch eine kontinuierliche Auslieferung zu gewährleisten sowie die nicht unerheblichen Kosten des Testens in den Griff zu bekommen. Dabei kann Automatisierung sowohl bei der Vorbereitung, beim Aufsetzen der Testumgebung, dem Deployment der zu testenden Anwendung wie auch bei der eigentlichen Testfallerstellung, Testdatengenerierung, Testausführung und Testauswertung zum Einsatz kommen.

Der Integrationstest verteilter und kooperierender Systeme stellt eine besondere Herausforderung dar, weil zusätzlich zum Verhalten eines einzelnen Systems die Kommunikation innerhalb des Gesamtsystems eine große Rolle spielt. Oft müssen die Systeme vor dem Test aufgesetzt und verteilt sowie der Test gegen verschiedenen Instanzen ausgeführt werden.

2 Einführung in Docker

Docker [2] ist eine Virtualisierungsumgebung, die im Gegensatz zu virtuellen Maschinen (VMs) sehr leichtgewichtig ist, was Ressourcenbedarf und die für das Clonen und Starten einer Instanz benötigte Zeit betrifft.

Viele Teile von Docker sind Open Source und unter der Apache 2.0 Lizenz veröffentlicht. Basis von Docker ist die Docker-Engine, die dazu dient, Instanzen zu erzeugen und zu verwalten.

Docker-Instanzen werden Container genannt, die eine Art schreibbaren Layer über zuvor erstellte sogenannte Images darstellen. Images sind unveränderbar, quasi ein Read-Only-Layer, und enthalten alle für den Betrieb eines Containers benötigten Applikationen. Innerhalb eines Containers läuft empfohlen nur ein Prozess, bspw. eine Datenbank, es können jedoch auch mehrere sein, allerdings sollten Container möglichst klein gehalten werden.

Docker bietet zwei Möglichkeiten, ein Image zu erzeugen. Ein laufender Container kann betreten und wieder verlassen werden. Änderungen, die in einem laufenden Container gemacht werden, z.B. die Installation weiterer Applikationen, verändern den Container. Dieser kann anschließend als neues Image abgelegt werden. Alternativ lässt sich ein Image durch ein Docker-spezifisches Script erzeugen, das Docker-File.

Docker-Images können anschließend in einer öffentlichen Docker-Registry global zur Verfügung gestellt oder in einer privaten Docker-Registry abgelegt werden.

Einen Überblick über den Docker-Lebenszyklus zeigt Abb. 1.

Daneben gibt es einige Add-Ons, die den Umgang mit Docker erleichtern. Bspw. koppelt Docker Compose verschiedene Container miteinander und mit Docker Swarm können verschiedene Docker-Hosts zu einem Cluster zusammengefasst werden.

3 Ein Beispielsystem aus der Praxis

Als Beispiel soll ein weltweit verteilt eingesetztes System zur Erstellung und Verteilung von Anwendungskonfigurationen dienen (siehe Abb. 2). Es besteht im Kern aus einer Serverkomponente mit diversen Schnittstellen, einem Web-Client und einem Rich-Client. Ein Anwendungsfall ist z.B. 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 Serverkomponente an verschiedenen Standorten verteilt im Einsatz. Die einzelnen Serverinstanzen kommunizieren miteinander, einerseits um Betriebsdaten auszutauschen, andererseits um Konfigurationsdaten weltweit zu replizieren und somit lokal zur Verfügung zu stellen. Die einzelnen Serverinstanzen bilden dabei eine Baumstruktur mit einem Top-Level-Knoten und Kindknoten über mehrere Ebenen; dabei kann ein Netzwerk leicht 10.000 und mehr Knoten umfassen.

Es gibt verschiedene Arten von Clients des Systems. Neben im Rahmen des Projekts entwickelten Clients zur Erstellung und Änderung der Konfigurationen sowie zur Überwachung des Servernetzwerks gibt es mehrere bekannte und auch unbekannte Clients, die sich über Schnittstellen des Systems mit Betriebsparametern versorgen.

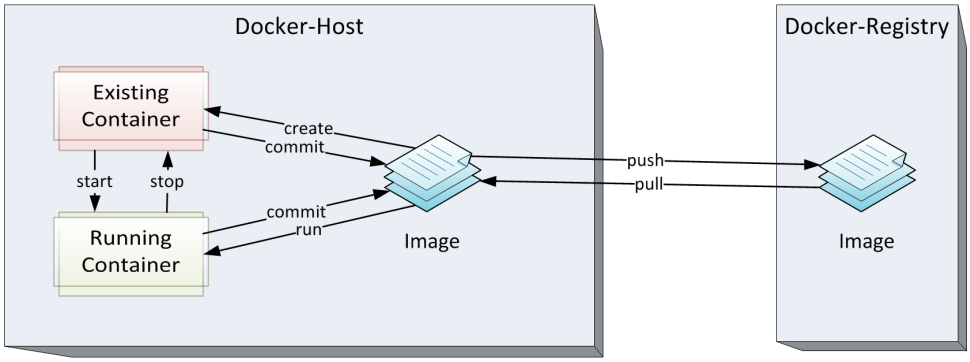

Abbildung 1: Docker-Lebenszyklus (vereinfacht)

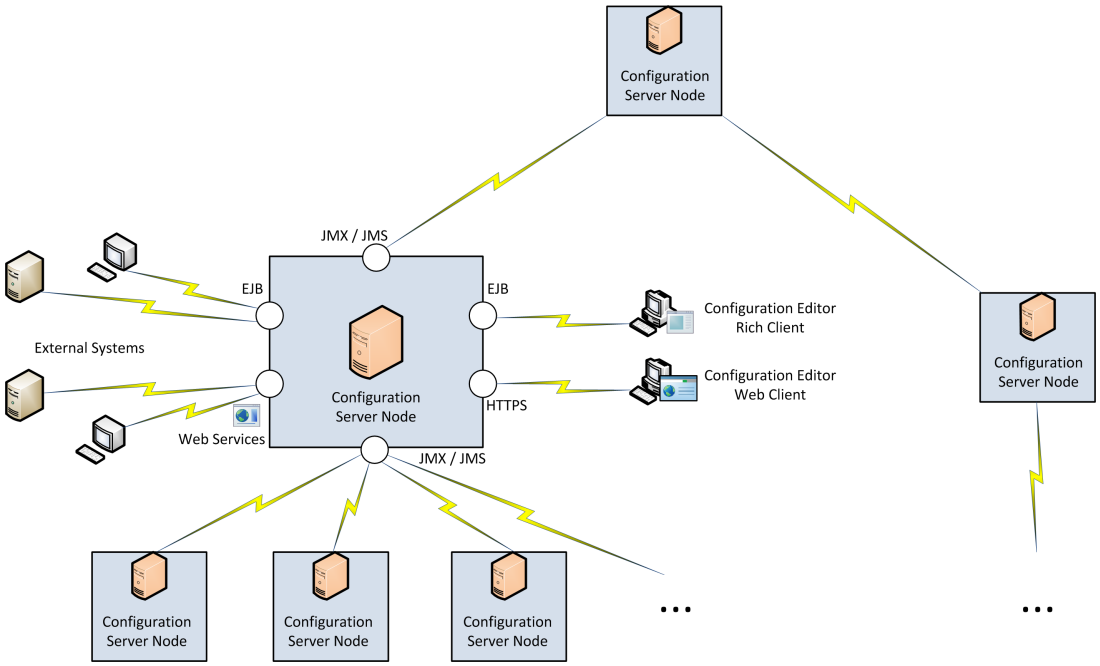

Abbildung 2: Beispielsystem

4 Docker in der Build- und Testumgebung

Oft muss für den Test eine Testumgebung aufgesetzt werden, auf die einzelne Instanzen zu verteilen und die Tests auszuführen sind. In unserem Fall wird durch den automatischen Test ein kleines Netzwerk aufgesetzt und u.a. die korrekte Replikation und Datenverarbeitung auf diesen Knoten überprüft. Jeder der Knoten benötigt einen Applikationsserver und eine Datenbank für ein realistisches Testszenario.

Um Tests parallel ausführen zu können, sollten diese Testumgebungen möglichst separiert sein. Als Beispiel seien hier die FitNesse-Runtime-Umgebung oder auch die Verteilung der zu testenden Software auf einen oder mehrere Applikationsservers oder der Abruf von Daten aus einer Datenbank genannt. Um Tests parallel laufen lassen zu können, müssen üblicherweise verschiedene Rechner / VMs für die Testausführung verwendet werden oder alternativ die Tests unter unterschiedlichen Ports oder gegen unterschiedliche Datenbankschemata laufen, um sich nicht gegenseitig zu stören. Beides hat Nachteile, insbesondere wenn verschiedene Versionen oder auch nur Feature-Branches der gleichen Software getestet werden sollen, da für diese Fälle üblicherweise der bestehende Buildjob kopiert und der Build selbst angepasst wird. Muss jedes Mal auch noch eine Anpassung der Testumgebung erfolgen, z.B. durch unterschiedliche Ports oder unterschiedliche Datenbankschemata, so ist dies aufwändig, fehlerträchtig und skaliert nicht.

Eine Alternative ist die Verwendung von Docker.

So lässt sich der Build bspw. mit Hilfe eines Jenkins CI-Servers [3] auf einen sogenannten Slave, einem weiteren Server, ausführen. Mit Hilfe des Docker-Plugins für Jenkins [4] läuft dieser Slave in einem Docker-Container und kann somit on-demand erzeugt werden (siehe Abb. 3). Der gesamte Build inklusive aller Tests wird dann vollkommen separiert von anderen Builds ausgeführt. Alle Buildartefakte sind nach

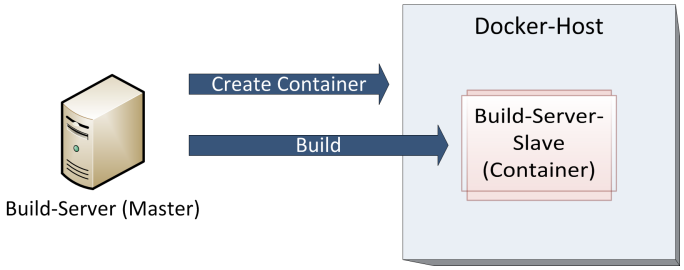

Abbildung 3: Erstellung Docker-Image aus Docker-File

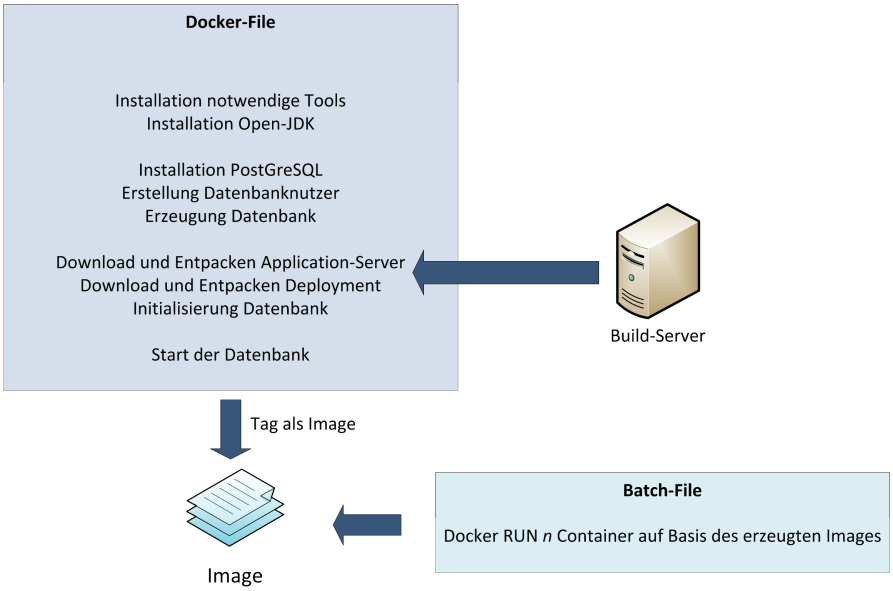

Abbildung 4: Erstellung Docker-Image aus Docker-File

dem Test in der aufrufenden CI-Server-Instanz (Master) verfügbar.

Die Verwendung von Docker hat zudem noch einen weiteren Vorteil. Im Buildprozess kommt es immer wieder einmal vor, dass die Testumgebung nicht komplett wieder gesäubert oder abgeräumt wird. Dies kann z.B. bei einem abgebrochenen oder fehlgeschlagenen Build passieren, wenn entsprechende Aufräumtasks nicht oder nur unvollständig durchgeführt werden. Ein Docker-Container dagegen wird aus einem unveränderlichen Image gestartet und nach dem Test wieder beendet. Selbst wenn das Beenden fehlgeschlagen ist, verwendet der nächste Test in jedem Fall einen neuen Container, der komplett von dem zuvor gestarteten separiert ist, der Stand ist also immer frisch und nie vom vorhergehenden Test beeinflusst. Der vom Jenkins-Master erzeugte Slave wird - je nach Konfiguration - nach dem Build entweder abgeräumt oder zur weiteren Verwendung mit einem Commit lokal oder mit einem Push in einer Docker-Registry aufbewahrt. Alternativ lassen sich durch Sharing von Ordnern im Dateisystem zwischen Jenkins-Master und -Slave die Artefakte und Testergebnisse direkt im Buildjob ablegen, so dass für den Nutzer die Verwendung des Buildservers völlig transparent ist.

5 Docker für Last- und Performancetests

Um Last- und Performancetests für die Replikation in unserem Beispielsystem durchführen zu können, müssen entsprechend viele Instanzen aufgesetzt werden, da die Anzahl der Kindknoten die vom Elternknoten zu verarbeitende Last beeinflusst. Nachdem die Anzahl der Knoten bei der Verwendung von vollwertigen VMs sehr schnell an ihre Grenzen gestoßen ist und zudem das Clonen und Starten von VMs relativ zeitintensiv war, musste ein anderer Weg gegangen werden.

Auch hierfür bietet sich Docker als leichtgewichtige Umgebung an, da einerseits das Erzeugen und Starten der Container sehr schnell geht, andererseits jeder Container deutlich weniger Ressourcen benötigt als eine VM und zudem die Images, aus denen die Container erstellt werden, in einer Registry versioniert abgelegt werden können.

Um ein Docker-Image zu erzeugen, gibt es prinzipiell zwei Möglichkeiten, erstens die Erstellung aus einem vorhandenen Container, der alle benötigten Applikationen enthält, oder zweitens die Erzeugen mit Hilfe eines Scripts, dem Docker-File. Wir haben uns

für den zweiten Weg entschieden, da bei Änderung der benötigten Applikationen, bspw. des Applikationsservers, nur eine Änderung des Scripts notwendig ist und nicht aufwändig neue Container erzeugt werden müssen, aus denen das neue Basis-Image erstellt wird. Die verwendeten Scripte können auch, wo nötig, parametrisiert werden, so z.B. um ein Script für gleich mehrere Versionen von bspw. Datenbank und Applikationsserver, aber auch zu testendem System verwenden zu können.

Der gesamte Prozess des Aufsetzens mit Hilfe eines Docker-Files ist schematisch in Abb. 4 dargestellt. Zunächst wird die benötigte Software per Kommando installiert. Anschließend werden der Applikationsserver sowie das Deployment für den Applikationsserver vom Buildserver heruntergeladen. Das so erzeugte, entsprechend getaggte Image kann nun mit dem Befehl zum Starten des Applikationsservers sowie der Identität des jeweiligen Containers gestartet werden. Für viele Container wird ein Batchscript verwendet, dass jedem Container durch Parametrisierung eine eigene eindeutige Identität zuweist, die gleichzeitig dem Hostnamen des Systems innerhalb des laufenden Containers entspricht. Alle Container kommunizieren innerhalb eines zuvor definierten Docker-Netzwerks miteinander. Der jeweils innerhalb eines Containers gestartete Knoten unseres Beispielsystems nutzt eine Art Autoinitialisierung auf Basis des Hostnamens, um automatisch seinen Platz innerhalb des Netzwerks zu ermitteln - dies erleichtert den Aufbau des Netzwerks erheblich.

Im Gegensatz zum Test mit Hilfe von VMs konnten unter Docker etwa fünfmal mehr Instanzen betrieben werden - bei gleicher zugrunde liegender Hardware, einem VMWare ESXi Host. Dies waren beim Test auf VMs maximal 60 Instanzen, beim Test mit Docker etwa 300 Instanzen. Die Anzahl der Instanzen ist bei Docker in unserem Beispiel um den Faktor 5 höher. Eine VM zu clonen und betriebsbereit zur Verfügung zu stellen, dauert mehr als 10 Minuten, einen Docker-Container zu starten dauert etwa 10 Sekunden. Der Geschwindigkeitsgewinn ergibt sich vor allem daraus, dass keine echte Datenkopie der dem Clone zugrunde liegenden VM erzeugt wird, sondern alle Docker-Container auf demselben Image, also auf dem gleichen Datenstand, basieren.

6 Alternativen

Neben Docker gibt es eine Reihe alternativer Virtualisierungsumgebungen, die jedoch zum größten Teil aktuell nicht die gleiche Bedeutung wie Docker besitzen und damit auch nicht die breite Unterstützung durch andere Werkzeuge. Deshalb wurden diese Alternativen in unserem Umfeld nicht in Betracht gezogen, sind jedoch einen weiteren Blick wert, insbesondere wenn die eigene Plattform durch Docker nur unzureichend oder gar nicht unterstützt wird. Eine Übersicht über die aktuell verfügbaren Docker-Alternativen bietet [5].

7 Zusammenfassung und Ausblick

Docker eignet sich sehr gut zur Testunterstützung. Es erleichtert insbesondere die Separierung von Tests, das Aufsetzen von Testumgebungen und den Test in unterschiedlichen Umgebungen sowie das Erstellen vieler Instanzen für Last- und Performancetests. Neben den gezeigten Beispielen gibt es eine Reihe weiterer Anwendungsfälle und Werkzeuge, bspw. unterstützt Arquilian mit Arquilian Cube [1] die Ausführung von Tests in einem Docker-Container.

Obwohl die genannten Beispiele, insbesondere der Last- und Performancetest, zunächst einmal für den Einsatz von Docker sprechen, so kann doch auch der Test auf herkömmlichen VMs sinnvoll sein, bspw. wenn das geplante Zielsystem möglichst realistisch abgebildet werden muss. Nicht alle Zielsysteme werden aktuell von Docker unterstützt und Einschränkungen von verfügbarem RAM oder Plattenplatz lassen sich unter Docker nicht definieren, zumindest in der von uns für die Tests verwendeten Dockerversion. So konnten wir nur mit VMs das letztendliche Einsatzszenario bei einem unserer Kunden eins zu eins nachstellen.

Aktuell gibt es immer noch sehr viel Bewegung im Docker-Umfeld. Es gibt in regelmäßigen Abständen neue Versionen, in denen neue Funktionen vorgestellt und des Öfteren alte Funktionen nicht mehr verfügbar sind oder verändert wurden. Laufend werden neue Add-Ons vorgestellt. Dies führt dazu, dass eine Migration auf eine neuere Docker-Version in der Regel mit einigem Aufwand verbunden ist, andererseits werden bisher nicht realisierbare Szenarien gegebenenfalls mit der nächsten Version oder dem nächsten Add-On möglich.

Referenzen

[1] Arquilian Cube.
http://arquillian.org/arquillian-cube.

[2] Docker.
https://www.docker.com.

[3] Jenkins CI.
https://jenkins.io.

[4] Jenkins Docker Plugin.
https://plugins.jenkins.io/docker-plugin.

[5] Diego Wyllie. App-Container: 5 professionelle Docker-Alternativen im Überblick. t3n News - Software & Infrastruktur,
http://t3n.de/news/docker-alternativen-container-783741/.