­­Dokumentation Assignment Cloud-Computing

# Infrastruktur:

|  |  |
| --- | --- |
| **Name** | **Zweck** |
| Putty[[1]](#footnote-1) | TelnetClient zur Serversteuerug und -verwaltung |
| SourceTree[[2]](#footnote-2) | Git Graphical User Interface (GUI) für eine einfachere Verwaltung des Repository |
| Git Repository auf GitHub[[3]](#footnote-3) | Bequeme Umverteilung auf den Server und auf einzelne Teammitglieder, Backupmöglichkeit, Portierung von z.B. Dockerfiles auf „private“ Testinstanzen |
|  |  |

Tabelle 1: Grundlegendes IT Setup

## Initiale Schritte

* Installation und Einrichtung aller obigen Programme
* Erstellung eines Git Repositories auf GitHub
* Installation von SourceTree
* Klonen des Repositories auf alle Teamrechner und dem Server

# Aufgaben

## Definitionen & Begriffserläuterungen

## CentOS Container

Ziel der Übung ist es,

|  |
| --- |
| $nano Dockerfile |

Code 1: Erstellen eines Dockerfiles

Der oben genannte Befehl ruft den Texteditor *nano[[4]](#footnote-4)* auf. Außerdem wird ein Dockerfile erstellt, welches als Grundlage für die Aufgabe dient- falls es nicht bereits vorhanden ist.

|  |
| --- |
| FROM centos:7  MAINTAINER "Florian Kaefer, Karlo Kraljic"  CMD /bin/bash |

Code 2: Dockerfile für das CentOS Image

In Code 2 befindet sich der Inhalt des Dockerfile. Das FROM Schlüsselwort gibt an, welches so genannte Base Image dem Container zugrunde liegt. Ein Base Image ist laut Docker Referenz ein Image, dem kein weiteres Image zugrunde liegt[[5]](#footnote-5). MAINTAINER gibt laut Docker Referenz die Maintainer des Containers an[[6]](#footnote-6), was im konkreten Fall Florian und Karlo sind.

CMD definiert laut Docker Referenz den Einstiegspunkt in den Container[[7]](#footnote-7). Das Kommando, welches hier übergeben wird, wird auf Konsolenebene innerhalb des startenden Containers ausgeführt. In unserem Beispiel, würde der Container die *bash* starten und den Nutzer in das Containerinterne Terminal leiten.

|  |
| --- |
| $ docker build --no-cache -t centos:7 . |

Code 3: Starten des Build Prozesses

Das Kommando in Code 3 startet den Build-Prozess Die Ergänzung --no-cache soll verhindern, das im Falle eines korrumpierten Pakets durch z.B. das instabile WLAN Netz an der DH nicht die fehlerhaft gecacheten Pakete bei erneuten Build Versuchen benutzt werden[[8]](#footnote-8).

|  |
| --- |
| Sending build context to Docker daemon 2.56 kB  Sending build context to Docker daemon  Step 0 : FROM centos:7  7: Pulling from centos  47d44cb6f252: Pull complete  8aa7f4a1dd13: Pull complete  e0f5867add13: Pull complete  eeb3a076a0be: Pull complete  Digest: sha256:b3da5267165bbaa9a75d8ee21a11728c6fba98c0944dfa28f15c092877bb4391  Status: Downloaded newer image for centos:7  ---> eeb3a076a0be  Step 1 : MAINTAINER "Florian Kaefer, Karlo Kraljic"  ---> Running in d1f76749f30b  ---> 890de4e7421c  Removing intermediate container d1f76749f30b  Step 2 : CMD /bin/bash  ---> Running in 8ee4493c13a8  ---> 7892c2457995  Removing intermediate container 8ee4493c13a8  Successfully built 7892c2457995 |

Code 4: Bash Output nach erfolgreichem Build Prozess

In Code 4 sehen wir den Output der Konsole. Man sieht, dass während Build-Prozesses zunächst die Pakete des Base Images heruntergeladen werden. Anschließend wird der Maintainer gesetzt und zum Schluss der Eingangspunkt in den Container definiert. Abgeschlossen wird der Build-Prozess durch eine Successfully built Nachricht. Weitere Informationen zum Build-Prozess und zum build Kommando finden sich in der Docker API[[9]](#footnote-9).

|  |
| --- |
| $docker run -t centos:7 |

Code 5: Starten der Bash im Container

Code 5 zeigt die eine run Instruktion. Diese Instruktion startet das Image mit dem Namen centos:7. Wie in Code 3 zu sehen ist, wurde unser CentOS image „centos:7“ benannt.

Der –t Flag sorgt dafür, dass unser Container erwartungsgemäß in unserem durch den CMD referenzierten Einstiegspunkt im Vordergrund startet. Der interessierte Leser kann für weitere Informationen zum run Kommando die Docker API nachschlagen[[10]](#footnote-10).

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Bash Kommando** | **Host OS** | **Innerhalb des Containers** |
| Laufende Prozesse  (Prozess ID + Prozess) | ps -A | 1 init      2 kthreadd      3 ksoftirqd/0      5 kworker/0:0H      7 rcu\_sched      8 rcu\_bh      9 rcuos/0     10 rcuob/0     11 migration/0     12 watchdog/0     13 khelper     14 kdevtmpfs     15 netns     16 perf     17 khungtaskd  […] | 1 bash  17 ps |
| Distribution | cat /etc/\*-release | Ubuntu 14.04.4 LTS, Trusty Tahr | CentOS Linux 7 |
| Kernel Version | uname -r | 3.19.0-49-generic | 3.19.0-49-generic |

Tabelle 2: Vergleichende Gegenüberstellung von Container und Host OS

In Tabelle 2 sieht man die Ergebnisse unseres Vergleiches von Container mit Host OS. Durch das Kommando ps –A wird laut Ubuntuusers eine Liste aller Prozesse angezeigt[[11]](#footnote-11).

Man sieht, dass sich auf dem Host OS eine Vielzahl an Prozessen in ihrer Ausführung befinden. Innerhalb des Container befinden sich genau 2 Prozesse in ihrer Ausführung; einerseits die Bash, wie wir sie im Dockerfile unter dem CMD Schlüsselwort definiert haben; andererseits der ps Prozess an sich.

Die Distributionen unterscheiden sich ebenfalls. Lennart Poettering schlägt vor, die Systeminformationen aus */etc/\*-release* herauszulesen[[12]](#footnote-12), was mit dem in der Tabelle dargestellten Kommando realisiert wird.

Um die Kernelversion zu untersuchen, wird in Ubuntuusers vorgeschlagen, den Befehl uname zu benutzen. Das –r Flag gibt die Releasenummer des Kernels aus[[13]](#footnote-13). Man sieht, dass derselbe Kernel benutzt wird, unabhängig vom tatsächlichen Betriebssystem.

Praktisch hat das folgende Auswirkung: Container sind nicht vollkommen portabel, da sie nur auf den Systemen laufen, die denselben Kernel nutzen wie es das Image verlangt.

## Grundlegende Containerbefehle

Die Docker Docs sind eine gute Quelle, um sich mit den Basisbefehlen der Container vertraut zu machen. In Tabelle 3 sieht man eine Auflistung von Befehlen, die laut Aufgabenstellung des Aufgabenblattes zu erklären sind.

|  |  |
| --- | --- |
| **Docker Befehl** | **Beschreibung** |
| docker ps <flags> | Gibt eine Liste mit den Laufenden Containern aus. Es können optional verschiedene Flags angehängt werden- der –l Flag gibt zum Beispiel den aktuellsten Container (unabhängig vom Ausführungsstatus, d.h. Running-Exited oder Paused) aus. Das –a Flag gibt alle verfügbaren Container aus.[[14]](#footnote-14) |
| docker info <cont.ID> | Gibt globale Docker Informationen aus, wie z.B.: Anzahl an Container, Anzahl an Containern in verschiedenen Zuständen, Anzahl an Images, Server Version, Betriebssyystem, Architektur uvm.[[15]](#footnote-15) |
| docker top <cont. ID> | Gibt die laufenden Prozesse eines Containers aus.[[16]](#footnote-16) |
| docker stop <cont. ID> | Beendet einen Container durch ein Betriebssystemsignal (SIGTERM und SIGKILL).[[17]](#footnote-17) |

Tabelle 3: Grundlegende Befehlsreferenz

## Docker Hub

Das Docker Hub ist eine Art Image Hosting Plattform, auf der Unternehmen und Teams ihre Images hosten und dadurch anderen Menschen zur Verfügung stellen können.

Das Docker Hub ist aus Nutzersicht kostenlos. Jeder kann sich die Images herunterladen ohne Entgelt aufbringen zu müssen. Für Zusatzfeatures, wie zum Beispiel die Möglichkeit Zugriff auf „sichere“ bzw. „vertrauensvolle“ Images zu erhalten, muss der Nutzer Entgelt aufbringen[[18]](#footnote-18).

Beispiele für vorgefertigte Images auf dem Docker Hub sind[[19]](#footnote-19):

* Busybox
* Nginx
* Ubuntu
* Redis
* CouchDB

## Aufsetzen eines Webservers

Die Aufgabenstellung besagt, dass der Webserver tutum/apache-php heruntergeladen werden soll.

Im Folgenden werden 2 Methoden gezeigt, wie man einen Webserver „downloaded“ und anschließend als Docker Container realisiert.

### Ohne Dockerfile – Download des Images vom Docker Hub

Bei der ersten der beiden Möglichkeiten wird kein vorher erstelltes Dockerfile mit einem anschließenden Build Prozess als Basis für das Image genutzt.

Stattdessen wird direkt der folgende Befehl ausgeführt:

|  |
| --- |
| $ docker run -d -p 9000:80 tutum/apache-php |

Code 6: Downloaden eines Images vom Docker Hub

Nach dem run Befehl befinden sich 2 Flags. Der –d Flag koppelt das Containerinterne Terminal vom Host OS Terminal ab, sodass wir weiterhin auf unserem Host OS Eingaben am Terminal vornehmen können. Der –p Flag sorgt dafür, dass wir unseren Webserver an einen Port binden können. Es fällt auf, dass 2 verschieden Ports angegeben müssen, die durch ein Doppelpunkt getrennt sind. Links neben dem Doppelpunkt befindet sich der externe Port, rechts davon befindet sich der interne Port des Containers. Weitere Informationen zu den Variationen des docker run Kommandos finden sich auf[[20]](#footnote-20)

Uns wurde Port 9000 durch die zufällige Verteilung der Gruppennamen zugewiesen, weswegen wir diesen nutzen. Port 80 wird standardmäßig vom Webserver benutzt (wird im Dockerfile von tutum/apache-php durch den Befehl EXPOSE 80 festgelegt, siehe Git Repository von tutum[[21]](#footnote-21)), weswegen jegliche andere Belegung durch den Maintainer zu Kommunikationsproblemen mit dem Webserver führen würde.

Anschließend erscheint der folgende Output durch die Konsole:

|  |
| --- |
| Unable to find image 'tutum/apache-php:latest' locally  latest: Pulling from tutum/apache-php  428b411c28f0: Pull complete  […]  cdced04212b6: Already exists  Digest: sha256:6d21118f57fd5415638b0744be140c16e40353e2bc06659c8c8ffeadae648cfc  Status: Downloaded newer image for tutum/apache-php:latest  510a50d510d75c382f3cc7848b1df98ab2f1f81461ed633a5f1317606b8c75d4 |

Code 7: Output beim Download eines Images vom Docker Hub

Wie man sieht, wird zunächst lokal nach einem Image mit dem Namen „tutum/apache-php:latest“ gesucht. Da dieses nicht gefunden wird, beginnt die Docker Engine auf dem Docker Hub nach einem gleichnamigen Image zu suchen. Dies war in diesem Beispiel erfolgreich, sodass der Download des Images erfolgreich abgeschlossen werden konnte.

Anschließend validieren wir das Ergebnis durch den Befehl

|  |
| --- |
| $ curl http://localhost:9000 |

Code 8: Validierung der Funktion des Webservers

Welches uns den Output des PHP Skriptes, welches auf Serverseite (innerhalb des Containers) ausgeführt wird, ausgibt:

|  |
| --- |
| <html>  <head>  <title>Hello world!</title>  <link href='http://fonts.googleapis.com/css?family=Open+Sans:400,700' rel='stylesheet' type='text/css'>  <style>  body {  background-color: white;  text-align: center;  padding: 50px;  font-family: "Open Sans","Helvetica Neue",Helvetica,Arial,sans-serif;  }  #logo {  margin-bottom: 40px;  }  </style>  </head>  <body>  <img id="logo" src="logo.png" />  <h1>Hello world!</h1>  <h3>My hostname is 510a50d510d7</h3> </body>  </html> |

Code 9: Output nach erfolgreicher Validierung des Webservers

### Manueller Build Prozess mit automatischem Download aller benötigten Files

Bei dieser Methode wird nicht das gesamte Image heruntergeladen. Stattdessen werden ausgehend vom Dockerfile alle nötigen, einzelnen Files als Pakete heruntergeladen und anschließend zu einem Image assembliert.

Da es sich beim Webserver von Apache um ein komplexeres Softwareprodukt handelt, benötigt man mehrere zusätzliche Dateien & Ordner, die man vom Git Hub Repository von Apache herunterladen kann[[22]](#footnote-22):

* makefile
* run.sh
* *samples/\*.\**

Das Dockerfile besitzt folgenden Inhalt:

|  |
| --- |
| FROM ubuntu:trusty  MAINTAINER Fernando Mayo <fernando@tutum.co>  RUN apt-get update && \  DEBIAN\_FRONTEND=noninteractive apt-get -yq install \  […]  RUN /usr/sbin/php5enmod mcrypt  RUN echo "ServerName localhost" >> /etc/apache2/apache2.conf && \  sed -i "s/variables\_order.\*/variables\_order = \"EGPCS\"/g" /etc/php5/apache2/php.ini  ENV ALLOW\_OVERRIDE \*\*False\*\*  ADD run.sh /run.sh  RUN chmod 755 /\*.sh  RUN mkdir -p /app && rm -fr /var/www/html && ln -s /app /var/www/html  ADD sample/ /app  EXPOSE 80  WORKDIR /app  CMD ["/run.sh"] |

Code 10: Apache PHP Webserver Dockerfile

Gut zu erkennen ist, wieso die beiden zusätzlichen Files makefile und run.sh zwingend benötigt werden: Durch den Befehl ADD werden laut Docker API die Dateien und Ordner vom als ersten Parameter angegeben relativen Pfad zur Dockerfile Quelle zum als zweiten Parameter angegeben Pfad (absolut) innerhalb des Containers kopiert[[23]](#footnote-23).

Durch den folgenden Befehl wird das Image erstellt:

|  |
| --- |
| $ docker build --no-cache -t tutum/apache-php . |

Code 11: Starten des Downloads der Rohdateien des Images – Start des Build Prozess

Anschließend kann man das Image ähnlich wie in Kapitel 2.5.2 starten, mit dem einzigen Unterschied, dass nun das Image sofort lokal gefunden wird und nicht erst vom Docker Hub heruntergeladen werden muss.

## Individualisierung des Webservers

Ziel der Aufgabe ist es, ein Image zu erstellen, welches den Login Namen als Tag beinhaltet.

Hierfür nehmen wir dasselbe Image wie in Kapitel Code 10 zu sehen ist. Durch das modifizieren des build Befehls kann man den Namen des Tags variieren. Um dies zu demonstrieren, wird einerseits Code 11, andererseits das folgende Code Snippet benutzt, um das Image zu bauen:

|  |
| --- |
| $ docker build --no-cache -t tutum/apache-php:mueller . |

Code 12: Modifizierter bzw. individualisierter Build Befehl

Die beiden Codes unterscheiden sich hinsichtlich ihrer Namensgebung des Images. In Code 12 wird durch ein Doppelpunkt (ähnlich wie beim Portbinding des Containers) einerseits links der Repository Name angegeben, andererseits rechts davon (optional) ein Tag verliehen[[24]](#footnote-24), was in diesem Beispiel der Tag „mueller“ wäre.

Anschließend vergleichen wir das Ergebnis der beiden Build Prozesse. Die Docker API schlägt hierfür folgenden Befehl vor[[25]](#footnote-25):

|  |
| --- |
| $ docker images -a |

Code 13: Anzeigen aller verfügbaren Images

Und sehen folgende Ergebnisse (irrelevante Images wurden aus Gründen der Lesbarkeit ignoriert):

|  |
| --- |
| REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE |
| tutum/apache-php mueller 907b6de8b09f 2 minutes ago 244.8 MB |
| tutum/apache-php latest cdced04212b6 9 months ago 244.7 MB |

Code 14: Vergleich zwischen den beiden unterschiedlichen Namensgebungen

Wie man sieht, haben wir erfolgreich den Tag in „mueller“ geändert.

## Änderung der Index.php innerhalb des laufenden Containers

Um die index.php File zu bearbeiten muss man mittels des folgenden Kommandos auf die Bash des Containers zugreifen:

|  |
| --- |
| $ docker exec -i -t <docker-id> bash |

Code 15: Öffnen der Bash eines laufenden Containers

Mit dem VIM Texteditor wird die index.php Datei anschließend mit dem folgenden Kommando geöffnet:

|  |
| --- |
| $ vm index.php |

Code 16: Öffnen der index.php im VIM Editor

Zum Einfügen neuer Textzeichen ist es notwendig, in den Insert-Mode innerhalb des VIM zu wechseln. Dies ist mit einem Tastendruck auf den Buchstaben "i" möglich. Nach der Eingabe des Textes kann mit :wq der VIM Editor verlassen und gleichzeitig die Datei gespeichert werden. Mehr Informationen zu den Modi und Befehlen finden sich auf Ubuntuusers[[26]](#footnote-26).

## Änderung des Dockerfiles um dasselbe Ergebnis zu erhalten wie in 2.7

### Variante 1: Ändern von index.php im tutum/apache-php Ordner

Die einfachere von 2 möglichen Varianten ist die Änderung des index.php Files in unserer Ordnerstruktur (siehe: *./sample/index.php*). Die Änderung, die vorgenommen wurde, ist rot markiert:

|  |
| --- |
| […]  <body>  <img id="logo" src="logo.png" />  <h1><?php echo "Hello Karlo and Florian greet you"; ?></h1>  <?php if($\_ENV["HOSTNAME"]) {?><h3>My hostname is <?php echo $\_ENV["HOSTNAME"]; ?></h3><?php } ?>  <?php  $links = [];  foreach($\_ENV as $key => $value) {  if(preg\_match("/^(.\*)\_PORT\_([0-9]\*)\_(TCP|UDP)$/", $key, $matches)) {  […] |

Code 17: Modifiziertes index.php

Der Build Befehl aus Code 12 startet den Build Prozess und generiert unser Image mit dem modifizierten index.php Files.

Nun starten wir unsere modifizierte Variante durch den folgenden Befehl:

|  |
| --- |
| $ docker run -d -p 9001:80 tutum/apache-php:mueller |

Code 18: Starten des modifizierten Containers

Und Verifizieren das Ergebnis durch Starten des Containers in der Bash mit dem folgenden Befehl:

|  |
| --- |
| $ docker exec -i -t 510a50d510d7 bash |

Code 19: Starten des Containers in der Bash

Der Output wird mit folgendem Befehl generiert:

|  |
| --- |
| $ root@9f3e441d4178: curl http://localhost:9001 |

Code 20: Validieren des Outputs

Beim Output beweist die folgende Zeile, dass die Änderung wirksam ist:

|  |
| --- |
| <h1>Hello Karlo and Florian greet you</h1> |

Code 21: Modifizierter Output index.php

### Variante 2: Überschreiben von index.php während des Builds

Die zweite, weniger performante Variante wäre das Überschreiben der index.php durch eine andere Variante von index.php, die sich im übergeordneten Ordner befindet (im selben wie das Dockerfile). Diese Datei wird beim Build Prozess in den Container kopiert. Dies wird durch einen zusätzlichem ADD Befehl realisiert[[27]](#footnote-27), der direkt im Anschluss an das Kopieren des /samples/ Ordners vom Host System zum Ordner durchgeführt wird. Der Rest der Intruktionen bleibt identisch. Nach der Modifikation sieht das Dockerfile wie folgt aus (die Änderung ist rot markiert):

|  |
| --- |
| FROM ubuntu:trusty  MAINTAINER Fernando Mayo <fernando@tutum.co>  # Install base packages  RUN apt-get update && \  […]  curl -sS https://getcomposer.org/installer | php -- --install-dir=/usr/local/bin --filename=composer  RUN /usr/sbin/php5enmod mcrypt  RUN echo "ServerName localhost" >> /etc/apache2/apache2.conf && \  sed -i "s/variables\_order.\*/variables\_order = \"EGPCS\"/g" /etc/php5/apache2/php.ini  ENV ALLOW\_OVERRIDE \*\*False\*\*  # Add image configuration and scripts  ADD run.sh /run.sh  RUN chmod 755 /\*.sh  # Configure /app folder with sample app  RUN mkdir -p /app && rm -fr /var/www/html && ln -s /app /var/www/html  ADD sample/ /app  ADD index.php /app/  EXPOSE 80  WORKDIR /app  CMD ["/run.sh"] |

Code 22: Modifiziertes Dockerfile Variante 2

Die index.php bleibt gleich wie die modifzierte Variante in Kapitel 2.8.1. Durch die restlichen Befehle aus Kapitel 2.8.1 wird zunächst der Build Prozess angestoßen und anschließend die Funktionsweise verifiziert. Der Output bleibt derselbe, was die Funktionsweise dieser Variante beweist.

## Ändern des logLevels eines Containers

Ziel dieser Aufgabe ist es, zunächst 2 Container zu starten, die als Entry Point einen laufenden SSH Server beinhalten. Beide Container sollen verschiedene Log Level anzeigen.

Um einen SSH Server zu erhalten, wird zunächst das folgende Dockerfile erstellt:

|  |
| --- |
| FROM ubuntu:latest  MAINTAINER "Florian Kaefer, Karlo Kraljic"  # Install base packages  RUN apt-get update && \  DEBIAN\_FRONTEND=noninteractive apt-get -yq install \  openssh-server \  curl \  apache2 \  libapache2-mod-php5 \  php5-mysql \  php5-mcrypt \  php5-gd \  php5-curl \  php-pear \  php-apc && \  rm -rf /var/lib/apt/lists/\* && \  curl -sS https://getcomposer.org/installer | php -- --install-dir=/usr/local/bin --filename=composer  RUN /usr/sbin/php5enmod mcrypt  RUN echo "ServerName localhost" >> /etc/apache2/apache2.conf && \  sed -i "s/variables\_order.\*/variables\_order = \"EGPCS\"/g" /etc/php5/apache2/php.ini  RUN sed -i 's/Port 22/Port 9000/g' /etc/ssh/sshd\_config  RUN sed -i 's/PasswordAuthentication yes/PasswordAuthentication no/g' /etc/ssh/sshd\_config  RUN sed -i '137s/.\*/LogLevel trace1/g' /etc/apache2/apache2.conf  ENV ALLOW\_OVERRIDE \*\*False\*\*  # Add image configuration and scripts  ADD run.sh /run.sh  RUN chmod 755 /\*.sh  # Configure /app folder with sample app  RUN mkdir -p /app && rm -fr /var/www/html && ln -s /app /var/www/html  ADD sample/ /app  ADD index.php /app/  EXPOSE 80  WORKDIR /app  CMD ["/run.sh"] |

Code 23: Dockerfile für einen SSH Server - Entwicklungsumgebung

Man sieht, dass im Gegensatz zu dem in Kapitel 2.5 vorgestellten Webserver das open-ssh Paket zusätzlich installiert wird.

**Wichtiger Hinweis:** Der SSH Server **MUSS** vor den anderen Paketen installiert werden. Ansonsten überschreibt die Installation des SSH Servers andere Dateien, die vom tutum/apache-php Server benötigt werden- mit der Konsequenz, dass der Container sich sofort beendet (Status: Exit). Konkret heißt das, dass kein Image das fertige tutum/apache-php Image als zugrundeliegendes Image nutzen kann. Stattdessen muss ein Base Image erstellt werden, in dem alle Installation in der richtigen Reihenfolge durchgeführt werden.

Außerdem wird der SSH Port, welcher per Default auf 22 liegt, auf 9000 geändert. Nach der Portänderung findet eine weitere Änderung in der sshd\_config Datei statt, welche das Aussetzen der Passwortauthentifizierung zur Folge hat. Für weitere Informationen zum sshd\_config File verweisen wir auf www.linux.die.net[[28]](#footnote-28).

Anschließend wird der Log Level in der */etc/apache2/apache2.conf* abgeändert, um den gewünschten LogLevel zu erhalten. Die apache2.conf ist für weite Teile der Serverkonfiguration zuständig- in unserem Fall editieren wir sie um das gewünschte LogLevel zu erreichen[[29]](#footnote-29).

Für die Produktivumgebung wird das gleiche Dockerfile benutzt. Anstatt LogLevel trace1 wird der LogLevel Error gesetzt, was im folgenden Code deutlich wird:

|  |
| --- |
| FROM ubuntu:latest  MAINTAINER "Florian Kaefer, Karlo Kraljic"  # Install base packages  RUN apt-get update && \  DEBIAN\_FRONTEND=noninteractive apt-get -yq install \  openssh-server \  […]  RUN sed -i '137s/.\*/LogLevel error/g' /etc/apache2/apache2.conf  […]  CMD ["/run.sh"] |

Code 24: Dockerfile für einen SSH Server- Produktivumgebung

Der Einstiegspunkt, welcher mit dem CMD Kommando ausgeführt wird, bleibt der Selbe.

Im run.sh Skript wird folgende Zeile ergänzt, um den SSH Service zu starten[[30]](#footnote-30):

|  |
| --- |
| service ssh start |

Code 25: Ergänzung im run.sh Skript

Um das Image zu Builden, Runnen, die Funktionsweise des Servers zu verifizieren und anschließend zu zeigen, dass der SSH Server funktionsbereit ist, werden die folgenden Kommandos ausgeführt:

|  |
| --- |
| $ docker build --no-cache -t ssh:ssh . |

Code 26: Build Befehl für den SSH Server

Dieser Befehl startet den Build Prozess für das Image.

Das Image wird mit dem folgenden Befehl gestartet, extern an den Port 9000 gebunden und intern an Port 80 gebunden:

|  |
| --- |
| $ docker run -d -p 9000:80 ssh:ssh |

Code 27: Run Befehl für den SSH Server

Aus Gründen der Einfachheit wird auf die Auflistung aller laufenden Container und die Statusüberprüfung des in Code 26 gestarteten Containers an dieser Stelle verzichtet.

Der letzte Schritt auf der Host Shell ist das Starten der Bash innerhalb des Containers:

|  |
| --- |
| $ docker exec -i -t cda764368879 bash |

Code 28: Starten der Bash in einem Laufenden Container

Die Verifizierung des tutum/apache-php Servers wird wie in Kapitel 2.8 beschrieben durchgeführt.

Nach erfolgreicher Verifizierung des Servers wird der Status des SSH Servers überprüft.

|  |
| --- |
| $ root@cda764368879: netstat –lnput | grep ssh |

Code 29: Überprüfen des SSH Servers

Netstat liefert laut Ubuntuusers ein „Diagnose-Werkzeug“ für Netzwerkschnittstellen. Um die Lauffähigkeit unseres SSH Servers zu beweisen, werden die Informationen für unseren SSH Server angegeben (der Port wurde wie in Code 23 beschrieben zu 9000 geändert)[[31]](#footnote-31):

|  |
| --- |
| tcp 0 0 0.0.0.0:9000 0.0.0.0:\* LISTEN 23/sshd tcp6 0 0 :::9000 :::\* LISTEN 23/sshd |

Code 30: Verifizierung des SSH Servers

Falls der SSH Server nicht lauffähig wäre, würde dieser Output nicht auf dem Terminal erscheinen.

Eine andere Möglichkeit, die Lauffähigkeit des SSH Servers zu überprüfen, liegt in der Verbindung mit dem SSH Server. Code 31 zeigt die hierfür nötigen Befehle[[32]](#footnote-32), Code 32 die Antwort des SSH Servers. Zur Erinnerung: Um mögliche Schnittstellen mit anderen Gruppen zu verhindern, haben wir denn Default Port 22 auf 9000 umgelegt.

|  |
| --- |
| $ ssh localhost -p 9000 |

Code 31: Verbindung zum SSH Server mithilfe von Ubuntu Befehlen

|  |
| --- |
| The authenticity of host '[localhost]:9000 ([127.0.0.1]:9000)' can't be established.  ECDSA key fingerprint is b1:ec:18:e5:90:5d:d7:dc:fa:be:f1:39:10:04:96:e1. |

Code 32: Antwort des SSH Servers

Der Output beweist, dass der SSH Server funktioniert.

Nun wird noch überprüft, ob der tutum/apache-php Server ebenfalls als Prozess innerhalb des Containers gelistet ist. Code 33 listet alle laufenden Prozesse im Container auf[[33]](#footnote-33), Code 34 stellt den (aus Platzgründen gekürzten) Output dar.

|  |
| --- |
| $ docker top 2adbbd1ebf16 |

Code 33:Auflisten aller Prozesse innerhalb von Containern

|  |
| --- |
| UID PID CMD  Root 29406 apache2 -D FOREGROUND  root 29447 /usr/sbin/sshd |

Code 34: Laufende Prozesse innerhalb des SSH Containers

Zu erkennen ist, dass sowohl der SSH Prozess als auch der apache Prozess am Laufen sind.

Zum Schluss werden die Änderungen an den beiden apache2.conf Dateien untersucht. Da die folgenden Zeilen innerhalb der apache2.config Dateien inkludiert sind, wurde auch diese Änderung funktionsbereit umgesetzt:

|  |
| --- |
| # LogLevel: Control the severity of messages logged to the error\_log.  # Available values: trace8, ..., trace1, debug, info, notice, warn,  # error, crit, alert, emerg.  # It is also possible to configure the log level for particular modules, e.g.  # "LogLevel info ssl:warn"  #  LogLevel trace1 |
| # LogLevel: Control the severity of messages logged to the error\_log.  # Available values: trace8, ..., trace1, debug, info, notice, warn,  # error, crit, alert, emerg.  # It is also possible to configure the log level for particular modules, e.g.  # "LogLevel info ssl:warn"  #  LogLevel error |

Code 35: Inhalt der beiden unterschiedlichen apache2.conf Dateien mit Fokus auf das LogLevel

## Verknüpfung einer wordpress instanz mit einer mysql-Datenbank

Ziel der Zusatzaufgabe ist ein tutum/apache-php Image, welches eine Wordpress Instanz enthält, die mit der MySQL-Datenbank verknüpft wird. Es werden hierbei zwei Docker Container mit jeweils tutum/apache und MySQL gestartet und anschließend verlinkt.

Folgende Dateien sind für die Aufgabe notwendig:

* Dockerfile
* run.sh
* wordpress.conf
* wordpress\_deployment.sh
* wp-config.php

In Code 33 ist ein Auszug des Wordpress-Dockerfile ersichtlich. Von zentraler Bedeutung ist die erste Zeile, welche eine bereits vorgefertigte tutum/apache-php aus dem Internet herunterlädt. Wir haben diese vorgefertigte Variante genutzt, um die Übersichtlichkeit in der Dokumentation zu steigern- das zugrunde liegende Image ist identisch mit den vorherigen Kapiteln. Die Kommandos, die spezifiziert wurden, dienen der Wordpress Installation.

|  |
| --- |
| FROM tutum/apache-php:latest  MAINTAINER "Florian Kaefer, Karlo Kraljic"  ENV WORDPRESS\_VER 4.3.1  WORKDIR /  […] |

Code 36 Auszug der Wordpress-Dockerfile

Das wordpress\_deployment.sh (das Shellscript wurde zuvor als „executable“ durch chmod +x gekennzeichnet[[34]](#footnote-34)). Es dient dazu, das Starten der Wordpress Instanzen zu vereinfachen:

|  |
| --- |
| docker build --no-cache -t wordpress .  docker run -d -e MYSQL\_PASS="password" --name db -p 3306:3306 tutum/mysql:5.5  docker run -d --link db:db -e DB\_PASS="password" -p 9000:80 wordpress |

Code 37 Shell Script zur Installation beider Container und anschließender Verlinkung

Die erste Zeile erstellt mithilfe des Dockerfiles ein Image mit dem Namen „wordpress“. Anschließend wird mittels des run Befehls ein MySQL-Datenbank Image von Apache vom Docker Hub heruntergeladen und ausgeführt. Zusätzlich wird das oben genannte Passwort gesetzt, und der interne sowie der externe Port auf 3306 gesetzt. Die MySQL Instanz im Container ist funktionsbreit.

Zuletzt wird das in der ersten Zeile erstellte „wordpress“ Image ausgeführt. Hierbei wird die Wordpress-Instanz mittels link mit der Datenbank verknüpft. Ein Link ist laut Docker Dokumentation eine sichere Verbindung zwischen zwei Containern.[[35]](#footnote-35) Zugrunde liegen Linux Bridges (virtuelle Switches), welche keine weiteren Schnittstellen zur Außenwelt haben[[36]](#footnote-36).

Wichtig hierbei ist, dass das Passwort auf beiden Seiten identisch ist. Der Beweis, dass sowohl der Wordpress Container als auch die Datenbank funktionieren, ist Abbildung 1: Es wird eine funktionsfähige Wordpress Instanz im Browser gezeigt.

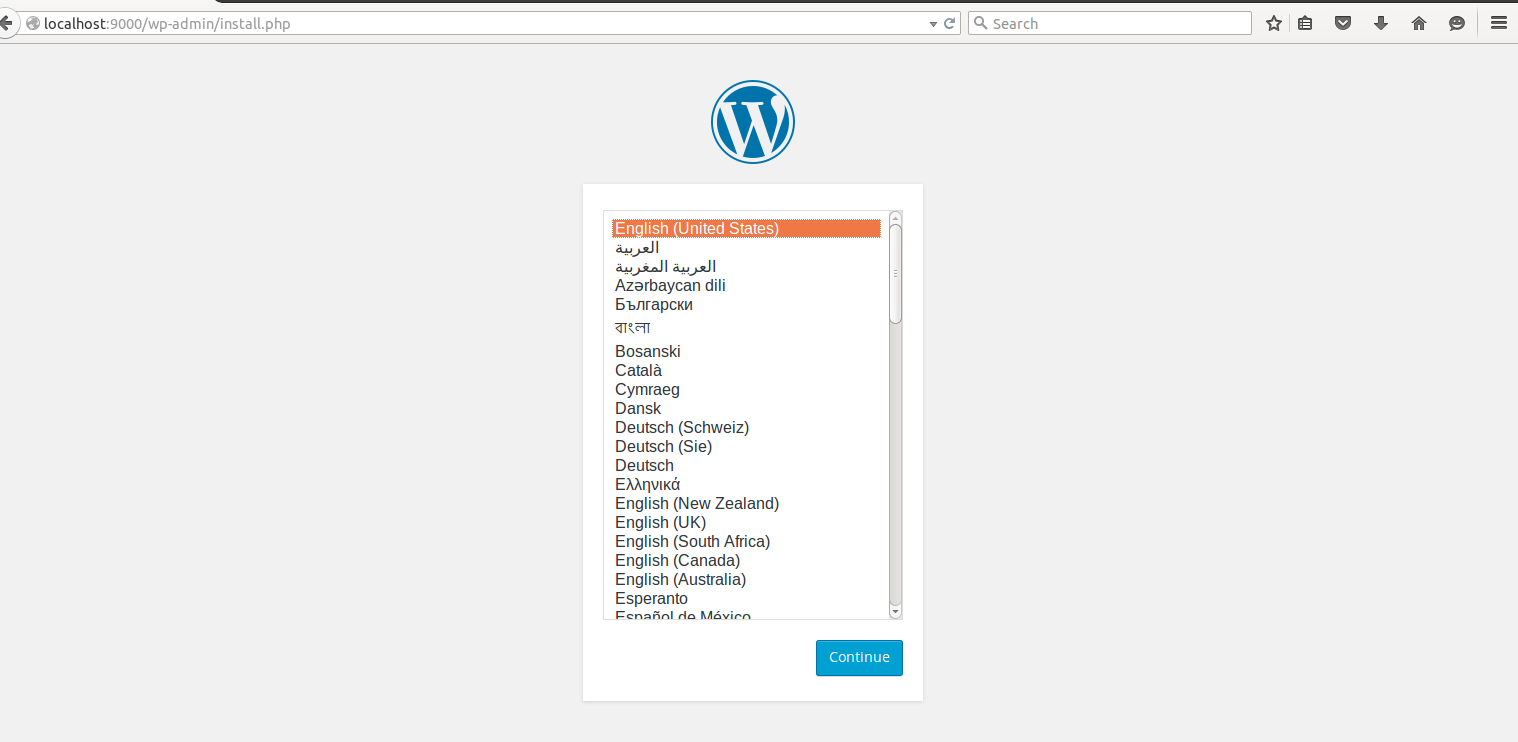


Abbildung 1 Laufende Wordpress-Instanz im Browser

Wir nutzten zum Beweis der Funktion eine virtuelle Maschine, da unsere Cloud Instanz keine grafische Oberfläche installiert hat.

**Wichtiger Hinweis:** Das Dockerfile von tutum/mysql ist unbrauchbar. Zum Einen fehlen in der Variante vom 25.04.16 in allen Versionen ein RUN chmod +x \*.sh um die inkludierten Shellscripts auszuführen (das Resultat ist ein Error beim Build Prozess). Zum Anderen läuft die Installation schief, was man daran erkennt, dass der MySQL Container direkt nach dem Absenden des docker run Befehls in den Status „Exit“ wechselt. Als Workaround nutzten wir das fertige Image vom Docker Hub und starteten den Container mit den oben genannten Befehlen.

1. http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html [↑](#footnote-ref-1)
2. https://www.sourcetreeapp.com/download/ [↑](#footnote-ref-2)
3. https://github.com/KKraljic/cloud-computing-assignments/ [↑](#footnote-ref-3)
4. https://wiki.ubuntuusers.de/Nano/ [↑](#footnote-ref-4)
5. https://docs.docker.com/engine/reference/glossary/#base-image [↑](#footnote-ref-5)
6. https://docs.docker.com/engine/reference/builder/#maintainer [↑](#footnote-ref-6)
7. https://docs.docker.com/engine/reference/builder/#cmd [↑](#footnote-ref-7)
8. https://docs.docker.com/compose/reference/build/ [↑](#footnote-ref-8)
9. https://docs.docker.com/engine/reference/commandline/build/ [↑](#footnote-ref-9)
10. https://docs.docker.com/engine/reference/run/ [↑](#footnote-ref-10)
11. https://wiki.ubuntuusers.de/ps/ [↑](#footnote-ref-11)
12. http://0pointer.de/blog/projects/os-release.html [↑](#footnote-ref-12)
13. https://wiki.ubuntuusers.de/uname/ [↑](#footnote-ref-13)
14. https://docs.docker.com/engine/reference/commandline/ps/ [↑](#footnote-ref-14)
15. https://docs.docker.com/engine/reference/commandline/info/ [↑](#footnote-ref-15)
16. https://docs.docker.com/engine/reference/commandline/top/ [↑](#footnote-ref-16)
17. https://docs.docker.com/engine/reference/commandline/stop/ [↑](#footnote-ref-17)
18. https://docs.docker.com/docker-hub/ [↑](#footnote-ref-18)
19. https://hub.docker.com/explore/ [↑](#footnote-ref-19)
20. https://docs.docker.com/engine/reference/run/ [↑](#footnote-ref-20)
21. https://github.com/tutumcloud/apache-php/blob/master/Dockerfile [↑](#footnote-ref-21)
22. https://github.com/tutumcloud/apache-php [↑](#footnote-ref-22)
23. https://docs.docker.com/engine/reference/builder/#add [↑](#footnote-ref-23)
24. [↑](#footnote-ref-24)
25. https://docs.docker.com/engine/reference/commandline/images/ [↑](#footnote-ref-25)
26. https://wiki.ubuntuusers.de/VIM/ [↑](#footnote-ref-26)
27. https://docs.docker.com/engine/reference/builder/#add [↑](#footnote-ref-27)
28. http://linux.die.net/man/5/sshd\_config [↑](#footnote-ref-28)
29. https://www.digitalocean.com/community/tutorials/how-to-configure-the-apache-web-server-on-an-ubuntu-or-debian-vps [↑](#footnote-ref-29)
30. https://wiki.ubuntuusers.de/Dienste/ [↑](#footnote-ref-30)
31. https://wiki.ubuntuusers.de/netstat/ [↑](#footnote-ref-31)
32. https://wiki.ubuntuusers.de/SSH/ [↑](#footnote-ref-32)
33. https://docs.docker.com/engine/reference/commandline/top/ [↑](#footnote-ref-33)
34. https://wiki.ubuntuusers.de/chmod/ [↑](#footnote-ref-34)
35. https://docs.docker.com/engine/userguide/networking/default\_network/dockerlinks/ [↑](#footnote-ref-35)
36. https://wiki.ubuntuusers.de/Netzwerkbr%C3%BCcke/ [↑](#footnote-ref-36)