­­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 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 in der 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.

Uns wurde Port 9000 durch die zufällige Verteilung der Gruppennamen zugewiesen, weswegen wir diesen nutzen. Port 80 wird standardmäßig vom Webserver benutzt (wird im Quellcode des Servers festgelegt), 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[[20]](#footnote-20):

* 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[[21]](#footnote-21).

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 der folgende 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 vom Doppelpunkt der Repository Name angegeben, andererseits rechts davon (optional) ein Tag verliehen[[22]](#footnote-22), 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[[23]](#footnote-23):

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

Code 13: Anzeigen aller verfügbaren Images

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

|  |
| --- |
| 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.

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://github.com/tutumcloud/apache-php [↑](#footnote-ref-20)
21. https://docs.docker.com/engine/reference/builder/#add [↑](#footnote-ref-21)
22. [↑](#footnote-ref-22)
23. https://docs.docker.com/engine/reference/commandline/images/ [↑](#footnote-ref-23)