

# **Docker**

-

## **Einsatz als Entwicklungstool im Projekt eTrading Pro**

Evaluation

Author	Roman Würsch, LIFHF
Rolle	Entwicklungsleiter, eTrading Pro
Im Auftrag der	Zürcher Kantonalbank

Juni 2015

## **Zusammenfassung**

Im Rahmen eines viertägigen Einsatzes ausserhalb der Räumlichkeiten der Zürcher Kantonalbank wird das Docker Ökosystem unter die Lupe genommen.

Es wird Docker als Entwicklungstool angeschaut und anhand eines Proof of Concept auf einen möglichen Einsatz im Projekt eTrading Pro getestet.

Der Proof of Concept beinhaltet einen Jetty WebSocket Server mit *yass* als Service Framework und einem TypeScript/HTML/CSS Client. Der Server wird als Docker-Image erstellt und in zwei verschiedenen Docker-Container laufen gelassen. Zu den Jetty WebSocker Servers soll ein LoadBalancer davor betrieben werden.

Es wird eine Empfehlung für den Einsatz von Docker als Entwicklungstool ausgesprochen.

Durch die Studie des Docker Ökosystems wird ebenfalls eine Empfehlung für den Einsatz von Docker in der Betriebsplattform ausgesprochen.

# Inhaltsverzeichnis

<b>1. Vorwort</b>	<b>1</b>
1.1. Rahmenbedingungen	1
1.2. Urheberrecht	1
1.3. Abgrenzung	1
<b>2. Einleitung</b>	<b>2</b>
2.1. Was ist Docker	2
2.1.1. Was ist ein Docker Image	2
2.1.2. Was ist ein Docker Container	2
2.2. Was ist Docker Hub	3
2.3. Linux Container vs. Virtual Machines	3
<b>3. Docker im Windows einsetzen</b>	<b>5</b>
3.1. Boot2Docker für Windows und Mac OS	5
<b>4. Docker Workflow</b>	<b>6</b>
4.1. Build	6
4.1.1. Dockerfile	6
4.1.2. Bauen via Shellscript	7
4.2. Ship	7
4.2.1. Via Docker Hub	7
4.2.2. Als .tar File	7
4.2.3. Via SSH Tunnel	8
4.3. Run	8
<b>5. Proof of Concept</b>	<b>9</b>
5.1. Fazit	9
<b>6. Pros und Cons</b>	<b>10</b>
6.1. Pros	10
6.2. Cons	10
<b>7. Empfehlung</b>	<b>12</b>
7.1. Einsatz als Entwicklungstool	12
7.2. Einsatz im Betrieb	12
<b>A. Abbildungsverzeichnis</b>	<b>a</b>
<b>B. Listingverzeichnis</b>	<b>a</b>

# **1. Vorwort**

Dieses Thema ist für mich aktuell und sehr spannend. Bei dem Besuch der JAX-W im Herbst 2014 in München bin ich das erste Mal mit Docker in Berührung gekommen. Ich habe gleich das Potential dieser neuen Technologie erkannt. Bei einem Vortrag wurde gesagt, dass es nicht die Frage ist, ob wir in der Zukunft mit Docker arbeiten, sondern wann. Diese Aussage ist mir geblieben und ich möchte Prüfen was dahinter steckt.

## **1.1. Rahmenbedingungen**

Diese Arbeit wurde im Einverständnis von Andreas Hofstetter, LIFH & Degu Dagne, LIFHF erstellt.

## **1.2. Urheberrecht**

Ich trete das Urheberrecht dieser Arbeit voll und ganz an die Zürcher Kantonalbank ab.

## **1.3. Abgrenzung**

Ich habe alles was ich erarbeitet habe mit meinem Mac Book Pro - Mid 2015 gemacht. Somit habe ich leider keine Aussage, ob Boot2Docker auch auf Windows einwandfrei funktioniert. Die Dokumentation von Docker sagt, das es alles auch auf Windows sauber funktionieren soll.

Ich verzichte auf ein Literaturverzeichnis im herkömmlichen Sinne.

## 2. Einleitung

Docker ist ein Ökosystem das von der Firma **Docker, Inc.** aufgebaut wird. Die Firma wurde im Jahr 2013 gegründet und ist dabei schon sehr erfolgreich in dem was sie tut.

Docker wurde in drei Runden mit 11 Investoren auf 150 Millionen Dollar gefounded.

Die Firma Entwickelt die Software **Docker** und sie betreibt auch einen Service der sich **Docker Hub** nennt. Zudem bietet sie mit ihren eigenen Datacenters die Möglichkeit Docker Hub zu nutzen.

### 2.1. Was ist Docker

“*Docker* ist eine Open-Source-Software, die beim Linux-Betriebssystem dazu verwendet werden kann, Anwendungen mithilfe von Betriebssystem Virtualisierung in Containern zu isolieren. Dies vereinfacht einerseits die Bereitstellung von Anwendungen, weil sich Container, die alle nötigen Pakete enthalten, leicht als Dateien transportieren und installieren lassen. Andererseits gewährleisten Container die Trennung der auf einem Rechner genutzten Ressourcen, sodass ein Container keinen Zugriff auf Ressourcen anderer Container hat.”<sup>1</sup>

Docker basiert auf einer ähnlichen Technologie wie Linux Containers (LXC).

#### 2.1.1. Was ist ein Docker Image

Ein Docker Image ist mehr oder weniger ein Betriebssystem Image. Mit Docker kann man Docker Images als .iso Dateien exportieren und importieren. Speziell bei Docker Images ist, dass sich ein diese in Layers aufteilen, somit kann ich ein bestehendes Docker Image nehmen und mein eigenes darauf aufbauen. Das ist meines Erachtens die grösste Stärke bei Docker, da man z.B. sein RHEL 7 Linux perfekt aufsetzen und härten kann und das als Docker Image speichert. Danach kann man alle weiteren Applikationen darauf aufbauen. Der Aufbau solcher Images kann man mittels einer Scriptspache in Dockerfiles beschreiben und somit automatisieren.

#### 2.1.2. Was ist ein Docker Container

Ein Docker Container ist eine Instanz eines Docker Images. Der Docker Container läuft auf dem Host Betriebssystem, in dem auch die Docker Software läuft. Aus einem Docker Image können beliebig viele Docker Container gestartet werden.

Die Dockersoftware kann beim starten eines Containers Umgebungsvariablen und Netzwerkports von aussen her setzen und somit den Kontext eines jeweiligen Containers bestimmen.

---

<sup>1</sup>Siehe: [http://de.wikipedia.org/w/index.php?title=Docker\\_\(Software\)&oldid=142961470](http://de.wikipedia.org/w/index.php?title=Docker_(Software)&oldid=142961470)

### 2.2. Was ist Docker Hub

“*Docker Hub* ist ein Repository für Docker-Images. Es teilt sich in einen öffentlichen und einen privaten Teil auf. Im öffentlichen Teil kann jeder Nutzer seine selbst erstellten Images hochladen und damit anderen Nutzern zur Verfügung stellen. Außerdem gibt es mittlerweile offizielle Images, z. B. von Linux-Distributoren. Im privaten Teil des Docker Hubs können Benutzer ihre Docker Images hochladen und dadurch einfach z. B. firmenintern verteilen, ohne dass diese damit sofort öffentlich auffindbar sind.

Die Hub Software wurde von Docker Inc. als Open-Source-Software veröffentlicht, sodass man die Vorteile des Hubs nun auch nutzen kann, ohne die eigenen Images auf die Server von Docker laden zu müssen.”<sup>2</sup>

Analog zu Nexus für Java Artefakte kann man Docker Hub einsetzen um Docker-Images zu verwalten.

### 2.3. Linux Container vs. Virtual Machines

Jede virtuelle Maschine hat nicht nur seine Applikations Binaries die ca. 100 MB gross ist, sondern auch die Binaries des gesamten Betriebssystems, das ca. 10 GB gross ist. Zudem läuft das Betriebssystem komplett in seiner eigenen Instanz.

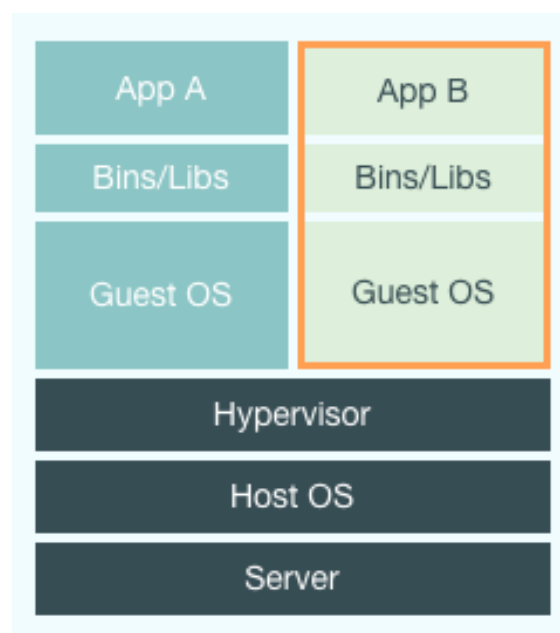


Abbildung 2.1.: Klassische Virtualisierung mit Host OS and Guest OS

Ein Linux Container läuft in einem eigenen Prozess im Host Betriebssystem. Der Linux Kernel isoliert den Prozess von allen anderen Prozessen und mittels Namespaces. Das gibt die Vorteile einer Virtualisierung, ist aber viel leichtgewichtiger und Ressourcen schonender, da der Betriebssystem overhead nur einmal anfällt.

---

<sup>2</sup>Siehe: [http://de.wikipedia.org/w/index.php?title=Docker\\_\(Software\)&oldid=142961470](http://de.wikipedia.org/w/index.php?title=Docker_(Software)&oldid=142961470)

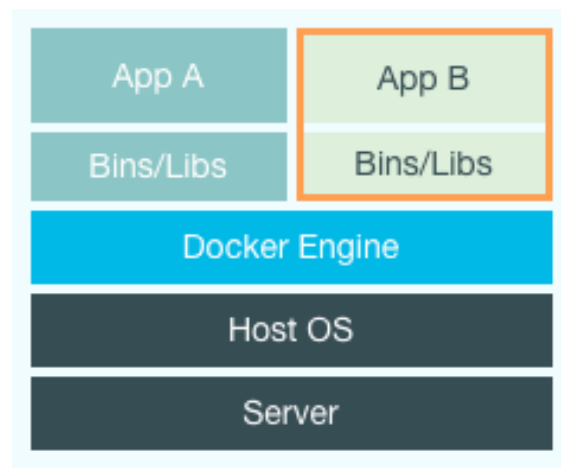


Abbildung 2.2.: Linux Container Virtualisierung

### 3. Docker im Windows einsetzen

Docker nutzt Features zur Prozess Virtualisierung des Linux Kernels, somit muss das Host Betriebssystem zwingend Linux sein. Wie kann man nun Docker in einem heterogenen Umfeld mit Windows, Mac OS, usw. nutzen?

Die Lösung heisst **Boot2Docker**

#### 3.1. Boot2Docker für Windows und Mac OS

Boot2Docker ist ein Tool das es ermöglicht im Windows und Mac OS auf einen Docker Daemon Prozess zuzugreifen. Dabei wird mit Oracles VirtualBox ein Spezielles Host Betriebssystem als richtige virtuelle Maschine gestartet. Dafür wird eine speziell für den Betrieb von Docker gebaute Linux Distribution namens boot2docker verwendet.

In der VirtualBox wird das Linux boot2docker gestartet und im Linux drin der Docker Daemon Prozess. Der Docker Daemon Prozess kann dann über die Kommandozeile gesteuert werden.

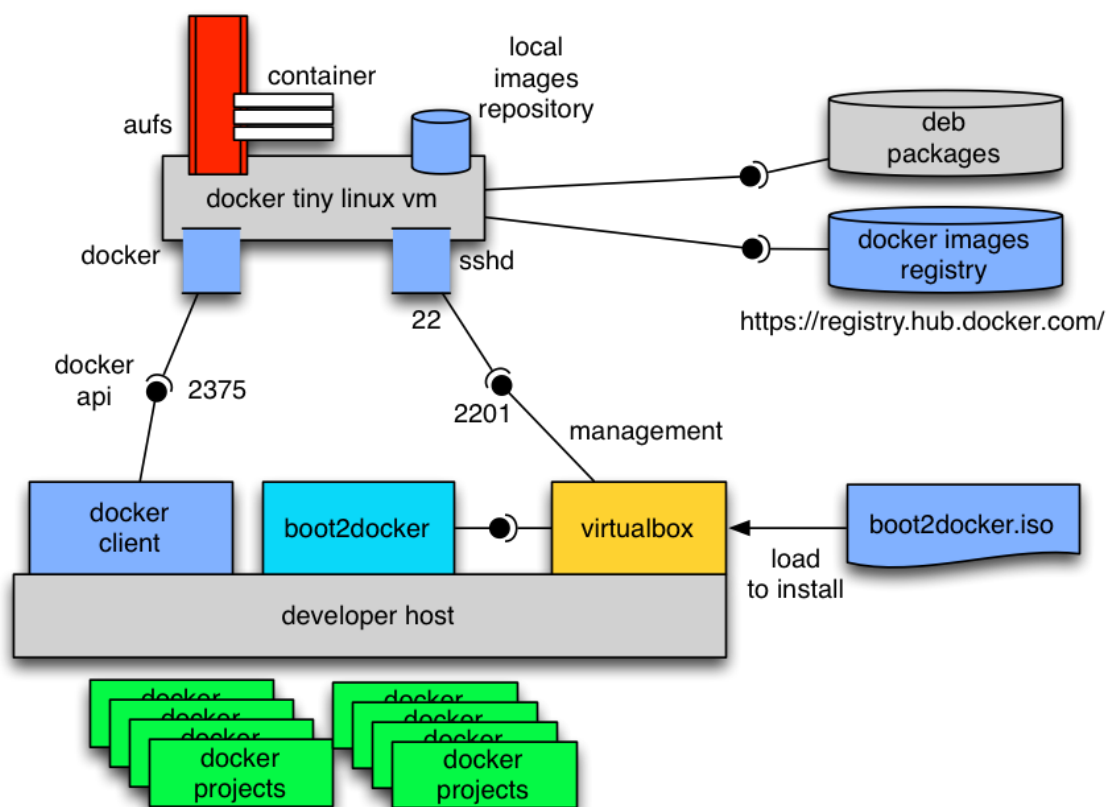


Abbildung 3.1.: Boot2Docker Architektur sieht dann so aus



## 4. Docker Workflow

Docker hat den Grundsätzlichen Workflow von Build, Ship, Run. Wie ich das im Proof of Concept gemacht habe, werde ich hier veranschaulichen.

### 4.1. Build

Um ein Image zu bauen kann man ein *Dockerfile* erstellen. Das Dockerfile ist wie ein Script-File, welches die Beschreibung eines Betriebssystems ist. Darin kann angegeben werden von welchem Docker Image abgeleitet wird. Welche Dateien in dieses Image gepackt werden sollen. In welchem Benutzerkontext das Image ausgeführt. Welche Netzwerkports nach aussen hin exposed werden. Und vieles mehr.

Als zweite Variante kann man ein bestehendes Image starten und live Änderungen am Image vornehmen. z.B. Konfigurationsdateien anpassen. Mit dem Befehl *docker commit* können dann jegliche Änderungen in ein neues Image gespeichert werden. Das hat Ähnlichkeit zum bestehenden Software Entwicklungsprozess, wo man seine Änderungen auch "committed".

#### 4.1.1. Dockerfile

Das Dockerfile welches ich für den Proof of Concept erstellt habe, sieht so aus:

```
1 FROM java:8
2 MAINTAINER Roman Wuersch
3
4 COPY . /usr/src/yass
5 WORKDIR /usr/src/yass
6
7 EXPOSE 9090
8
9 CMD [ "./start-in-docker.sh" ]
```

Listing 4.1: Dockerfile

**Zeile 1** Das Image leitet vom offiziellen OpenJDK 8 Image ab und baut darauf auf.

**Zeile 4** Kopiert alle Dateien im aktuellen Pfad "." in das Image in den Pfad "/usr/src/yass". Das sehe ich als eine der grössten Stärken an. Man kann sein Setup Lokal auf seiner Maschine so aufsetzen und testen bis es alles sauber funktioniert und dann mit dieser Zeile genau den Inhalt in das Image replizieren.

**Zeile 5** Setzt das Startverzeichnis

**Zeile 7** Gibt an, dass der Port 9090 von einem Prozess verwendet wird und zukünftig angebunden werden kann

**Zeile 9** Der Befehl, welcher ausgeführt wird, wenn ein Container aus dem Image gestartet wird.

### 4.1.2. Bauen via Shellscript

Um das Image zu bauen habe ich ein Shell-Script gemacht:

```
1 #!/usr/bin/env bash
2 docker rmi -f sushicutta/docker-test
3 docker build -t sushicutta/docker-test .
```

Listing 4.2: Bauen eines Docker Images

Dieses Shell-Script sucht im aktuellen Verzeichnis ein Dockerfile und baut mit dessen Angaben das Image.

**Zeile 2** Löscht das bestehende Image sushicutta/docker-test aus dem Docker Hafen

**Zeile 3** Erstellt das Image neu anhand der Datei *Dockerfile* unter dem selben Namen.

## 4.2. Ship

### 4.2.1. Via Docker Hub

Um das Image zu liefern, kann man nun Docker Hub verwenden.

```
1 docker push sushicutta/docker-test
```

Listing 4.3: Push to Docker Hub

### 4.2.2. Als .tar File

Oder man kann das Image als .tar Datei speichern.

```
1 docker save -o docker-test.tar sushicutta/docker-test
```

Listing 4.4: Docker Image exportieren

Das gelieferte Image kann man dann irgendwo wieder importieren.

```
1 docker load -i docker-test.tar
```

Listing 4.5: Docker Image importieren

### 4.2.3. Via SSH Tunnel

Oder gleich das Image per SSH transportieren zippen und unzippen on the fly, und mit pv den Pipe traffic darstellen:

```
1 docker save <image> | bzip2 | pv | \  
2   ssh user@host 'bunzip2|docker load'
```

Listing 4.6: SSH Transport mit zippen und network traffic Visualisierung

## 4.3. Run

Das Image ist jetzt gebaut worden und im Docker Hafen angekommen, und es trägt den Namen sushicutta/docker-test.

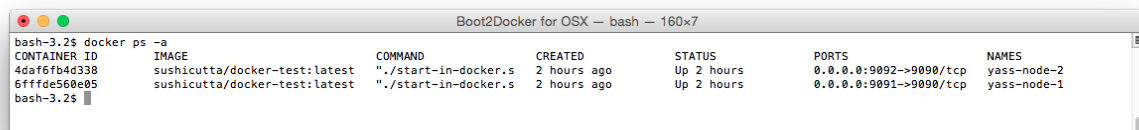
Dieses Image kann nun beliebig viele Male gestartet werden.

Um das Image zweimal zu starten, benötigt man folgende Commands:

```
1 docker run -d -p 9091:9090 --name yass-node-1 sushicutta/docker-test  
2 docker run -d -p 9092:9090 --name yass-node-2 sushicutta/docker-test
```

Listing 4.7: Image zweimal starten auf den Ports 9091 und 9092

Das Image läuft jetzt zweimal, der Inhalt ist zu 100% identisch. Der unterschied besteht darin, dass Docker nun den Ports 9091 des Host Betriebssystems auf den Port 9090 des ersten Containers mapped und analog den Port 9092 des Host Betriebssystems auf den Port 9090 des zweiten Containers.



CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
4daf6fb4d338	sushicutta/docker-test:latest	"/start-in-docker.s	2 hours ago	Up 2 hours	0.0.0.0:9092->9090/tcp	yass-node-2
6ffffde560e05	sushicutta/docker-test:latest	"/start-in-docker.s	2 hours ago	Up 2 hours	0.0.0.0:9091->9090/tcp	yass-node-1

Abbildung 4.1.: Zwei identische Container auf verschiedenen Ports

Was jetzt noch zum tragen kommt, ist der Start Befehl, der im Dockerfile konfiguriert wurde. In unserem Beispiel start-in-docker.sh:

```
1 #!/usr/bin/env bash  
2 java -Dfile.encoding=UTF-8 -classpath \  
3   "/usr/src/yass/lib/*:/usr/src/yass/build/classes/tutorial\  
4   ch.softappeal.yass.tutorial.server.web.JettyServer 2>&1
```

Listing 4.8: Start Script im Docker Container



```
bash-3.2$ docker logs yass-node-1
2015-06-15 08:46:54.237:INFO::main: Logging initialized @601ms
2015-06-15 08:46:54.623:INFO:oejs.Server:main: jetty-9.2.10.v20150310
2015-06-15 08:46:55.207:INFO:oejs.ContextHandler:main: Started o.e.j.s.ServletContextHandler@612679d6{/,null,AVAILABLE}
2015-06-15 08:46:55.293:INFO:oejs.ServerConnector:main: Started ServerConnector@64f6106c{HTTP/1.1}{0.0.0.0:9090}
2015-06-15 08:46:55.294:INFO:oejs.Server:main: Started @1665ms
started
session 1 created
session 1 opened
08:49:45 | 1 | client | entry | echo | [ "hello from server" ]
08:49:45 | 1 | server | entry | reload | [ true 987654 ]
08:49:45 | 1 | server | entry | getInstruments | [ ]
08:49:45 | 1 | server | entry | subscribe | [ [ 987654321 ] ]
```

Abbildung 4.2.: Output am Stdout und Stderr sichtbar machen vom Host Betriebssystem

Den Output der Prozesse kann man mit dem Befehl `docker logs` anschauen.

## 5. Proof of Concept

In einem Proof of Concept habe ich mir zum Ziel gesetzt Docker direkt auszuprobieren. Das heisst Hands-On. Den kompletten Proof of Concept habe ich mit meinem Mac Book Pro - Mid 2010 durchgeführt. Dabei habe ich nur IntelliJ IDEA 14.1.3 und das Terminal von Mac OS verwendet.

Als Proof of Concept habe ich mir ausgedacht, ich würde gerne zwei Jetty Websocket Server mit yass als Service Framework erstellen. Als Applikation habe ich das Tutorial von yass verwendet. Vor diese beiden Server soll ein Load Balancer mit Sticky-Sessions platziert werden und die jeweiligen WebSocket-HTTP-Request, als Reverse Proxy an einen der Jetty Websocket Server weiterleiten.

Da ich dabei alles Open-Source-Software verwendet habe, habe ich den erstellten Code des Proof of Concept auf Github publiziert:

<https://github.com/sushicutta/yass/tree/testDocker>

Das ganze habe ich hingekriegt und es läuft einwandfrei.

### 5.1. Fazit

Docker ist ein sehr cooles Ökosystem. In der kurzen Zeit, habe ich es geschafft zwei Docker Images zu erstellen, eines mit einem Java WebSocket Backend auf Basis von Jetty und yass, und eines mit einem WebServer, der statischen Content, sprich den Client in HTML und JavaScript, ausliefert und dann auch noch gleich das Load-Balancing mit Sticky-Sessions bietet.

Docker ist ein sehr Entwickler freundliches Tool, da man alles von der Kommandozeile her steuern kann. Zudem ist die Dokumentation von Docker einwandfrei auf dessen Homepage verfügbar.

Ich würde mich freuen, wenn wir in Zukunft auf die Stärken von Docker zurückgreifen könnten.

## 6. Pros und Cons

### 6.1. Pros

- Docker Images sind Immutable, das garantiert, dass ein Docker Image nachdem es gebaut wurde nicht mehr verändert wird. Das ist im Entwicklungsprozess von Vorteil, wenn man viele Engineering und Staging Infrastrukturen hat und die Images von einer zur nächsten transportieren muss.
- Docker Images sind komplette Betriebssysteme. Alles was es braucht um die Applikation zu betreiben, z.B. JRE - Java Runtime Environment, kann in einem Image mitgeliefert werden und gehören somit ebenfalls zur Applikation.
- Docker kann über die Kommandozeile gesteuert werden.
- Docker fügt sich reibungslos in den Entwicklungsprozess ein, bei welchem Build-Automatisierungs Tools eingesetzt werden. Durch die Steuerung per Kommandozeile können alle Schritte im Docker Entwicklungsprozess mittels Shell-Scripts automatisiert werden. Solche Shell-Scripts können dann z.B. via Jenkins im Buildprozess verwendet werden.
- Docker Container lassen sich als *root* ausführen, es kann aber auch ein spezieller Benutzer im Dockerfile angegeben werden.
- Der Start eines Docker Containers ist sehr schnell, meistens im Bereich  $< 1$  Sekunde. Gegenüber klassischer Virtualisierung bietet das eine starke Verbesserung.
- IntelliJ ab Version 14.1 bietet eine Docker Integration.
- Docker Images bauen auf anderen Docker Images auf, das passt perfekt in den DRY, don't repeat yourself, Ansatz. Es können standard Images für verschiedene Anwendungszwecke gebaut werden. So kann für den JBoss ein Image Inhouse erstellt und gehärtet werden, welches dann von den jeweiligen Applikationen weiterverwendet werden kann.
- Jegliche Applikationen werden neu zu Docker Images, daraus folgt, 1 Applikation ist somit auch nur genau 1 Datei. Wir erinnern uns, man kann ein Docker Images als .tar Datei exportieren.
- Docker bietet eine sehr gute Dokumentation aller Aspekte. Es wird ausführlich beschrieben wie man ein Dockerfile richtig gestalten soll, oder auch wie man per Kommandozeile den Docker Daemon richtig bedient.

### 6.2. Cons

- Yet another Dev Tool.
- Docker kommt mit einer sehr steilen Lernkurve daher. Auch für mich war es eine Herausforderung in dieser kurzen Zeit alles was das Docker Ökosystem bietet richtig einzuordnen.

- Für die Ausführung eines Docker Images braucht es zwingend immer einen Docker Deamon Prozess der in einem Linux Host Betriebssystem läuft, Das macht es für den Betrieb schwierig Docker in der Produktion einzusetzen.

Zumindest kommt da die Frage auf, wie sicher "bullet proof" is der Docker Deamon Prozess? und wie geht man mit Security Updates zur Laufzeit um, welche für den Docker Deamon Prozess erscheinen, oder auch für die Container, welche im Rahmen eines Docker Deamon Prozesses laufen.

- Docker existiert erst seit kurzer Zeit und es gibt noch nicht viele Entwickler und Administratoren, welche sich professionell mit diesem Thema auseinander gesetzt haben. Somit wird es schwer werden Leute für die Sache zu gewinnen. Vielerorts existiert eine Akzeptanzhürde bei neuen Technologien, welche bestehendes auf den Kopf stellen könnten.

## **7. Empfehlung**

Wenn ich eine eigene Firma betreiben würde, wäre für mich klar, dass ich meine Systeme mit Docker betreiben würde.

Im Proof of Concept habe ich mein Image ohne Probleme bei Digital Ocean auf einen angemieteten Server in Frankfurt installiert. Dort habe ich mit Docker in ein paar Minuten zwei Container meines Jetty Webserver gestartet und von lokal angebunden. Das war für mich ein riesiges Highlight, da ich noch nie so leicht eine Applikation bei einem Hosting Partner installieren konnte.

### **7.1. Einsatz als Entwicklungstool**

Ich empfehle Docker als Entwicklungstool.

Der Grund dafür ist, dass wir im eTrading Pro ein kompliziertes Setup haben. Mit Docker habe ich die Möglichkeit ein Setup mit vielen Komponenten nahe an der Realität abzubilden. Docker gibt mir dann die Möglichkeit schnell und einfach eine Komponente zu starten oder zu stoppen. Somit können Failover Tests, oder Reconnect Scenarios einfach Lokal nachgestellt werden. Darüber hinaus kann entsprechender Code direkt entwickelt und getestet werden. Es fallen die lästigen Deployments weg, welche vorher dafür notwendig waren.

### **7.2. Einsatz im Betrieb**

Mit meinem aktuellen Kenntnisstand würde ich Docker nicht für den breiten Einsatz im Betrieb empfehlen.

Dennoch würde ich anhand einer kleineren Applikation den Einsatz wagen, und dabei die nötigen Erfahrungen zu sammeln.

## A. Abbildungsverzeichnis

2.1. Klassische Virtualisierung mit Host OS and Guest OS . . . . .	3
2.2. Linux Container Virtuallisierung . . . . .	4
3.1. Boot2Docker Architektur sieht dann so aus . . . . .	5
4.1. Zwei identische Container auf verschiedenen Ports . . . . .	8
4.2. Output am Stdout und Stderr sichtbar machen vom Host Betriebssystem . . . . .	9

## B. Listingverzeichnis

4.1. Dockerfile . . . . .	6
4.2. Bauen eines Docker Images . . . . .	7
4.3. Push to Docker Hub . . . . .	7
4.4. Docker Image exportieren . . . . .	7
4.5. Docker Image importieren . . . . .	7
4.6. SSH Transport mit zippen und network traffic Visualisierung . . . . .	8
4.7. Image zweimal starten auf den Ports 9091 und 9092 . . . . .	8
4.8. Start Script im Docker Container . . . . .	8