

Docker

# Docker & různé OS

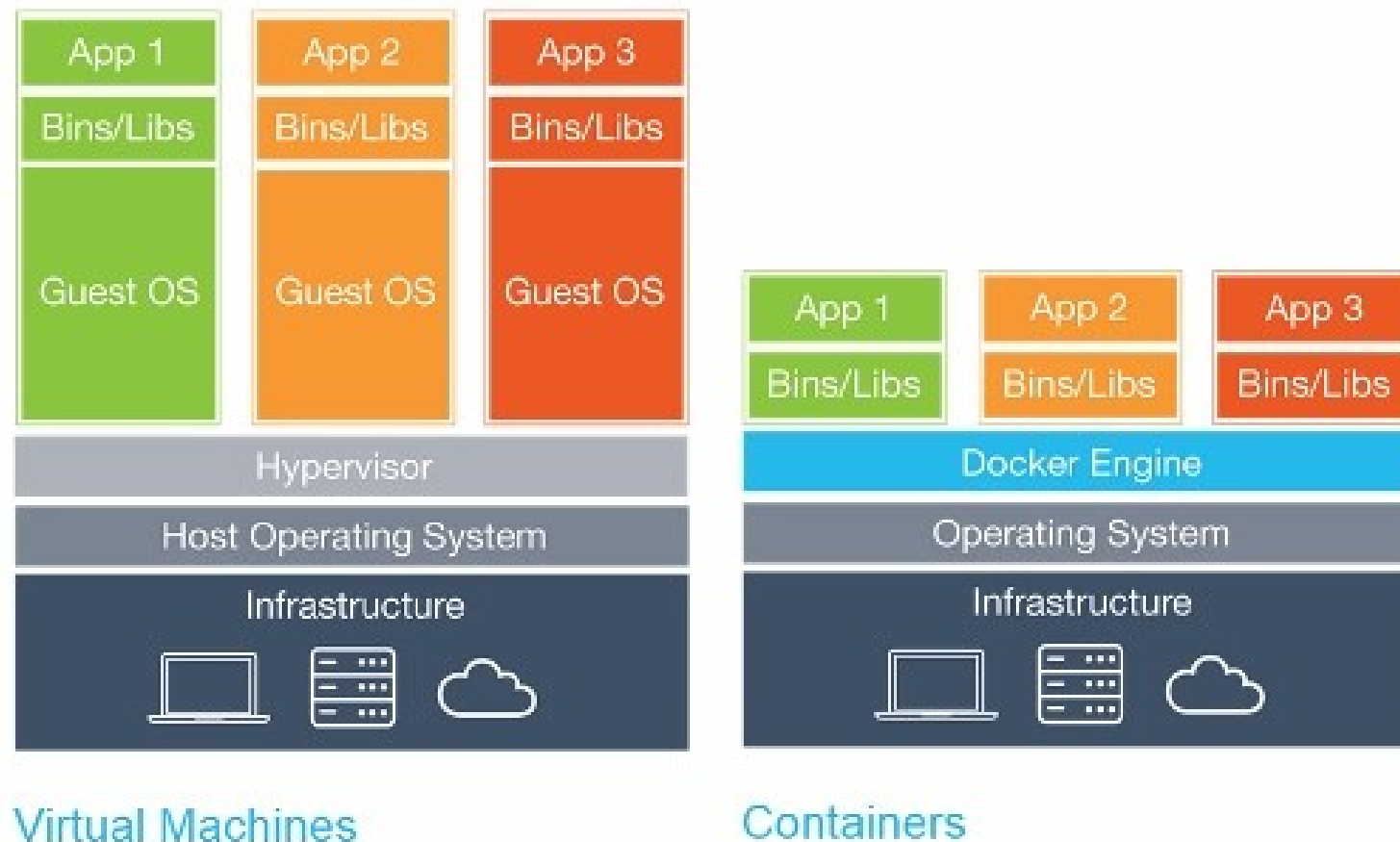
- Čistě teoreticky existuje Docker for Windows / Mac a celá řada věcí je přenositelná, ale Docker containery jsou v současnosti 100% přenositelné pouze v případě Linuxu.
  - Zejména u Windows 7 je hromada problémů (například nefungují symbolické linky), protože Docker (konkrétně boot2docker) běží uvnitř Oracle VM VirtualBox.
    - Tip: Na boot2docker je možné se připojit pomocí:  
`docker-machine ssh`
- Instalace na Ubuntu 20.04:
  - <https://docs.docker.com/engine/install/ubuntu/>

**Docker & Windows 10:** Docker vyžaduje Hyper-V, což ale na druhou stranu nesmí být zapnuté při běhu VirtualBox. Pokud potřebujete jak Docker, tak VirtualBox, pak je možné Hyper-V zapnout/vypnout (vyžaduje restart Windows) ... nebo použít WSL 2:  
<https://ugetfix.com/ask/how-to-disable-hyper-v-in-windows-10/>

# Docker & Proxy

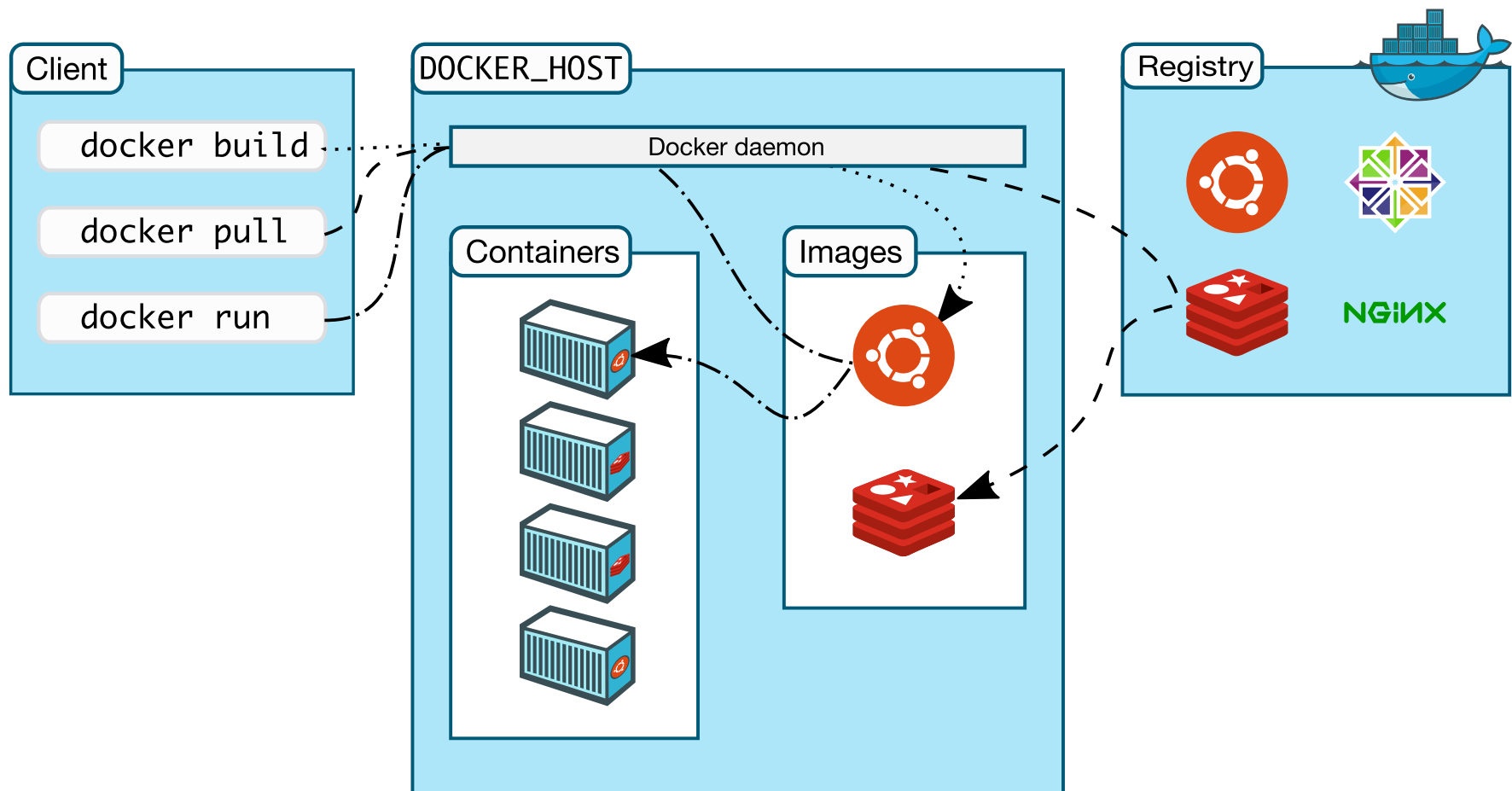
- Jak rozchodit Docker s proxy:
  - <https://stackoverflow.com/questions/23111631/cannot-download-docker-images-behind-a-proxy>
- Poznámka: Pro proxy vyžadující uživ. jméno a heslo použijte tento formát URL:  
`http://<username>:<password>@<proxy_host>:<proxy_port>`

# Docker vs. virtualizace








Poznámka: Pro dosažení požadované funkcionality používá Docker Linux namespaces:  
[https://en.wikipedia.org/wiki/Linux\\_namespaces](https://en.wikipedia.org/wiki/Linux_namespaces)

# Docker Workflow



- Docker běží na Linuxu a Windows 10 na localhostu, na Windows 7 s Docker Tools běží na IP adrese 192.168.99.100

# Základní příkazy

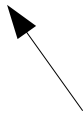
- Stažení image:
  - `docker pull httpd`  To samé jako: `docker pull httpd:latest`  
Poznámka: pull nejprve zjistí, jestli existuje novější verze image a pokud ano, pak ji stáhne.
  - `docker pull httpd:alpine`
  - [https://hub.docker.com/\\_/httpd/](https://hub.docker.com/_/httpd/)
- Vytvoření containeru s názvem apache:
  - `docker run -p 80:80 -d --name apache httpd:alpine`  Daemon, vypnutí:  
`docker stop apache`
  - `docker run -p 80:80 -it --rm httpd:alpine`
  - `docker run -p 80:80 -d --name apache \`  
`--restart=always httpd:alpine`
    -  Uživatelský  
název
    -  Bude běžet v  
interaktivním  
terminálu a  
nebude se ukládat
    -  Po restartu počítače (dockeru)  
se container automaticky nastartuje

# Image tag

- Každá image může mít tagy. Používání „latest“ u produkčních aplikací není best practice, protože to prakticky znamená - SNAPSHOT.

# Základní příkazy

- `docker run -p 80:80 -it --rm httpd:alpine`



Port-forwarding, port 80 z containeru bude forwardován na port 80 hosta.

Syntaxe je: `HOST_PORT:CONTAINER_PORT`

- Vypnutí containeru:
  - `docker stop apache`
- Restart containeru:
  - `docker restart apache`
- Smazání containeru:
  - `docker rm -v apache`
- Smazání image:
  - `docker rmi httpd`

Také je možné použít:

```
docker run --net=host -it --rm \
httpd:alpine
```

V tomto případě container používá network stack hosta.

<https://docs.docker.com/engine/userguide/networking/>

**Poznámka:** Port forwarding automaticky zpřístupní port i skrz iptables. Port se dá ale také namapovat tak, aby byl přístupný pouze z localhostu:

<https://stackoverflow.com/questions/22100587/docker-expose-a-port-only-to-host>



# Základní příkazy

- Seznam všech běžících containerů:
  - `docker ps`
- Seznam všech containerů:
  - `docker ps -a`
- Seznam stažených images:
  - `docker images`
- Logy (STDOUT a STDERR):
  - `docker logs --name=apache`
- Podrobné informace o containeru:
  - `docker inspect apache`

# Základní příkazy

- Kdekoli je možné použít nápovědu:
  - `docker run --help`
- Online dokumentace:
  - <https://docs.docker.com/>
  - <http://blog.codepipes.com/containers/docker-for-java-big-picture.html>

# Jméno containeru

- Při spouštění containeru jsem používal `--name=apache`. Tím se definuje uživatelské jméno containeru. To ale není jediné jméno. Když se container vytvoří jako daemon (`-d`), pak se do konzole vypíše text jako:

`1fbb72a3ce70773d48aa313184f5864803c601bcf1bbec623892be464e319037`

- Toto je automaticky vygenerovaný název. A když spustíme `docker ps -q`, pak se vrátí: `1fbb72a3ce70` (prvních 12 písmen dlouhého názvu). Všechny tyto názvy je možné použít pro identifikování containeru.
- Praktické použití:
  - Vypnutí všech běžících containerů:
    - `docker stop $(docker ps -q)`

Poznámka: Obdobně mají svůj „strojový“ název image a příkaz:  
`docker images -q`  
vrátí všechny tyto názvy

# DNS

- Out-of-the-box Docker používá DNS definovanou v `/etc/resolv.conf` a jako fallback používá Google DNS: 8.8.8.8
- V některých firmách je Google DNS zablokována, pak je nutné explicitně nastavit DNS:
  - <https://development.robinwinslow.uk/2016/06/23/fix-docker-networking-dns/>
  - Dělá to problémy když se uvnitř kontejneru snažíme například něco nainstalovat

# Volumes I.

- `docker run -p 80:80 -it --rm \`  
  `-v c:/dkr/html/:/usr/local/apache2/htdocs:ro \`  
  `-v c:/dkr/conf/httpd.conf:/usr/local/apache2/conf/httpd.conf:ro \`  
  `-v /tmp`  
  `httpd:alpine`

Read only



**Pomocí -v se přidává volume. Jedná se o místo, kam container může zapisovat data.**

Syntaxe je:

`-v ADRESAR_CONTAINERU`

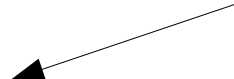
Nebo:

`-v ADRESAR_HOSTA:ADRESAR_CONTAINERU`

Nebo:

`-v SOUBOR_HOSTA:SOUBOR_CONTAINERU`

Na hostu se vytvoří adresář, který se použije jako volume.



Adresář hosta „překryje“ adresář containeru.



V „inspect“ jsou Volumes pod „Mounts“

# Volumes II.

- Také je možné vytvořit volume jako další container a potom ho přilinkovat k jinému containeru.
- Docker neřeší to, kdyby se dva containery snažily zapisovat do jednoho volumu ve stejnou chvíli (může dojít k poškození dat nebo k výjimce).
- Když se maže container, pak je best practice ho mazat pomocí příkazu:
  - `docker rm -v apache`
  - Přepínač „-v“ řekne, že se také smažou všechny vytvořené volumes (vytvořené pomocí -v CONT, nikoli -v HOST:CONT)
- <https://docs.docker.com/engine/tutorials/dockervolumes/>

Jak přidat volume ke stávajícímu containeru:

<https://stackoverflow.com/questions/28302178/how-can-i-add-a-volume-to-an-existing-docker-container>

# Volumes – best practices

- Na produkci je best practice, aby byl container „stateless“, aby neobsahoval žádná uživatelská data, logy apod. (pouze „temporary“ data), takže když container smažeme a nahradíme ho jiným, tak o naše data nepřijdeme.
- Od Docker 17.06 také existuje Bind Mount, což je to samé jako Volume, ale má explicitnější syntaxi:
  - <https://docs.docker.com/storage/bind-mounts/>

# Logy I.

- V dockerizované aplikaci je best practice logovat na standardní výstup (stdout) s tím, že se o logy následně stará Docker
  - Ve výchozím nastavení ukládá jenom jeden log soubor s neomezenou délkou:
    - `/var/lib/docker/containers/<container-id>/<container-id>-json.log`
  - To se dá změnit (dá se nastavit počet těchto souborů a jejich maximální velikost):
    - [https://medium.com/@Quigley\\_Ja/rotating-docker-logs-keeping-your-overlay-folder-small-40cfa2155412](https://medium.com/@Quigley_Ja/rotating-docker-logs-keeping-your-overlay-folder-small-40cfa2155412)
    - <https://docs.docker.com/config/containers/logging/json-file/>
    - <https://docs.docker.com/config/containers/logging/configure/>



# Logy II.

- Gelf (Graylog Extended Format Logging Driver):
  - <https://docs.docker.com/config/containers/logging/gelf/>
    - Docker také obsahuje nativní způsob jak dostat logy do Graylog, Logstash nebo Fluentd.
      - Pozor!
        - <https://www.docker.com/blog/adventures-in-gelf/>

# Vnitřek Docker containeru I.

- Existuje spousta způsobů jak se dostat dovnitř containeru. Osobně preferuji tyto způsoby:

```
docker exec -it NAZEV_CONTAINERU /bin/bash
```

- Tímto se dostanu do aktuálně běžícího containeru.
- Další varianty tohoto příkazu:

```
docker exec -it NAZEV_CONTAINERU bash
```

```
docker exec -it NAZEV_CONTAINERU sh
```

```
docker attach NAZEV_CONTAINERU
```

- Tímto se připojím k běžícímu procesu aktuálně běžícího containeru. Ven se dostanu CTRL + P + Q

```
docker export NAZEV_CONTAINERU > container.tar
```

- Exportuje container do tar souboru (container nemusí běžet).

# Vnitřek Docker containeru / image II.

- Další způsoby:
  - Nainstalovat do image sshd a připojit se dovnitř
    - Tento způsob se používal zejména v minulosti.
  - Docker commit vytvoří snapshot containeru do nové image
    - <https://docs.docker.com/engine/reference/commandline/commit/>
    - POZOR! Do nové image se nekopírují volumes!!!
  - Tímto se dostaneme do image:
    - `docker run --rm -it --entrypoint /bin/bash <imageID>`

# Kopírování souborů

- Jak zkopírovat soubor z hosta dovnitř docker containeru:

```
docker cp file_name container_name:/file_name
```

- <https://stackoverflow.com/questions/22907231/copying-files-from-host-to-docker-container>

# Read only

- Docker container ve výchozím nastavení NENÍ read only. Aplikace v něm běžící může libovolně vytvářet adresáře a soubory KDEKOLI na disku.
- To se dá změnit pomocí flagu `--read-only`
  - Potom je možné zapisovat pouze do volume adresářů.
- Ale i po restartu containeru tam zůstanou data. Pomocí flagu `--tmpfs` můžete nastavit temporary adresář, který se při startu containeru smaže:
  - <http://www.projectatomic.io/blog/2015/12/making-docker-images-write-only-in-production/>
- Je best practice spouštění Docker containeru s těmito parametry:
  - `docker run --read-only --tmpfs /tmp`

# Dockerfile

- Dockerfile je soubor, na základě kterého se vytváří image. Nemusí se jmenovat „Dockerfile“, ale je to best practice.
- Aplikace v Containeru MUSÍ běžet na popředí, teprve při spuštění containeru pomocí přepínačů rozhodujeme, jestli bude skutečně běžet na popředí, nebo jestli bude démonem.
- Příklad: V době psaní tohoto příkladu neexistoval Docker container, ve kterém by byl Apache s podporou HTTP 2 a PHP.
  - Chrome pro podporu HTTP 2 vyžaduje OpenSSL 1.0.2, které je v současnosti v Ubuntu 16.04 nebo v Alpine Linux.
  - Alpine je lepší v tom, že je výrazně menší. Důrazně ale doporučuji si při tvorbě aplikací na Alpine nejprve aplikaci v něm skutečně otestovat (například ve virtuální mašině).

Spustí se při  
buildování image

# Alpine + Apache + PHP

Na tomto image se bude stavět.

Je možné stavět vždy pouze na jednom image!!!

FROM alpine:3.4

RUN apk update && \

apk add openssh-client apache2 \

php5-apache2 openssl && \

mkdir -p /run/apache2

Poznámka: Je možné mít víc RUN operací

ENTRYPOINT ["httpd"]

EXPOSE 80

CMD ["-D", "FOREGROUND", "-e", "info", "-f", "/etc/apache2/httpd.conf"]



Argumenty aplikace,  
co se bude spouštět  
(httpd)

NEZAPOMENOUT  
NA TEČKU!!!

Build image se spustí pomocí:  
docker build --tag alpine-apache .

# Alpine + Apache + PHP II.

Je možné mít víc RUN operací:

```
FROM alpine:3.4
```

```
RUN apk update
```

```
RUN apk add openssh-client apache2 php5-apache2 openssl
```

```
RUN mkdir -p /run/apache2
```

```
ENTRYPOINT ["httpd"]
```

```
EXPOSE 80
```

```
CMD ["-D", "FOREGROUND", "-e", "info", "-f", "/etc/apache2/httpd.conf"]
```

Co je ale best practice?

<https://stackoverflow.com/questions/39223249/multiple-run-vs-single-chained-run-in-dockerfile-what-is-better>

<https://medium.com/@jessgreb01/digging-into-docker-layers-c22f948ed612>



# Apt best practices

- Pokud budete používat Debian (Ubuntu), tak je zde pár poznatků:
  - Je zapotřebí volat `apt install -y XXX`
  - Není best practice volat `apt upgrade` a `dist-upgrade`
  - Je best practice volat v jednom RUN příkazu `apt update` a `apt install`, protože když `apt install` skončí chybou, tak se znovu nezavolá `apt update`, protože to je již v build cache.
  - Je best practice mazat obsah adresáře `/var/lib/apt/lists`, kde se nachází výsledek `apt update`
- Čili je best practice volat:

```
RUN apt-get update && apt-get install -y \  
    pkg1 \  
    pkg2 \  
    && rm -rf /var/lib/apt/lists/*
```

# Security best practices I.

- Preferujte base image (ubuntu, alpine)
- Ve výchozím nastavení běží aplikace v Dockeru pod ROOTem. To není best practice ...

```
FROM ubuntu
```

```
RUN mkdir /app
```

```
RUN groupadd -r lirantal && \
```

```
    useradd -r -s /bin/false -g lirantal lirantal
```

```
WORKDIR /app
```

```
COPY . /app
```

```
RUN chown -R lirantal:lirantal /app
```

```
USER lirantal
```

```
CMD node index.js
```

# Security best practices II.

- Pozor na hesla / klíče / certifikáty, které potřebujete při buildování aplikace, ale nakonec je chcete z image odstranit. Když je přidáte v jedné vrstvě a odeberete ve druhé, tak to ve skutečnosti v image stále bude. Jak tento problém vyřešit?
  - Buď si dát na to pozor a s citlivými „temporary“ daty pracovat pouze v rámci jedné vrstvy.
  - Nebo použít multi-stage build
- <https://snyk.io/blog/10-docker-image-security-best-practices/>

# Docker Linter

- Pro validování Dockerfile jestli je splňuje obecně přijímané best practices lze použít hadolint:
  - <https://github.com/hadolint/hadolint>

# Spring Boot aplikace

```
FROM openjdk:8-jre
WORKDIR /app
VOLUME ["/tmp"]
COPY target/app.jar app.jar
RUN sh -c 'touch app.jar'
ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom", \
            "-jar","app.jar"]
```

Aktuální adresář když se použije relativní cesta k souboru

Uvnitř image se vytvoří adresář /tmp

Zkopíruje target/app.jar do /app/app.jar

Nastaví časové razítko

↑  
Spustí  
aplikaci

Poznámka: Ještě je velice užitečný parametr ENV pro definování proměnných prostředí

Rozdíl v CMD a ENTRYPOINT je v tom, že parametry v ENTRYPOINT nejdou změnit z příkazové řádky při spuštění containeru:  
<http://goinbigdata.com/docker-run-vs-cmd-vs-entrypoint/>

Příklad:

```
docker run --rm -it -p 8081:8081 test --server.port=8081
```

# Dockerfile drobnosti

- Při přidávání adresářů může být užitečné použít soubor `.dockerignore`, který funguje jako `.gitignore` a slouží k vyloučení souborů / adresářů při kopírování adresářů.
  - <https://docs.docker.com/engine/reference/builder/#dockerignore-file>
- Při tvorbě image Docker z důvodu optimalizace out-of-the-box používá build cache. To se dá vypnout pomocí `--no-cache=true`.
  - [https://docs.docker.com/develop/develop-images/dockerfile\\_best-practices/#leverage-build-cache](https://docs.docker.com/develop/develop-images/dockerfile_best-practices/#leverage-build-cache)
- <https://vsupalov.com/docker-arg-vs-env/>

# FROM scratch

- Jestli budete někdy potřebovat vytvořit FROM scratch Dockerfile, pak doporučuji tento článek pro jeho pochopení:
  - <https://www.mgasch.com/post/scratch/>

# Docker Buildkit

- Docker 18.09 integruje Buildkit, nový způsob buildování images, který je zejména rychlejší než stávající způsob:
  - [https://docs.docker.com/develop/develop-images/build\\_enhancements/](https://docs.docker.com/develop/develop-images/build_enhancements/)
  - <https://github.com/moby/buildkit>



# Docker multi-stage build

- Od Docker 17.0.5 existuje nově multi-stage build (v jednom Dockerfile může být více FROM příkazů, díky kterým je možné více dělat operací v jednom Dockerfile, zatímco doted' bylo nutné vytvářet více Dockerfile souborů):
  - [https://codefresh.io/blog/node\\_docker\\_multistage/](https://codefresh.io/blog/node_docker_multistage/)
- Také je Multi-stage build skvělý pro optimalizaci velikosti image:
  - <https://docs.docker.com/develop/develop-images/multistage-build/>
  - <https://blog.realkinetic.com/building-minimal-docker-containers-for-python-applications-37d0272c52f3>

# JIB, Spring Boot

- Existuje Maven plugin, který neuvěřitelně zjednodušuje tvorbu Docker image bez nutnosti psaní Dockerfile:
  - <https://github.com/GoogleContainerTools/jib/tree/master/jib-maven-plugin>
  - <https://www.baeldung.com/jib-dockerizing>
- Pěkný tutorial jak na Docker se Spring Bootem (bez JIB, s JIB, best practices):
  - <https://spring.io/blog/2018/11/08/spring-boot-in-a-container>
- Spring Boot 2.3 má ještě lepší podporu:
  - <https://spring.io/blog/2020/01/27/creating-docker-images-with-spring-boot-2-3-0-m1>

# Distroless

- JIB out-of-the-box používá distroless image:
  - <https://github.com/GoogleContainerTools/distroless>

# Tomcat & WAR

```
FROM tomcat:7
```

```
COPY probe.war /usr/local/tomcat/webapps/probe.war
```

```
COPY tomcat-users.xml /usr/local/tomcat/conf/tomcat-users.xml
```

```
ENTRYPOINT ["/usr/local/tomcat/bin/catalina.sh", "run"]
```

tomcat-users.xml:

```
<tomcat-users>
```

```
  <role rolename="manager-gui" />
```

```
  <user username="tomcat" password="tomcat" roles="manager-gui" />
```

```
</tomcat-users>
```

- Aplikace Psi probe ke stažení zde:
  - <https://github.com/psi-probe/psi-probe/releases>

# Docker & Maven I.

- V současnosti je nejpoužívanější tento plugin:
  - <https://javalibs.com/plugin/io.fabric8/docker-maven-plugin>
  - <https://dmp.fabric8.io/>
- Použití se Spring Boot aplikací:
  1. Vytvořte `src/main/docker/Dockerfile` ve kterém bude nastavení jako před pár snímky s drobným rozdílem:
    - Místo: `COPY target/app.jar app.jar`
    - Bude: `COPY maven/app.jar app.jar`
  2. Přidejte do `pom.xml` plugin (viz. následující stránka)
- Poznámka: Nefunguje na Windows 7 s Docker Toolbox

# Docker & Maven II.

Poznámka:

Stará image se automaticky smaže!

To se dá změnit pomocí:

Configuration → images → image →  
build → <cleanup>none</cleanup>

```
<plugin>
```

```
<groupId>io.fabric8</groupId>
```

```
<artifactId>docker-maven-plugin</artifactId>
```

```
<version>0.27.2</version>
```

```
<configuration>
```

```
<images>
```

```
<image>
```

```
<name>app</name>
```

```
<build>
```

```
<dockerFileDir>${project.basedir}/src/main/docker</dockerFileDir>
```

```
<assembly>
```

```
<mode>dir</mode>
```

```
<targetDir>/app</targetDir>
```

```
<descriptorRef>artifact</descriptorRef>
```

```
</assembly>
```

```
</build>
```

```
</image>
```

```
</images>
```

```
</configuration>
```

```
</plugin>
```

Název výsledné image

Kde se nachází Dockerfile

Kam se nakopíruje JAR soubor (výsledný artifact)

Soubory se  
pouze zkopírují

Informace o tom, že se do image  
bude kopírovat pouze výsledný artifact

**Build image:**

mvn clean package docker:build

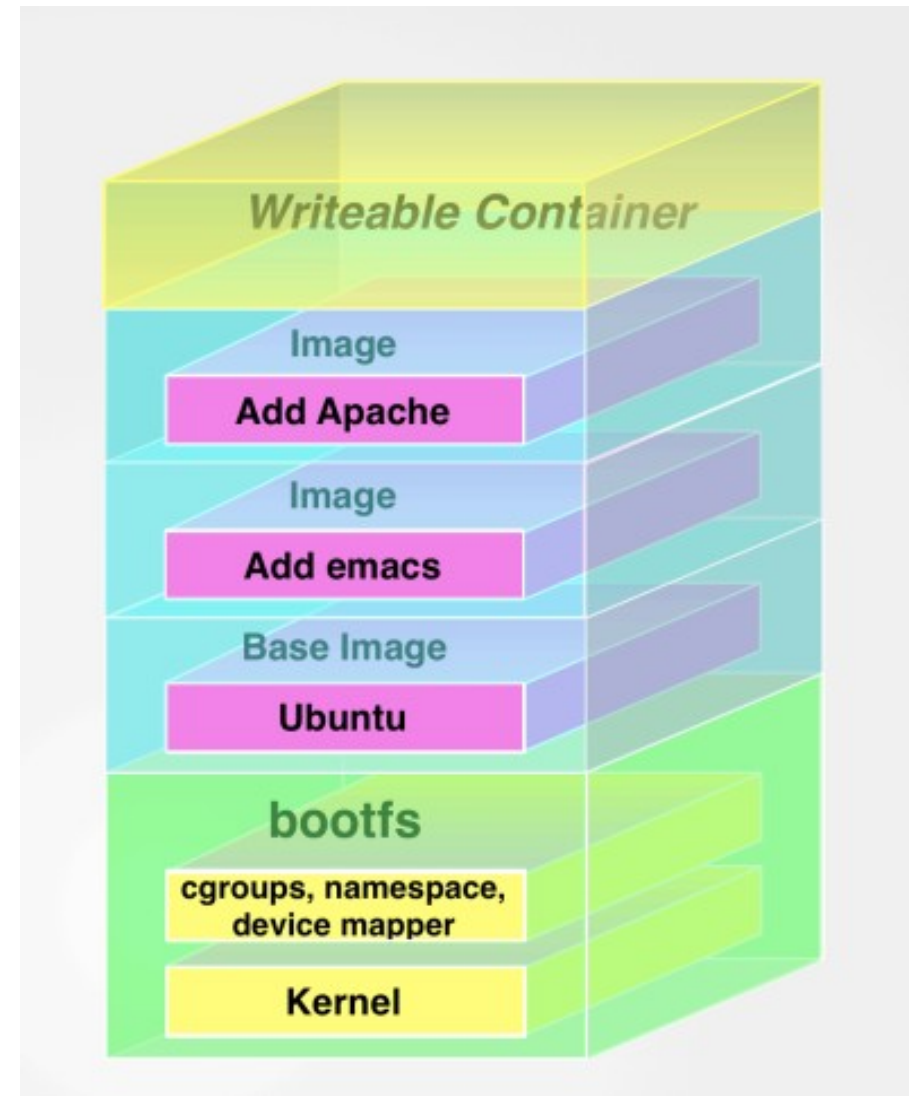
# Docker & Maven III.

```
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>docker-maven-plugin</artifactId>
  <version>0.27.2</version>
  <configuration>
    <images>
      <image>
        <name>app</name>
        <build>
          <dockerFileDir>${project.basedir}/src/main/docker</dockerFileDir>
          <assembly>
            <mode>dir</mode>
            <targetDir>/app</targetDir>
            <descriptorRef>artifact</descriptorRef>
          </assembly>
        </build>
      </image>
    </images>
  </configuration>
  <executions>
    <execution>
      <id>build</id>
      <phase>install</phase>
      <goals>
        <goal>build</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

**Build image:**  
mvn clean install

# Union File System

- Docker používá pro sgrupování images Union File System
  - [https://washraf.gitbooks.io/the-docker-ecosystem/content/Chapter%201/Section%203/union\\_file\\_system.html](https://washraf.gitbooks.io/the-docker-ecosystem/content/Chapter%201/Section%203/union_file_system.html)
  - <http://stackoverflow.com/questions/32775594/why-does-docker-need-a-union-file-system>
  - <https://en.wikipedia.org/wiki/UnionFS>





# Memory constraints & Java 10

- Poznámka: Je hodně dobrý nápad používat Docker minimálně s JDK 10 (resp. JDK 11 :-)
  - <https://blog.csanchez.org/2017/05/31/running-a-jvm-in-a-container-without-getting-killed/>
  - <https://blog.csanchez.org/2018/06/21/running-a-jvm-in-a-container-without-getting-killed-ii/>
  - Při spuštění Docker containeru je možné specifikovat maximální velikost alokované paměti pomocí `--memory 1000m`
  - Java automaticky přidělí heapu 1/4 této velikosti (256MB)
  - Množství alokované paměti pro heap se dá ovlivnit pomocí přepínačů
    - `-XX:MaxRAMPercentage=50` (toto nastaví velikost heapu na cca. 50% přidělené paměti, čili cca. 500MB)
- další přepínače: `-XX:MinRAMPercentage`, `-XX:InitialRAMPercentage`

# Memory constraints & Java 8u131

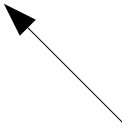
- Od Java 8u131 lze používat (od Java 10 deprecated) přepínače:

-XX:+UnlockExperimentalVMOptions


-XX:+UseCGroupMemoryLimitForHeap

-XX:MaxRAMFraction=2

Bere v úvahu  
nastavení --memory



↑  
Heap bude tvořit 50% nastavení --memory



Poznámka: Toto nastavení je hodně hloupé,  
protože umí pracovat pouze s celými čísly.  
1 = heap by tvořil 100% nastavení --memory.  
Ale v Javě není jenom Heap ...

# CPU constraints

- Obdobně je možné containeru nastavit maximální množství jader (out-of-the-box používá všechny jádra) pomocí přepínače (od Docker 1.13):
  - `--cpus=1.5`
  - [https://docs.docker.com/config/containers/resource\\_constraints/](https://docs.docker.com/config/containers/resource_constraints/)

# Networking

- Když se nainstaluje Docker, vytvoří automaticky tři sítě:
  - bridge (default)
  - host
  - none
- Také je možné vytvářet vlastní sítě (bude probráno později).  
Seznam všech sítí získáte pomocí:
  - `docker network ls`

# Networking – bridge I.

- `docker run --net=bridge`
  - Docker démon vytvoří na hostu bridge s názvem `docker0`
    - HOST:
      - `ip addr show docker0` (například 172.17.0.1)
    - CONTAINER:
      - `ip addr show eth0` (například 172.17.0.2)
    - CONTAINER:
      - `route` (je vidět že IP adresa hosta – 172.17.0.1 je nastavena jako default route)
    - CONTAINER:
      - `ping 172.17.0.1` FUNGUJE

# Networking – bridge II.

- Jenom pozor na to, aby něco komunikaci mezi containerem a hostem neustříhlo. Ve výchozím nastavení je možné se například k PostgreSQL připojit pouze z localhostu (je to definované v postgresql.conf). A na druhou stranu pozor na to, aby k databázi nemohl každý :-) ... od toho jsou například iptables.
  - PostgreSQL:
    - <http://www.cyberciti.biz/tips/postgres-allow-remote-access-tcp-connection.html>
  - MySQL:
    - <http://serverfault.com/questions/139323/mysql-bind-to-more-than-one-ip-address>

# Networking – bridge III.

- Jak získat IP adresu hosta z containeru:
  - `/sbin/ip route | awk '/default/ { print $3 }'`
- Pozor! Musíte ji získat při startu containeru, nikoli při vytváření image / containeru!!!!
- Pokud chceme forwardovat port z containeru na hosta, pak použijeme při startu `-p 80:80`, což jsme už používali.
  - Poznámka: Také existuje přepínač `-P`, který forwarduje VŠECHNY porty z containera na hosta. Toto není doporučené používat!!!

# Networking – host

- `docker run --net=host`
  - Container bude sdílet network stack s hostem. Kdybych zavolal na hostu i containeru: `ip addr show eth0`, pak bych získal stejné informace včetně stejné IP adresy.
- Poznámky k networkingu:
  - <http://stackoverflow.com/questions/24319662/from-inside-of-a-docker-container-how-do-i-connect-to-the-localhost-of-the-mach>



# Networking – host IP address test

## Dockerfile:

```
FROM alpine:3.4
WORKDIR /app
VOLUME ["/app"]
COPY run.sh run.sh
ENTRYPOINT ["/bin/sh", \
  "run.sh", "arg0", "arg1"]
```

Pozor! Tohle na jeden řádek!

run.sh:

```
IP_ADDR=$(/sbin/ip route | awk
'/default/ { print $3 }')
echo "aplikace funguje"
echo "argumenty: $@"
echo "docker0 ip addr:"
echo $IP_ADDR
```

## Test:

```
docker build --tag testImage .
docker run --rm -it testImage arg2 arg3
```

# Custom networking

- Pro propojení containerů se dřív používal link:
  - [https://docs.docker.com/engine/userguide/networking/default\\_network/dockerlinks/](https://docs.docker.com/engine/userguide/networking/default_network/dockerlinks/)
- V současnosti je best practice vytvořit custom network:
  - <https://docs.docker.com/engine/userguide/networking/#/user-defined-networks>
- Vytvoření vlastní sítě:
  - `docker network create mynetwork`
    - Jedná se o custom bridge network.
- Při vytvoření containeru se specifikuje, jakou síť má používat:
  - `docker run --network=mynetwork -d --name httpd1 httpd`
    - Při kooperaci více containerů je lepší používat docker-compose (viz. dále), které používá pro networking stejný mechanismus jako jsem popsal zde.

# Custom networking - příklad

- Spust'te tyto příkazy:
  - `docker network create pgnet`
  - `docker run --rm -it --name pg --network pgnet postgres`
  - `docker run --rm -it -p 5050:5050 --network pgnet fenglc/pgadmin4`
- Přejděte na <http://localhost:5050>
  - username: pgadmin4@pgadmin.org
  - password: admin
- Přihlašovací údaje k Vaší PostgreSQL databázi:
  - název serveru: pg
  - username: postgres
  - password: admin

# Nástroje pro práci s Dockerem

- docker (CLI)
- docker-compose (CLI + YAML soubor)
- Kitematic (GUI – Windows, Mac)
- Eclipse, IntelliJ Idea, NetBeans (GUI)
  - <https://github.com/docker/labs/tree/master/developer-tools/java>
- Visual Studio Code (GUI)
  - S pluginem „Docker“
- Portainer:
  - <https://www.portainer.io/installation/>

# Docker Compose

- Je možné konfigurovat container pomocí „docker run ...“, ale jakmile je konfigurace víc, pak to začne být zdlouhavé. Také dříve nebo později budeme chtít spouštět více containerů, navíc „ve správném pořadí“. Nebo potom je všechny budeme chtít najednou vypnout / odstranit. V takových situacích pomůže příkaz docker-compose.
- Vstupem je soubor docker-compose.yml, ve kterém se definuje konfigurace containerů.

# docker-compose.yml

- Příklad ze začátku přednášky předělaný do docker-compose.yml:

```
version: '3'
services:
  web:
    image: httpd
    ports:
      - "80:80"
    volumes:
      - c:/Users/jirka/DOCKER/html/:/usr/local/apache2/htdocs:ro
      - /tmp
```

**Spuštění:** docker-compose up -d

**Vypnutí:** docker-compose stop

**Smazání:** docker-compose rm -v -f

**Vypnutí a smazání:** docker-compose down

# Docker Compose app & db:

version: '3'

application.properties (Spring Boot):

services:

app:

image: app

ports:

- "8080:8080"

depends\_on:

- mypostgres

mypostgres:

image: postgres

ports:

- "5432:5432"

environment:

- POSTGRES\_PASSWORD=password
- POSTGRES\_USER=postgres
- POSTGRES\_DB=mydb

```
spring.datasource.url=jdbc:postgresql://mypostgres:5432/mydb
spring.datasource.username=postgres
spring.datasource.password=password
spring.jpa.hibernate.ddl-auto=create
```

# Docker Compose & wait-for-it

- Při používání Docker Compose je nejlepší tvořit služby které na sobě závisí tak, aby nezáviselo na pořadí jejich spouštění.
- Ne vždy je ale možné toho docílit. V takových situacích je velice užitečný wait-for-it skript, pomocí kterého je možné jednoduše počkat až se například nashartuje databáze a poté nashartovat aplikaci:
  - <https://github.com/vishnubob/wait-for-it>
- Protože je příklad na toto malinko komplexnější, tak jsem vytvořil následující repozitář, kde je kompletně rozchozený:
  - <https://github.com/jirkapinkas/spring-boot-postgresql-docker-compose>



# Docker Compose Build

- Pomocí docker-compose je možné také provést build custom images:
  - <https://stackoverflow.com/questions/50230399/what-is-the-difference-between-docker-compose-build-and-docker-build>

# Docker Prune

- Časem začnou staré image / containery zabírat na disku hodně místa. Dřív se používal Docker GC:
  - <https://github.com/spotify/docker-gc>
- Od Docker 1.13 jsou nové příkazy:
  - `docker system df` (obdoba `df` v Linuxu)
  - `docker system prune` (smaže všechny nepoužívaná data)
  - `docker image prune` (smaže všechny dangling image)
  - `docker image prune --all` (smaže všechny image, které nejsou používané žádným kontejnerem)
  - `docker volume prune` (smaže nepoužívané volumes)
  - `docker container prune` (smaže stoplé kontejnery)

Další užitečné příkazy:

<https://gist.github.com/bastman/5b57ddb3c11942094f8d0a97d461b430>

# Docker stats

- Statistiky používání disku / paměti, monitorování kontejnerů:
  - `docker ps --size`
  - `docker stats`
    - <https://docs.docker.com/config/containers/runmetrics/>
  - Kontejner pro export metrik ostatních kontejnerů (umožňuje export metrik do Prometheus):
    - <https://github.com/google/cadvisor>
  - Export metrik Dockeru do Prometheus:
    - <https://docs.docker.com/config/daemon/prometheus/>

# Docker registry I.

- Registry je něco jako Git pro Docker images.
  - <https://docs.docker.com/registry/deploying/>
- Na localhostu (i pro Windows 7) nevyžaduje žádnou speciální konfiguraci. Stačí jenom spustit:
  - `docker run -d -p 5000:5000 --restart=always --name registry registry:2`
- Potom nějakou image „otagovat“:
  - `docker pull ubuntu && docker tag ubuntu localhost:5000/ubuntu`
  - <https://docs.docker.com/engine/reference/commandline/tag/>
- Push do registry:
  - `docker push localhost:5000/ubuntu`
- Pull z registry:
  - `docker pull localhost:5000/ubuntu`

# Docker registry II.

- Konfigurace Docker registry někde jinde než na localhostu je o něco složitější a předpokládá se, že bude běžet na https s certifikátem od důvěryhodné authority. V celé řadě situací (zejména když registry běží pouze v intranetu) může být vhodné toto zabezpečení snížit:
  - <https://docs.docker.com/registry/insecure/>
- Bohužel když je registry nakonfigurované špatně, tak Docker v současné době vrací nic neříkající chyby jako:
  - Get https://SERVER:5000/v1/\_ping: http: server gave HTTP response to HTTPS client
    - Při startu registry nebyl přidán certifikát
      - <https://docs.docker.com/registry/deploying/#/get-a-certificate>
  - Get https://SERVER:5000/v1/\_ping: x509: certificate is valid for , not SERVER
  - Atd.

# Maven v Dockeru

- Docker umožňuje spouštět jakékoli konzolové aplikace. Příklad spuštění mvn clean install:

```
docker run -it --rm \
-v "$PWD":/usr/src/mymaven -w /usr/src/mymaven \
maven:3.5-jdk-8 mvn clean install
```

Poznámka: Celý příkaz je obvykle na jednom řádku  
Dokumentace: [https://hub.docker.com/\\_/maven/](https://hub.docker.com/_/maven/)

# Docker & X11

- Docker umožňuje spouštět i GUI aplikace. Příklad spuštění Eclipse:

xhost +

```
docker run --rm -it -e DISPLAY=$DISPLAY \
-v /tmp/.X11-unix:/tmp/.X11-unix psharkey/eclipse
```

# Docker alternatives

- Rkt (Rock-it)
  - <https://github.com/coreos/rkt>



# Clustering

- Clustering je důležitý na produkci pro zajištění HA. Pokud jsou Docker containers stateless, pak je možné je zapojit do clusteru ... jenom musí být něco, co bude provádět tzv. orchestraci containerů (zapojení do clusteru apod.) V současnosti jsou nejpopulárnější:
  - Kubernetes (od společnosti Google)
  - Docker Swarm (od Docker 1.12)
  - Apache Mesos
- V současnosti se v 90% případů používá Kubernetes a vypadá to, že se časem bude používat v 99% případů. Docker se snaží integrovat Kubernetes, ale obvykle se Kubernetes v současnosti používá napřímo.

# Kubernetes

- Existuje mnoho způsobů jak pracovat s Kubernetes, od Docker 18.06 je podpora pro Kubernetes integrovaná přímo v Dockeru:

```
set DOCKER_ORCHESTRATOR=kubernetes
```

```
docker stack deploy nginxes -c docker-compose.yml
```

```
docker-compose.yml:
```

```
version : '3'
```

```
services :
```

```
  nginx :
```

```
    image : nginx:alpine
```

```
    ports :
```

```
      - "80:80"
```

```
  deploy :
```

```
    replicas : 5
```

Poznámky:

- Zatím je tato podpora spíš experimentální
- Úplně na začátku se musí Kubernetes zapnout (na Windows je to v Docker settings)

# Swarm I.

- 1. inicializace:
  - Manager:

```
docker swarm init --advertise-addr IP_ADRESA
docker node ls
```
  - Worker:

```
docker swarm join ...
docker node ls
```

# Swarm II.

- 2. Ad-hoc service:

```
docker service create --name registry --publish 5000:5000  
registry:2
```

```
docker service ls
```

```
docker service scale registry=5
```

```
docker service ps registry
```

```
docker service rm registry
```

# Swarm III.

- 3. Stack (pozor! docker-compose.yml musí být verze 3!!!):  
    `docker stack deploy --compose-file=docker-compose.yml myapp`  
    `docker service scale myapp=2`  
    `docker stack rm myapp`

# Awesome Docker

- Awesome Docker:
  - <https://github.com/veggemonk/awesome-docker>