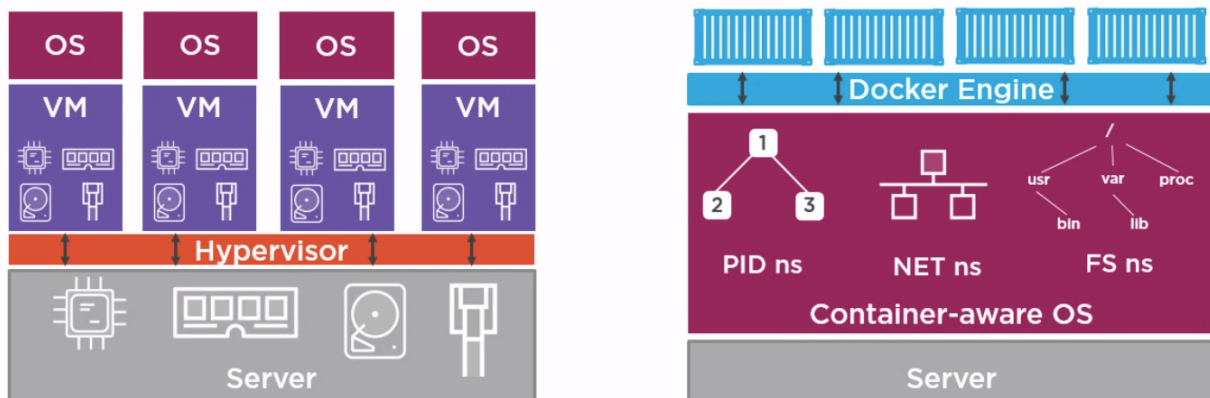


# Docker

## 1. Motivacija

Sa pojavom virtuelnih mašina (VM), omogućeno je izbegavanje situacija gde se fizički serveri koriste na takav način da je iskoristivost resursa vrlo mala, što je u prošlosti često bio slučaj (iskoristivost resursa često bude od 10-20%). Virtuelne mašine su apstrakcija fizičkog hardvera koje omogućuju pretvaranje jednog servera u više manjih servera. Svaka VM-a uključuje punu kopiju operativnog sistema, aplikacije, biblioteke, pri čemu se ispod njih nalazi hypervisor, odnosno softver koji omogućuje kreiranje, pokretanje i izvršavanje više VM-a na jednom fizičkom računaru (type 1, postoji i type 2 hypervisor) i omogućuje deljenje fizičkih resursa (memoriju, procesor) između njih. Dakle, virtuelne mašine (virtuelni serveri) su jeftiniji od fizičkih servera, s obzirom da troše deo resursa istog. Pored manje cene, omogućuju lakše upravljanje, bolje skaliranje, konzistentno okruženje za izvršavanje aplikacija što ih čini odličnom podlogom za pružanje usluga web servisa.

Sa pojavom virtuelnih mašina „svet je postao bolje mesto”, ali i dalje je bilo prostora za napredak. Ono što je negativna strana VM-a, jeste da svaka zahteva *underlying* OS što znači da će se deo resursa koristiti za podizanje i izvršavanje operativnog sistema. Takođe operativni sistemi uključuju potencijalni *overhead* u obliku dodatnih potreba za licencama, potreba za administracijom (*updates, patches*) itd.



Slika 1 - Virtuelne mašine vs kontejneri

Ovi nedostaci su u priču uključili kontejnere. Za razliku od virtuelnih mašina, gde svaka ima sopstveni OS i oslanja se na *hypervisor*, kontejneri se oslanjaju na **jedan** *host* OS i dele njegove funkcije kernela (takođe i *binaries, libraires* itd.) i samim tim su lakši

(*lightweight*) i u priličnoj meri se smanjuje *overhead* koji donose VM-e. Kontejnerske tehnologije su bile prisutne duže vremena, ali nisu bile previše popularne jer je kreiranje i upravljanje kontejnerima bilo dosta kompleksno ali je *Docker* uspeo to da promeni.

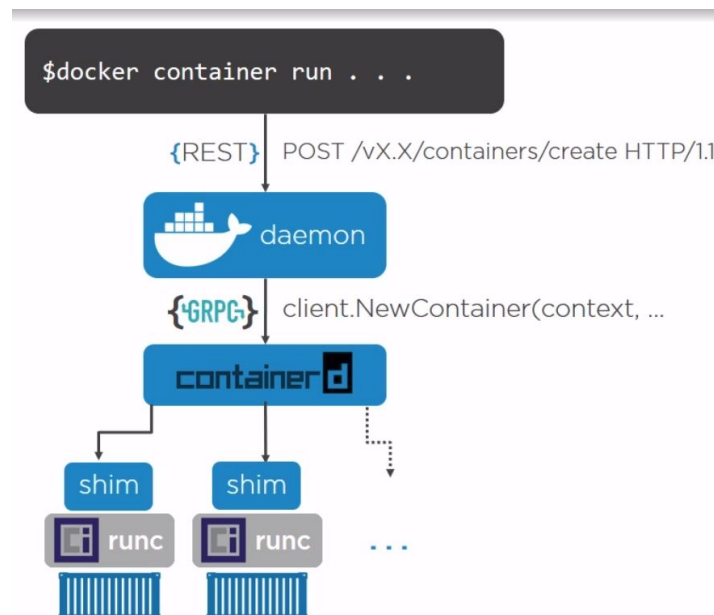
## 2. Šta je Docker i koje su njegove komponente?

*Docker* je *open-source* platforma koja automatizuje proces *deployment*-a aplikacija u softverske kontejnere. On dodaje *application deployment engine* na vrh *virtuelized container execution environment*-a pri čemu je dizajniran tako da omogući lagano i brzo okruženje za izvršavanje naših aplikacija kao i izuzetno lako premeštanje aplikacija iz jednog okruženja u drugo. (*test* -> *production*).

Njegove osnovne komponente su:

1. *Docker Engine*,
2. *Docker Images*,
3. *Registries (Docker Hub)*,
4. *Docker containers*,
5. *Docker Swarm*

Kada pričamo o *Docker Engine*-u, govorimo o klasičnoj klijent-server aplikaciji. *Docker klijent* nam pruža *cli (command line interface)* putem kojeg unosimo komande, na osnovu kojih se generišu API request-ovi koji se šalju serveru (*Docker daemon*-u) koji ih obrađuje.



## Slika 2 - Arhitektura Docker engine-a

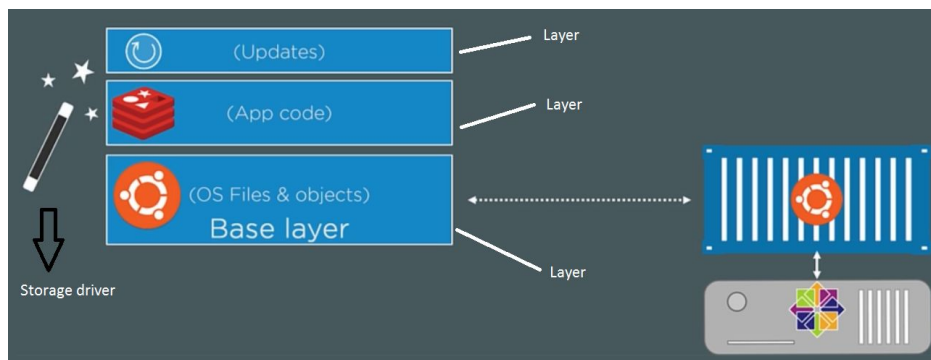
Sam *Docker daemon* je nakon refaktorisanja (zbog toga što narastao u jedan veliki monolit) ostao bez ikakvog koda koji zaista kreira i pokreće kontejnere. On se obraća putem gRPC API-a preko lokalnog linux socket-a *containerd*-u (*long running daemon*-u) koji predstavlja API „fasadu” koja omogućuje startovanje *containerd-shim*-a, odnosno roditeljskog procesa za svaki kontejner, gde *runc* (*container runtime*) vrši kreiranje kontejnera. Sloj ispod *containerd*-a vrši kompletan rad sa kernelom, odnosno koristi njegove funkcije.

Iako arhitektura izgleda prilično kompleksno, ovakva podela omogućuje da se pojedine komponente bez ikakvih problema zamenjuju, a da to ne utiče na pokrenute kontejnere, što sa administratorske tačke gledišta puno olakšava stvari. Na primer, moguće je promeniti verziju *Docker*-a a da se pri tome ne moraju zaustavljati već pokrenuti kontejneri.

**Napomena:** Arhitektura na Windows operativnom sistemu izgleda malo drugačije. Za detalje, pogledati zvaničnu dokumentaciju.

## 3. Šta su Docker slike?

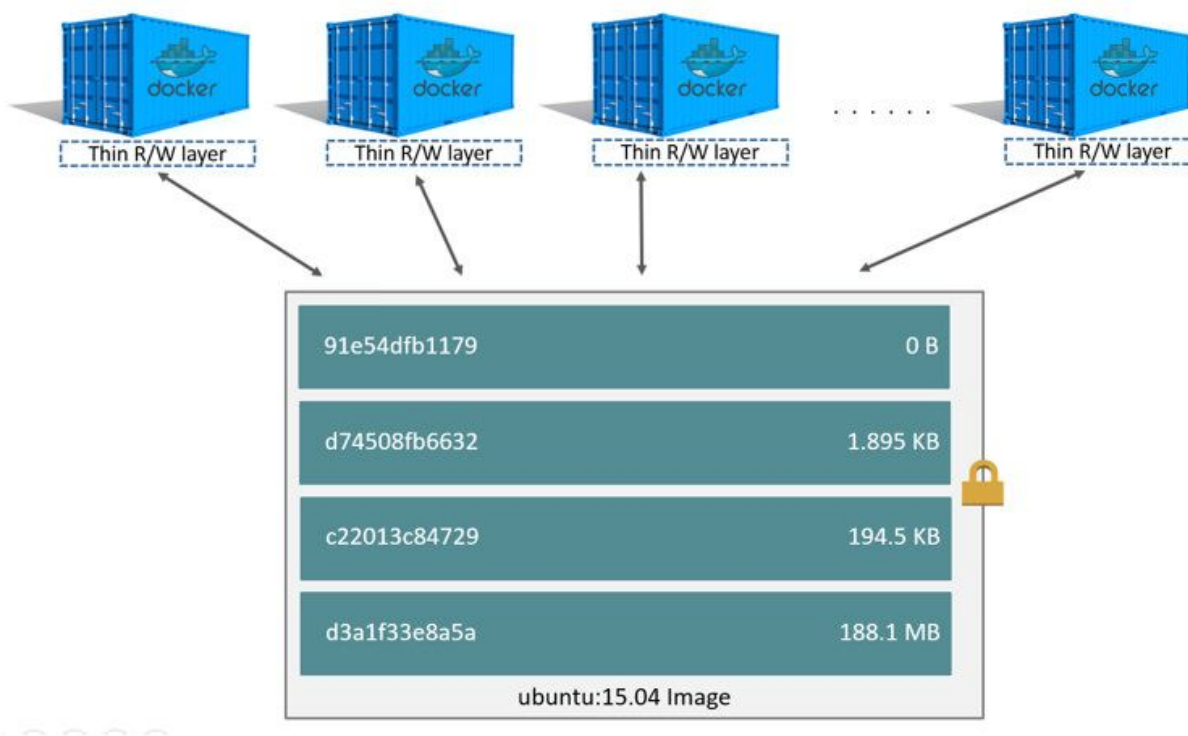
Generalno je poznat koncept slike kada je priča o virtuelnim mašinama. Za sličnu stvar se koriste i *Docker slike*, odnosno predstavljaju *build-time* konstrukt od kojih nastaju kontejneri ali se tu sličnost završava. *Docker slike* predstavljaju skup *read-only layer*-a, gde svaki sloj predstavlja različitosti u fajlsistemu u odnosu na prethodni sloj pri čemu uvek postoji jedan bazni (*base*) sloj. Upotrebom *storage driver*-a, skup svih slojeva čini *root filesystem* kontejnera, odnosno svi slojevi izgledaju kao jedan unificirani fajlsistem.





Slika 3 - Image layer-i

Svi ovi *read-only* slojevi predstavljaju osnovu za svaki kontejner koji se pokreće i ne mogu se menjati. Prilikom pokretanja svakog kontejnera, *Docker* dodaje još jedan sloj koji je *read-write* tipa i u koji se upisuju nove datoteke i sve izmene. Ukoliko želimo da menjamo neki fajl koji se nalazi u nekom *read-only* sloju, taj fajl će biti kopiran u *read-write* sloj, biće izmenjen i kao takav dalje korišćen. Originalna verzija će i dalje postojati (nepromenjena), ali nalaziće se „skrivena” ispod nove verzije.



Slika 4 - Svaki kontejner dobije svoj *read-write* sloj

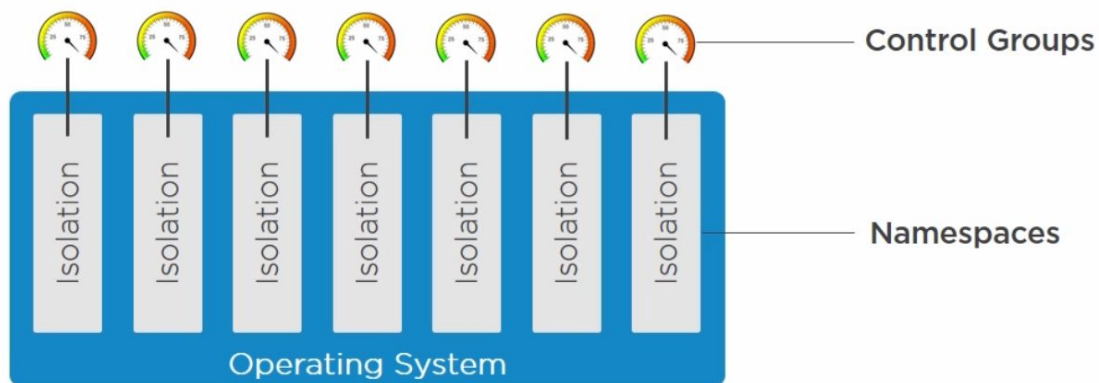
Ovakav mehanizam se zove *Copy-on-write* i delom čini *Docker* zaista moćnim. Koliko god kontejnera da kreiramo, *read-only* slojevi će uvek biti isti, tj. ostaće nepromenjeni, samo će svaki kontejner dobiti sopstveni *read-write* sloj. Na ovaj način se štedi jako puno prostora na disku, jer kada smo jednom preuzeli/kreirali sliku, koliko god kontejnera da pokrenemo, slika ostaje apsolutno nepromenjena.

## 4. Odakle se preuzimaju postojeće slike?

*Docker* čuva slike u registrima, pri čemu postoje dva tipa, odnosno javni i privatni. Javni registar kojim upravlja *Docker, Inc.* se zove *DockerHub* i na njemu svako može da napravi nalog i da tamo čuva i deli sopstvene slike. Postoje dva tipa slika, a to su oficijelne, koje žive na top nivou *DockerHub* namespace-a (npr. *Ubuntu*, *Redis* itd.) i neoficijelne (korisničke). Takođe je moguće napraviti privatni registar u kome se mogu čuvati slike i sve to sakriti iza *firewall*-a, što je ponekad neophodno za pojedine organizacije.

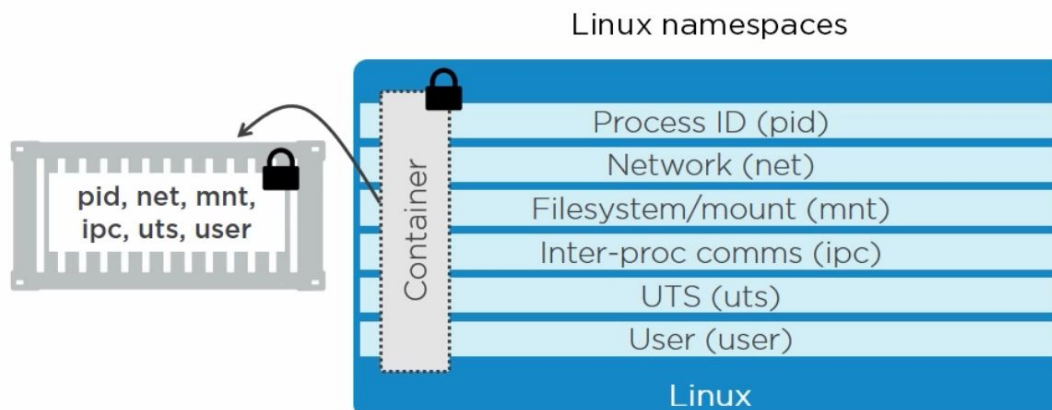
## 5. Šta predstavljaju kontejneri?

Kako slike predstavljaju *build-time* konstrukt, tako su kontejneri *run-time* konstrukt. Gruba analogija odnosa između slike i kontejnera se može posmatrati kao klasa i instanca te klase. Kontejneri predstavljaju *lightweight execution environment* koji omogućuju izolovanje aplikacije i njenih zavisnosti koristeći kernel *namespaces* i *cgroups* mehanizme.



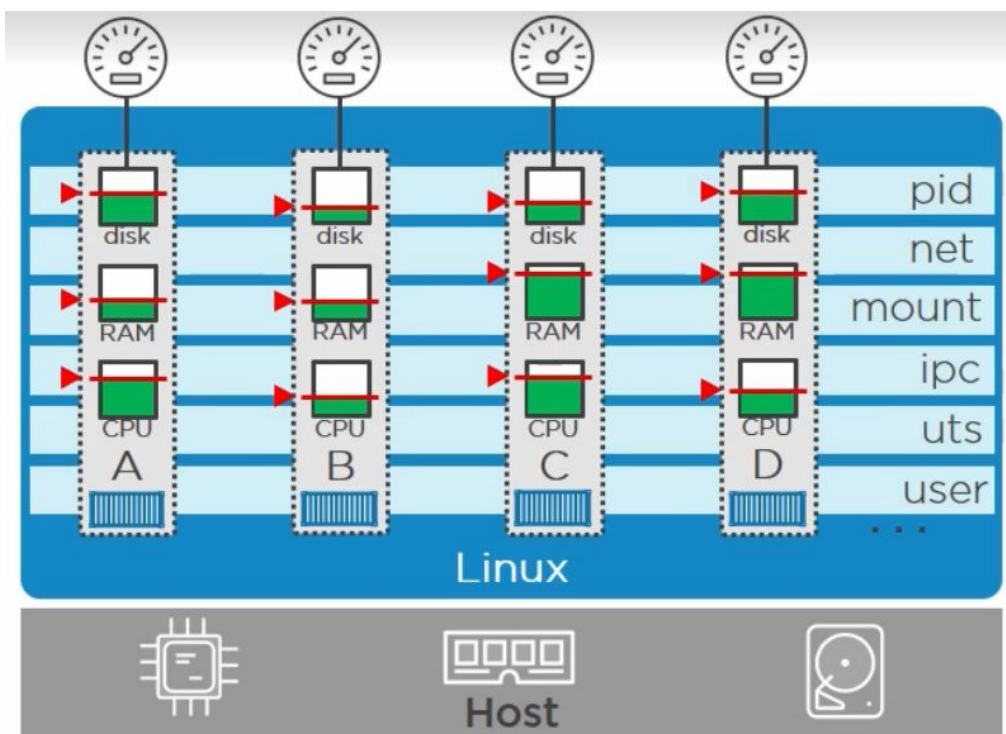
Slika 5 - Kernel features

*Namespaces* nam omogućuju izolaciju, odnosno da podelimo naš operativni sistem na manje izolovanih virtuelnih operativnih sistema (kontejnera). Odnosno kontejneri *smells-and-feels* kao zasebni operativni sistemi (kao slučaj kod VM-a), samo što to nisu, jer svi dele isti kernel na host OS-u. Svaki kontejner ima sopstveni skup *namespace*-a (kada pričamo o Linux-u, to su *namespace*-ovi sa slike 6) pri čemu je njegov pristup ograničen isključivo na taj prostor imena, odnosno svaki kontejner nije uopšte svestan postojanja drugih kontejnera.



Slika 6 - Linux namespaces

Međutim, iako imamo potpunu izolaciju, to nam nije skroz dovoljno. Kao i svaki *multi-tenant* sistem, uvek postoji opasnost od *noisy neighbors*-a, odnosno neophodan nam je mehanizam kojim ćemo ograničiti upotrebu resursa host OS-a od strane svih kontejnera kako se ne bi desilo da jedan kontejner troši mnogo više resursa od drugih. To nam omogućava *control groups* (*cgroups*) kernel mehanizam (slika 7).



Slika 7 - Linux control groups

## 6. Kako raditi sa kontejnerima?

Pre nego što bi mogli bilo šta da radimo sa kontejnerima, neophodno je izvršiti instalaciju *Docker CE-a (Community edition)*. Kompletan *guide* za instalaciju za bilo koji operativni sistem (u primerima će biti korišćen Ubuntu) postoji u zvaničnoj dokumentaciji na sledećem linku: <https://docs.docker.com/install/linux/docker-ce/ubuntu/>.

Nakon instalacije, neophodno je proveriti da li je instalacija bila uspešna. U terminalu otkucati komandu: `sudo docker info`

Napomena: Ukoliko ne želite da izvršavate Docker naredbe sa povišenim privilegijama (da kucate sudo), onda je neophodno nakon instalacije ispratiti par koraka ispisanih u dokumentaciji: <https://docs.docker.com/install/linux/linux-postinstall/>.



```
stefan@stefan-MS-7817: ~  
File Edit View Search Terminal Help  
stefan@stefan-MS-7817 ~$ sudo docker info  
[sudo] password for stefan:  
Containers: 4  
  Running: 0  
  Paused: 0  
  Stopped: 4  
Images: 5  
Server Version: 18.09.2  
Storage Driver: overlay2  
  Backing Filesystem: extfs  
  Supports d_type: true  
  Native Overlay Diff: true  
Logging Driver: json-file  
Cgroup Driver: cgroupfs  
Plugins:  
  Volume: local  
  Network: bridge host macvlan null overlay  
  Log: awslogs fluentd gcplogs gelf journald json-file local logentries splunk syslog  
Swarm: inactive  
Runtimes: runc  
Default Runtime: runc  
Init Binary: docker-init  
containerd version: 9754871865f7fe2f4e74d43e2fc7ccd237edcbce  
runc version: 09c8266bf2fcf9519a651b04ae54c967b9ab86ec  
init version: fec3683
```

Rezultat naredbe jesu informacije o broju kontejnera, broju slika, *storage driver*-u i ostalim bazičnim konfiguracijama.

Ukoliko želimo da pokrenemo neki kontejner, kucamo komandu: *docker run naziv\_slike*. U konkretnom slučaju otkucaćemo: *docker run -i -t ubuntu /bin/bash*

```
root@739a84e7e100: /  
File Edit View Search Terminal Help  
stefan@stefan-MS-7817 ~$ docker run -i -t ubuntu /bin/bash  
Unable to find image 'ubuntu:latest' locally  
latest: Pulling from library/ubuntu  
6abc03819f3e: Pull complete  
05731e63f211: Pull complete  
0bd67c50d6be: Pull complete  
Digest: sha256:f08638ec7ddc90065187e7eabdfac3c96e5ff0f6b2f1762cf31a4f49b53000a5  
Status: Downloaded newer image for ubuntu:latest  
root@739a84e7e100:/#
```

Dakle šta se najpre dogodilo? Docker nije uspeo da pronađe sliku sa datim nazivom na lokalnom računaru, pa se obratio javnom registru (*DockerHub*-u) i krenuo da povlači poslednju *stable* verziju (označena tagom *latest*) slike. Rekli smo da se slike sastoje iz više *layer*-a, pa je preuzeo svaki sloj (linije koje se završavaju sa *Pull complete*). Nakon preuzimanja, pokrenuo je nov kontejner. Ovde smo dodali i dva flega



prilikom pokretanja komande. Fleg **-i** i **-t**. Prvi naglašava da je neophodno održati standard input (STDIN), dok drugi fleg dodeljuje pseudo terminal (terminal koji ima funkcije kao i pravi fizički terminal). Nakon naziva slike, zadali smo i komandu koja je pokrenula **Linux shell** pri čemu nam se pokretanje kontejnera prikazuje kao na slici. Kada pokrenemo **top** komandu unutar kontejnera, vidimo da je to jedini proces koji je zapravo pokrenut u našem kontejneru.

```

root@739
File Edit View Search Terminal Help
top - 09:31:54 up 1:46, 0 users, load average: 0.24, 0.27, 0.26
Tasks: 2 total, 1 running, 1 sleeping, 0 stopped, 0 zombie
%Cpu(s): 4.9 us, 1.0 sy, 0.0 ni, 94.1 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 16369116 total, 9501912 free, 3551952 used, 3315252 buff/cache
KiB Swap: 7999484 total, 7999484 free, 0 used. 12421752 avail Mem

  PID USER      PR  NI   VIRT   RES   SHR  S  %CPU  %MEM     TIME+ COMMAND
   1 root        20   0   18508   3508  3096  S   0.0   0.0    0:00.03 bash
  10 root        20   0   36616   3232  2792  R   0.0   0.0    0:00.00 top

```

Sa komandom **exit** napuštamo kontejner i vraćamo se na glavni terminal. Ono što je bitno razumeti je da smo sa ovom komandom ugasili glavni proces kontejnera i samim tim smo ugasili i kontejner.

Sa komandom **docker ps** smo zatražili izlistavanje svih pokrenutih kontejnera.

```

stefan@stefan-MS-7817: ~
File Edit View Search Terminal Help
stefan@stefan-MS-7817: ~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS              NAMES
739a84e7e100       ubuntu             "/bin/bash"        21 minutes ago      Exited (0) About a minute ago      eager_visvesvaraya
stefan@stefan-MS-7817: ~$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS              NAMES
739a84e7e100       ubuntu             "/bin/bash"        21 minutes ago      Exited (0) About a minute ago      eager_visvesvaraya
stefan@stefan-MS-7817: ~$ docker ps -l
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS              NAMES
739a84e7e100       ubuntu             "/bin/bash"        21 minutes ago      Exited (0) About a minute ago      eager_visvesvaraya
stefan@stefan-MS-7817: ~$ docker ps -n 1
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS              NAMES
739a84e7e100       ubuntu             "/bin/bash"        21 minutes ago      Exited (0) About a minute ago      eager_visvesvaraya
stefan@stefan-MS-7817: ~$

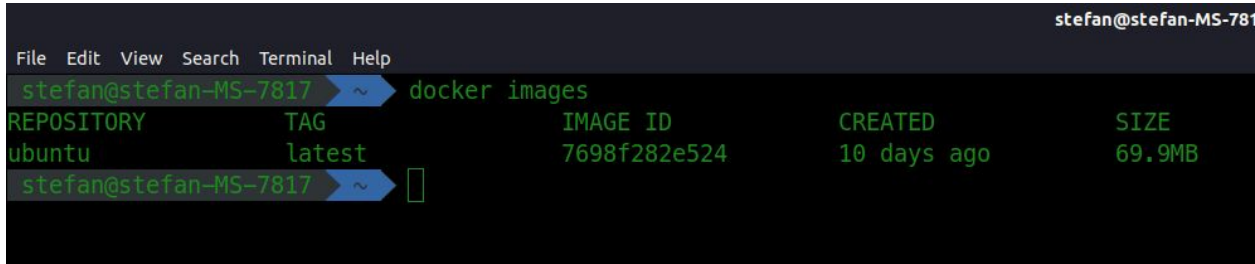
```

S obzirom da smo sa **exit** ugasili glavni proces našeg kontejnera (samim tim i njega), prilikom izvršenja gorepomenute komande neće biti izlistane informacije o kontejneru. Dodavanjem flega **-a**, izlistavamo i pokrenute i zaustavljene kontejnere, dok sa flegom **-l** izlistavamo informacije o poslednjem kontejneru koji je bio pokrenut, bez obzira da li je i dalje pokrenut ili je zaustavljen. Sa flegom **-n x** slična priča kao i sa **-l**, s tim što ovde eksplicitno naglašavamo za koliko kontejnera želimo da vidimo informacije. Konkretno stvari koje nam se prikazuju jesu:

1. **ID** - Identifikator kontejnera.
2. **IMAGE** - Slika od koje je kreiran kontejner.
3. **COMMAND** - izvršena komanda.
4. **STATUS** - Status našeg kontejnera (koliko je dugo pokrenut/ugašen).

5. **PORTS** - Izloženi portovi.
6. **NAMES** - Naziv kontejnera (Ako nije eksplicitno zadat putem flega, biće generisano ime).

Sa komandom **docker images** izlistavamo informacije o svim preuzetim i kreiranim slikama.

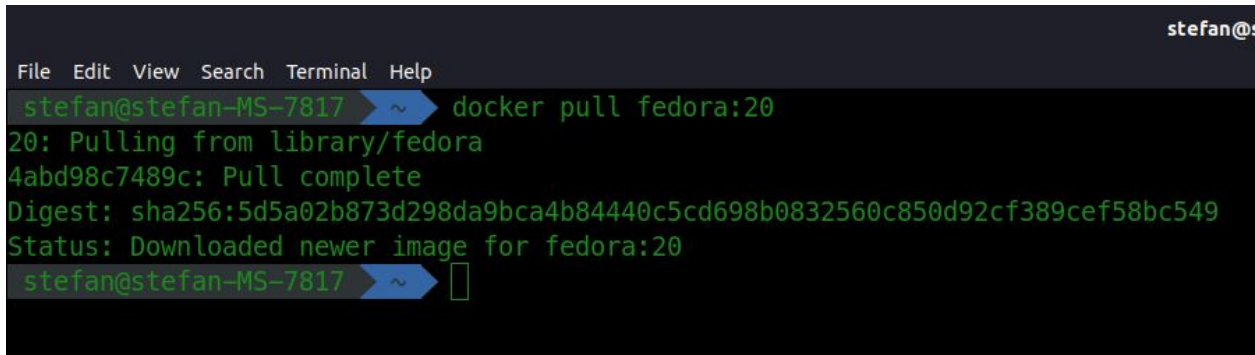


```
stefan@stefan-MS-7817 ~$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
ubuntu              latest             7698f282e524       10 days ago        69.9MB
stefan@stefan-MS-7817 ~$
```

Informacije koje nam se prikazuju su:

1. **REPOSITORY** - Repozitorijum sa koje je slika preuzeta.
2. **TAG** - Oznaka, koja najčešće ima za ulogu da prikaže verziju slike (npr. za *Ubuntu* je to 18.04/18.10 itd.). Ukoliko ne naglasimo koji tag želimo, biće preuzeta poslednja *stable* verzija slike.
3. **IMAGE ID** - Identifikator slike.
4. **CREATED** - Kada je slika kreirana.
5. **SIZE** - Veličina slike.

Sa komandom **docker run** smo istovremeno preuzeli sliku i odmah pokrenuli kontejner od nje. Možemo izvršiti i samo preuzimanje slike bez naknadnog pokretanja putem komande **docker pull naziv\_slike:tag**.



```
stefan@stefan-MS-7817 ~$ docker pull fedora:20
20: Pulling from library/fedora
4abd98c7489c: Pull complete
Digest: sha256:5d5a02b873d298da9bca4b84440c5cd698b0832560c850d92cf389cef58bc549
Status: Downloaded newer image for fedora:20
stefan@stefan-MS-7817 ~$
```

U konkretnom slučaju, preuzeli smo Fedora sliku gde smo sa tagom nazačili verziju 20.

Neke od vrlo korisnih komandi:

1. **docker rm naziv\_kontejnera** (dodatno fleg **-f** za brisanje kontejnera koji je pokrenut. Umesto naziva se može koristiti i id).

2. `docker start naziv_kontejnera` (pokretanje kontejnera sa zadatim nazivom, može se koristiti i id).
3. `docker stop naziv_kontejnera` (zaustavljanje kontejnera sa zadatim nazivom, može se koristiti i id).
4. `docker exec` (omogućuje izvršavanje komandi unutar kontejnera).
5. `docker rmi naziv_slike` (omogućuje brisanje slike po nazivu).

Postoji naravno još komandi i puno dodatnih flegova za svaku komandu i dodatne informacije o svakoj se mogu naći u odličnoj zvaničnoj dokumentaciji: <https://docs.docker.com/engine/reference/commandline/docker/>

## 7. Kako kreirati sopstvene slike?

Videli smo kako da pokrenemo kontejnere na osnovu već postojećih slika, ali ono što nas konkretno interesuje jeste kako da kreiramo sopstvene slike i da pomoću njih pokrenemo naše kontejnere u kojima će se izvršavati neki konkretan mikroservis (u primeru neka *Spring-Boot* aplikacija).

Za potrebe kreiranja naše slike, neophodno je da kreiramo *Dockerfile* (sa tim nazivom), odnosno tekstualnu datoteku (najbolja praksa je da se ona nalazi u root direktorijumu **projekta**) koja koristi bazični *DSL* sa instrukcijama za kreiranje slika. Kada kreiramo taj fajl, komandom `docker image build` ćemo kreirati našu sliku izvršavanjem instrukcija koje smo napisali i zatim ćemo od te slike startovati kontejner.

Format je relativno jednostavan:

```
# Comment
INSTRUCTION arguments
```

Instrukcije koje postoje su:

1. **FROM** - Pomoću ove instrukcije definišemo koja je bazna slika za predstojeće instrukcije koje će biti izvršene. Svaki fajl **mora** početi FROM **instrukcijom**, s tim što je moguće imati više FROM instrukcija u istom *Dockerfile-u*. Bazična slika bi trebala da bude oficijelna i po potrebi sa *latest* tagom, jer su te slike proverene.
2. **ADD** - Ova instrukcija kopira fajlove sa zadate destinacije u fajlsistem slike na određenoj destinaciji (biće dodat novi sloj u slici).
3. **RUN** - Omogućuje izvršavanje komande, pri čemu će rezultat biti novi sloj (*layer*) u samoj slici.
4. **COPY** - Slično kao i ADD instrukcija, s tim što ADD omogućuje da **source** bude i *URL*, dok COPY zahteva fizičku putanju na disku (biće dodat novi sloj u slici).

5. **WORKDIR** - Postavlja putanju odakle će pojedine komande biti izvršene.
6. **EXPOSE** - Definišemo port kako bi mogli da odradimo mapiranje portova da bi kontejneri mogli da komuniciraju sa spoljašnjim svetom.
7. **ENTRYPOINT** - Postavljamo *executable* koji će biti pokrenut sa pokretanjem kontejnera.
8. **ENV** - Podešavanje *environment* varijabli.
9. **LABEL** - Dodaje metapodakte slike, poput verzije, *maintainer* itd.

Postoji još instrukcija koje se mogu definisati u *Dockerfile*-u, i više informacija o njima kao i o formatu argumenata instrukcija možete naći u zvaničnoj dokumentaciji: <https://docs.docker.com/engine/reference/builder/>. Takođe, preporuke koje diktira najbolja praksa možete naći u zvaničnoj dokumentaciji: [https://docs.docker.com/develop/develop-images/dockerfile\\_best-practices/](https://docs.docker.com/develop/develop-images/dockerfile_best-practices/).

Dakle tok koji treba ispoštovati je pisanje koda, pa kad s tim završimo, kreiramo *Dockerfile*, pokrećemo **docker image build** kako bi kreirali sliku, i zatim startujemo kontejner na osnovu slike koju smo kreirali.



Kreiranja *Dockerfile*-a i *build*-ovanje slike biće ilustrovano na **Spring-boot** aplikaciji iz materijala (folder first-app).

```
FROM openjdk:8-jdk-alpine
LABEL maintainer="danijelradakovic@uns.ac.rs"
EXPOSE 8080
WORKDIR /app
COPY ./servers.jar ./
CMD ["java", "-jar", "servers.jar"]
```

Dockerfile

U samom *Dockerfile*-u definisali smo 5 instrukcija:

1. Sa prvom instrukcijom smo rekli šta je bazna slika. U ovom slučaju smo izabrali oficijalnu sliku koja se oslanja na *Alpine-linux* koji dolazi sa instaliranim **jdk**-om i verzije 8.

2. Sa drugom instrukcijom smo zadali metapodatke koje se odnose na kreatora i održavaoca slike.
3. EXPOSE-ujemo još port na kom aplikacija sluša unutar kontejnera.
4. WORKDIR podešava trenutni radni direktorijum unutar kontejnera na */app*.
5. COPY instrukcija vrši kopiranje servers.jar fajla sa host fajl sistema u trenutni radni direktorijum unutar kontejnera (odnosno kopira u */app* direktorijum)
6. CMD instrukcija navodi šta će biti *executable* i šta će biti pokrenuto sa samim pokretanjem kontejnera.

Kada smo kreirali *Dockerfile*, pozicioniramo se u korenski direktorijum konkretne aplikacije i izvršimo komandu **docker image build -t first-app .** . Sa flegom **-t** definišemo naziv naše slike (potencijalno možemo dodati i tag, ali ako ga ne dodamo, biće *latest*) i sa tačkom **.** definišemo šta je *build context*, odnosno lokaciju našeg izvornog koda. Ako smo pozicionirani u korenskom direktorijumu aplikacije, onda je lokacija tekući direktorijum. Rezultat izvršavanja komande je prikazan u pratećoj slici.

```
[daniyel@asus first-app]$ docker build -t first-app .
Sending build context to Docker daemon  47.5MB
Step 1/6 : FROM openjdk:8-jdk-alpine
--> a3562aa0b991
Step 2/6 : LABEL maintainer="daniyelradakovic@uns.ac.rs"
--> Running in b7d41adac86b
Removing intermediate container b7d41adac86b
--> 619436c5cde2
Step 3/6 : EXPOSE 8080
--> Running in 4db8edc00d10
Removing intermediate container 4db8edc00d10
--> 31ee9d3a0ec9
Step 4/6 : WORKDIR /app
--> Running in 4136db1599bf
Removing intermediate container 4136db1599bf
--> b35ca4b6360a
Step 5/6 : COPY ./servers.jar ./
--> 5de02d7ec8dc
Step 6/6 : CMD ["java", "-jar", "servers.jar"]
--> Running in 958c21d30a8c
Removing intermediate container 958c21d30a8c
--> d29bc4587902
Successfully built d29bc4587902
Successfully tagged first-app:latest
[daniyel@asus first-app]$
```

Ukoliko ukucamo komandu **docker images**, kreirana slika će nam biti prikazana kao i sve ostale preuzete slike.

S obzirom na to da aplikacija skladišti podatke u MySQL bazu, neophodno je pokrenuti MySQL bazu. Da bi smo uspesno realizovali komunikaciju između baze i aplikacije poželjno je da se nalaze unutar iste mreže.



1. Kreirati novu mrežu pod nazivom *first-network*:

***docker network create first-network***

2. Pokrenuti MySQL kontejner i dodati kontejner u *first-network* mrežu (sve ovo je jedna komanda):

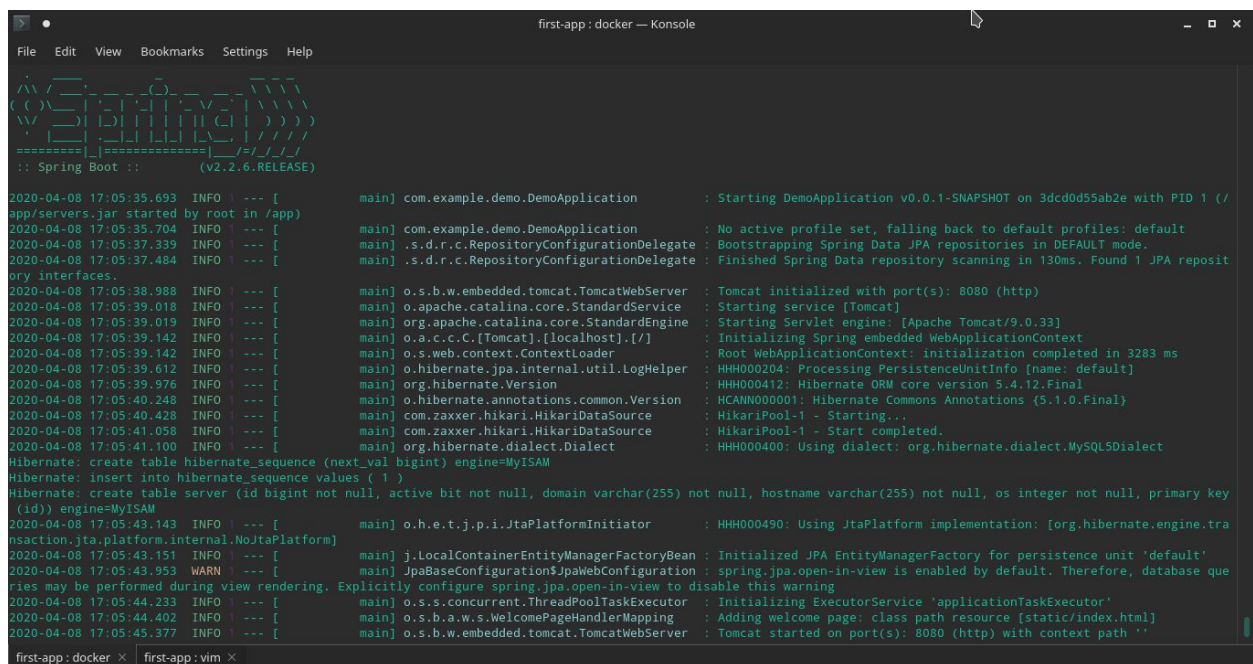
**docker run -d --network first-network --name mysql -e MYSQL\_ROOT\_PASSWORD=password -e MYSQL\_USER=sa -e MYSQL\_PASSWORD=zgadija -e MYSQL\_DATABASE=servers mysql:8.0.19**

3. Nakon izvršene komande sačekati par minuta da se baza podigne,

4. Pokrenuti kontejner Spring-Boot aplikacije i dodati kontejner u *first-network* mrežu (sa **-p** flegom kažemo da mapiramo port iz kontejnera na port na hostu) :

**docker run -it --network first-network --name app -p 8089:8080 -e DATABASE\_USERNAME=sa -e DATABASE\_PASSWORD=zgadija -e DATABASE\_DOMAIN=mysql -e DATABASE\_PORT=3306 first-app**

(Nakon izvršene komande treba da se dobije rezultat kao na sledećoj slici)

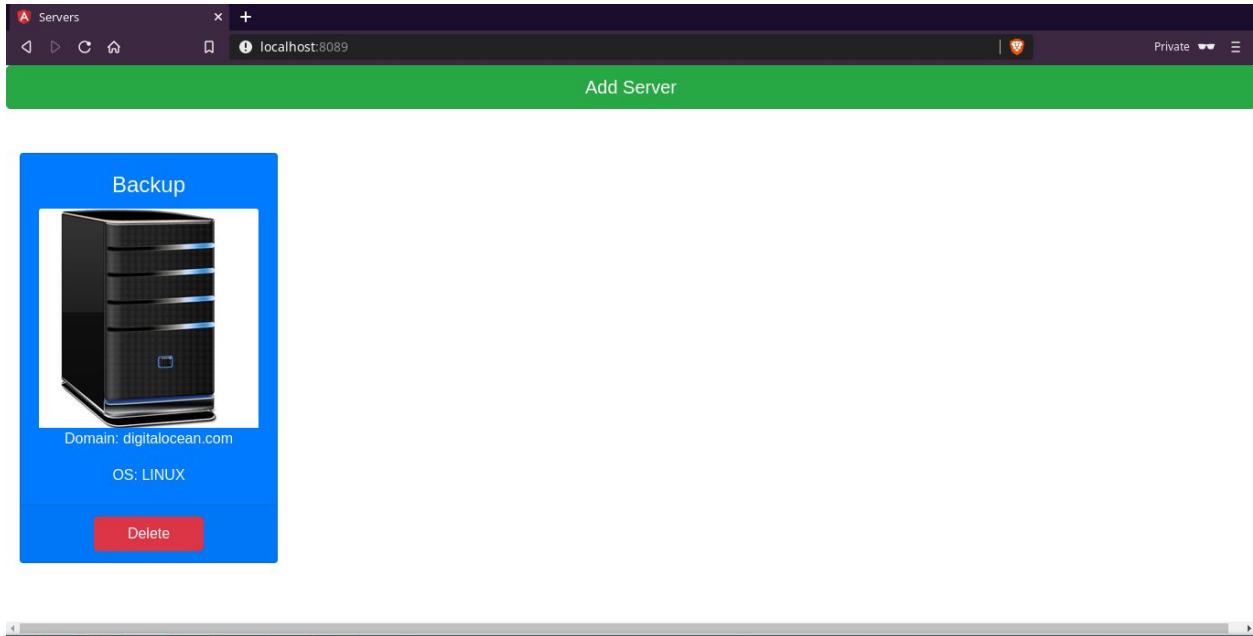


```
first-app: docker — Konsole
File Edit View Bookmarks Settings Help

:: Spring Boot :: (v2.2.6.RELEASE)

2020-04-08 17:05:35.693 INFO --- [main] com.example.demo.DemoApplication : Starting DemoApplication v0.0.1-SNAPSHOT on 3dc0d55ab2e with PID 1 (/app/servers.jar started by root in /app)
2020-04-08 17:05:35.704 INFO --- [main] com.example.demo.DemoApplication : No active profile set, falling back to default profiles: default
2020-04-08 17:05:37.339 INFO --- [main] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JPA repositories in DEFAULT mode.
2020-04-08 17:05:37.484 INFO --- [main] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 130ms. Found 1 JPA repository interfaces.
2020-04-08 17:05:38.988 INFO --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2020-04-08 17:05:39.018 INFO --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2020-04-08 17:05:39.019 INFO --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.33]
2020-04-08 17:05:39.142 INFO --- [main] o.s.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2020-04-08 17:05:39.142 INFO --- [main] o.s.web.context.ContextLoader : Root WebApplicationContext: initialization completed in 3283 ms
2020-04-08 17:05:39.612 INFO --- [main] o.hibernate.jpa.internal.util.LogHelper : HHH000204: Processing PersistenceUnitInfo [name: default]
2020-04-08 17:05:39.976 INFO --- [main] org.hibernate.Version : HHH0000412: Hibernate ORM core version 5.4.12.Final
2020-04-08 17:05:40.248 INFO --- [main] o.hibernate.annotations.common.Version : HCANN000001: Hibernate Commons Annotations {5.1.0.Final}
2020-04-08 17:05:40.428 INFO --- [main] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starting...
2020-04-08 17:05:41.058 INFO --- [main] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start completed.
2020-04-08 17:05:41.100 INFO --- [main] org.hibernate.dialect.Dialect : HHH000400: Using dialect: org.hibernate.dialect.MySQL5Dialect
Hibernate: create table hibernate_sequence (next_val bigint) engine=MyISAM
Hibernate: insert into hibernate_sequence values ( 1 )
Hibernate: create table server (id bigint not null, active bit not null, domain varchar(255) not null, hostname varchar(255) not null, os integer not null, primary key (id)) engine=MyISAM
2020-04-08 17:05:43.143 INFO --- [main] o.h.e.t.j.p.i.JtaPlatformInitiator : HHH000490: Using JtaPlatform implementation: [org.hibernate.engine.transaction.jta.platform.internal.NoJtaPlatform]
2020-04-08 17:05:43.151 INFO --- [main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
2020-04-08 17:05:43.953 WARN --- [main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries may be performed during view rendering. Explicitly configure spring.jpa.open-in-view to disable this warning
2020-04-08 17:05:44.233 INFO --- [main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2020-04-08 17:05:44.402 INFO --- [main] o.s.b.a.w.s.WelcomePageHandlerMapping : Adding welcome page: class path resource [static/index.html]
2020-04-08 17:05:45.377 INFO --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
```

Zatim je potrebno pomoću *browser*-a pristupiti na adresu *localhost:8089* kako pristupili aplikaciji. Slika ispod prikazuje aplikaciju nakon uspešnog dodavanja novog servera.



U zavisnosti od potrebe, nekad je neophodno kreirati više *Dockerfile*-a, odnosno više slika za različite faze razvoja aplikacije. Jedna varijanta jeste da svaki *Dockerfile* se nalazi u zasebnom direktorijumu. Drugi način jeste zadavanje drugačijeg imena/ekstenzije. Primer *Dockerfile.dev*, *Dockerfile.test* itd. Voditi računa prilikom *build*-a, da ne bi došlo do konkretnih problema, odnosno iskoristiti fleg **-f/-file** i zadati naziv konkretnog *Dockerfile*-a.

Napredne stvari:

- [Multi-stage build](#)
- [Kreiranje base slike](#)

## 8. Docker volumes?

U poglavlju u kome su opisivane slike, bilo je reči o *read-only* slojevima i *read-write* sloju koji se dodaju iznad prethodnih slojeva za svaki kontejner koji je pokrenut. Sve promene i sav sadržaj se upisuju u taj sloj. Problem sa tim jeste da kada se kontejner obrisu, promene će biti potpuno izgubljene.

Zato je *Docker* uveo nov koncept pod nazivom *volumes*. Da bi mogli da čuvamo konkretan sadržaj (*persist*), i po potrebi ga delimo između različitih kontejnera, kreiramo poseban *volume* koji je prosto rečeno, ništa drugo do skup direktorijuma/fajlova koji se



nalaze izvan *default*-nog *UFS*-a i koji naravno postoje kao direktorijumi/fajlovi na host fajlsistemu.

Kreiranje *volume*-a je moguće odraditi sa komandom **docker volume create naziv**. *Mount*-ovanje se radi prilikom pokretanja sa flegom **--volume** ili **-v**. Primer: **docker run -i -t -v primer1:/nekiPodaci ubuntu /bin/bash**. Dakle najobičnija komanda (koju smo već videli), proširena flegom **-v** gde smo zadali naziv *volume*-a i gde će biti izvršeno *mount*-ovanje u okviru samog kontejnera.

```
stefan@stefan-MS-7817 ~$ docker volume create primer1
primer1
stefan@stefan-MS-7817 ~$ docker run -i -t -v primer1:/nekiPodaci ubuntu /bin/bash
root@114d6d9edbc4:/# ls
bin boot dev etc home lib lib64 media mnt nekiPodaci opt proc root run sbin srv sys tmp usr var
root@114d6d9edbc4:/# cd nekiPodaci/
root@114d6d9edbc4:/nekiPodaci# touch NekiFajl.txt
root@114d6d9edbc4:/nekiPodaci# ls
NekiFajl.txt
root@114d6d9edbc4:/nekiPodaci# exit
exit
stefan@stefan-MS-7817 ~$ docker run -i -t -v primer1:/nekiPodaci ubuntu /bin/bash
root@36646f39e61d:/# ls
bin boot dev etc home lib lib64 media mnt nekiPodaci opt proc root run sbin srv sys tmp usr var
root@36646f39e61d:/# cd nekiPodaci/
root@36646f39e61d:/nekiPodaci# ls
NekiFajl.txt
root@36646f39e61d:/nekiPodaci#
```

Na slici je prikazano najpre kreiranje *volume*-a, a zatim je pokrenut kontejner kome smo *mount*-ovali prethodno kreirani *volume* na putanji *nekiPodaci*. U okviru prvog kontejnera smo i kreirali običan tekstualni fajl. Zatim smo izvršili *exit* (ugasili glavni proces */bin/bash* i samim tim i ugasili kontejner) i pokrenuli nov kontejner kome smo takođe *mount*-ovali isti *volume* na istoj putanji (apsolutno ne mora biti ista) i kada smo ušli u sam folder, datoteka koju smo prethodno kreirali iz totalno drugog kontejnera i dalje postoji.

## 9. Šta raditi sa ostalim mikroservisima?

U prethodnom poglavlju je objašnjena manipulacija *volume*-a, kako kreirati sopstvenu sliku i kako od nje kreirati kontejner. Međutim, postavlja se pitanje šta raditi ukoliko imamo više aplikacija, od kojih je neke neophodno pokrenuti u više instanci (kontejnera), koji moraju da komuniciraju međusobno. Tada posao pojedinačnog kreiranja slika i pokretanja kontejnera nije baš idealan. Zato se koristi alat *docker-compose* koji nam značajno olakšava stvari po tom pitanju. Omogućuje nam

pokretanje i zaustavljanje *stack*-a aplikacija, kao i zajednički ispis logova svih aplikacija na jedan pseudo terminal.

Sve što je neophodno jeste da se instalira alat (uputstvo dostupno u dokumentaciji <https://docs.docker.com/compose/install/>) i da kreiramo fajl pod nazivom *docker-compose.yml*. Na slici je prikazan deo iz *docker-compose.yml* fajla koji se nalazi unutar *demo* direktorijuma.

```
version: "3.7"
services:

  servers:
    build: ./servers
    image: danijelradakovic/servers:1.0.0
    container_name: servers
    restart: on-failure
    networks:
      - demo
    ports:
      - 8080:8080
    environment:
      DATABASE_USERNAME: sa
      DATABASE_PASSWORD: zgadija
      DATABASE_DOMAIN: mysql
      DATABASE_SCHEMA: servers
      RMQ_HOST: rabbitmq-broker
    depends_on:
      - mysql
      - rabbitmq-broker

  logan:
    build: ./Log-Analyzer
    image: danijelradakovic/logan:1.0.0
    container_name: logan
    restart: on-failure
    networks:
      - demo
    ports:
      - 8081:8080
    environment:
      LOG_STORAGE: /var/log/web-traffic.log
      RMQ_HOST: rabbitmq-broker
    volumes:
      - logs:/var/log
    depends_on:
      - rabbitmq-broker
```

U fajlu za konkretan primer je definisano više direktiva:

1. **version** - Ovde naglašavamo koju verziju formata želimo da koristimo. Ovo polje je uvek neophodno i dovoljno je navesti verziju 3 (poslednja verzija formata).
2. **services** - U ovoj sekciji se definiše niz objekata gde svaki predstavlja servis, odnosno kontejner i takođe ova sekcija je obavezna. Dalje unutar servisa definišemo:
  1. **build** - Ova direktiva ako je definisana, govori da je neophodno kreirati slike pri čemu se definišu odnosno putanja do direktorijuma na kojoj se nalazi Dockerfile.
  2. **image** - Definiše naziv slike koja će nastati prilikom *build*-ovanja.
  3. **container-name** - Definiše naziv kontejnera koji će biti pokrenut.
  4. **restart** - Definiše pod kojim okolnostima kontejner treba restartovati
  5. **networks** - definiše mrežu (mreže) u kojoj kontejner treba da se nalazi.
  6. **ports** - Vršiti se mapiranje portova.
  7. **environments** - Postavlja vrednost environment varijable koje se nalaze u kontejneru.
  8. **volumes** - Definiše volume za koje se kontejner kači.
  9. **depends\_on** - Govori prilikom pokretanja servisa koje su zavisnosti između njih, odnosno koji servisi moraju biti pokrenuti pre nego što se pokrene konkretan servis.

Za dodatne direktive i njihove vrednosti možete pogledati u zvaničnoj dokumentaciji <https://docs.docker.com/compose/>.

Pozicioniramo se na putanju do direktorijuma u kojem se nalazi *docker-compose.yml* i pozovemo naredbu: **docker-compose up --build**. Sa ovim pokrećemo sve naše servise (kontejnere). Rezultat izvršavanja *docker-compose* naredbe nam je u pseudo terminalu spojio logove sa svih pokrenutih servisa.

```

mysql | 2020-04-08T15:59:26.767300Z 0 [Warning] [MY-011010] [Server] Insecure configuration for --pid-file: Location '/var/run/mysqld' in the path is accessible to all OS users. Consider choosing a different directory.
mysql | 2020-04-08T15:59:26.789198Z 0 [System] [MY-010931] [Server] /usr/sbin/mysqld: ready for connections. Version: '8.0.19' socket: '/var/run/mysqld/mysqld.sock' port: 3306 MySQL Community Server - GPL.
mysql | 2020-04-08T15:59:26.913608Z 0 [System] [MY-011323] [Server] X Plugin ready for connections. Socket: '/var/run/mysqld/mysqlx.sock' bind-address: '::: port: 33060
servers
servers
servers
servers
servers
servers
servers
servers
servers
:: Spring Boot :: (v2.2.6.RELEASE)
servers
servers | 2020-04-08 15:59:29.050 INFO 1 --- [main] com.example.demo.DemoApplication : Starting DemoApplication v0.0.1-SNAPSHOT on 1c3f1750fdc2 with PID 1 (/app/servers.jar started by root in /app)
servers | 2020-04-08 15:59:29.055 INFO 1 --- [main] com.example.demo.DemoApplication : No active profile set, falling back to default profile: default
servers | 2020-04-08 15:59:30.172 INFO 1 --- [main] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JPA repositories in DEFAULT mode.
servers | 2020-04-08 15:59:30.260 INFO 1 --- [main] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 75ms. Found 1 JPA repository interfaces.
servers | 2020-04-08 15:59:31.220 INFO 1 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
servers | 2020-04-08 15:59:31.241 INFO 1 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
servers | 2020-04-08 15:59:31.242 INFO 1 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.33]
servers | 2020-04-08 15:59:31.335 INFO 1 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
servers | 2020-04-08 15:59:31.335 INFO 1 --- [main] o.s.web.context.ContextLoader : Root WebApplicationContext: initialization completed in 2167 ms
servers | 2020-04-08 15:59:31.612 INFO 1 --- [main] o.hibernate.jpa.internal.util.LogHelper : HHH0000204: Processing PersistenceUnitInfo [name: default]
servers | 2020-04-08 15:59:31.850 INFO 1 --- [main] org.hibernate.Version : HHH0000412: Hibernate ORM core version 5.4.12.Final
servers | 2020-04-08 15:59:32.085 INFO 1 --- [main] o.hibernate.annotations.common.Version : HCANN000001: Hibernate Commons Annotations {5.1.0.Final}
servers | 2020-04-08 15:59:32.273 INFO 1 --- [main] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starting...
servers | 2020-04-08 15:59:33.167 INFO 1 --- [main] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start completed.

```

## 10. Docker Swarm

*Docker Swarm* je alat koji omogućava orkestraciju nad kontejnerima. *Docker Swarm*-a ima implementiran *load balancer* i *discovery service*, servisi koji su neophodni u mikroservisnoj arhitekturi. Za aktiviranje *Docker Swarm*-a neophodno je podesiti *Docker* da radi u *swarm* režimu komandom: ***docker swarm init***. Za pokretanje prethodnog primera pomoću *Docker Swarm*-a, neophodne je dopuniti određene stvari u *docker-compose.yml* fajlu. Na slici je prikazan deo *stack-file.yml* fajla koji se nalazi u unutar *demo* direktorijuma

```
version: "3.7"
services:

  servers:
    image: danijelradakovic/servers:1.0.0
    networks:
      - demo
    ports:
      - 8080:8080
    environment:
      DATABASE_USERNAME: sa
      DATABASE_PASSWORD: zgadija
      DATABASE_DOMAIN: mysql
      DATABASE_SCHEMA: servers
      RMQ_HOST: rabbitmq-broker
    depends_on:
      - mysql
      - rabbitmq-broker
    deploy:
      replicas: 2
      update_config:
        parallelism: 2
        delay: 10s
      restart_policy:
        condition: on-failure

  logan:
    image: danijelradakovic/logan:1.0.0
    networks:
      - demo
    ports:
      - 8081:8080
    environment:
      LOG_STORAGE: /var/log/web-traffic.log
      RMQ_HOST: rabbitmq-broker
    volumes:
      - logs:/var/log
    depends_on:
      - rabbitmq-broker
    deploy:
      replicas: 2
      update_config:
        parallelism: 2
        delay: 10s
      restart_policy:
        condition: on-failure
```

U fajlu je dodata *deploy* sekcija koja govori kako treba da se uradi *deployment* servisa:

1. **replicas** - koliko instanci kontejnera treba da bude aktivno
2. **parallelism** i **delay** - ukoliko broj aktivnih kontejnera manji od specificiranog, *Docker* pokreće nove instance kontejnera dok ne postigne željeni broj. Ova sekcija definiše kako je to potrebno postići. Na primer ukoliko definišemo *replicas: 10*, *parallelism:2* i *delay: 10s*, u slučaju da su pali svih 10 kontejnera, *Docker* će istovremeno podizati 2 kontejnera pri čemu će nakog uspešnog podizanja oba kontejnera sačekati 10 sekundi i opet pokušavati da podigne istovremeno 2 kontejnera. Ovaj proces se ponavlja sve dok se ne postigne željeni broj instanci kontejnera.
3. **restart\_policy** - definiše pod kojim okolnostima je neophodno podizati nove kontejnere.

Pokretanje se vrši pomoću naredne komande (*demo* predstavlja naziv *stack*-a koji može biti proizvoljan, dok **-c** flagom definišemo putanju do *yml* fajla):

**docker stack deploy -c stack-file.yml demo**

Za praćenje stanja servisa neophodne je izvršiti narednu komandu:

**docker stack services demo**