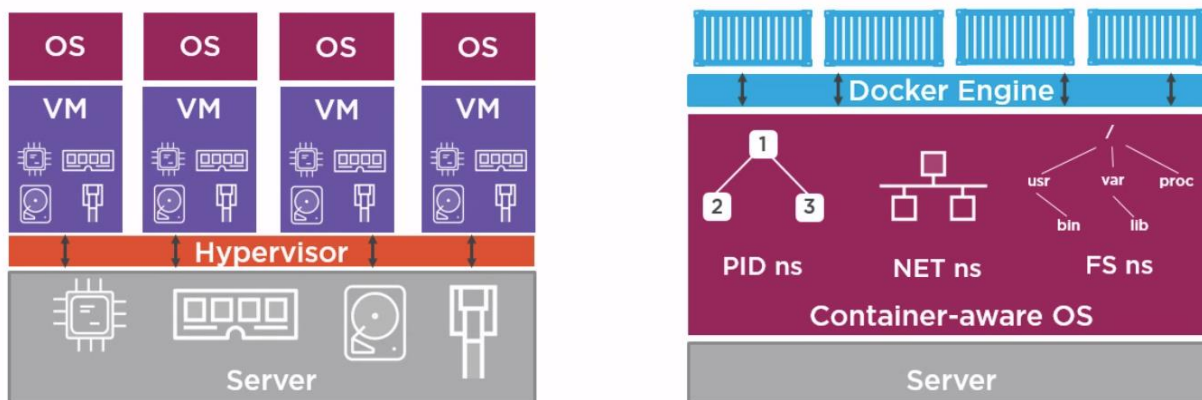


1. Motivacija

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 što je *Docker* nastojao i uspeo da promeni.

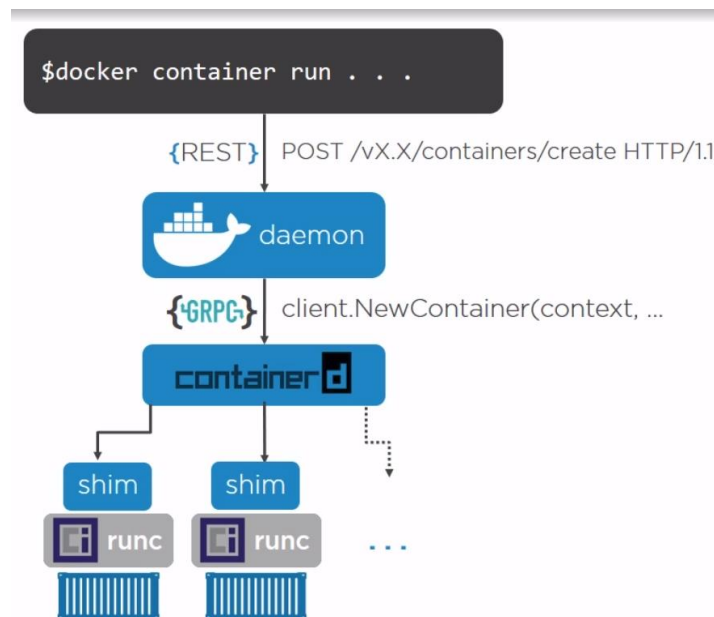
2. Šta je Docker i koje su njegove komponente?

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*,
4. *Docker containers*.

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

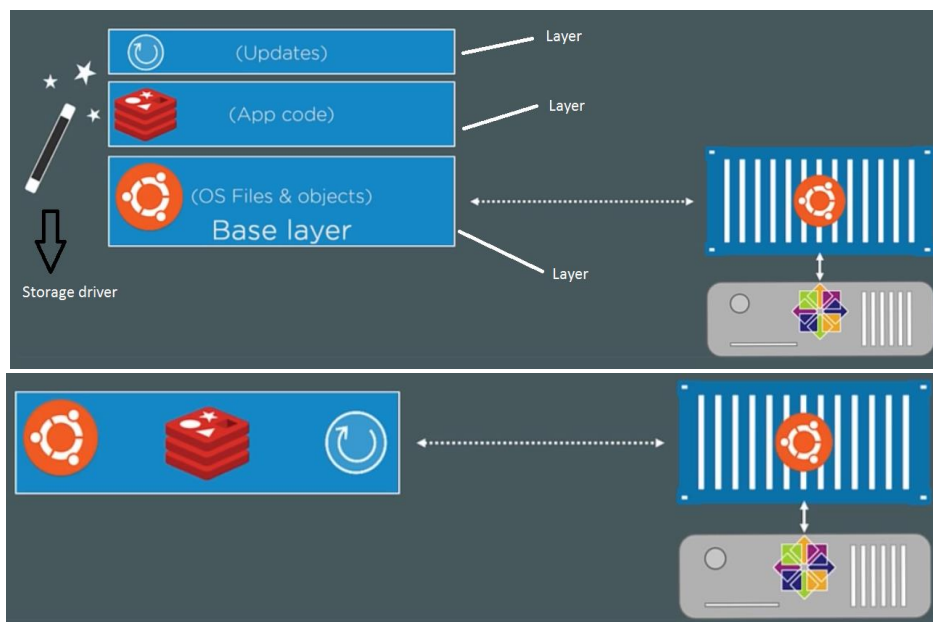
Sam *Docker daemon* je nakon refaktorisanja (zbog toga što narastao u jedan veliki monolit) ostao bez ikakvog koda koji zaista kreira i pokreće kontejnere. On se obraća putem gRPC API-a preko lokalnog linux socket-a *containerd-u (long running daemon-u)* koji predstavlja API „fasadu” koja omogućuje startovanje *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 (npr. moguće je dosta lakše održati up-time, pošto kontejneri ne moraju da se restartuju u slučaju izmene neke komponente). Kod *Windows*-a, arhitektura izgleda malo drugačije, ali je poenta ista.

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

3. Šta su Docker slike?

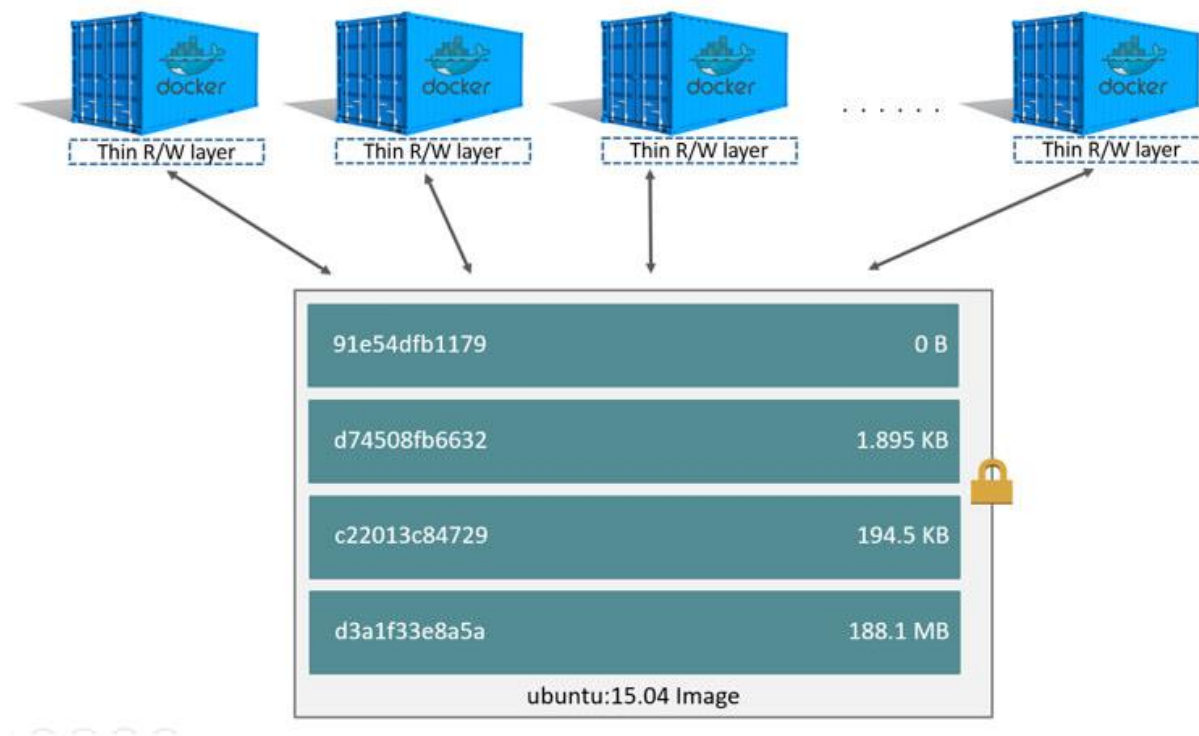
Generalno je poznat koncept slike kada je priča o virtuelnim mašinama. Za sličnu stvar se koriste i *Docker slike*, odnosno predstavljaju *build-time* konstrukt od kojih nastaju kontejneri ali se tu sličnost završava. *Docker slike* predstavljaju skup *read-only layer*-a, gde svaki sloj predstavlja različitosti 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 „bazu” za svaki kontejner koji se pokreće, i kao što im ime kaže, ne mogu se menjati. Prilikom startovanja svakog kontejnera, *Docker mount*-uje još jedan sloj koji je *read-write* tipa i u koji se upisuju nove datoteke i sve izmene. Kod izmena, 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

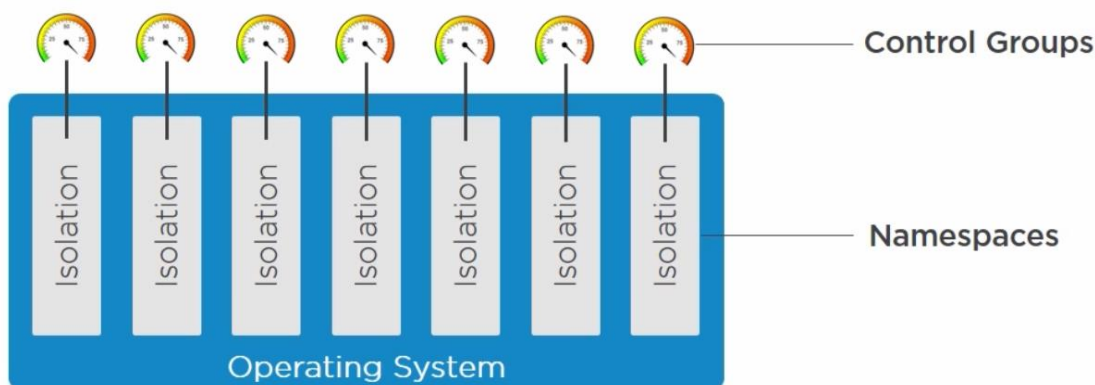
Ovakav mehanizam se zove *Copy-on-write* i delom čini *Docker* zaista moćnim. Koliko god kontejnera da kreiramo, *read-only* slojevi će uvek biti isti, tj. ostaće nepromenjeni, samo će svaki kontejner dobiti sopstveni *read-write* sloj. Na ovaj način se štedi jako puno prostora na disku, jer kada smo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

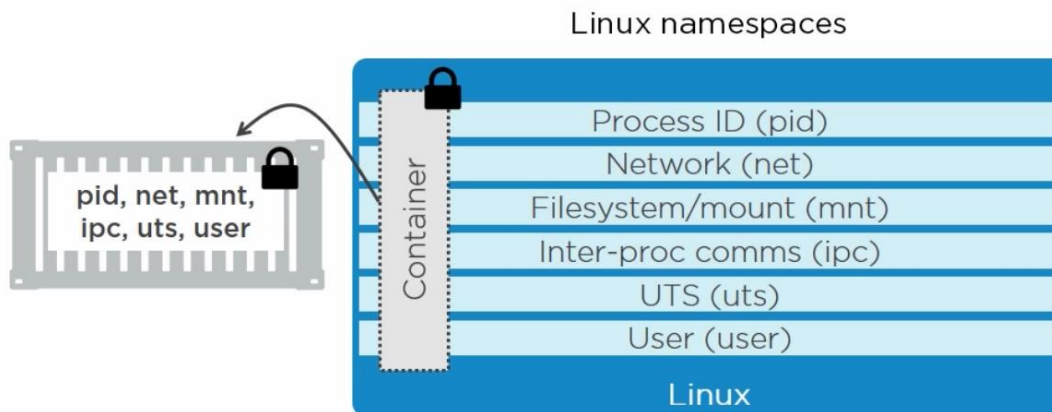
5. Šta predstavljaju kontejneri?

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

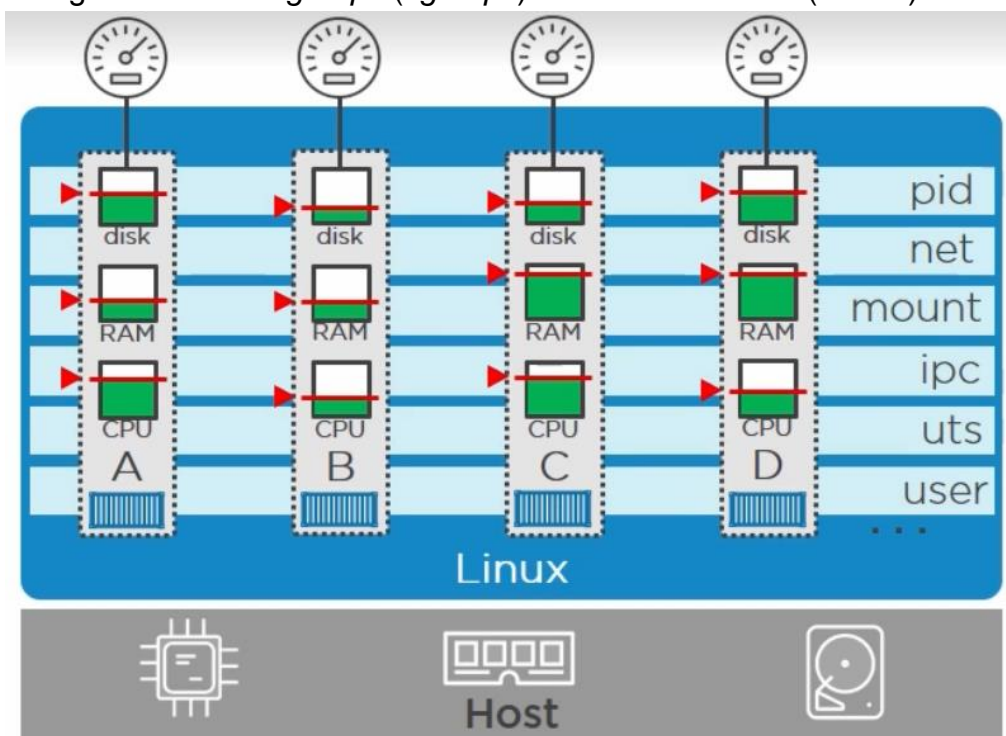
Namespaces nam omogućuju izolaciju, odnosno da podelimo naš operativni sistem na manje izolovanih virtuelnih operativnih sistema (kontejnera). Odnosno kontejneri *smells-and-feels* kao zasebni operativni sistemi (kao slučaj kod VM-a), samo što to nisu, jer svi dele isti kernel na host OS-u. Svaki kontejner ima sopstevni 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 sve kako treba. 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: 09c8266bf2fc9519a651b04ae54c967b9ab86ec  
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
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
File Edit View Search Terminal Help
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
File Edit View Search Terminal Help
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 je 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. **MAINTAINER** - Podešavanje punog imena i prezimena, kao i *email-a* kreatora slike.

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/>.

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 *containerized-discovery Spring-boot* aplikaciji iz materijala. Konkretna aplikacija predstavlja *Eureka service registry* koji nam u mikroservis arhitekturi omogućava *load-balancing* bez hardkodiranja hostova i portova.

```
Dockerfile x
1 FROM anapsix/alpine-java
2 MAINTAINER Stefan Colic <stefan.colic4@gmail.com>
3 ADD target/containerized-discovery-0.0.1-SNAPSHOT.jar containerized-discovery.jar
4 ENTRYPOINT ["java", "-jar", "/containerized-discovery.jar"]
5 EXPOSE 8761
6
```

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

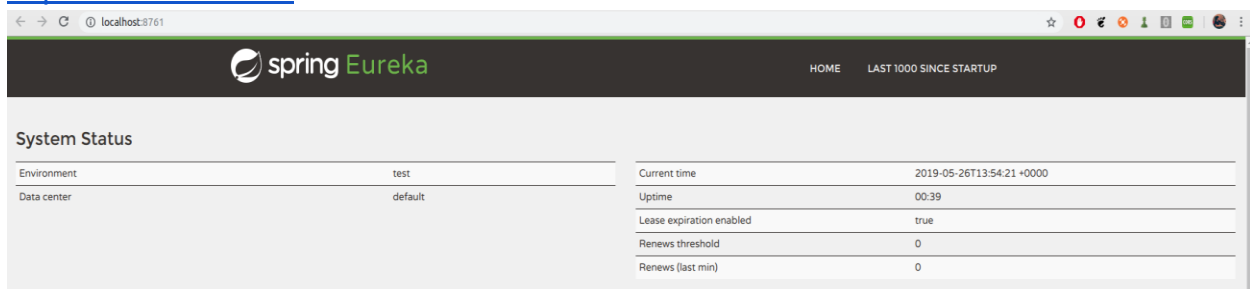
1. Sa prvom instrukcijom smo rekli šta je bazna slika. U ovom slučaju smo izabrali neoficijelnu sliku (za ozbiljne projekte uvek koristiti oficijelne slike) koja se oslanja na *Alpine-linux (ultralightweight)* koji dolazi sa instaliranim *jdk*-om.
2. Sa drugom instrukcijom smo zadali metapodatke koje se odnose na kreatora i održavaoca slike.
3. ADD instrukcija vrši kopiranje .jar fajla i njegovo preimenovanje.
4. Sa ENTRYPOINT instrukcijom navodimo šta će biti *executable* i šta će biti pokrenuto sa samim pokretanjem kontejnera.
5. I expose-ujemo još port na kom aplikacija sluša unutar kontejnera.

Kada smo kreirali *Dockerfile*, pozicioniramo se u korenski direktorijum konkretne aplikacije i izvršimo komandu **docker image build -t discovery-service ..** 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.

```
stefan@stefan-M5-7817: ~/spring-boot-microservice-eureka-zuul-docker-master/containerized-discovery
File Edit View Search Terminal Help
stefan@stefan-M5-7817: ~/spring-boot-microservice-eureka-zuul-docker-master/containerized-discovery docker image build -t discovery-service .
Sending build context to Docker daemon 44.96MB
Step 1/5 : FROM anapsix/alpine-java
latest: Pulling from anapsix/alpine-java
169185f82c45: Pull complete
1e929b64ace7: Pull complete
Digest: sha256:1d24bc352e07b84c073acfff8bf913c213d1cfc73cdf876b181d714870968819
Status: Downloaded newer image for anapsix/alpine-java:latest
----> c45785c254c5
Step 2/5 : MAINTAINER Stefan Colic <stefan.colic4@gmail.com>
----> Running in 66fc67e0a700
Removing intermediate container 66fc67e0a700
----> 3bd06ba06c7d
Step 3/5 : ADD target/containerized-discovery-0.0.1-SNAPSHOT.jar containerized-discovery.jar
----> 95dbe60312ce
Step 4/5 : ENTRYPOINT ["java", "-jar", "/containerized-discovery.jar"]
----> Running in b6274a2c7dc3
Removing intermediate container b6274a2c7dc3
----> 343162feb364
Step 5/5 : EXPOSE 8761
----> Running in 140016093fa4
Removing intermediate container 140016093fa4
----> 9175e6ed6dca
Successfully built 9175e6ed6dca
Successfully tagged discovery-service:latest
stefan@stefan-M5-7817: ~/spring-boot-microservice-eureka-zuul-docker-master/containerized-discovery
```

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

Ostalo nam je još da pokrenemo kontejner, što činimo sa sledećom komandom: **docker run --detach -p 8761:8761 discovery-service:latest**. Sa **--detach** flegom kažemo da se kontejner izvršava u pozadini, bez da imamo bilo kakav prikaz i ispis u našem terminalu. Sa **-p** flegom kažemo da mapiramo port iz kontejnera na port na hostu. Ukoliko je sve prošlo kako treba, aplikacija bi trebala da bude dostupna na <http://localhost:8761>.



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.

8. Šta raditi sa ostalim mikroservisima?

U prethodnom poglavlju je objašnjeno 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 najidealniji. 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 instaliramo 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 primera fajla iz materijala.

```
docker-compose.yml
1  version: '3'
2  services:
3
4    discovery:
5      image: service-discovery
6      container_name: service-discovery
7      build:
8        context: ./containerized-discovery
9        dockerfile: Dockerfile
10     ports:
11       - "8761:8761"
12
13   gateway:
14     image: service-gateway
15     container_name: service-gateway
16     build:
17       context: ./containerized-gateway
18       dockerfile: Dockerfile
19     ports:
20       - "8762:8762"
21     depends_on:
22       - discovery
23     links:
24       - discovery:discovery
```


U .yaml 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. **image** - Ova direktiva govori kako da se nazove slika nakon što se kreira.
 2. **container-name** - Dodeljujemo naziv kontejneru koji će biti pokrenut.
 3. **build** - Ova direktiva ako je definisana, govori da je neophodno kreirati slike pri čemu se definišu **build-context**, odnosno putanja na kojoj se nalazi Dockerfile i **dockerfile**, odnosno setuje se alternativni naziv za *Dockerfile*.
 4. **ports** - Vršiti se mapiranje portova.
 5. **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/>.

Kada smo kreirali *docker-compose.yaml*, pozicioniramo se na putanju fajla i pozovemo naredbu: **docker-compose up --build --scale service1=3 --scale service2=3**

Sa ovim pokrećemo sve naše servise (kontejnere), pri čemu smo eksplicitno naglasili da *service1* želimo da pokrenemo u 3 instanci, gde će svih 3 biti registrovano u Eureka servisu (isto i za *service2*).

HOME LAST 1000 SINCE STARTUP

System Status

Environment	test	Current time	2019-06-01T05:23:32 +0000
Data center	default	Uptime	00:01
		Lease expiration enabled	true
		Renews threshold	13
		Renews (last min)	0

THE SELF PRESERVATION MODE IS TURNED OFF.THIS MAY NOT PROTECT INSTANCE EXPIRY IN CASE OF NETWORK/OTHER PROBLEMS.

DS Replicas

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CONTAINERIZED-GATEWAY	n/a (1)	(1)	UP (1) - 7b7fe46cc5aa:containerized-gateway:8762
CONTAINERIZED-SERVICE1	n/a (3)	(3)	UP (3) - 3f1701fbc55:containerized-service1:2222 , 40b7f6591bce:containerized-service1:2222 , 3fa2992ed1b6:containerized-service1:2222
CONTAINERIZED-SERVICE2	n/a (3)	(3)	UP (3) - 3d181f004477:containerized-service2:2222 , 9a6003e403ff:containerized-service2:2222 , e464fb54aaa5:containerized-service2:2222

Rezultat izvršavanja *docker-compose* naredbe nam je u pseudo terminalu spojio logove sa svih pokrenutih servisa.


```
File Edit View Search Terminal Help
docker-compose up --build --scale service1=3 --scale service2=3
service-discovery | 2019-06-01 05:23:19.717 INFO 1 --- [nio-8761-exec-1] c.n.e.registry.AbstractInstanceRegistry : Registered instance CONTAINERIZED-SERVICE2/e464fb54aaa5:containerized-service2:2222 with status UP (replication=false)
service2_3 | 2019-06-01 05:23:19.719 INFO 1 --- [infoReplicator-0] com.netflix.discovery.DiscoveryClient : DiscoveryClient_CONTAINERIZED-SERVICE2/e464fb54aaa5:containerized-service2:2222 - registration status: 204
service2_3 | 2019-06-01 05:23:19.722 INFO 1 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 2222 (http) with context path ''
service2_3 | 2019-06-01 05:23:19.723 INFO 1 --- [main] .s.c.n.e.s.EurekaAutoServiceRegistration : Updating port to 2222
service2_3 | 2019-06-01 05:23:19.727 INFO 1 --- [main] a.u.f.s.ContainerizedService2Application : Started ContainerizedService2Application in 21.853 seconds (JVM running for 24.924)
service1_3 | 2019-06-01 05:23:22.177 INFO 1 --- [freshExecutor-0] com.netflix.discovery.DiscoveryClient : Disable delta property : false
service1_3 | 2019-06-01 05:23:22.177 INFO 1 --- [freshExecutor-0] com.netflix.discovery.DiscoveryClient : Single vip registry refresh property : null
service1_3 | 2019-06-01 05:23:22.178 INFO 1 --- [freshExecutor-0] com.netflix.discovery.DiscoveryClient : Force full registry fetch : false
service1_3 | 2019-06-01 05:23:22.178 INFO 1 --- [freshExecutor-0] com.netflix.discovery.DiscoveryClient : Application is null : false
service1_3 | 2019-06-01 05:23:22.178 INFO 1 --- [freshExecutor-0] com.netflix.discovery.DiscoveryClient : Registered Applications size is zero : true
service1_3 | 2019-06-01 05:23:22.178 INFO 1 --- [freshExecutor-0] com.netflix.discovery.DiscoveryClient : Application version is -1: true
service1_3 | 2019-06-01 05:23:22.179 INFO 1 --- [freshExecutor-0] com.netflix.discovery.DiscoveryClient : Getting all instance registry info from the eureka server
service1_3 | 2019-06-01 05:23:22.253 INFO 1 --- [freshExecutor-0] com.netflix.discovery.DiscoveryClient : The response status is 200
service1_2 | 2019-06-01 05:23:23.530 INFO 1 --- [freshExecutor-0] com.netflix.discovery.DiscoveryClient : Disable delta property : false
service1_2 | 2019-06-01 05:23:23.530 INFO 1 --- [freshExecutor-0] com.netflix.discovery.DiscoveryClient : Single vip registry refresh property : null
service1_2 | 2019-06-01 05:23:23.530 INFO 1 --- [freshExecutor-0] com.netflix.discovery.DiscoveryClient : Force full registry fetch : false
service1_2 | 2019-06-01 05:23:23.531 INFO 1 --- [freshExecutor-0] com.netflix.discovery.DiscoveryClient : Application is null : false
service1_2 | 2019-06-01 05:23:23.531 INFO 1 --- [freshExecutor-0] com.netflix.discovery.DiscoveryClient : Registered Applications size is zero : true
service1_2 | 2019-06-01 05:23:23.531 INFO 1 --- [freshExecutor-0] com.netflix.discovery.DiscoveryClient : Application version is -1: true
service1_2 | 2019-06-01 05:23:23.531 INFO 1 --- [freshExecutor-0] com.netflix.discovery.DiscoveryClient : Getting all instance registry info from the eureka server
service1_2 | 2019-06-01 05:23:23.600 INFO 1 --- [freshExecutor-0] com.netflix.discovery.DiscoveryClient : The response status is 200
service1_1 | 2019-06-01 05:23:23.658 INFO 1 --- [freshExecutor-0] com.netflix.discovery.DiscoveryClient : Disable delta property : false
service1_1 | 2019-06-01 05:23:23.658 INFO 1 --- [freshExecutor-0] com.netflix.discovery.DiscoveryClient : Single vip registry refresh property : null
service1_1 | 2019-06-01 05:23:23.658 INFO 1 --- [freshExecutor-0] com.netflix.discovery.DiscoveryClient : Force full registry fetch : false
service1_1 | 2019-06-01 05:23:23.659 INFO 1 --- [freshExecutor-0] com.netflix.discovery.DiscoveryClient : Application is null : false
service1_1 | 2019-06-01 05:23:23.659 INFO 1 --- [freshExecutor-0] com.netflix.discovery.DiscoveryClient : Registered Applications size is zero : true
service1_1 | 2019-06-01 05:23:23.659 INFO 1 --- [freshExecutor-0] com.netflix.discovery.DiscoveryClient : Application version is -1: true
service1_1 | 2019-06-01 05:23:23.659 INFO 1 --- [freshExecutor-0] com.netflix.discovery.DiscoveryClient : Getting all instance registry info from the eureka server
service1_1 | 2019-06-01 05:23:23.729 INFO 1 --- [freshExecutor-0] com.netflix.discovery.DiscoveryClient : The response status is 200
service-discovery | 2019-06-01 05:23:54.198 INFO 1 --- [a-EvictionTimer] c.n.e.registry.AbstractInstanceRegistry : Running the evict task with compensationTime 0ms
service-discovery | 2019-06-01 05:24:54.199 INFO 1 --- [a-EvictionTimer] c.n.e.registry.AbstractInstanceRegistry : Running the evict task with compensationTime 0ms
^[[3~
```

Ukoliko budemo gađali `service1_X` (<http://localhost:8762/service1/api/service1/hello>) kroz *gateway*, prilikom svakog gađanja zahtev će otići skroz drugoj instanci, odnosno *load-balancer* (*Ribbon*) će zahteve prosleđivati naizmenično pokrenutim instancama, a za adrese gde se same instance nalaze, *gateway* će kontaktirati Eureka servis i dobiti realne adrese na kojima su te instance pokrenute.

9. Docker volumes?

U poglavlju u kome su opisivane slike, bilo je reči o *read-only* slojevima i *read-write* sloju koji se *mount*-uje 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 prekine (što je operacija koju ćete često raditi), 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.