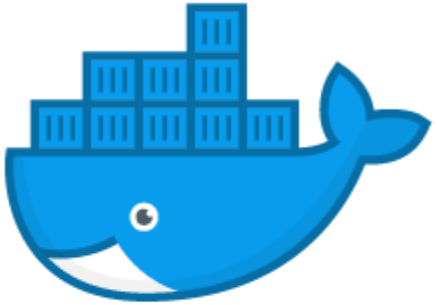


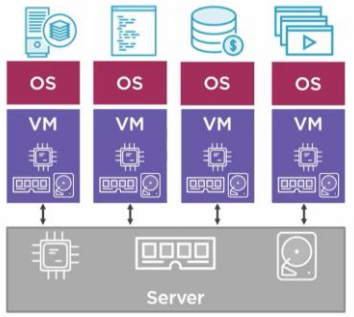
# docker

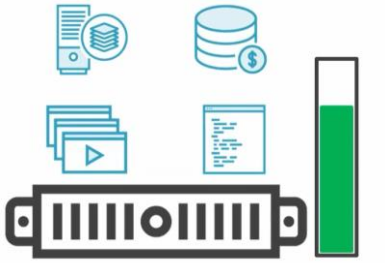
Hönnun og smíði hugbúnaðar

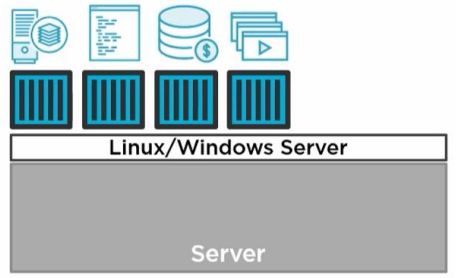
Haust 2022

# The Bad Old Days

- The old old days
  - Einn server fyrir eitt app
  - Server-ar voru(og eru dýrir)
  - Server-ar voru oft massively overpowered
  - Allt „on-premise“ það voru engar skýjaþjónustur
- The old days
  - Hægt að keyra mörg application á server með VMs
  - En VMs eru resource þung
    - Þurfa sitt eigið stýrikerfi
    - Þurfa sitt eigið virtual RAM, virtual disk, virtual CPU etc.
    - Miklir resources fara í hjúpunina
- The new days
  - Container tæknin kom fram
  - Loksins hægt að keyra mörg application, hvert í sínu isolated umhverfi, á einum server í einu stýrikerfi
  - Fáum kostina við VMs án gallana
  - Skýja þjónustur og container orchestration tól gera auðvelt að maintain-a, deploy-a og scale-a öppin okkar



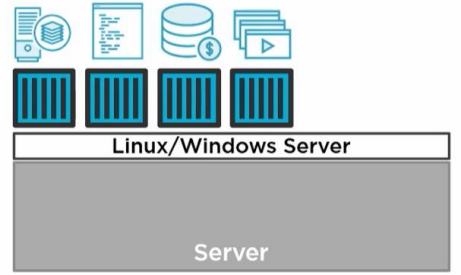


# Hvað er Container?

- Standardized unit of software
  - Container pakkar saman kóða og öllum hans dependency-um
  - Hægt að keyra app hratt og áreiðanlega frá einu umhverfi yfir í annað
  - Hjúpun á keyrslu forrits

- Leysir „but it works on my machine“ vandámálið

- Many sizes
  - Allt frá litlum microservices til stór monoliths




- VM 2.0
  - Ákveðin gerð af virtualization en ekki full blown machine virtualization
  - Innihalda ekki sitt eigið stýrikerfi -> meira lightweight og portable en VMs
  - Er einangrað svæði af stýrikerfinu með resource usage limitations
  - Basically Virtual Operating System með sitt eigið root file system, process tré, root user etc. en deila öll sama undirliggjandi OS kernel

# Hvað er Container?

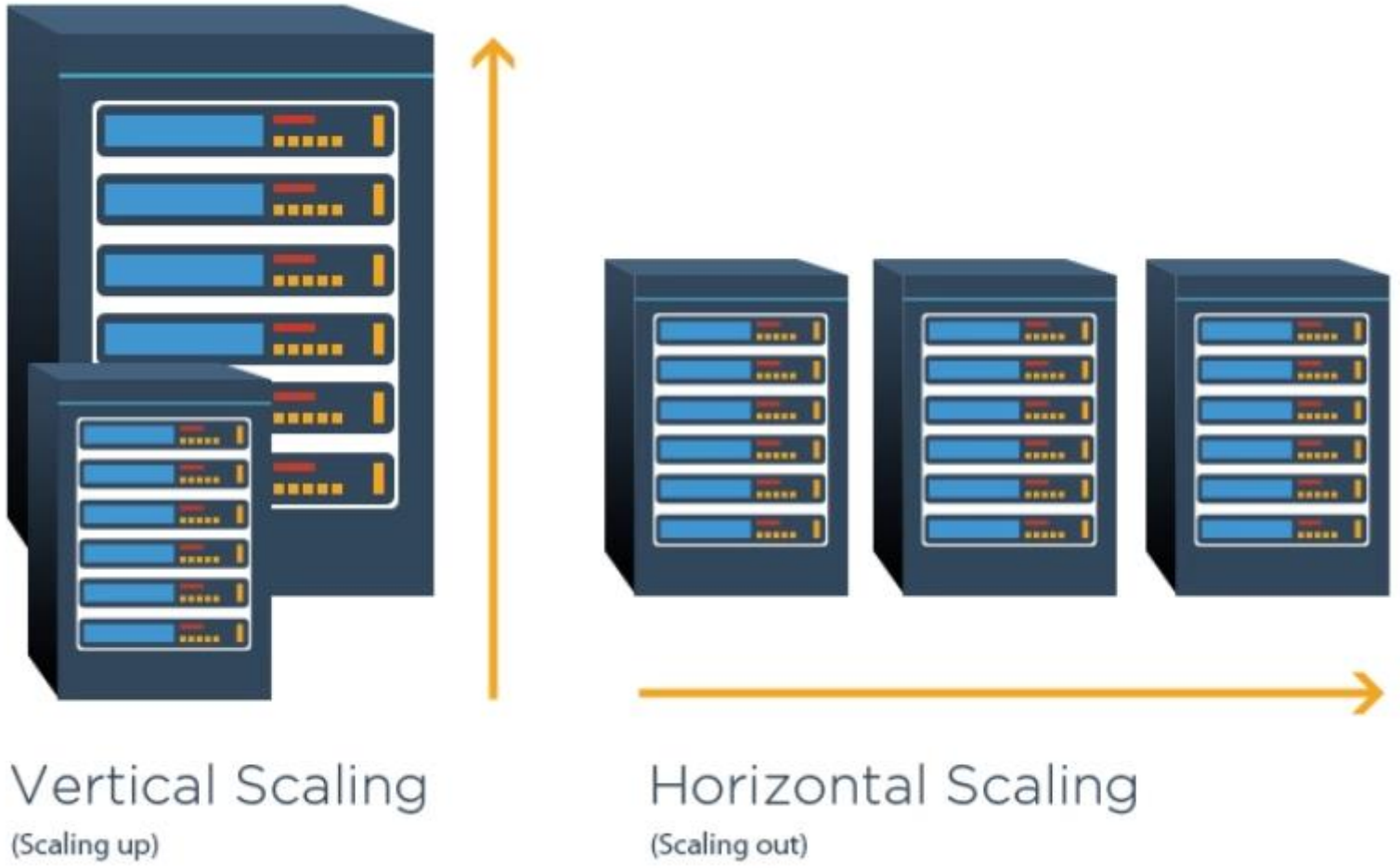
- Hægt að stjórna með container orchestration tólum
  - T.d. Kubernetes

- Margir container-ar geta keyrt á einni vél
  - Alveg eins og með VMs nema geta verið mun fleiri
  - Hver container er isolated frá hvor öðrum

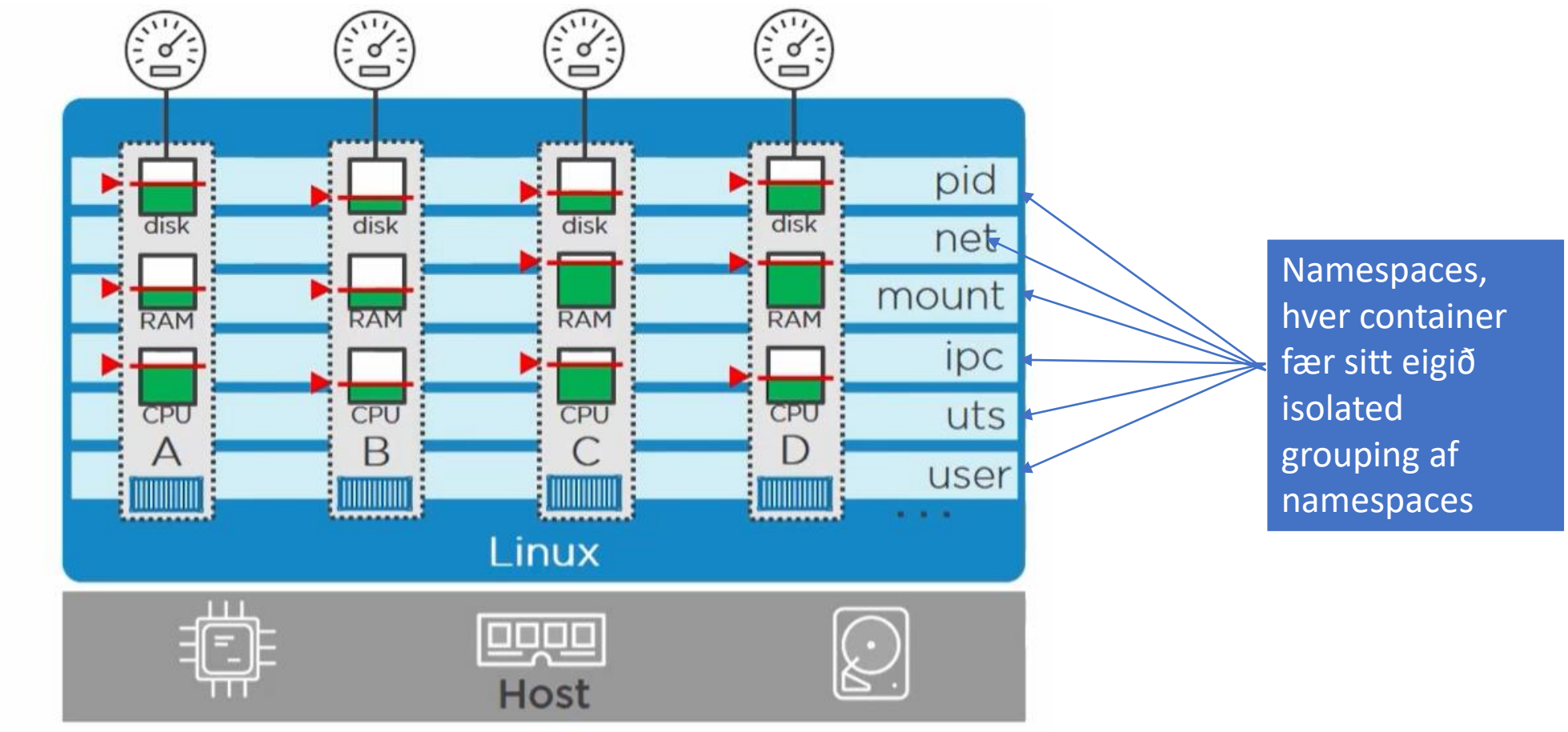
- Container ætti eingöngu að hafa eitt concern
  - Við ættum að decouple-a applications í marga container-a sem hjálpar við horizontal scaling, maintainability, og reuse
  - T.d.  Gæti web application samanstaðið af tveimur containerum -> container fyrir framenda og container fyrir bakenda

# Container kostir

- Minna overhead
  - Container-ar þurfa minna af system resources en traditional VMs
  - Fljótari að start-a upp og stoppa
  - Hægt að hafa mun fleiri container-a keyrandi á einni vél en virtual machines (Mbs vs Gbs)
- Meira portability
  - Containerized applications geta verið deploy-uð auðveldlega á mörg mismunandi stýrikerfi og platform
- Meira consistent operation
  - Þú veist að container-inn mun keyra og að það muni keyra sama hvar það er deploy-að.
  - Leysir “but it works on my machine” vandamálið
- Meira efficiency
  - Hjálpa við hröð deployment
- Scaling
  - Getur haft marga container-a keyrandi fyrir sama application. **(horizontal scaling)**
- Betra development umhverfi
  - Styðja vel við agile og DevOps tækni t.d. hvað varðar testing CI/CD, deployment etc.

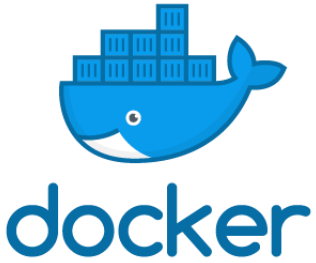

Vertical Scaling
(Scaling up)
Horizontal Scaling
(Scaling out)

Control groups segja okkur t.d. Að container A eigi bara að fá x mikið disk pláss, y mikið RAM, z mikið af CPU-inum etc.



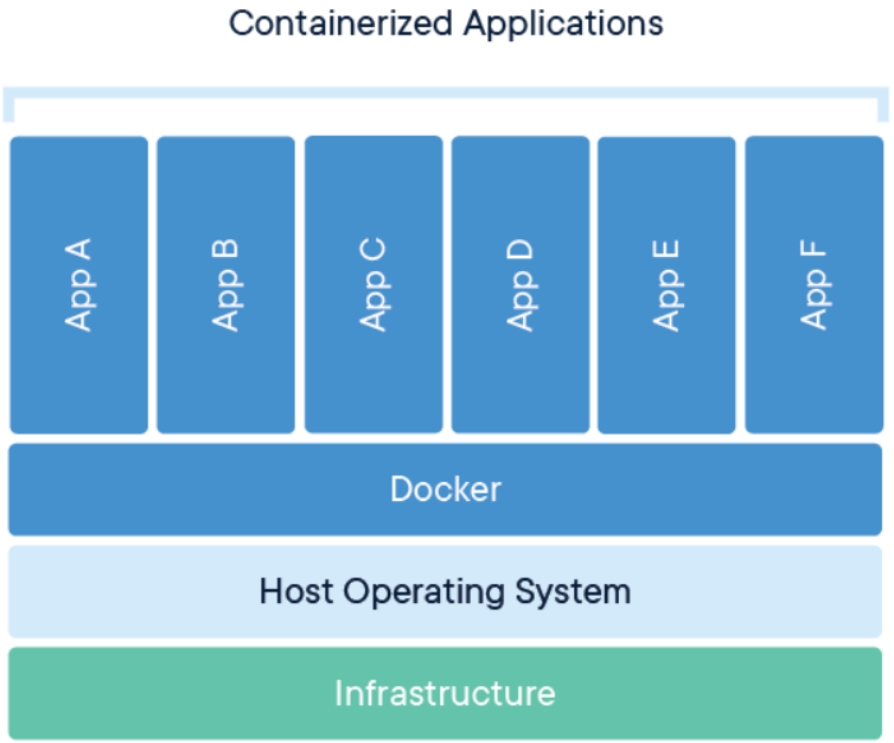

Namespaces, hver container fær sitt eigið isolated grouping af namespaces

# Hvað er Docker?





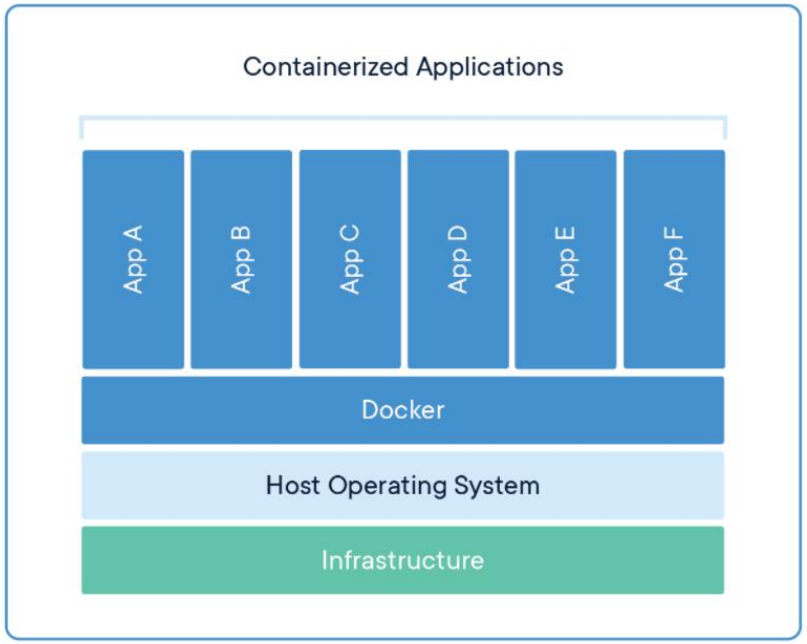
- Docker er Industry standard tól-ið til að búa til og manage-a container-a
- Docker sér um low-level kernel virknina ti að búa til container-a (control groups, namespaces etc. etc.)
- Með Docker getum við auðveldlega keyrt marga container-a samtímis á einni tölvu í einu stýrikerfi
- Docker býður upp á tól til að manage-a container-ana og lifecycle-ið á þeim
- Til að búa til container-a með docker er fyrst búið til docker image sem er ákveðið blueprint eða template fyrir container-a með upplýsingar um hvernig á að búa til Docker container-ana og síðan er notað það image til að búa til/spin-a upp nýju tilviki af því image-i í formi containers
- Býður upp á community og enterprise edition
- Til fyrir Windows, Mac og Linux
- Það er til heilt ecosystem í kringum docker

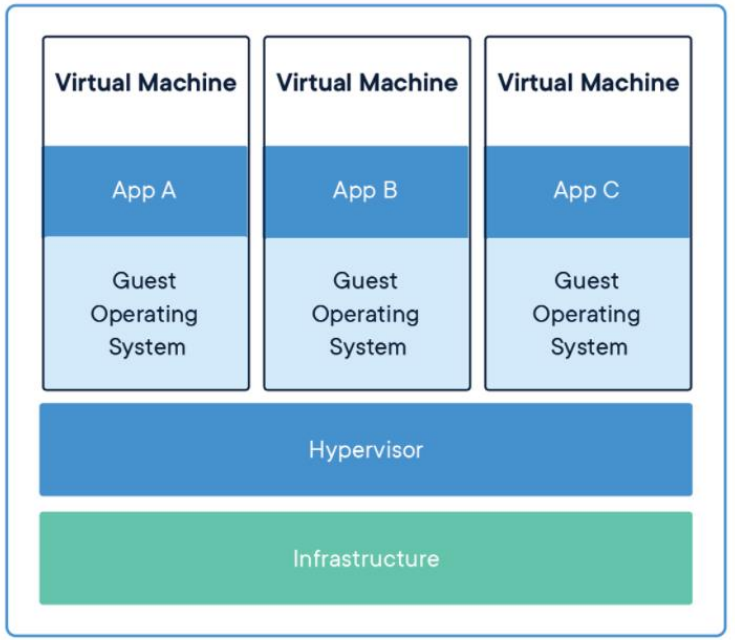


# Virtual Machines vs Docker Containers

Hafa svipað motivation og resource isolation en virka öðruvísi því virtual machines virtualize-a hardware en container-ar virtulize-a stýrikerfið sem gerir container-a meira portable, lightweight og efficient

Container-ar eru abstraction á app layer-inu sem pakkar saman kóða og dependency-um saman. Margir container-ar geta keyrt á sömu vél og deilt stýrikerfi með öðrum container-um, þar sem hver container keyrir í isolated process. Container-ar taka minna pláss en VMs og geta höndlað fleiri applications. Container-ar eru yfirleitt tugir MBs og eru mjög fljótir að start-a upp




Virtrual Machines(VMs) eru abstraction á physical hardware-i, breytandi einum server í marga server-a. Hypervisor-inn leyfir mörgum VMs að keyra á einni vél. Hvert VM inniheldur fullt copy af stýrikerfi, application-inu og öllum dependency-um sem tekur upp tugi GBs og geta verið hæg að start-a upp

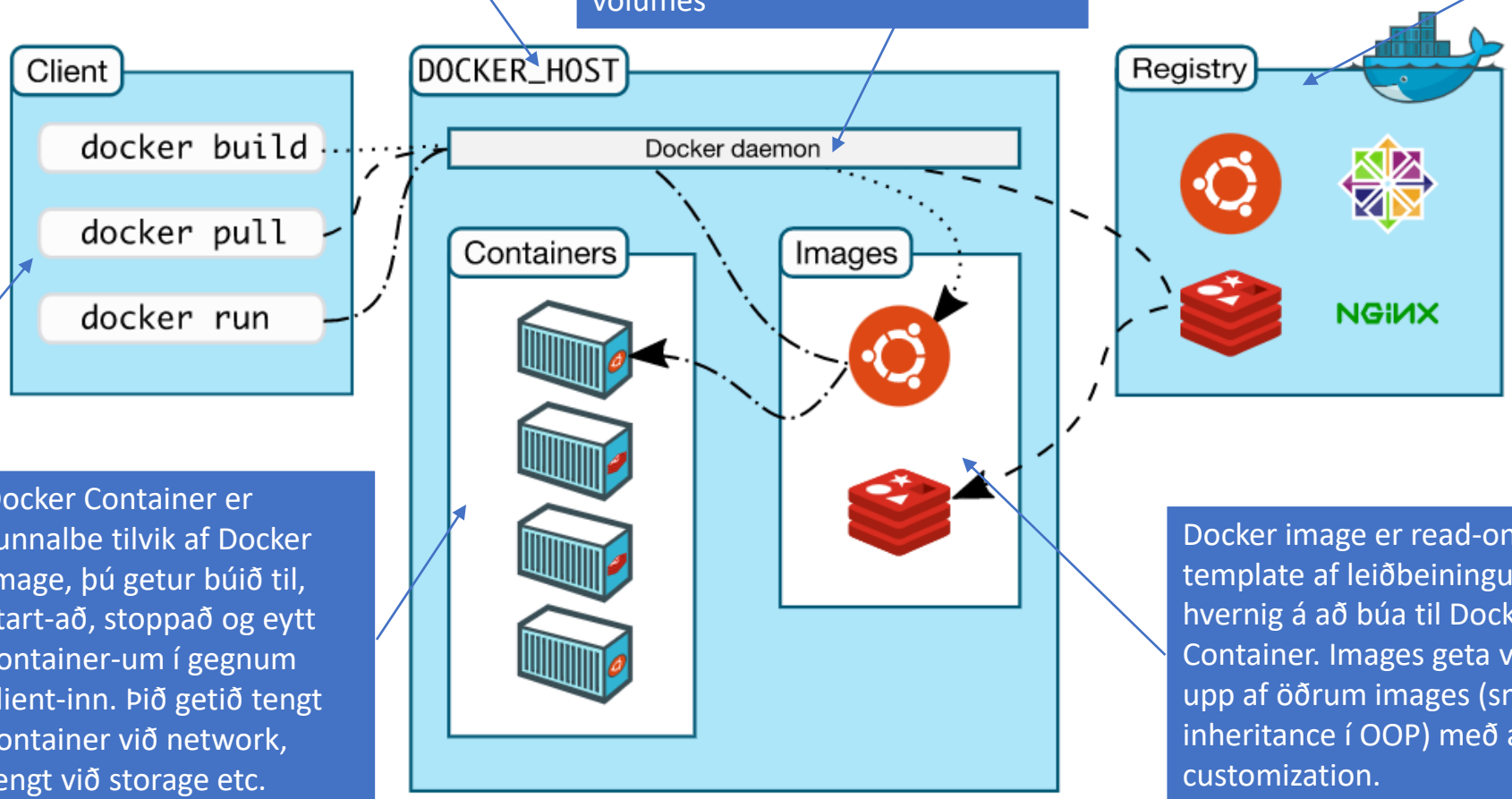
# Docker Architecture

Docker daemon-inn og client-inn geta keyrt í sama kerfi eða client-inn getur talað við daemon-inn remotely.

Docker daemon-inn sér um allt heavy lifting eins og að build-a og keyra container-ana sem og að manage-a docker hluti eins og images, container-a networks og volumes

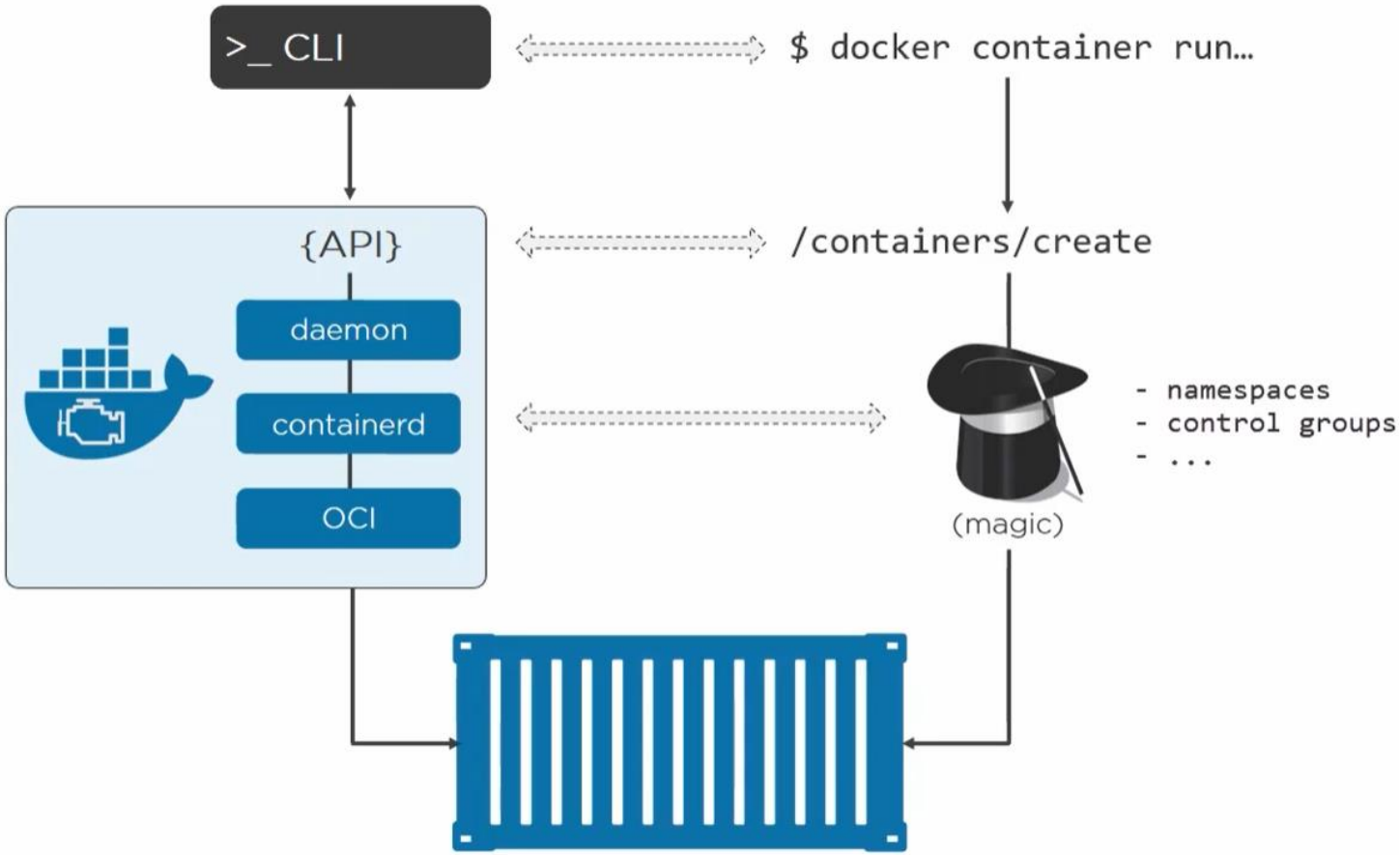
Docker registry-ið geymir Docker images (hugsiði github fyrir Docker images). Docker Hub er default registry-ið en það er hægt að hafa private registry

Docker client-inn talar við Docker Daemon-inn og sendir request-ur á hann um að t.d. Pull-a docker images og build-a og keyra container-a frá þeim image-um, þið hafið *samskipti* við Docker í gegnum client-inn, getið gert þetta í gegnum terminal-inn eða í gegnum Docker desktop GUI tólið




Docker Container er runnalbe tilvik af Docker Image, þú getur búið til, start-að, stoppað og eytt container-um í gegnum client-inn. Þið getið tengt container við network, tengt við storage etc.

Docker image er read-only template af leiðbeiningum um hvernig á að búa til Docker Container. Images geta verið byggð upp af öðrum images (smá eins og inheritance í OOP) með auka customization.

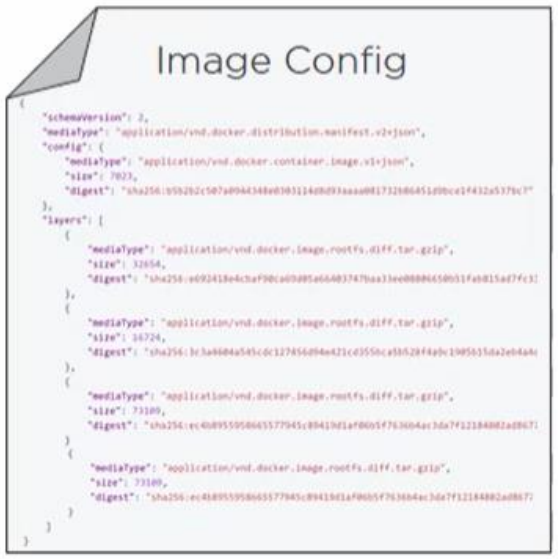

>_ CLI
$ docker container run…
{API}
/containers/create
daemon
containerd
OCI
- namespaces
- control groups
- ...
(magic)

# Docker Images

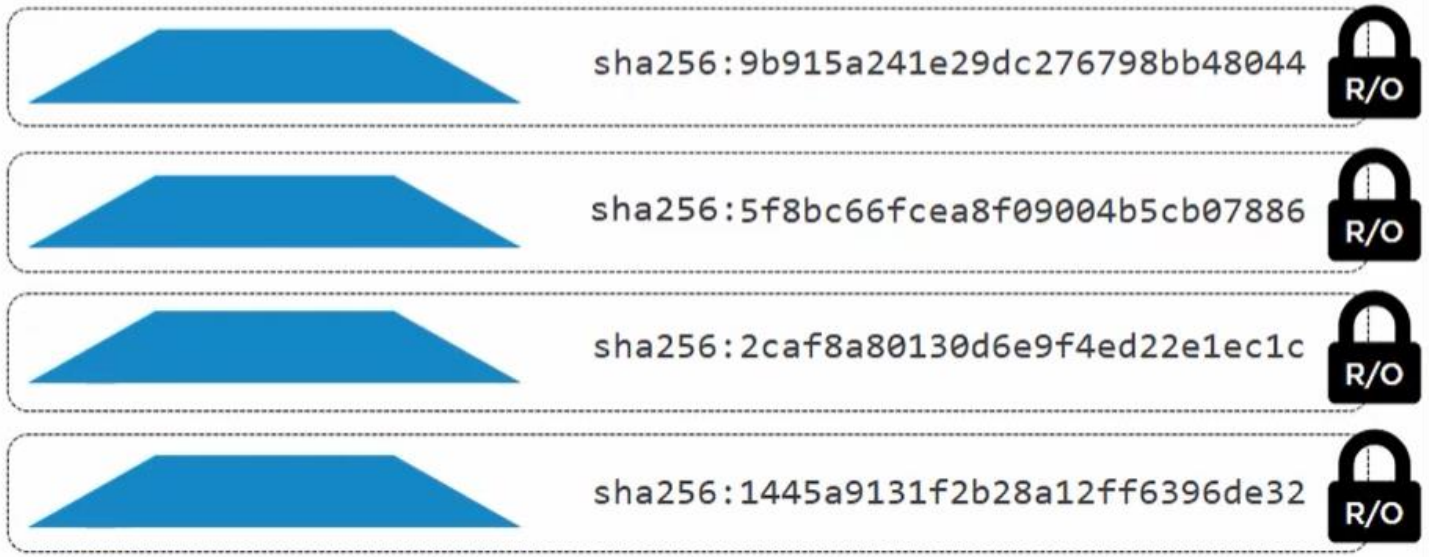

- Docker Image er read-only template/blueprint með leiðbeiningum um hvernig á að smíða container
  - Útvega container-um með isolated file system
  - Innihalda allt sem þarf til að keyra application -> dependencies, configuration, scripts, binaries etc.
- Image getur verið byggt upp af öðru Image
  - Smá eins og inheritance í OOP
  - Getur t.d. byggt upp frá Ubuntu image-i
  - Getur t.d. Byggt upp frá image-i sem er nú þegar með python install-að
- Image er samansett af mörgum independent layer-um
  - Layer-in mynda saman Image-ið sem býr til container-inn
  - Layer-in eru tilgreind sem skref í **Dockerfile** (ekki öll skref bara sum)
  - Eru cached
- Getur búið til þitt eigið image eða notað image frá öðrum
  - Til að búa til þitt eigið býrðu til svokallað **Dockerfile**
  - Getur fundið safn af öðrum image-um á **image registries**
    - https://hub.docker.com/ (default registry-ið)
    - Getið hugsað um registries sem github fyrir docker images
    - Þegar þú pull-ar image vistast það í local registry

# Docker Layers

Docker image er samansett út mörgum layer-um þar sem ákveðið image config bindir layer-in saman í image




sha256:d124781fc06a73b05



# Nokkur Docker Image CLI Commands


```
$ docker build [OPTIONS] PATH | URL | -
```

Byggir image frá dockerfile

**Note** ekki tæmandi listi: https://docs.docker.com/engine/reference/commandline/docker/

```
$ docker images [OPTIONS] [REPOSITORY[:TAG]]
```

Listi af öllum image-um

```
$ docker image pull [OPTIONS] NAME[:TAG|@DIGEST]
```

Sækir image frá registry (docker hub er default)

```
$ docker image push [OPTIONS] NAME[:TAG]
```

Push-ar image á einhvað registry (docker hub er default)

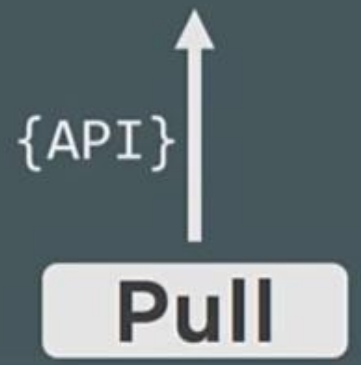
```
$ docker image rm [OPTIONS] IMAGE [IMAGE...]
```

Fjarlægir image

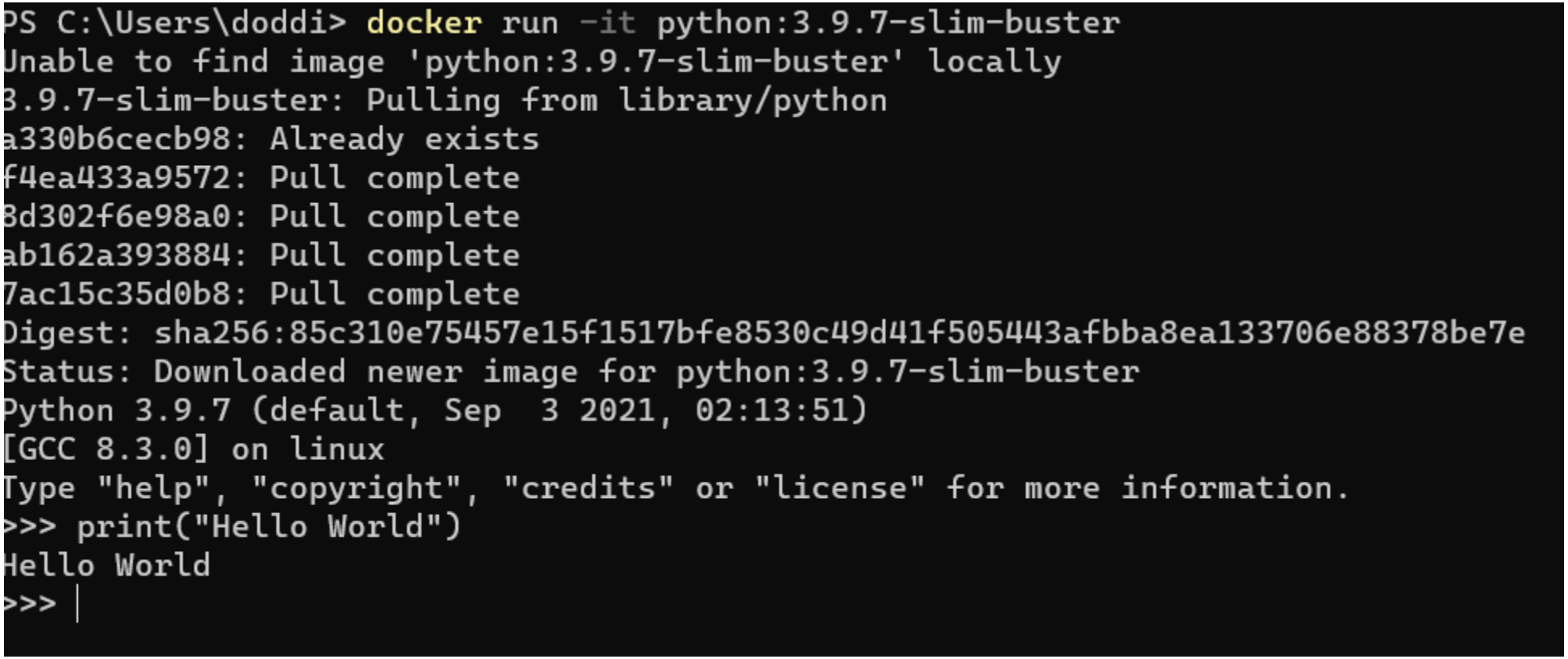
```
root@Linux1:/home/ubuntu#
root@Linux1:/home/ubuntu#
root@Linux1:/home/ubuntu#
root@Linux1:/home/ubuntu# docker image pull redis
Using default tag: latest
latest: Pulling from library/redis
065132d9f705: Pull complete
fc32c7d9b0f4: Pull complete
ad60cc6fa431: Pull complete
b21c99d8cf03: Pull complete
357908014789: Pull complete
e27e1cb0ca43: Pull complete
Digest: sha256:fe77356e6e8d5c5200b9800e50ae71147efdc446a3cc4f601c607fbfd218015e
Status: Downloaded newer image for redis:latest
root@Linux1:/home/ubuntu#
```

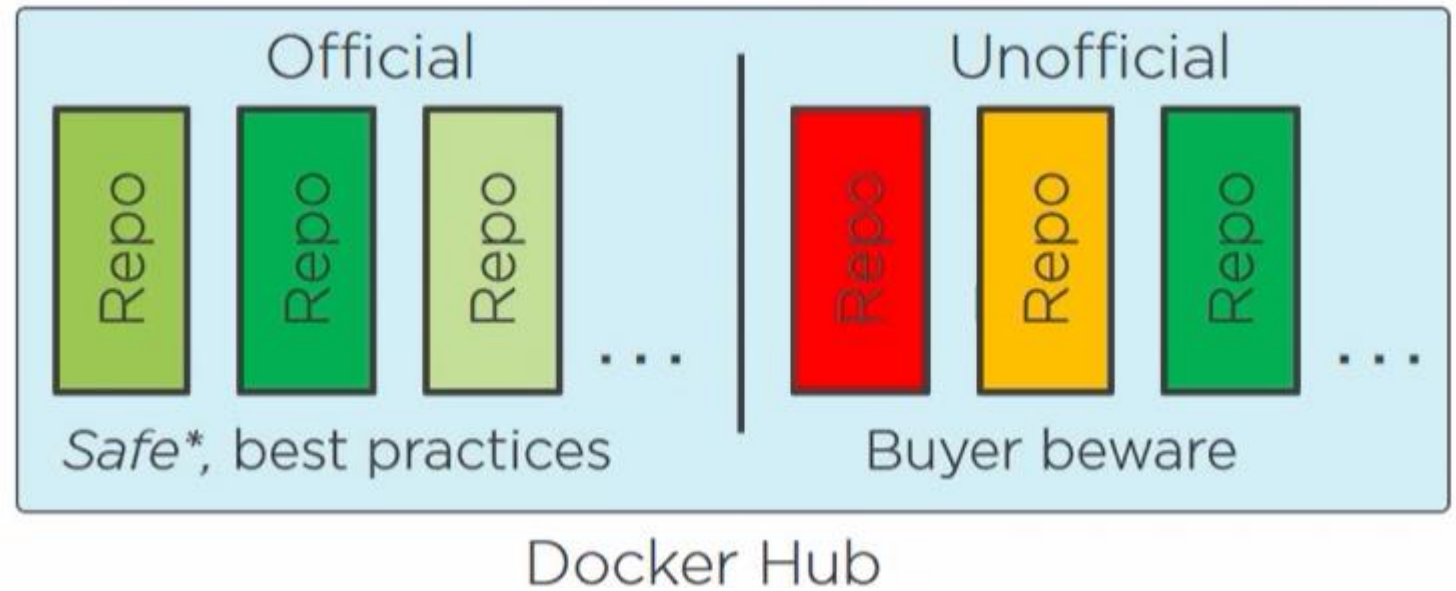
Individual layer-in sem saman mynda image-ið sem við pull-um frá docker hub

Image með latest tag-ið er sótt by default ef image tagg-ið er ekki specify-að (images geta haft mörg tags), latest þarf ekki endilega að vera „latest“/nýjasta þetta er tag sem er sett manually á

Við getum t.d. Pull-að official python 3.9.7 image frá docker hub og keyrt upp container af því image-i með því commandinu: `docker-run -it python:3.9.7-slim-buster` þar sem default command-ið á container-num er að start-a upp python interpreter

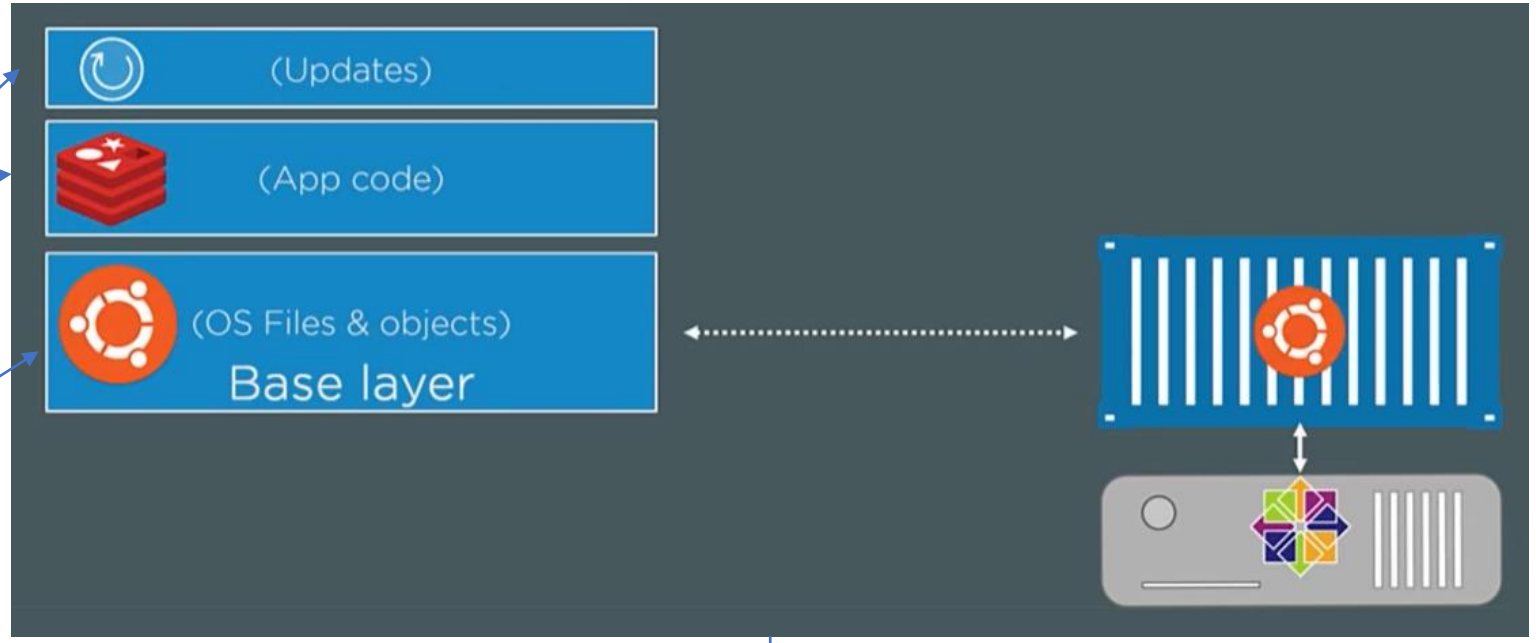




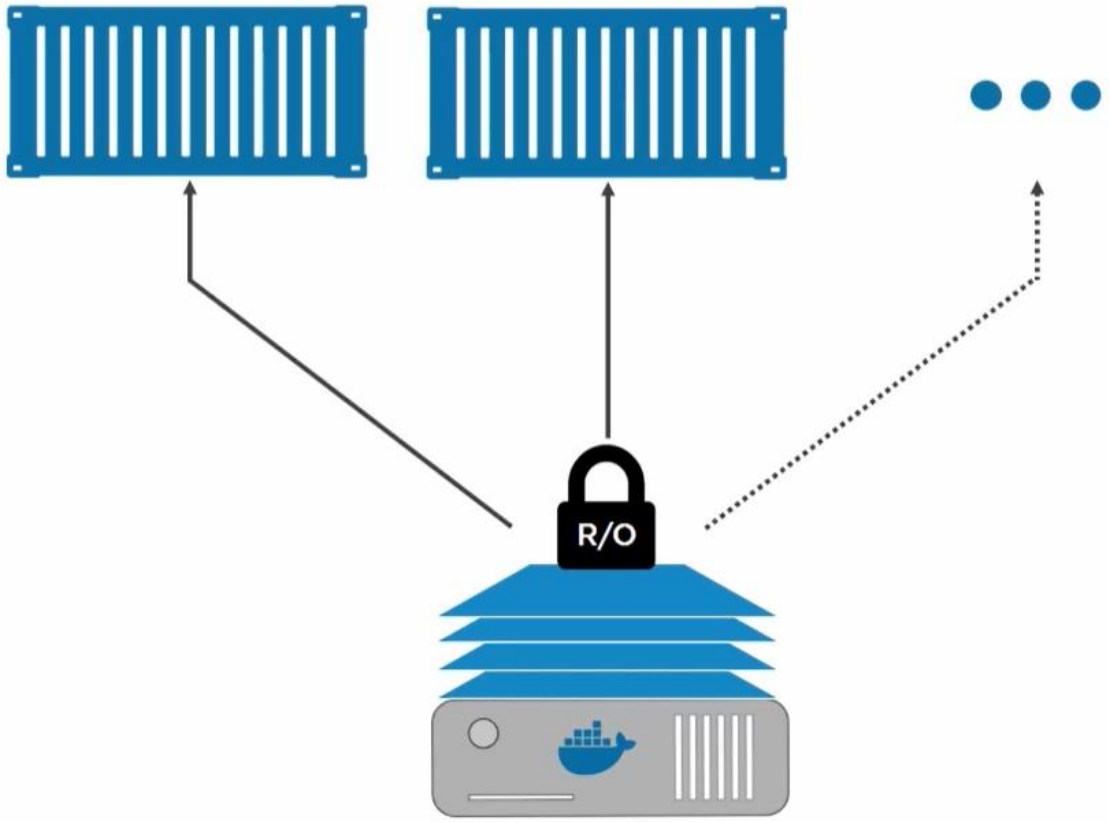

https://docs.docker.com/docker-hub/official_images/

Layers sem við bætum við

Base Image sem við byggjum Image-ið okkar á

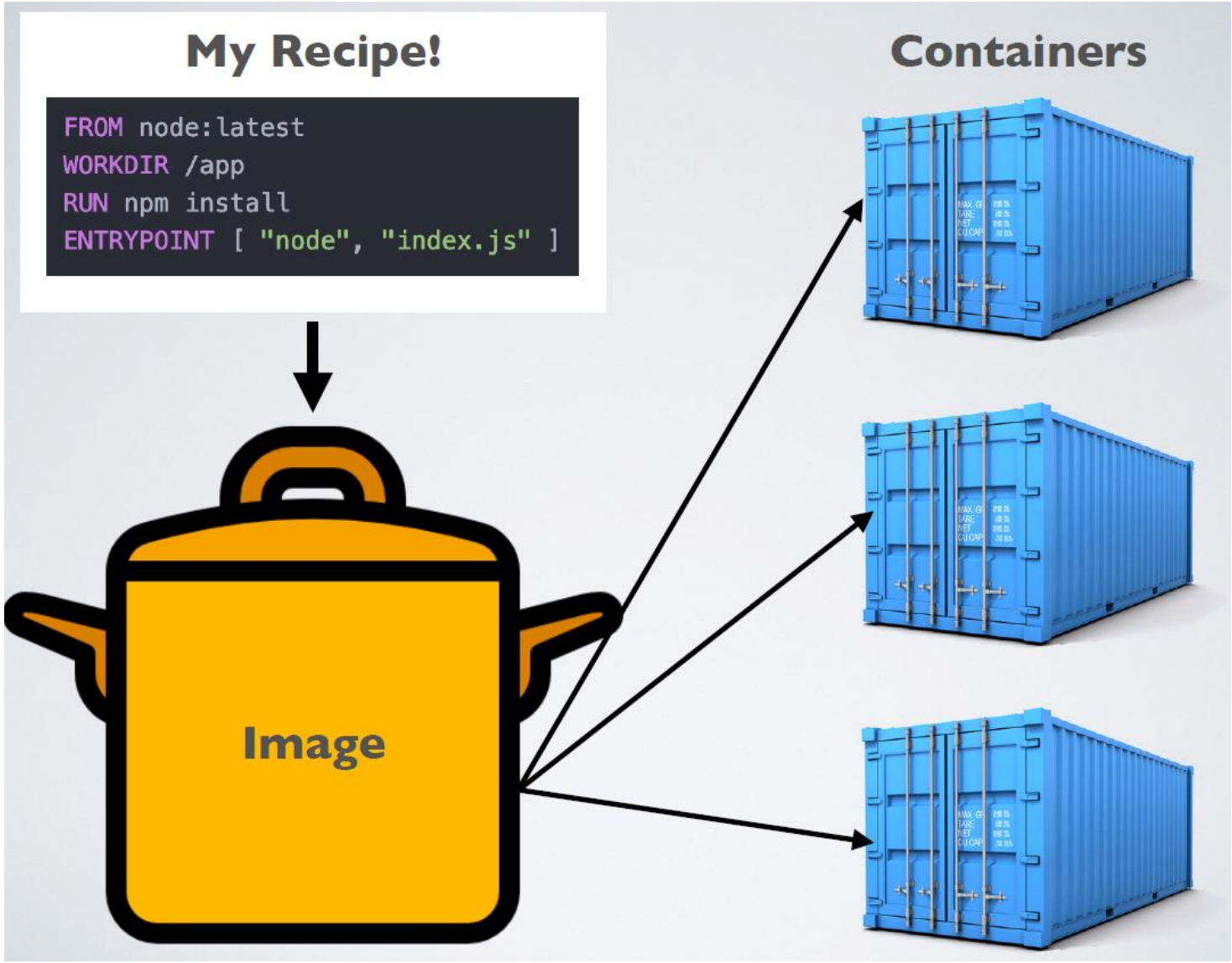



Við hugsum um Image-ið okkar sem unfied heild þegar við vinnum með það


R/O

# Nokkur Docker Image best practices


- Nota Official images
  - https://docs.docker.com/docker-hub/official_images/
  - Official images eru/ættu að vera safe og vel document-uð
  - Unofficial images eru anybody‘s game.
- Keep you images small
  - Small images eru fljótari að pull-a yfir netið og fljótar-i að load-a í minni þegar container-ar eru start-aðir.
  - Gert t.d. með því að reyna að hafa fá layer í image.
  - Hafa lítið base image
  - Nota multistage builds
  - Getur haft lítið production base image og síðan haft annað image fyrir development sem notar production image-ið en bætir við layer-um til að leyfa debugging og aðrar develop virknir
- Image layer caching
  - Hjálpa við build tíma
  - Layer sem breytast oft viljum við t.d. Skilgreina seint í Dockerfile þannig að við getum cache-að öll önnur layer og því eingögnu þurft að build-a layer-ið sem breytist.
- Be explicit með image tags
  - Default „latest“ tag-ið er alltaf það sem þú vilt því latest í gær gæti ekki verið latest á morgun
  - Ættuð frekar að gera t.d. `pull alphine/3.6`
- Security check images
  - Góð hefð að security scan-a image-ið ykkar með `docker scan <image-name>`


My Recipe!
FROM node:latest
WORKDIR /app
RUN npm install
ENTRYPOINT [ "node", "index.js" ]
Image
Containers

# Docker Files

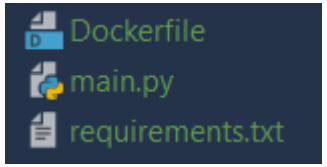
- Docker files eru text-based scriptur af leiðbeiningum sem eru notaðar til að búa til container image

- Þegar við segjum Docker að build-a image-ið okkar með því að keyra `docker build` þá les Docker skipanirnar í Dockerfile og býr til Docker image-ið

- Note að Docker files hafa ekki nein file extension
  - heita yfirleitt `Dockerfile` en geta heitið öðrum nöfnum (Docker gerir default ráð fyrir docker file sem heitir `Dockerfile`)

# Python dæmi

NOTE: þið getið fylgt með dæminu með því að nota docker playground –> https://labs.play-with-docker.com/ án þess að hafa install-að docker locally hjá ykkur. (hægt að drag and drop-a file-um)




Dockerfile

```
# syntax=docker/dockerfile:1

FROM python:3.8-slim-buster

WORKDIR /app

COPY requirements.txt requirements.txt

RUN pip install -r requirements.txt

COPY . .

CMD ["python", "main.py"]
```

main.py

```
import pyfiglet

pyfiglet.print_figlet('hello world’)
```

requirements.txt

```
pyfiglet==0.8.post1
```

Dockerfile verður að byrja á FROM skipuninni, FROM skipuninn specify-ar parent image-ið sem við erum að byggja ofan á.

COPY <src> <dst> leiðbeiningin copy-ar files eða directories frá src til dst (þar sem dst er relative við WORKDIR)

Það getur bara verið ein CMD instruction í Dockerfile, provide-ar default skipun til að keyra fyrir executing container (hægt að yfirskrifa)

```
# syntax=docker/dockerfile:1

FROM python:3.8-slim-buster

WORKDIR /app

COPY requirements.txt requirements.txt

RUN pip install -r requirements.txt

COPY . .

CMD ["python", "main.py"]
```

Syntax segir okkur hvaða syntax Dockerfile-inn notar.

Docker/dockerfile:1 er það sem er mælt með, það bendir á latest stable version af version 1 dockerfile syntax

WORKDIR leiðbeiningin setur working directory-ið fyrir aðrar skipanir þannig skilgreind path fyrir t.d. COPY, RUN, CMD munu vera relative á working directory path-ið.

RUN instruction-ið exectue-ar command-ið sem er specify-að, inni í container-num

docker build byggir image frá Dockerfile

Nafnið á image-inu sem er verið að búa til, by default fær image-ið „latest“ tag-ið ef það er ekki specify-að eins og hér

Leitar að Dockerfile í current directory-inu

```
PS C:\Users\doddi\Desktop\docker\python_demo> docker build --tag python-docker-example .
[+] Building 3.5s (16/16) FINISHED
 => [internal] load build definition from Dockerfile                                                    0.0s
 => => transferring dockerfile: 32B                                                                     0.0s
 => [internal] load .dockerignore                                                                       0.0s
 => => transferring context: 2B                                                                         0.0s
 => resolve image config for docker.io/docker/dockerfile:1                                              1.9s
 => [auth] docker/dockerfile:pull token for registry-1.docker.io                                        0.0s
 => CACHED docker-image://docker.io/docker/dockerfile:1@sha256:9e2c9eca7367393aecc68795c671f93466818395a2693498de  0.0s
 => [internal] load build definition from Dockerfile                                                    0.0s
 => [internal] load .dockerignore                                                                       0.0s
 => [internal] load metadata for docker.io/library/python:3.8-slim-buster                               1.1s
 => [auth] library/python:pull token for registry-1.docker.io                                           0.0s
 => [1/5] FROM docker.io/library/python:3.8-slim-buster@sha256:e9087bce99087a2eca8cb5cc8278de16d7907aee25c47b33bd  0.0s
 => [internal] load build context                                                                       0.1s
 => => transferring context: 158B                                                                       0.0s
 => CACHED [2/5] WORKDIR /app                                                                           0.0s
 => CACHED [3/5] COPY requirements.txt requirements.txt                                                 0.0s
 => CACHED [4/5] RUN pip install -r requirements.txt                                                    0.0s
 => [5/5] COPY . .                                                                                      0.0s
 => exporting to image                                                                                  0.1s
 => => exporting layers                                                                                 0.0s
 => => writing image sha256:727f8fe52019588a8ec325f0600110d67d171151716dfe9b4bdbffebaf8bc7b8            0.0s
 => => naming to docker.io/library/python-docker-example                                                0.0s

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them
PS C:\Users\doddi\Desktop\docker\python_demo>
```

Image-ið sem við vorum að búa til

```
PS C:\Users\doddi\Desktop\docker\python_demo> docker images
REPOSITORY              TAG       IMAGE ID       CREATED          SIZE
python-docker-example   latest    727f8fe52019   6 minutes ago    126MB
PS C:\Users\doddi\Desktop\docker\python_demo>
```

docker images gefur lista af local images sem er búið að búa til

Fær latest sem default tag því við tókum ekkert TAG fram

Við getum gefið image-inu sem við vorum að búa til explicit tag

```
PS C:\Users\doddi\Desktop\docker\python_demo> docker tag python-docker-example:latest python-docker-example:v1.0.0
PS C:\Users\doddi\Desktop\docker\python_demo> docker images
REPOSITORY              TAG       IMAGE ID       CREATED          SIZE
python-docker-example   latest    727f8fe52019   11 minutes ago   126MB
python-docker-example   v1.0.0    727f8fe52019   11 minutes ago   126MB
```

Takið eftir að þetta er ennþá sama image-ið, það er bara núna með tveimur töggum

Við getum líka fjarlægt tag-ið sem við vorum að setja á

```
PS C:\Users\doddi\Desktop\docker\python_demo> docker rmi python-docker-example:v1.0.0
Untagged: python-docker-example:v1.0.0
PS C:\Users\doddi\Desktop\docker\python_demo> docker images
REPOSITORY              TAG       IMAGE ID       CREATED          SIZE
python-docker-example   latest    727f8fe52019   14 minutes ago   126MB
```

docker run build-ar og start-ar container-num okkar og keyrir CMD skipunina eins og er specify-að í Dockerfile-num

```
PS C:\Users\doddi\Desktop\docker\python_demo> docker run python-docker-example
 _                 _ _                                          _       _
| |__   ___| | | ___   __      _____  _ __| | __| |
| '_ \ / _ \ | |/ _ \  \ \ /\ / / _ \| '__| |/ _` |
| | | |  __/ | | (_) |  \ V  V / (_) | |  | | (_| |
|_| |_|\___|_|_|\___/    \_/\_/ \___/|_|  |_|\__,_|
```

docker ps –a gefur lista af öllum(-a stendur fyrir all) container-um sem við erum búin að búa til

Við getum séð að container-inn okkar hætti í keyrslu þegar skipunin sem CMD keyrði klárast

```
PS C:\Users\doddi\Desktop\docker\python_demo> docker ps -a
CONTAINER ID   IMAGE                  COMMAND              CREATED          STATUS                      PORTS     NAMES
71960d52ede0   python-docker-example   "python main.py"   3 minutes ago   Exited (0) 3 minutes ago              friendly_ellis
PS C:\Users\doddi\Desktop\docker\python_demo>
```

```
import pyfiglet

pyfiglet.print_figlet('hello world’)
```

Prufum að breyta main.py

```
import pyfiglet

name = input('What\'s your name? ')
pyfiglet.print_figlet(name)
```

Ef við búum til og run-um nýjum container þá fáum við ennþá gömlu skilaboðin, ástæðan fyrir því er að við erum ennþá að byggja frá gömlu image-i sem var byggt upp frá gamla kóðanum

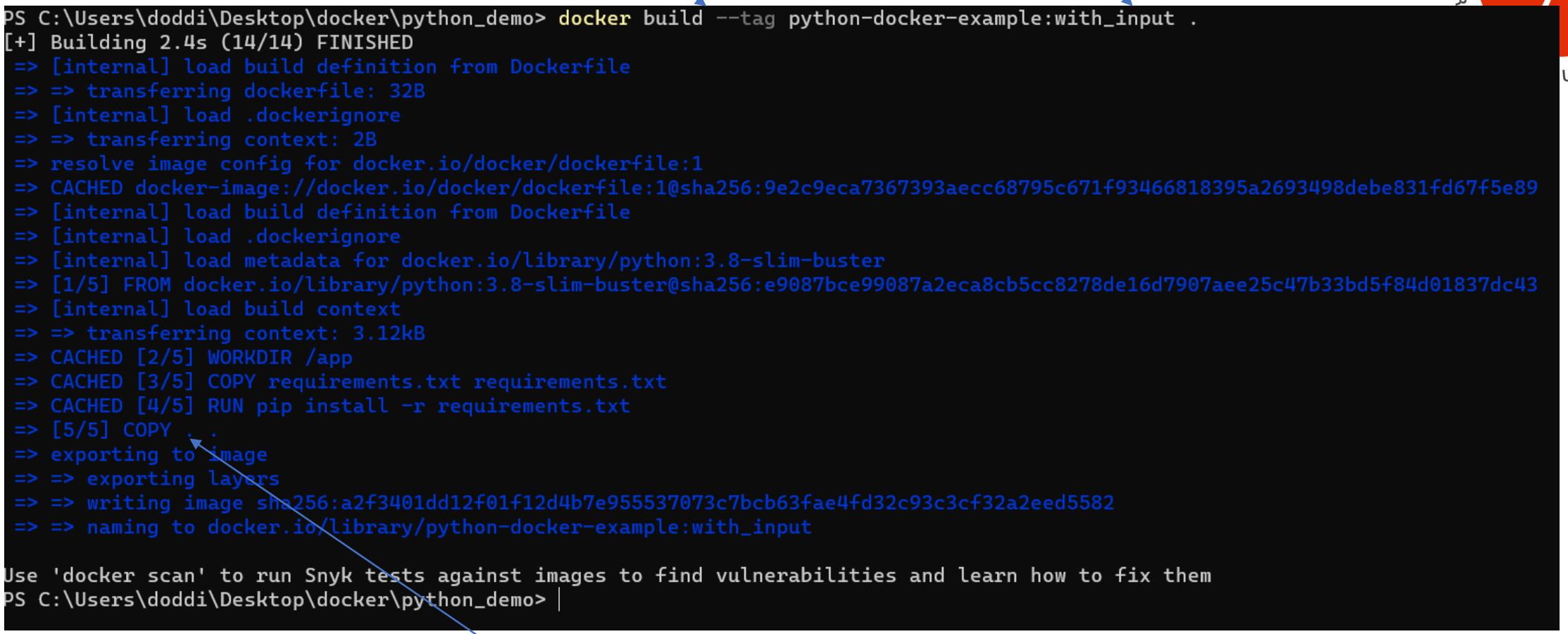
```
PS C:\Users\doddi\Desktop\docker\python_demo> docker run python-docker-example
 _           _ _                                        _       _
| |__   ___| | | ___   __      _____  _ __| | __| |
| '_ \ / _ \ | |/ _ \  \ \ /\ / / _ \| '__| |/ _` |
| | | |  __/ | | (_) |  \ V  V / (_) | |  | | (_| |
|_| |_|\___|_|_|\___/    \_/\_/ \___/|_|  |_|\__,_|
```

Til að sjá að við bjuggum til nýjan container getum við notað docker ps -a

```
PS C:\Users\doddi\Desktop\docker\python_demo> docker ps -a
CONTAINER ID   IMAGE                    COMMAND             CREATED          STATUS                      PORTS     NAMES
4818490309ba   python-docker-example    "python main.py"    5 minutes ago    Exited (0) 5 minutes ago              elated_mccarthy
71960d52ede0   python-docker-example    "python main.py"    18 minutes ago   Exited (0) 18 minutes ago             friendly_ellis
PS C:\Users\doddi\Desktop\docker\python_demo>
```

Búum til nýtt image með nýju breytingunum

Gefum því explicit tag-ið *with_input*



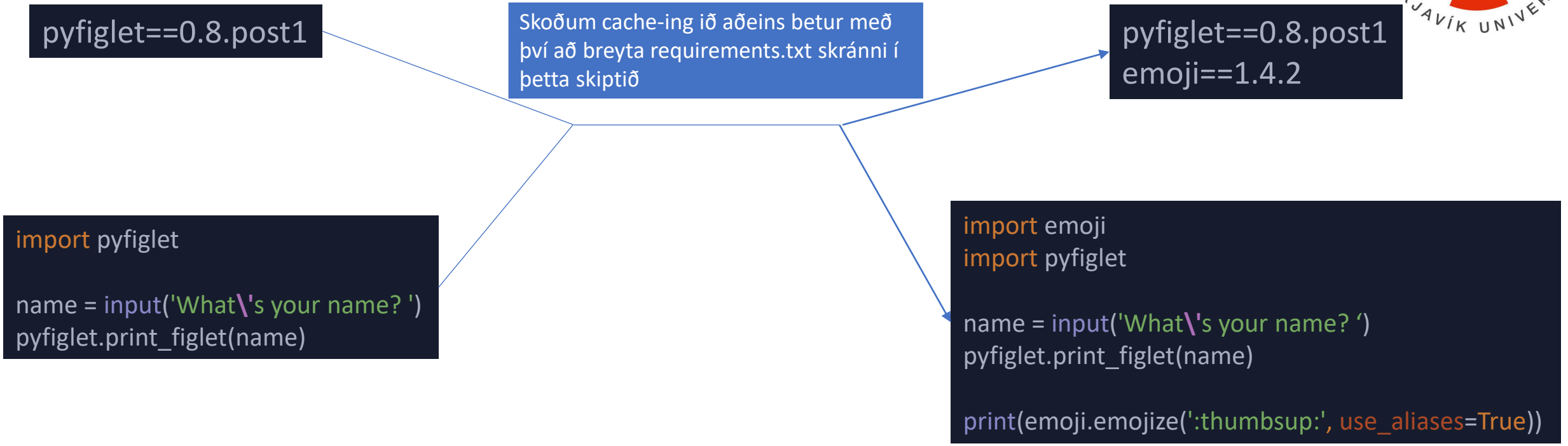

Athugið að layer-ið sem COPY býr til er ekki cache-að, þetta er af því docker tekur hash af öllum file-um sem við copy-um og ef eitthvað file breytist þá breytist hash-ið líka og docker veit þá að það þarf að búa til layer-ið sem COPY skipunin gerir aftur

Ef við keyrum docker run eins fyrir nýja image-ið eins og við gerðum áðan þá fáum við error því input býst við input-i frá terminal-inu en það er engin til að gefa input í container terminal-inu

```
PS C:\Users\doddi\Desktop\docker\python_demo> docker run python-docker-example:with_input
What's your name? Traceback (most recent call last):
  File "main.py", line 3, in <module>
    name = input('What\'s your name? ')
EOFError: EOF when reading a line
```

Við getum lagað þetta með því að bæta við –it flaggið, þetta tengir terminal-ið í container-num við terminal-ið okkar

```
PS C:\Users\doddi\Desktop\docker\python_demo> docker run -it python-docker-example:with_input
What's your name? Þórður


 _ __ _   _ _ __
| '__| | | | '__|
| |  | |_| | |
|_|   \__,_|_|
```

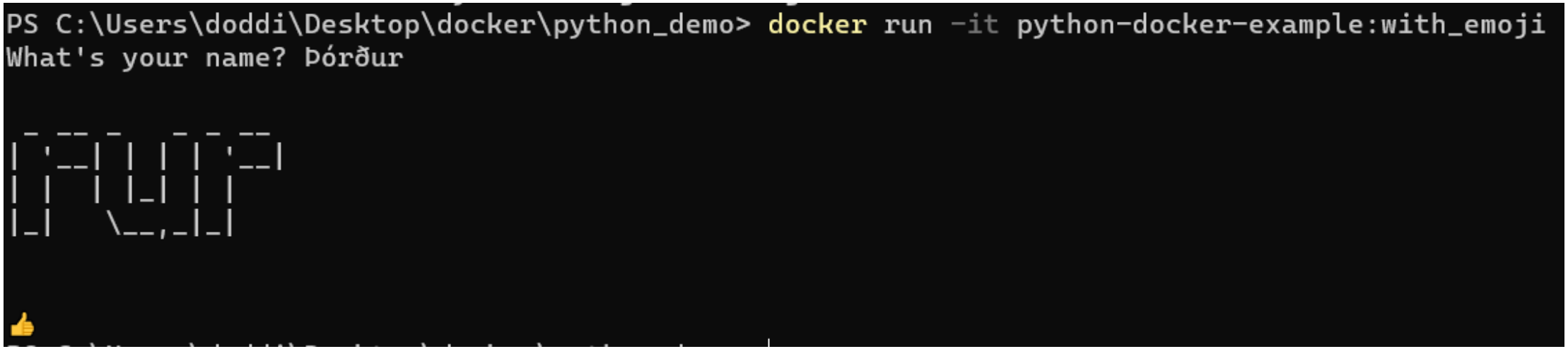

pyfiglet==0.8.post1
Skoðum cache-ing ið aðeins betur með því að breyta requirements.txt skránni í þetta skiptið
pyfiglet==0.8.post1
emoji==1.4.2
import pyfiglet
name = input('What\'s your name? ')
pyfiglet.print_figlet(name)
import emoji
import pyfiglet
name = input('What\'s your name? ‘)
pyfiglet.print_figlet(name)
print(emoji.emojize(':thumbsup:', use_aliases=True))

Búum til nýtt image með explicit tag-i eins og áður

```
PS C:\Users\doddi\Desktop\docker\python_demo> docker build --tag python-docker-example:with_emoji .
[+] Building 8.5s (16/16) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 32B
 => [internal] load .dockerignore
 => => transferring context: 44B
 => resolve image config for docker.io/docker/dockerfile:1
 => [auth] docker/dockerfile:pull token for registry-1.docker.io
 => CACHED docker-image://docker.io/docker/dockerfile:1@sha256:9e2c9eca7367393aecc68795c671f93466818395a2693498debe831fd67f5e89
 => [internal] load .dockerignore
 => [internal] load build definition from Dockerfile
 => [internal] load metadata for docker.io/library/python:3.8-slim-buster
 => [auth] library/python:pull token for registry-1.docker.io
 => [1/5] FROM docker.io/library/python:3.8-slim-buster@sha256:e9087bce99087a2eca8cb5cc8278de16d7907aee25c47b33bd5f84d01837dc43
 => [internal] load build context
 => => transferring context: 3.91kB
 => CACHED [2/5] WORKDIR /app
 => [3/5] COPY requirements.txt requirements.txt
 => [4/5] RUN pip install -r requirements.txt
 => [5/5] COPY . .
 => exporting to image
 => => exporting layers
 => => writing image sha256:b34d5a7ac6c794516ea603e84458e27da7b464f230a71bb7a77abd372ce4fb2a
 => => naming to docker.io/library/python-docker-example:with_emoji
```

Tekið eftir að í þetta skiptið er eingögngu WORKDIR layer-ið cache-að, ástæðan fyrir því er að við breyttum requirements skjalinu þannig COPY layer-ið fyrir það getur ekki verið endurnýtt og ekki heldur neitt af layer-unum fyrir neðan það einnig ekki hægt að endurnýta layer-in sem eru fyrir neðan það því layer-in fyrir neðan byggja á layer-inu á undan (t.d. RUN pip install –r requirements.txt)

Keyrum emoji container til gamans



By default gefur docker container-um random nafn ef við tökum ekki sérstaklega eitthvað container nafn fram

```
PS C:\Users\doddi\Desktop\docker\python_demo> docker ps -a
CONTAINER ID   IMAGE                                COMMAND           CREATED              STATUS                          PORTS     NAMES
603b67cf61dc   python-docker-example:with_emoji     "python main.py"  5 minutes ago        Exited (0) 5 minutes ago                  elegant_nightingale
a4b87ef27a58   python-docker-example:with_input     "python main.py"  55 minutes ago       Exited (0) 55 minutes ago                 angry_ramanujan
63d582d1a637   python-docker-example:with_input     "python main.py"  58 minutes ago       Exited (1) 58 minutes ago                 quizzical_hoover
6d5423af6877   727f8fe52019                         "python main.py"  About an hour ago    Exited (0) About an hour ago              distracted_mcnulty
4818490309ba   python-docker-example                "python main.py"  About an hour ago    Exited (0) About an hour ago              elated_mccarthy
71960d52ede0   python-docker-example                "python main.py"  2 hours ago          Exited (0) 2 hours ago                    friendly_ellis
```

Rename-um container-inn sem er byggður á emoji image-inu sem emoji_container

```
PS C:\Users\doddi\Desktop\docker\python_demo> docker rename elegant_nightingale emoji_container
PS C:\Users\doddi\Desktop\docker\python_demo> docker ps -a
CONTAINER ID   IMAGE                                COMMAND           CREATED              STATUS                          PORTS     NAMES
603b67cf61dc   python-docker-example:with_emoji     "python main.py"  8 minutes ago        Exited (0) 8 minutes ago                  emoji_container
a4b87ef27a58   python-docker-example:with_input     "python main.py"  58 minutes ago       Exited (0) 58 minutes ago                 angry_ramanujan
63d582d1a637   python-docker-example:with_input     "python main.py"  About an hour ago    Exited (1) About an hour ago              quizzical_hoover
6d5423af6877   727f8fe52019                         "python main.py"  About an hour ago    Exited (0) About an hour ago              distracted_mcnulty
4818490309ba   python-docker-example                "python main.py"  About an hour ago    Exited (0) About an hour ago              elated_mccarthy
71960d52ede0   python-docker-example                "python main.py"  2 hours ago          Exited (0) 2 hours ago                    friendly_ellis
```

Við getum start-að gamla container-inum okkar aftur án þess að þurfa að búa til nýjan

```
PS C:\Users\doddi\Desktop\docker\python_demo> docker start -ai emoji_container
What's your name? doddi
    _                 _     _ _
  __| | ___   __| | __| (_)
 / _` |/ _ \ / _` |/ _` | |
| (_| | (_) | (_| | (_| | |
 \__,_|\___/ \__,_|\__,_|_|


👍
PS C:\Users\doddi\Desktop\docker\python_demo> docker ps -a
CONTAINER ID   IMAGE                                COMMAND             CREATED             STATUS                          PORTS     NAMES
603b67cf61dc   python-docker-example:with_emoji     "python main.py"    16 minutes ago      Exited (0) 44 seconds ago                 emoji_container
a4b87ef27a58   python-docker-example:with_input     "python main.py"    About an hour ago   Exited (0) About an hour ago              angry_ramanujan
63d582d1a637   python-docker-example:with_input     "python main.py"    About an hour ago   Exited (1) About an hour ago              quizzical_hoover
6d5423af6877   727f8fe52019                         "python main.py"    About an hour ago   Exited (0) About an hour ago              distracted_mcnulty
4818490309ba   python-docker-example                "python main.py"    2 hours ago         Exited (0) 2 hours ago                    elated_mccarthy
71960d52ede0   python-docker-example                "python main.py"    2 hours ago         Exited (0) 2 hours ago                    friendly_ellis
```

Við getum sett nafnið á container-num þegar við búum hann til líka

```
PS C:\Users\doddi\Desktop\docker\python_demo> docker create -it --name emoji_container2 python-docker-example:with_emoji
5ac81d45a72529617e41178cb08aa31fbb6d5d9993a60a101b71006aa11a8689
PS C:\Users\doddi\Desktop\docker\python_demo> docker ps -a
CONTAINER ID   IMAGE                              COMMAND             CREATED             STATUS                          PORTS     NAMES
5ac81d45a725   python-docker-example:with_emoji   "python main.py"    26 seconds ago      Created                                   emoji_container2
603b67cf61dc   python-docker-example:with_emoji   "python main.py"    31 minutes ago      Exited (130) 4 minutes ago                emoji_container
a4b87ef27a58   python-docker-example:with_input   "python main.py"    About an hour ago   Exited (0) About an hour ago              angry_ramanujan
63d582d1a637   python-docker-example:with_input   "python main.py"    About an hour ago   Exited (1) About an hour ago              quizzical_hoover
6d5423af6877   727f8fe52019                       "python main.py"    2 hours ago         Exited (0) 2 hours ago                    distracted_mcnulty
4818490309ba   python-docker-example              "python main.py"    2 hours ago         Exited (0) 2 hours ago                    elated_mccarthy
71960d52ede0   python-docker-example              "python main.py"    2 hours ago         Exited (0) 2 hours ago                    friendly_ellis
```

Terminal 1 er að keyra emoji_container2

```
PS C:\Users\doddi\Desktop\docker\python_demo> docker start -ai emoji_container2
What's your name?
```

Terminal 2 er að keyra emoji_container

```
PS C:\Users\doddi> docker start -ai emoji_container
What's your name?
```

Terminal 3 sýnir okkur að þessir container-ar eru keyrandi á sama tíma

```
PS C:\Users\doddi> docker ps
CONTAINER ID   IMAGE                                COMMAND             CREATED          STATUS          PORTS     NAMES
5ac81d45a725   python-docker-example:with_emoji     "python main.py"    2 minutes ago    Up 40 seconds             emoji_container2
603b67cf61dc   python-docker-example:with_emoji     "python main.py"    33 minutes ago   Up 6 seconds              emoji_container
```

Við getum síðan auðvitað fjarlægt emoji_container2

```
PS C:\Users\doddi> docker container rm emoji_container2
emoji_container2
PS C:\Users\doddi> docker ps -a
CONTAINER ID   IMAGE                                COMMAND             CREATED             STATUS                          PORTS     NAMES
603b67cf61dc   python-docker-example:with_emoji     "python main.py"    37 minutes ago      Exited (130) 7 seconds ago                emoji_container
a4b87ef27a58   python-docker-example:with_input     "python main.py"    About an hour ago   Exited (0) About an hour ago              angry_ramanujan
63d582d1a637   python-docker-example:with_input     "python main.py"    2 hours ago         Exited (1) 2 hours ago                    quizzical_hoover
6d5423af6877   727f8fe52019                         "python main.py"    2 hours ago         Exited (0) 2 hours ago                    distracted_mcnulty
4818490309ba   python-docker-example                "python main.py"    2 hours ago         Exited (0) 2 hours ago                    elated_mccarthy
71960d52ede0   python-docker-example                "python main.py"    2 hours ago         Exited (0) 2 hours ago                    friendly_ellis
PS C:\Users\doddi>
```

Búum til repository á docker hub til að push-a image-inu sem við erum búin að búa til

https://hub.docker.com/repository/create

## Create Repository

doddi321 | python-docker-example

Description

### Visibility

Using 0 of 1 private repositories. Get more

**Public**
Appears in Docker Hub search results

**Private**
Only visible to you

Cancel | Create

Pro tip

You can push a new image to this repository using the CLI

```
docker tag local-image:tagname new-repo:tagname
docker push new-repo:tagname
```

Make sure to change *tagname* with your desired image repository tag.

doddi321 | python-docker-example

Til að við getum push-að local image-inu okkar á docker hub þá þurfum við fyrst að endurskýra myndina þannig að það spegli repo-ið á docker hub þ.e. <username>/<image_name>:<image_tag>

```
PS C:\Users\doddi> docker tag python-docker-example:with_emoji doddi321/python-docker-example:with_emoji
```

Næst getum við push-að image-inu okkar á docker hub

```
PS C:\Users\doddi> docker push doddi321/python-docker-example:with_emoji
The push refers to repository [docker.io/doddi321/python-docker-example]
3c58270a4b87: Pushed
4523a8cbcc5d: Pushed
1439d3bd307b: Pushed
bcc97ea8799a: Pushed
55035799b7c9: Pushed
1e9b19e6cd49: Pushed
4e74918601aa: Pushed
55ad5d8b8f0f: Pushed
1000633a5681: Pushed
with_emoji: digest: sha256:fad7f0a650c9bbb567bc61d1566312f76cb094e1f97a74eef8614f69d0e04c4f size: 2203
PS C:\Users\doddi>
```

```
[node1] (local) root@192.168.0.8 ~
$ docker pull doddi321/python-docker-example:with_emoji
with_emoji: Pulling from doddi321/python-docker-example
a330b6cecb98: Pull complete
319bfff7a4e7: Pull complete
e16783809314: Pull complete
2331071d0972: Pull complete
ab05c39cc860: Pull complete
b1179e3a75f9: Pull complete
5a9e5f8976f4: Pull complete
e633526ddb70: Pull complete
aec7e1ec4fd5: Pull complete
Digest: sha256:fad7f0a650c9bbb567bc61d1566312f76cb094e1f97a74eef8614f69d0e04c4f
Status: Downloaded newer image for doddi321/python-docker-example:with_emoji
docker.io/doddi321/python-docker-example:with_emoji
```

```
[node1] (local) root@192.168.0.8 ~
$ docker run -it doddi321/python-docker-example:with_emoji
What's your name? doddi
     _                 _       _ _
  __| | ___     __| | __| (_)
 / _` |/ _ \   / _` |/ _` | |
| (_| | (_) | | (_| | (_| | |
 \__,_|\___/   \__,_|\__,_|_|

👍
```

Til að sjá the power of docker þá getum við núna farið á https://labs.play-with-docker.com/ og pull-að image-ið okkar á algjörlega ótengda tölvu og keyrt container af því image-i án þess að þurfa að gera nokkuð setup

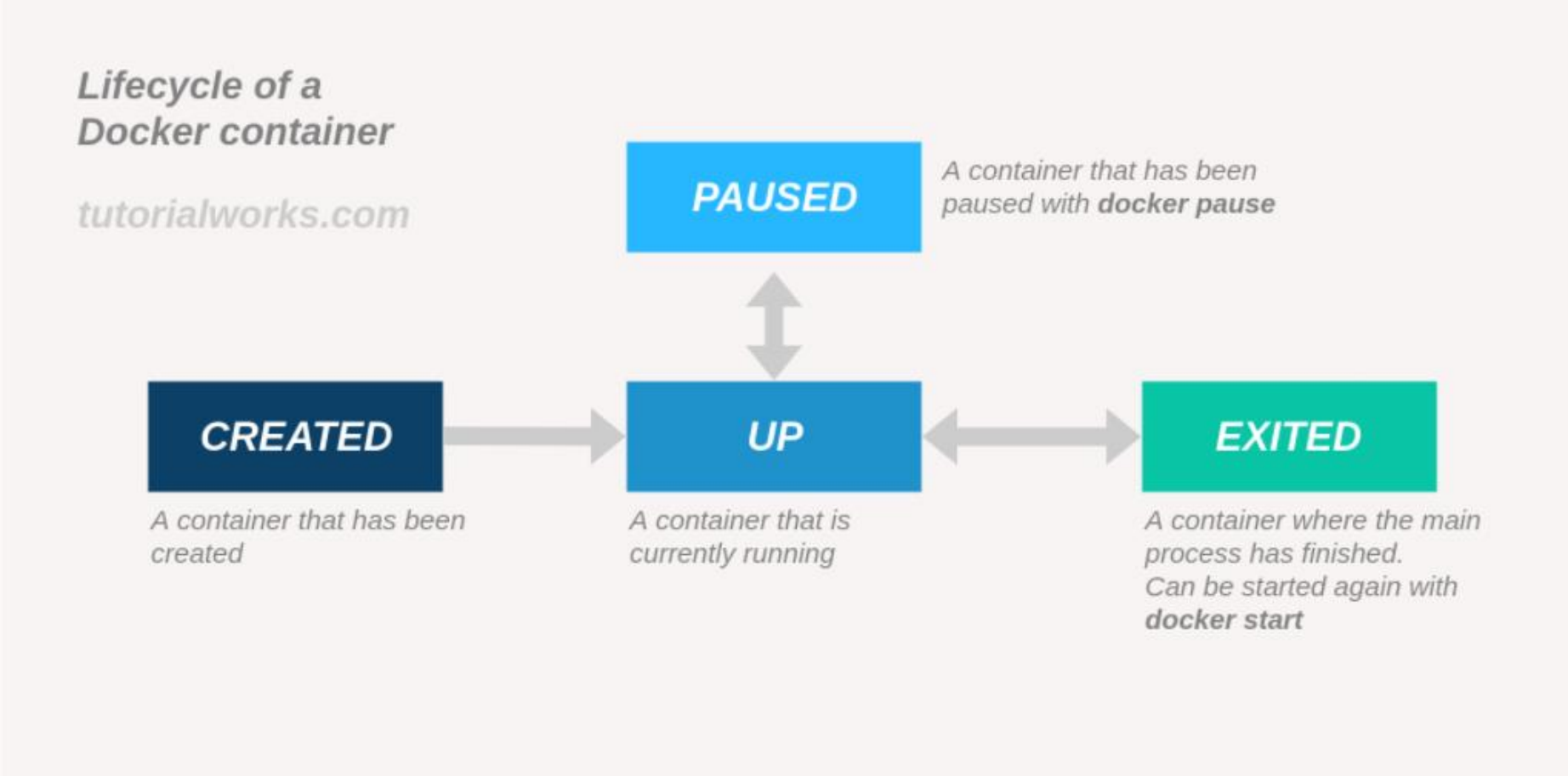
# CMD vs RUN vs ENTRYPOINT

RUN:
- RUN keyrir þegar við erum að búa til image-ið okkar
- það er image build skref og býr til nýtt layer í myndinni (og þar afleiðandi cache-að)
- Það geta verið margar RUN skipanir í einum Dockerfile

CMD:
- CMD setur default command sem keyrist þegar container er startaður upp (hægt að yfirskrifa þetta command við container startup)
- Getur bara verið ein CMD skipun í Dockerfile (ef fleiri þá er seinasta CMD skipunin sú sem gildir)

ENTRYPOINT:
- Entrypoint leyfir okkur að configure-a container til að keyra sem executable
- Hefur svipað syntax og CMD
- Munurinn á CMD og ENTRYPOINT er að ENTRYPOINT **command-in eru ekki ignore-uð þegar docker keyrir með command line parameters**
- Prefer ENTRYPOINT yfir CMD þegar ákveðið command þarf alltaf að keyrast og ekki hægt að yfirskrifa(auðveldlega)

```
# syntax=docker/dockerfile:1

FROM python:3.8-slim-buster

WORKDIR /app

COPY requirements.txt requirements.txt

RUN pip install -r requirements.txt

COPY . .

CMD ["python", "main.py"]
```


Lifecycle of a
Docker container
tutorialworks.com
PAUSED
A container that has been paused with docker pause
CREATED
A container that has been created
UP
A container that is currently running
EXITED
A container where the main process has finished. Can be started again with docker start

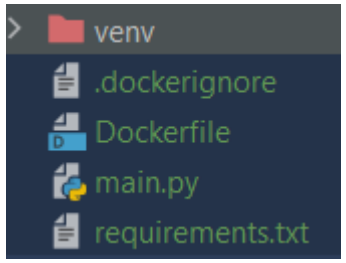
# Meira um Image Layers


- Images eru layer based
  - þ.e.a.s að images eru búin til úr mörgum layer-um
  - Með Dockerfile-num okkar erum við basically að setja ný layer ofan á parent image-ið og búa til nýtt custom image þannig.
- Nokkur Command/skref í Dockerfile-um búa til sér layer
  - RUN
  - COPY
  - ADD
- Layer eru cache-uð
  - Þegar við build-um ákveðið image þá eru eingöngu þau layer þar sem eitthvað breyttist (og öll layer sem sitja undir því layer-i) endur smíðuð
  - Þetta gerir build process-ið mun fljótara því við getum endurnýtt þau layer sem eru óbreytt
  - Röð á skrefum í dockerfile-um er því mikilvæg (t.d. Mikilvægt að gera pip install –r requirements.txt áður en application kóðinn er keyrður því application kóðinn breytist oft)
  - Sjá glæru
  - Hægt að koma í veg fyrir að nota cache með `--no-cache` flag-inu

# .Dockerignore

- Sama hugmynd og .gitignore
  - Og er með svipuð exclusion patterns
- Við getum skilgreint hluti í .Dockerignore sem við viljum að Docker ignore-i þegar við build-um image-ið okkar
- Þetta getur leitt til fljótari build tíma
  - Bæði því við erum t.d. ekki að copy-a einhverja file-a og folder-a sem við þurfum ekki
  - En líka því **þetta getur hjálpað okkur með caching** því t.d. ef við erum með files eða folder-a sem skipta engu máli í container-num en breytast oft hjá okkur locally þá myndi t.d.  cache af COPY . . Skipuninni ekki verið nýtt því þau file breytast.

# .Dockerignore dæmi




.dockerignore

```
.idea
venv
```

```
PS C:\Users\doddi\OneDrive - Reykjavik University\kennsludót\HONN-haust-2021\prufa\docker\1> docker images
REPOSITORY                  TAG                     IMAGE ID       CREATED              SIZE
python-docker-example       with_dockerignore       80f524b63fd9   5 seconds ago        126MB
python-docker-example       without_dockerignore    a598a5ed9e36   40 seconds ago       145MB
```

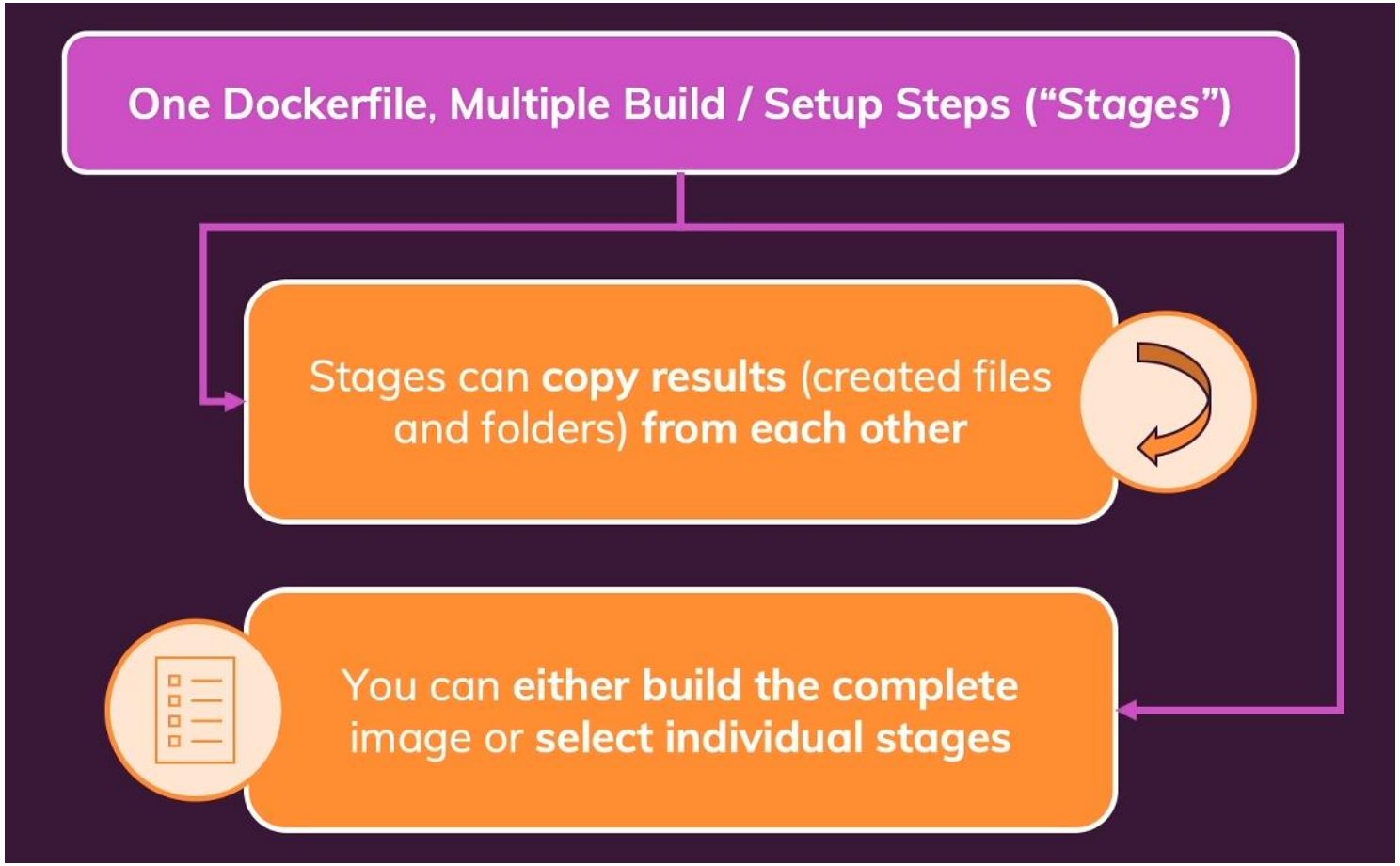
Þegar við build-um image með .dockerignore file þá sjáum við töluverðan mun á image size-inu bara út af venv folder-num

# Dockerfile dæmi Node

```
FROM node:14

# Create app directory
WORKDIR /usr/src/app

# Install app dependencies
# A wildcard is used to ensure both package.json AND package-lock.json are copied
# where available (npm@5+)
COPY package*.json ./

RUN npm install
# If you are building your code for production
# RUN npm ci --only=production

# Bundle app source
COPY . .

EXPOSE 8080
CMD [ "node", "server.js" ]
```

# Multi-stage builds


- Minnka image size
  - Multi-stage builds hjálpa okkur að minnka final size-ið á image-inu okkar
- Mörg FROM statements
  - Með multi-stage build-um þá notum við mörg FROM statements  í Dockerfile-num okkar
  - FROM skipanirnar geta notað mismunandi base image
  - Hver FROM skipun byrjar nýtt stage af build-inu
- Notum aðeins það sem við þurfum
  - Með multi stage buildum getum við copy-að artifact-a frá einu build-i í annað og þannig skilið eftir allt sem við þurfum ekki í final image-inu


One Dockerfile, Multiple Build / Setup Steps (“Stages”)
Stages can copy results (created files and folders) from each other
You can either build the complete image or select individual stages

# Multi-stage builds React dæmi

Dockerfile

```
FROM node:14-alpine

WORKDIR /app

COPY package.json .

RUN npm install

COPY . .

EXPOSE 3000

CMD [ "npm", "start" ]
```

Dockerfile.prod

```
FROM node:14-alpine as build

WORKDIR /app

COPY package.json .

RUN npm install

COPY . .

RUN npm run build

FROM nginx:stable-alpine

COPY --from=build /app/build /usr/share/nginx/html

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]
```

# Multi-stage builds python dæmi


Fyrsta build skrefið(compile-image) keyrir fyrst einhverjar skipanir sem eru nauðsynlegar til að geta compile-að c++ kóða, býr síðan til virtualenv og pip install-ar síðan þessu python project-i (sem presumably krefst að compile c++ kóða)

Seinna build-ið notar síðan bara afurðina/artifact-inn af pip install-inu

```
FROM python:3.7-slim AS compile-image
RUN apt-get update
RUN apt-get install -y --no-install-recommends build-essential gcc

RUN python -m venv /opt/venv
# Make sure we use the virtualenv:
ENV PATH="/opt/venv/bin:$PATH"

COPY requirements.txt .
RUN pip install -r requirements.txt

COPY setup.py .
COPY myapp/ .
RUN pip install .

FROM python:3.7-slim AS build-image
COPY --from=compile-image /opt/venv /opt/venv

# Make sure we use the virtualenv:
ENV PATH="/opt/venv/bin:$PATH"
CMD ['myapp']
```

https://pythonspeed.com/articles/multi-stage-docker-python/

# Nokkur Docker Files best practices


- Nota Multi stage builds
  - Multi stage builds geta drastically minnkað final size-ið á image-inu okkar
- Leverage Build Cache
  - Þegar við búum til dockerfile-in okkar er gott að hafa layer caching í huga og hafa þær skipanir sem leiða til layer-a sem breytast sjaldan ofar en þau sem breytast oftar.
- .dockerignore
  - Við getum notað .dockerignore til að ignore-a þau file sem okkur er sama um og þannig minnkað image size-ið okkar og mögulega bjargað okkur frá því að þurfa að endursmíða ákveðið layer í stað þess að nota cache-ið
- Minnka fjölda skipana sem búa til nýtt layer
  - Við getum reynt að minnka image size-ið okkar með því að minnka fjölda skipana sem búa til nýtt layer í Dockerfile-num okkar
  - T.d. með því að sameina margar RUN skipanir í eina.

# Persisting data á milli container-a

Gerum enn aðra breytingu í main.py file-num okkar þar sem við prentum ekki nöfnin út í console-ið heldur skrifum við þau niður í file:

```
import emoji

import pyfiglet

name = input('What\'s your name? ‘)

with open('data/names.txt', 'a+', encoding='utf-8') as names_file:
    jazzed_name = pyfiglet.figlet_format(name)
    thumbs_up = emoji.emojize(':thumbsup:', use_aliases=True)
    names_file.write(f'{jazzed_name} {thumbs_up}')
```

# Persisting data á milli container-a vandamál


Build-um image-ið

```
docker build --tag python-docker-example:persistance .
```

Keyrum síðan upp container instance af þessu image-i

```
PS C:\> docker run -it --name persistance_container_1 python-docker-example:persistance
What's your name? doddi
PS C:\>
```

```
PS C:\> docker start persistance_container_1
persistance_container_1
PS C:\> docker exec -it persistance_container_1 cat data/names.txt
     _                _       _ _
  __| | ___   __| | __| (_)
 / _` |/ _ \ / _` |/ _` | |
| (_| | (_) | (_| | (_| | |
 \__,_|\___/ \__,_|\__,_|_|

 👍
```

Skoðum síðan innihaldið í names.txt file-num okkar með því að start-a upp sama container og prenta-a út names.txt

# Persisting data á milli container-a vandamál frh.

Búum til nýtt container tilvik af image-inu okkar

```
PS C:\> docker run -it --name persistance_container_2 python-docker-example:persistance
What's your name? name
```

Endurræsum síðan container-inn okkar og sjáum að bæði tilvikin eru keyrandi

```
PS C:\> docker start persistance_container_2
persistance_container_2
PS C:\> docker ps
CONTAINER ID   IMAGE                                  COMMAND              CREATED          STATUS          PORTS     NAMES
9459ab136747   python-docker-example:persistance      "python main.py"     44 seconds ago   Up 9 seconds              persistance_container_2
aa68e08b03b7   python-docker-example:persistance      "python main.py"     8 minutes ago    Up 5 minutes              persistance_container_1
```

```
PS C:\> docker exec -it persistance_container_1 cat data/names.txt
     _                _      _ _
  __| | ___   __| | __| (_)
 / _` |/ _ \ / _` |/ _` | |
| (_| | (_) | (_| | (_| | |
 \__,_|\___/ \__,_|\__,_|_|

 👍
PS C:\> docker exec -it persistance_container_2 cat data/names.txt
     _                _      _ _ ____
  __| | ___   __| | __| (_)___ \
 / _` |/ _ \ / _` |/ _` | | __) |
| (_| | (_) | (_| | (_| | |/ __/
 \__,_|\___/ \__,_|\__,_|_|_____|

 👍
```

Sjáum að þeir eru með sín eigin aðskilin names.txt file, þetta er af því hver container hefur sitt eigið isolated filesystem

Vandamál:

- Ef container-arnir eyðast(sem mun gerast) þá munu names.txt gögnin glatast
- Hvernig getum við tryggt að það gerist ekki?
- Þ.e.a.s að þegar container start-ar upp þá notar hann sömu gögn og hafa verið vistuð áður af öðrum container-um

# Volumes

- Docker filesystem og host machine filesystem tenging
  - Volumes gera manni kleift að tengja specific filesystem paths í container-num við svokölluð *volumes* á host machine-inu
- Folder-ar og files eru mounted
  - Folders í container geta verið *mounted*(attached)af folder-um svokölluðum *volumes* í host machine-inu og þannig myndast mapping frá container til host machine-sins
- Breytingar verða þannig persistent á milli container-a
  - Breytingar í þessum folder-um/file-um í container-um rata þar af leiðandi til folder-a/file-a í host machine-inu og þegar nýr container er start-aður þá hefur hann aðgang að þessum breytingum líka.
- Tvær tegundir af volumes
  - Named volumes
  - Anonymous Volumes

# Named volumes


- Bucket of data
  - Hægt að hugsa um named volumes sem *bucket of data*
- Docker sér um vinnuna
  - Við biðjum docker um að búa til þessi volumes (þessu *bucket of data*) fyrir okkur
  - Docker sér um physical location-ið af þessum volume á host machine-inu
  - Eina sem við þurfum að muna er nafnið á volume-inu sem við bjuggum til
- Command til að búa til volume:
  - `docker volume create <volume-name>`
- Command til að mount-a volume
  - `docker run -v <volume-name>:<container-folder-name> <container-name>`

# Persisting data á milli container-a lausn frh:
# Named Volumes

Byrjum á því að búa til volume-ið okkar

```
PS C:\> docker  volume create names
names
```

Við getum séð details um volume-ið sem við vorum að búa til. T.d. Hvar volume-ið er geymt á host machine-inu

```
PS C:\> docker volume inspect names
[
    {
        "CreatedAt": "2021-09-20T18:26:12Z",
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/names/_data",
        "Name": "names",
        "Options": {},
        "Scope": "local"
    }
]
```

# Persisting data á milli container-a lausn frh:
# Named Volumes

Stoppum og eyðum container-unum okkar sem eru nú þegar til.

```
PS C:\> docker stop persistance_container_2 persistance_container_1
persistance_container_2
persistance_container_1
PS C:\> docker rm persistance_container_2 persistance_container_1
persistance_container_2
persistance_container_1
```

Keyrum container-ana okkar aftur alveg eins og áðan

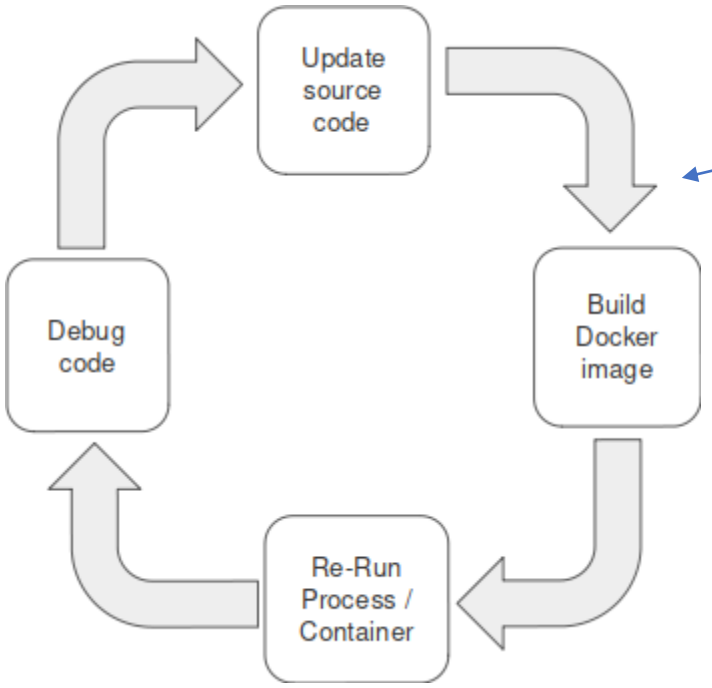
```
PS C:\> docker run -v names:/app/data --name persistance_container_1 -it python-docker-example:persistance
What's your name? doddi
PS C:\> docker run -v names:/app/data --name persistance_container_2 -it python-docker-example:persistance
What's your name? doddi2
PS C:\> docker start persistance_container_1
persistance_container_1
PS C:\> docker exec -it persistance_container_1 cat data/names.txt
    _                 _       _ _
 __| | ___   __| | __| (_)
/ _` |/ _ \ / _` |/ _` | |
| (_| | (_) | (_| | (_| | |
 \__,_|\___/ \__,_|\__,_|_|

 👍      _                 _       _ _ ____
 __| | ___   __| | __| (_)___ \
/ _` |/ _ \ / _` |/ _` | | __) |
| (_| | (_) | (_| | (_| | |/ __/
 \__,_|\___/ \__,_|\__,_|_|_____|

 👍
```

Við sjáum núna að gögnin eru vistuð einhvers staðar á sameiginlegum stað og ef við myndum eyða container-num okkar og start-a aftur þá myndu þessi gögn ennþá vera þar

# Bind Mounts

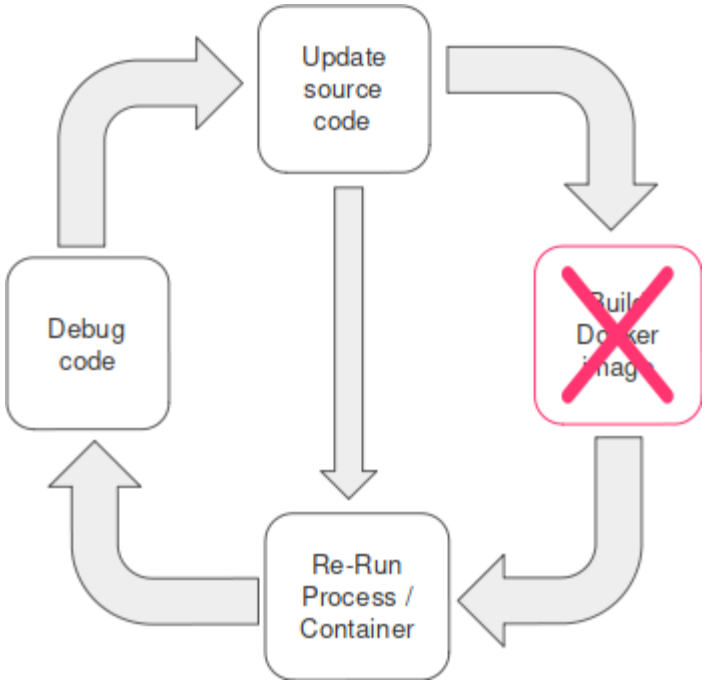

- Svipuð og volumes
  - Nema með bind mounts stjórnum við nákvæmlega mountpoint-inu á host machine-inu
- Notað til að provide-a auka gögn í container-inn
  - Við getum notað bind mounts til að persist-a gögn eins og með Named Volumes en þetta er yfirleitt notað til að provide-a auka gögn í container-inn okkar
  - **Við getum t.d. Notað Bind Mounts til að *mount*-a kóðann okkar inn í container-inn til að sjá kóða breytingar strax.** Þetta er mikilvægt við development því það væri mjög þreytandi að þurfa að endurbyggja image-ið fyrir hverja kóða breytingu
- Folder-ið þarf ekki að vera til á host machine-inu
  - Til að nota bind mounts þarf folder-ið á host machine-inu ekki að vera til nú þegar
  - Ef það er ekki til þá verður það búið til on demand
- Yfirleitt bara notað í development
  - Bind mounts eru yfirleitt eingöngu notuð við development.
  - Kóði á t.d. ekki að geta breyst í production
  - Oft bara notað til að gera development auðveldara
  - Í production ætti volumes frekar að vera notað

# Bind Mounts til að auðvelda development




Án bind mounts myndum við alltaf þurfa að build-a image-ið okkar aftur við hvert source code update, þetta myndi verða virkilega þreytandi

Við getum optimize-að development ferlið með því að nota bind mounts og þannig *mount-að* source kóðann okkar á local tölvunni okkar við source kóðann í container-num og þannig uppfært kóða í container-num þegar við uppfærum kóða locally hjá okkur



# Bind Mounts til að auðvelda development frh

Mount-um kóðann okkar í container-inn þegar við búum hann til

```
PS C:\Users\doddi\OneDrive - Reykjavik University\kennsludót\HONN-haust-2021\prufa\docker\1> docker run -v ${pwd}:/app:ro -v names:/app/data --name bind_mou
nts_container -it python-docker-example:persistance
```

${pwd} er syntax í windows powershell til að fá absoulte path-ið í terminal-inu, þetta mun vera öðruvísi eftir hvaða stýrikerfi/skel þú ert að nota, virkar líka að setja absolute path-ið beint inn

ro stendur fyrir read-only sem við látum fylgja með því við viljum ekki að container-inn geti yfirskriað neitt af kóðanum okkar (ro er óþarfi en góður siður að hafa)

```python
import emoji

import pyfiglet

name = input('What\'s your name? ')

with open('data/names.txt', 'a+', encoding='utf-8') as names_file:
    jazzed_name = pyfiglet.figlet_format(name)
    thumbs_up = emoji.emojize(':thumbsup:', use_aliases=True)
    names_file.write(f'{jazzed_name} {thumbs_up}')
    print(jazzed_name)
    print(thumbs_up)
```

Kóða breyting, prentum líka út nafnið og thumbs up

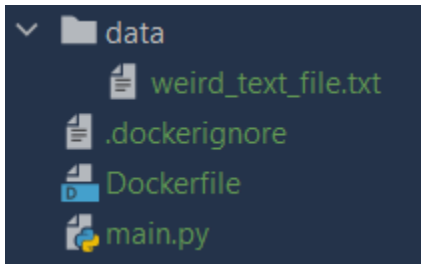
Núna án þess að þurfa að build-a image-ið þá fáum við kóða breytingar í gegn strax

```
PS C:\> docker start -ai bind_mounts_container
What's your name? doddi
      _                 _     _ _
   __| | ___   __| | __| (_)
  / _` |/ _ \ / _` |/ _` | |
 | (_| | (_) | (_| | (_| | |
  \__,_|\___/ \__,_|\__,_|_|


👍
```

# Anonymous volumes

- Anonymous
  - Anonymous volumes hafa ekkert nafn og við búum þau ekki til explicitly eins og  named volumes
- Við getum skilgreint anonymous volumes í Dockerfile
  - `VOLUME [<path-in-container>]`
- Fyrir ákveðinn container
  - Anonymous volumes eru búin til fyrir ákveðinn container
  - Ekki hægt að share-a á milli container-a
  - Anonymous volumes eru bara til á með container-inn er til
- Notkunardæmi
  - Anonymous volumes geta verið **gagnleg fyrir gögn sem eiga bara að lifa eins lengi og container-inn sjálfur**, t.d. einhver temp gögn geta verið geymd sem anonymous volumes fyrir auka performance boost.
  - Anonymous volumes geta verið mjög **gagnleg til að láta bind mounts ekki yfirskrifa eitthvað í container-num**, t.d. Ef við vorum búin að copy-a eða install-a einhverju í Dockerfile-num okkar og viljum ekki yfirskrifa það með bind mounts.

# Anonymous volume með Bind Mounts dæmi




Local weird_text_file.txt

```
weird local data
```

.dockerignore

```
Data
```

Dockerfile

```
# syntax=docker/dockerfile:1

FROM python:3.8-slim-buster

WORKDIR /app

RUN mkdir data
RUN echo 'weird container data' > "/app/data/weird_text_file.txt“

COPY . .

CMD ["python", "main.py"]
```

Búum til okkar eigið data og weird_text_file.txt í Dockerfile

main.py

```
with open('data/weird_text_file.txt') as weird_text_file:
    print(weird_text_file.readline())
```

# Anonymous volume með Bind Mounts dæmi frh.

Build-um image-ið okkar

```
docker build --tag anon_vol_test .
```

```
PS C:\Users\doddi\OneDrive - Reykjavik University\kennsludót\HONN-haust-2021\prufa\docker\anon_vol> docker run -v ${pwd}:/app:ro -it anon_vol_test
weird local data
```

Ef við keyrum container upp með bind mounts núna þá yfirskrifum við weird_text_file.txt sem við skilgreindum í image-inu með local weird_text_file.txt

# Anonymous volume með Bind Mounts dæmi frh.

Við getum skilgreint Anonymous volume í docker file-num okkar með nákvæmara path-i fyrir /app/data og það mun því vera notað í staðinn fyrir ónákvæmara path-ið sem er specify-að í bind mountinu(/app). Anonymous volume-ið er líka hægt að skilgreina með docker CLI eins og við erum búin að sjá með named volumes og bind mounts

```
# syntax=docker/dockerfile:1

FROM python:3.8-slim-buster

WORKDIR /app

RUN mkdir data
RUN echo 'weird container data' > "/app/data/weird_text_file.txt“

COPY . .

VOLUME ["/app/data"]

CMD ["python", "main.py"]
```

Build-um image-ið aftur

```
docker build --tag anon_vol_test .
```

Keyrum container upp aftur með sömu skipunum

```
PS C:\Users\doddi\OneDrive - Reykjavik University\kennsludót\HONN-haust-2021\prufa\docker\anon_vol> docker run -v ${pwd}:/app:ro -it anon_vol_test
weird container data
```

Núna sjáum við að við yfirskrifum ekki weird_text_file.txt með local gögnunum heldur notum við það sem við skilgreindum í Dockerfile

# Bind Mounts vs Volumes

| Bind Mounts | Volumes |
|---|---|
| Stjórnum nákvæmlega mountpoint á host machine | Látum docker sjá um að búa til volumes fyrir okkur |
| Folder-ið þarf ekki að vera til á host machine-inu ef það er ekki til þá er það búið til | Volumes þurfa að vera til nú þegar |
| Yfirleitt bara notað í development | Notað bæði í development og production |
| Ekki fullkomlega óháð stýrikerfinu því við þurfum að sjá um að gefa nákvæmt mountpoint | Docker sér um volumes, þess vegna eru volumes óháð stýrikerfinu eða directory structure, volumes mynda þannig ákveðið abstraction |
| Bind mounts verða að vera á host-machine | **Volume drivers leyfa okkur að geyma volumes á remote hosts** eða í cloud providers. |

Bind mounts gefa aðgang að sensitive data -> hægt að breyta host machine filesystem-inu í gegnum container-a

Preferred leiðin til að persist-a data

Auðveldara að back-a upp volumes en bind mounts

Hægt að manage-a volumes með Docker CLI

Hægt að deila á milli margra container-a með meira öryggi

Ný volumes geta verið pre-populated af container

# COPY vs Bind Mounts

Einhver er kannski að velta fyrir sér af hverju við þurfum COPY ennþá þegar við erum með Bind Mounts

Ástæðan fyrir því að við erum ennþá með COPY er af því Bind Mounts(eins og ég sýndi það) er eingöngu fyrir development. Þegar við erum á production server þá viljum nota COPY því það er ekkert source code sem er að uppfærast á production server-num. Við viljum hafa Snapshot af kóðanum okkar á production.

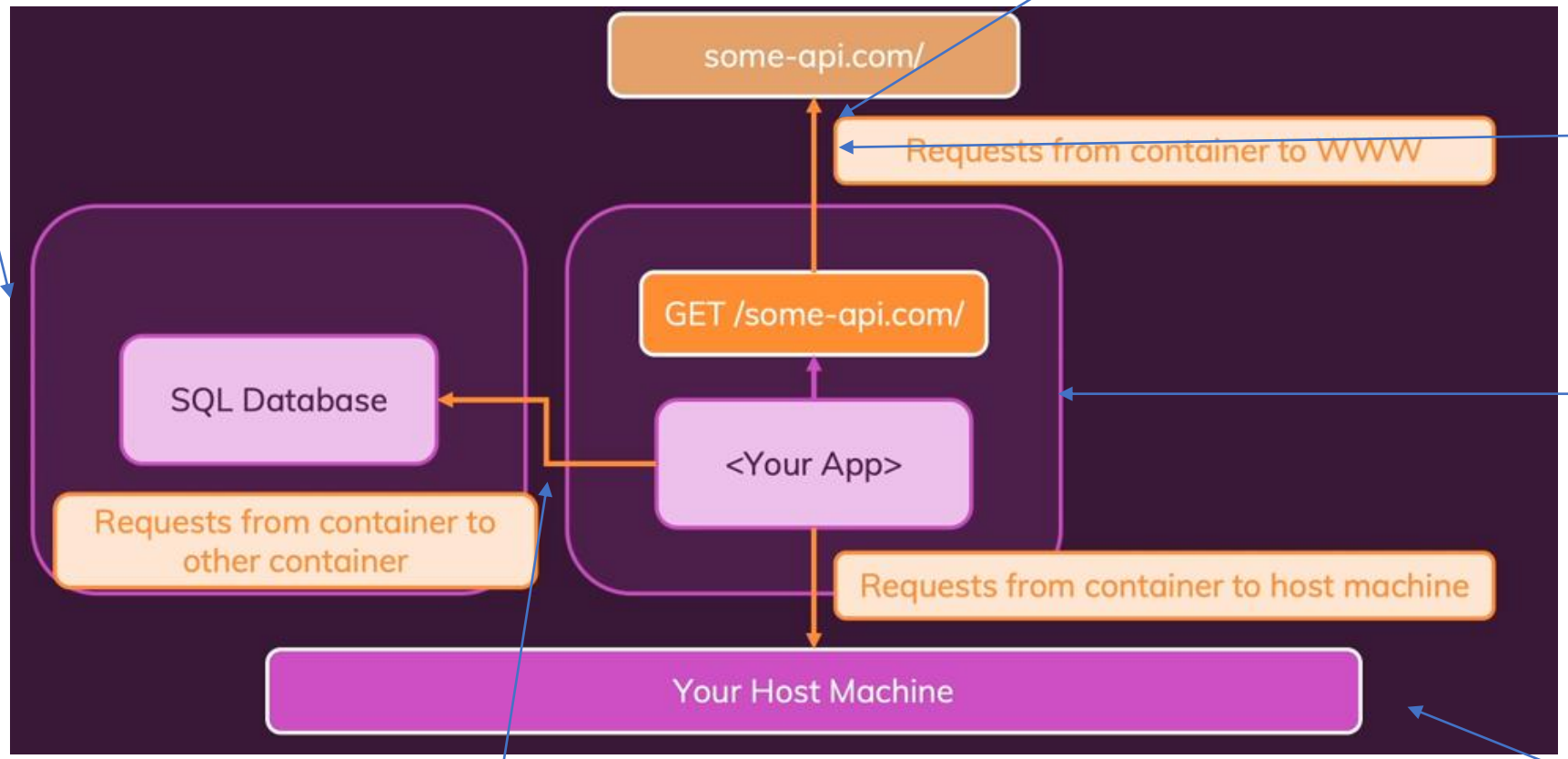
# Multi container apps

- Flest application samanstanda af mörgum container-um
  - T.d. framend, bakendi, background keyrslur
- “each container should do one thing and do it well”
  - Góðar líkur að þú þarft að scale-a upp mismunandi component-a öðruvisi, t.d. þarftu kannski að scale-a framenda og bakenda öðruvísi
  - Hægt að version-a og update-a í isolation

# Container communication

Communication við netið (WWW) virkar out of the box

SQL gagnagrunnurinn er í sér container




**Nr 1.** Container þarf oft að geta haft samskipti við ákveðin resources í gegnum netið (WWW)

Appið okkar er í sér container

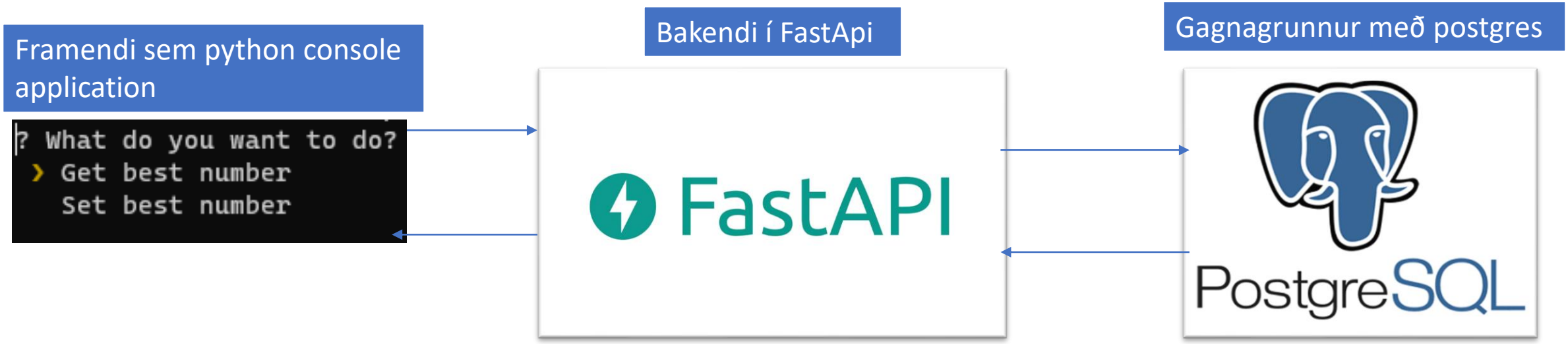
**Nr 2.** Container þarf kannski að geta haft samskipti við einhver resources á local vélinni sem container-inn keyrir í (t.d. ef við erum að test-a á dev)

**Nr 3.** Container þarf oft að tala við aðra container-a (cross-container communication)

# Container Communication og Multi Container App

## dæmi

Application-ið sem við munum nota er mjög einfalt kerfi sem vistar og sækir eina tölu(*best number*). Framendinn er gerður sem python console app, bakendinn í **FastApi** og gagnagrunnurinn er **Postgres.** Við byrjum með undockerized version-> https://github.com/thordurf-ru/multi_container_app/releases/tag/v1.0



# Dæmi frh. Backend containerization

```
# syntax=docker/dockerfile:1
FROM python:3.8-slim-buster

WORKDIR /app

COPY requirements.txt requirements.txt

RUN pip install -r requirements.txt

COPY . .

EXPOSE 8000

CMD ["python", "main.py"]
```

Kóðann eftir backend containerization-ið má finna: https://github.com/thordurf-ru/multi_container_app/releases/tag/v2.0

# Backend Containerization vandamál

Build-um image-ið

```
docker build --tag best-number-backend .
```

Reynum að keyra upp container eins og vanalega

```
docker run best-number-backend
```

Þegar við reynum að start-a container-num okkar þá fáum við eftirfarandi error

```
Traceback (most recent call last):
  File "main.py", line 21, in <module>
    conn = psycopg2.connect(
  File "/usr/local/lib/python3.8/site-packages/psycopg2/__init__.py", line 122, in connect
    conn = _connect(dsn, connection_factory=connection_factory, **kwasync)
psycopg2.OperationalError: could not connect to server: Connection refused
        Is the server running on host "localhost" (127.0.0.1) and accepting
        TCP/IP connections on port 5432?
could not connect to server: Cannot assign requested address
        Is the server running on host "localhost" (::1) and accepting
        TCP/IP connections on port 5432?
```

Villan kemur af því við erum að reyna að tala við gagnagrunn sem er keyrandi á localhost(127.0.0.1) á port-i 5432. Vandamálið er að Container-inn hefur sínar eigin hugmyndir um hvað „localhost“ og port 5432 er.

# Backend Containerization lausn

Við getum lagað vandamálið með því að breyta „localhost“ eða „127.0.0.1“ í kóðanum okkar með special „docker domain“ sem docker skilur og map-ar yfir í ip address-ið á host-machine-inu okkar. Þetta special domain er **host.docker.internal**

```
conn = psycopg2.connect(
    host='host.docker.internal',
    database='best_number',
    user='postgres',
    password='password’,
    port='5432'
)
```

Build-um image-ið aftur

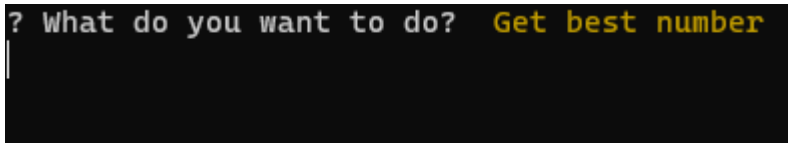
```
docker build --tag best-number-backend .
```

Núna startar container-inn og bakendinn

```
PS C:\Users\doddi\Desktop\multi_container_app\2-best-number-backend> docker run best-number-backend
INFO:     Will watch for changes in these directories: ['/app']
INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
INFO:     Started reloader process [1] using statreload
INFO:     Started server process [9]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```

# Backend Containerization vandamál 2

Núna erum við þó að lenda í ákveðnu vandamáli sem er að framendinn okkar nær ekki að tala við bakendann okkar og frýs þar afleiðandi




Ástæðan er að framendinn býst við að bakendinn er keyrandi á porti 8000 en container-inn er með sitt eigið port 8000 sem mappast ekki sjálfkrafa á local machine-ið

# Backend Containerization lausn 2

Til að leysa þetta port vandamál þá getum við keyrt container-inn upp með sérstöku flag-i þ.e.a.s –p <host port> <container port>

```
docker run -p 8000:8000 best-number-backend
```

```
? What do you want to do?  Get best number
12
```

Núna náum við að tengjast við bakendann og sækja *best number*

# Dæmi frh. Fronted Containerization

```
# syntax=docker/dockerfile:1
FROM python:3.8-slim-buster

WORKDIR /app

COPY requirements.txt requirements.txt

RUN pip install -r requirements.txt

COPY . .

CMD ["python", "main.py"]
```

Kóðann eftir frontend containerization-ið má finna: https://github.com/thordurf-ru/multi_container_app/releases/tag/v3.0

# Frontend Containerization vandamál

Build-um image-ið

```
docker build -t best-number-frontend .
```

Keyrum container-inn

```
PS C:\Users\doddi\Desktop\docker\multi_container_app\1-best-number-frontend> docker run -it best-number-frontend
? What do you want to do?  Get best number
Traceback (most recent call last):
  File "/usr/local/lib/python3.8/site-packages/urllib3/connection.py", line 174, in _new_conn
    conn = connection.create_connection(
  File "/usr/local/lib/python3.8/site-packages/urllib3/util/connection.py", line 96, in create_connection
    raise err
  File "/usr/local/lib/python3.8/site-packages/urllib3/util/connection.py", line 86, in create_connection
    sock.connect(sa)
ConnectionRefusedError: [Errno 111] Connection refused
```

Vandamálið í þetta skiptið er að núna er framendinn í sínum eigin container og hefur því sínar eigin hugmyndir um hvað „localhost“ er þannig þegar við reynum að kalla á GET fyrir http://localhost:8000/bestnumber inni í container-num þá erum við ekki lengur að tala við local host machine-ið sem keyrir container-inn

# Frontend Containerization Lausn 1

Ein (naive) lausn til að laga vandamálið er að nota ip töluna sem container-inn sem við erum að reyna að tala við er með(ip talana á bakend container-num), við getum notað command-ið: `docker container inspect <container>`

```
$ docker container inspect clever_lamarr | grep IPAddress
            "SecondaryIPAddresses": null,
            "IPAddress": "172.17.0.2",
                    "IPAddress": "172.17.0.2",
```

Container ip address af container-num „ clever_lamarr“ sem er bakenda container-inn okkar

```
if answers[ACTION] == GET_BEST_NUMBER:
    response = requests.get('http://172.17.0.2:8000/bestnumber')
    print(response.text)


elif answers[ACTION] == SET_BEST_NUMBER:
    response = requests.post('http://172.17.0.2:8000/bestnumber', json={'bestNumber': int(answers[BEST_NUMBER])})
    if response.status_code == 200:
        print('Best number saved')
    else:
        print('Failed to save best number')
```

Skiptum localhost út fyrir 172.17.0.2

Þá fáum við réttar niðurstöður

```
? What do you want to do?  Get best number
12
```

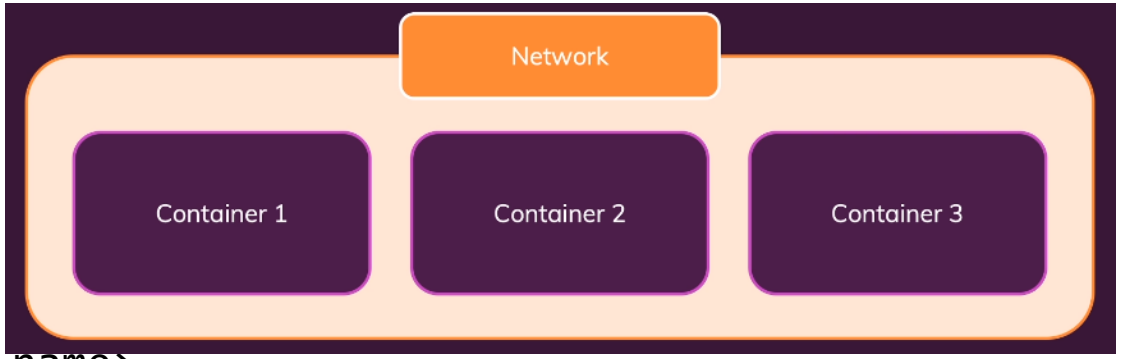
# Frontend Containerization Lausn 1 vandamál


- Þurfum alltaf að keyra `container inspect <container>` til að finna ip-ið á container-num

- Þetta ip mun vera öðruvísi fyrir hvern nýjan container

# Networks

- Docker networks er *rétta* leiðin til að látta container-a tala saman
- Tveir container-ar geta eingöngu haft samskipti sín á milli ef þeir eru á sama networki
- Ip address-ur eru automatically resolved.
  - Með networks þá í staðinn fyrir að gera eitthvað eins og http://localhost/bestnumber eða host.docker.internal þá myndum við nota nafn-ið á container-num sem við erum að reyna að tala við sem domain-ið t.d. þá http://**best-number-backend**/bestnumber
- Docker default network
  - Þegar við búum til container þá fer það sjálfkrafa á network sem heitir “bridge”
  - Ástæðan af hverju fyrra dæmið okkar virkaði með að manually plögg-a inn container ip tölunum er af því báðir container-arnir eru á þessu sama “bridge” network-i og þessar ip tölur eru ip tölur container-arana fyrir það network.
- Commands
  - `docker network create <network-name>`
    - Býr til nýtt network
  - `docker network ls`
    - Prentar út lista af öllum network-um
  - `docker network inspect <network-name>`
    - Prentar út details af ákveðnu networki
  - `Docker run --network <network-name> <image-name>`
    - Býr til container sem er í ákveðnu networki



# Frontend Containerization Lausn 2


Búum til nýtt network

```
docker network create best-number-network
```

ATH að við þurfum ekki lengur að taka map-a port-ið á local machine port-ið okkar því samskiptin fara öll núna fram í gegnum docker network-ið

Búum til og keyrum bakenda container-inn okkar með nafni og á network-inu okkar

```
docker run --name best-number-backend --network best-number-network best-number-backend
```

Breytum domain-inu í container nafnið

```
if answers[ACTION] == GET_BEST_NUMBER:
    response = requests.get('http://best-number-backend:8000/bestnumber')
    print(response.text)


elif answers[ACTION] == SET_BEST_NUMBER:
    response = requests.post('http://best-number-backend:8000/bestnumber', json={'bestNumber': int(answers[BEST_NUMBER])})
    if response.status_code == 200:
        print('Best number saved')
    else:
        print('Failed to save best number')
```

Keyrum upp framend-ann á sama network-i

```
PS C:\> docker run -it --name best-number-frontend --network best-number-network best-number-frontend
? What do you want to do?  Get best number
12
```

# Database containerization

Dockerfile

```
FROM postgres

ENV POSTGRES_USER "postgres"
ENV POSTGRES_PASSWORD "password"
ENV POSTGRES_DB "best_number"

EXPOSE 5432

COPY init.sql /docker-entrypoint-initdb.d/
```

init.sql

```
CREATE TABLE public.best_number
(
    "number" integer,
    created timestamp with time zone
)

TABLESPACE pg_default;

ALTER TABLE public.best_number
    OWNER to postgres;
```

Kóðann eftir frontend containerization-ið má finna: https://github.com/thordurf-ru/multi_container_app/releases/tag/v4.0

Build-um image-ið

```
docker build -t best-number-db .
```

Keyrum container-inn

Fáum persistance á milli container-a með volume

```
docker run --name best-number-db --network best-number-network -v best-number-db-data:/var/lib/postgresql/data best-number-db
```

# Database containerization frh.

```
conn = psycopg2.connect(
    host='best-number-db',
    database='best_number',
    user='postgres',
    password='password',
    port='5432'
)
```

Breyta domain-inu í nafnið á db container-num í bakendanum

Re-build-a og endurræsa bakendanum

```
docker build -t best-number-backend .
```

```
docker run --name best-number-backend --network best-number-network best-number-backend
```

Og kerfið okkar virkar þá eðlilega

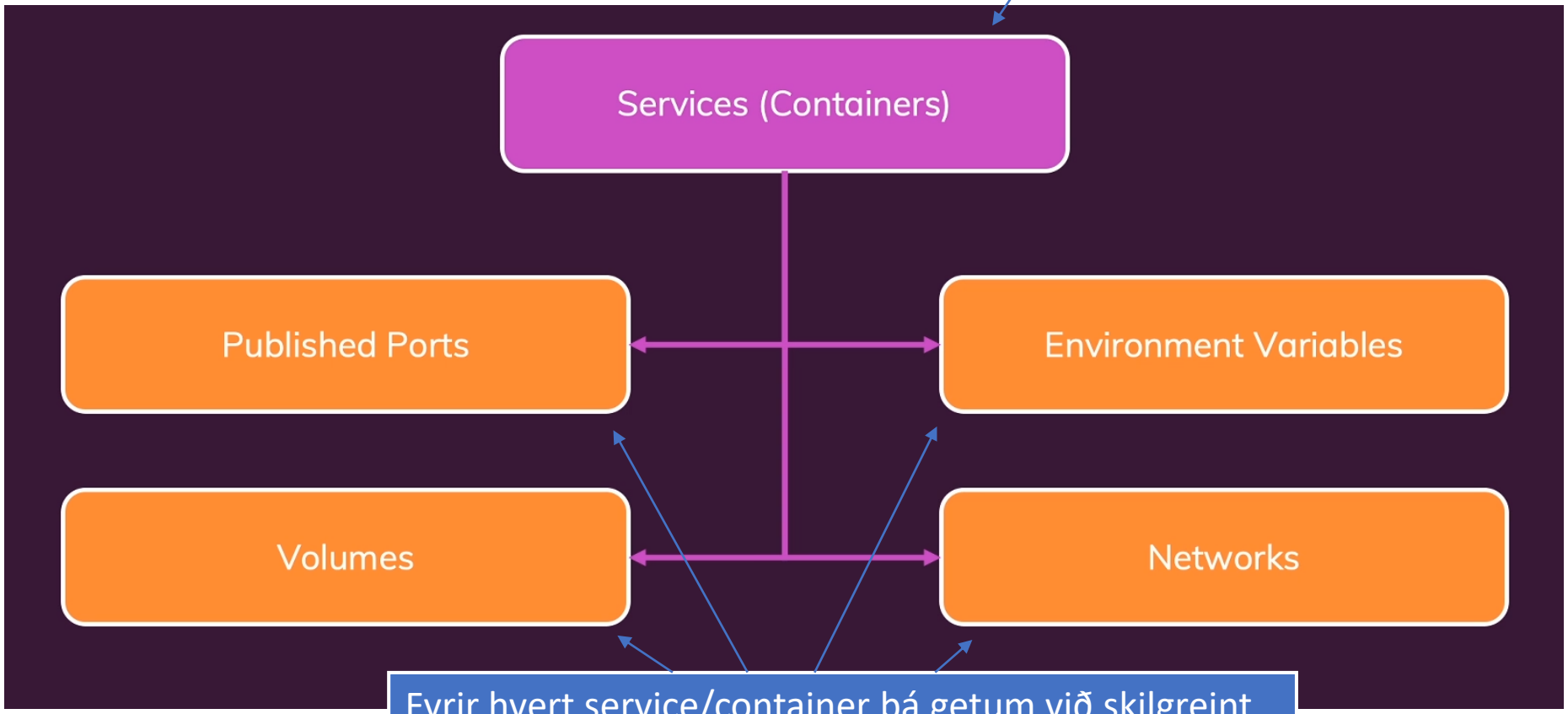
```
PS C:\> docker run -it --name best-number-frontend --network best-number-network best-number-frontend
? What do you want to do?  Set best number
? Enter you favorite number:   10
Best number saved
PS C:\> docker rm best-number-frontend
best-number-frontend
PS C:\> docker run -it --name best-number-frontend --network best-number-network best-number-frontend
? What do you want to do?  Get best number
10
```

# Docker Compose


- Erfitt líf hingað til að starta upp kerfinu
  - Hingað til höfum við þurft að start-a alla container-ana okkar manually með löngum run skipunum.
  - Það er til betri leið til að start-a upp multi-container application og það er með *docker compose*
- Docker Compose
  - Compose er tól til að skilgreina og keyra multi container applications.
  - Með Compose notum við YAML file til að configure-a öll application service-in okkar
  - Við getum síðan startað öllu kerfinu upp með einni stuttri command skipun.
  - Compose notar project names til að isolate-a umhverfi frá hvort öðru (t.d. fyrir dev, staging, prod etc.)
- Endurnýtir container-a
  - Docker compose cache-ar configuration-ið notað til að búa til container-ana. Þegar þú restart-ar service-i sem hefur ekki breyst þá endurnotar docker compose container-a sem eru til nú þegar.
- Mjög gagnlegt í development umhverfi
  - Gerir auðvelt að start-a upp og stoppa allt application-ið sem getur verið mjög hentugt á meðan maður er að þróa application-ið
- Gott Documentation
  - Compose gefur mannig gott documentation um allt kerfið og hvernig það er configure-að
- Ekki ætlað til að manage-a multiple containers á mismunandi host machines
  - Getur því verið vesen í production þegar þú þarft multiple host machines (Kubernetes er yfirleitt notað í staðinn fyrir þannig þarfir)

# Docker Compose frh.

Docker Compose file samanstendur af configuration-um fyrir services/containers (t.d. Fyrir framenda service, bakenda service og gagangrunn)




Fyrir hvert service/container þá getum við skilgreint configuration fyrir það t.d. Ports, volumes, environment varaibles, networks etc. alveg eins og þegar við keyrum docker run ...

docker-compose.yaml

```yaml
version: "3.8"
services:
  best-number-frontend:
    build: ./1-best-number-frontend
    stdin_open: true
    tty: true
    depends_on:
      - best-number-backend

  best-number-backend:
    build: ./2-best-number-backend
    depends_on:
      - best-number-db

  best-number-db:
    image: postgres
    environment:
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=password
      - POSTGRES_DB=best_number
    volumes:
      - best-number-db-data:/var/lib/postgresql/data
      - ./3-best-number-db:/docker-entrypoint-initdb.d/

volumes:
  best-number-db-data:
```

Stdin_open er eins og –i í run skipuninni og tty er eins og –t í run skipuninni saman mynda þau því –it flagg-ið

depend_on segir að þetta service ætti að vera start-að eftir að öll service-in sem það „depend“-ar á eru komin upp

Við tókum út dockerfile-inn fyrir gagnagruninn og færðum alla logic-ina yfir í compose file-ið í staðinn

Við þurfum að taka fram named volumes hérna líka

Við skilgreinum service-in okkar, service nöfnin er það sem við getum notað sem domain fyrir cross-container communication

Fyrir hvert service getum við skilgreint staðsetninguna á dockerfile-num fyrir það service þannig ef það þarf að build-a image-ið þá er það gert

**NOTE:** Compose setur öll service-in í sama network by default

# Docker Compose frh.

- Yfirleitt start-ar maður upp öllu kerfinu með `docker-compose up` eða með `docker-compose up –d` til að gera það í detached mode og fá ekki upp container startup log-in

- En í okkar dæmi þá þurfum við að nota skipunina `docker-compose run --rm best-number-frontend`

```
$ docker-compose run --rm best-number-frontend
Starting multi_container_app_best-number-db_1 ... done
Starting multi_container_app_best-number-backend_1 ... done
Creating multi_container_app_best-number-frontend_run ... done
? What do you want to do?  Get best number
10
```

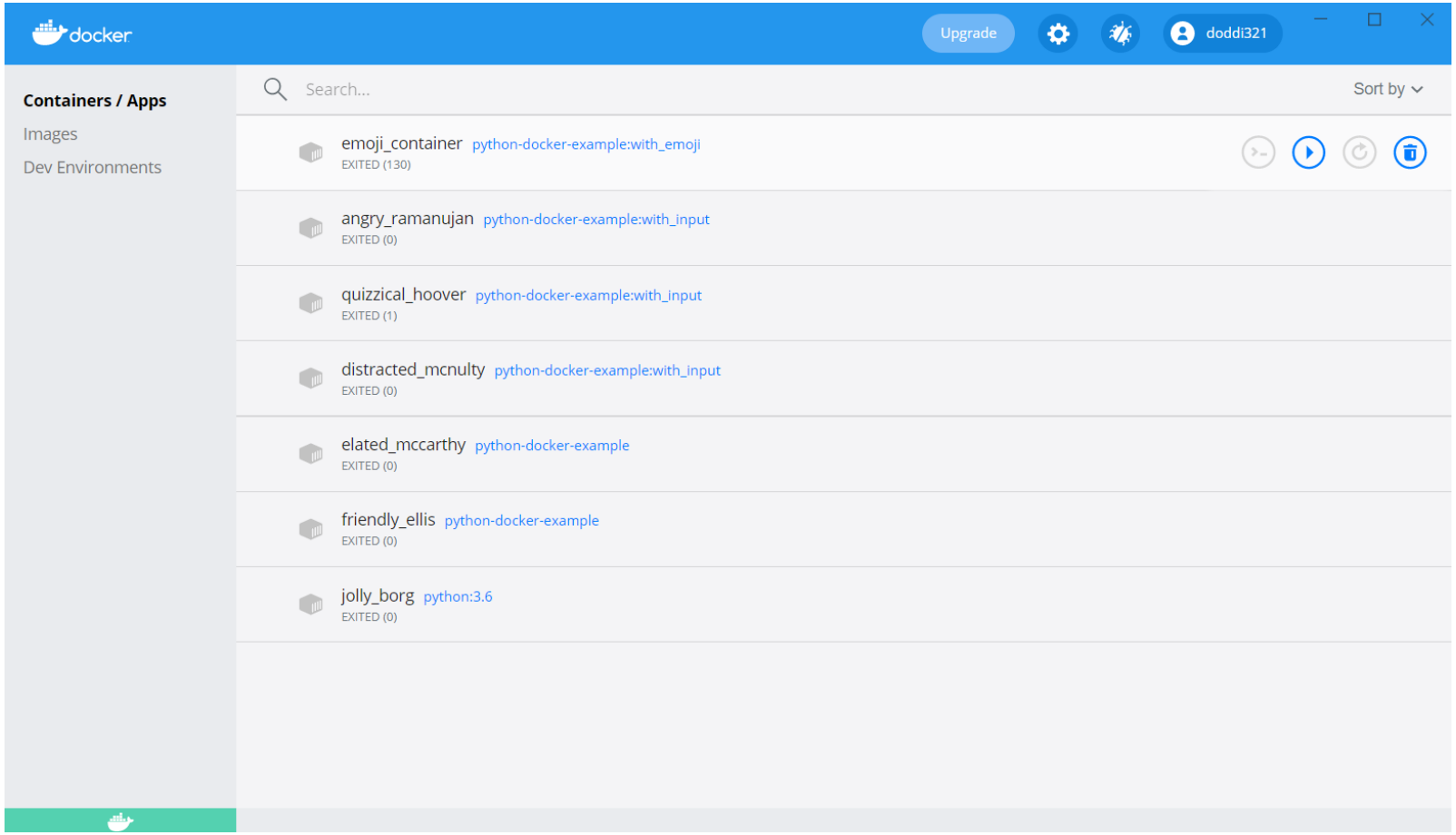
Loka niðurstöðu má sjá hér: https://github.com/thordurf-ru/multi_container_app/releases/tag/v5.0

# Docker og Databases


- Database Containerization
  - Eins og við höfum séð þá getum við manage-að okkar eigin Database containers og notað volumes fyrir persistance
  - Kostir
    - Isolated umhverfi
    - Auðvelt að setja allt kerfið með t.d. Docker-compose
    - Getur gert development mun auðveldara
    - Getum sett upp sandbox umhverfi fyrir kerfið okkar
  - Gallar
    - Scaling getur orðið erfiðara (ekki bara hægt að duplicate-a container-ana)
    - Managing availability getur verið erfiðara
    - Performance getur orðið vandamál (sérstaklega útaf traffic spikes)
    - Backups og security getur verið erfiðara
    - **Almennt ekki mælt með að hafa gagnagrunn í container í production**
- Alternative approach: **Managed Database Services**
  - Managed database services eru þjónustur á netinu sem bjóða upp á gagnagrunns hýsingu og sjá um erfiðu hlutina fyrir okkur eins og **security, scaling, availability, backups etc.**
  - Dæmi: https://cloud.google.com/sql, https://aws.amazon.com/free/database/ , https://www.digitalocean.com/try/managed-databases-postgresql , https://azure.microsoft.com/en-us/services/postgresql/, https://www.heroku.com/postgres , https://www.elephantsql.com/
  - Getur verið gott að keyra gagnagrunn í docker í development umhverfi en síðan nota Managed Database Services í production en getum líka auðvitað notað Managed Database Services bæði í development og í production

# Docker Desktop

GUI tól fyrir docker sem getur hjálpað okkur með containers og images og einnig development environment deployments af Swarm eða Kubernetes

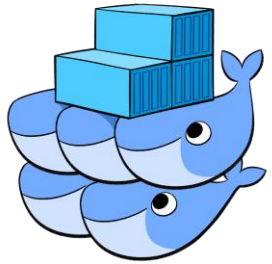


# Ecosystem

Það er heilt ecosystem af products sem styðja við og/eða eru byggð í kringum docker








...

# Container Orchestration

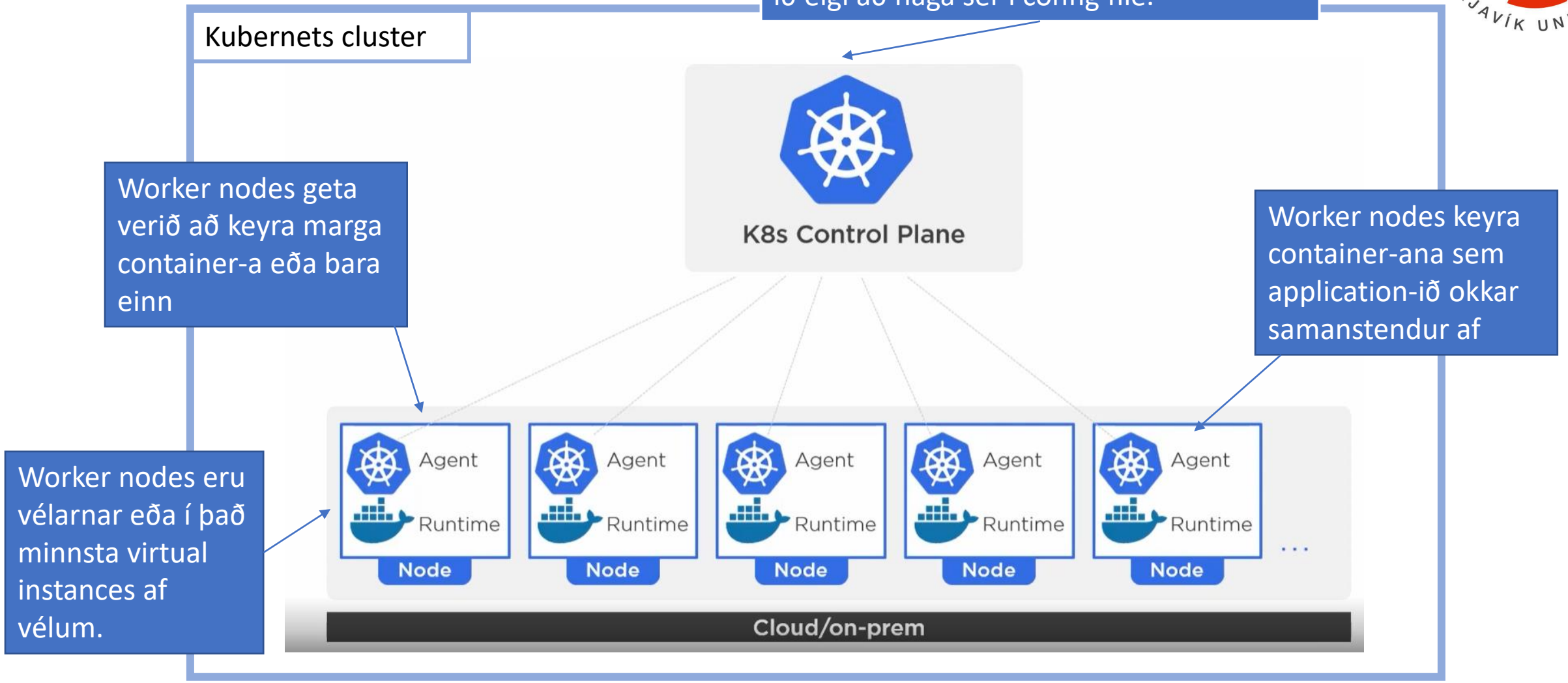
- Orchestrators
  - Tól sem manage-a, scale-a og maintain-a containarized applications eru kölluð *orchestrators*
- Hvað gera þau
  - Þessi tól automate-a management og maintanance á application container-um
  - Geta replace-að failed containerum með nýjum automatically
  - Geta scale-að application-ið upp með því að búa til ný tilvik af container-um
- Helstu dæmi um orchestration tól eru
  - Docker Swarm
  - Kubernetes
  - Margir cloud service providers bjóða upp á einfaldar lausnir fyrir þessa hluti einnig t.d. autoscaling, health checking og auto-redeployment, load balancers etc. (en þetta er yfirleitt ekki kallað orchestration tól því þetta býður upp á mjög limit lausnir og læsir okkur við það cloud service)

# Hvað er Kubernetes?


- Kubernetes er portable, extensible open-source platform til að manage-a containerized workloads og services.
- Kubernetes er gríslt orð sem þýðir stýrismaður
- Oft kallað K8s sem stytting (kemur frá því að það eru 8 stafir á milli k og s í orðinu Kubernetes)
- Google Open-Source-aði Kubernetes árið 2014
- Kubernetes býður upp á framework til að keyra distributed kerfi resiliently.
- Kubernetes getur séð um
  - Deployment patterns
    - Getur t.d. Manage-að canary deployment strategy-una sem er það að release-a application-inu incremenatlly til subset af notendum
  - Scaling
    - Getur horizontal scale-að application-ið okkar með því að bæta við fleiri container-um
  - Failover og self-healing
    - Getur restart-að container-um, kill-að þá, replace-að þá þegar þeir fail-a
  - Service discovery
    - Kubernetes getur expose-að conainer með DNS nafni eða þeirra eigin IP addressu
  - Load balancing
    - Kubernetes getur distribute-að network traffic ef traffic er mjög há
  - Storage orchestration
    - Kubernetes leyfir okkur að automatically mount-a storage system t.d. Local storages, cloud providers etc.
  - Automated rollouts og rollbacks
  - Automatic bin packing
    - Við getum sagt kubernetes hversu mikið CPU og memory hver container þarf, kubernetes getur þá fit-að container-ana á worker nodes til að nýta resources sem best.
  - Secret og configuration management
    - Kubernetes leyfir okkur að manage-a sensitive upplýsingar eins og password, Oauth tokens, SSH lykla etc. Við getum deploy-að og uppfært secrets og application configurtaion án þess að þurfa að rebuilda container image-in.
- https://kubernetes.io/

# Kubernetes

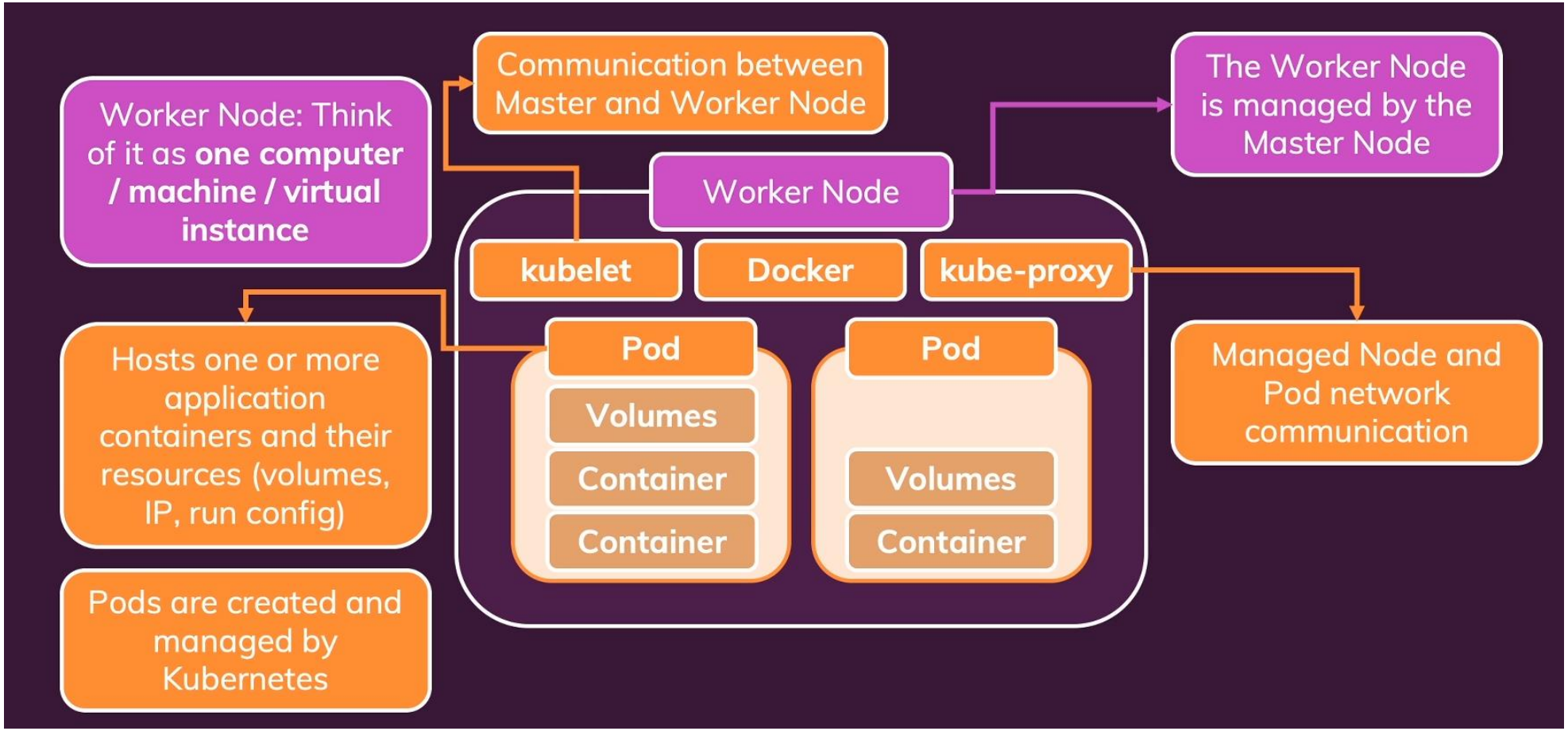
Control panel-ið eða *master node* stjórnar worker nodes og deployment á þeim eftir því hvernig við specify-um að application-ið eigi að haga sér í config file.

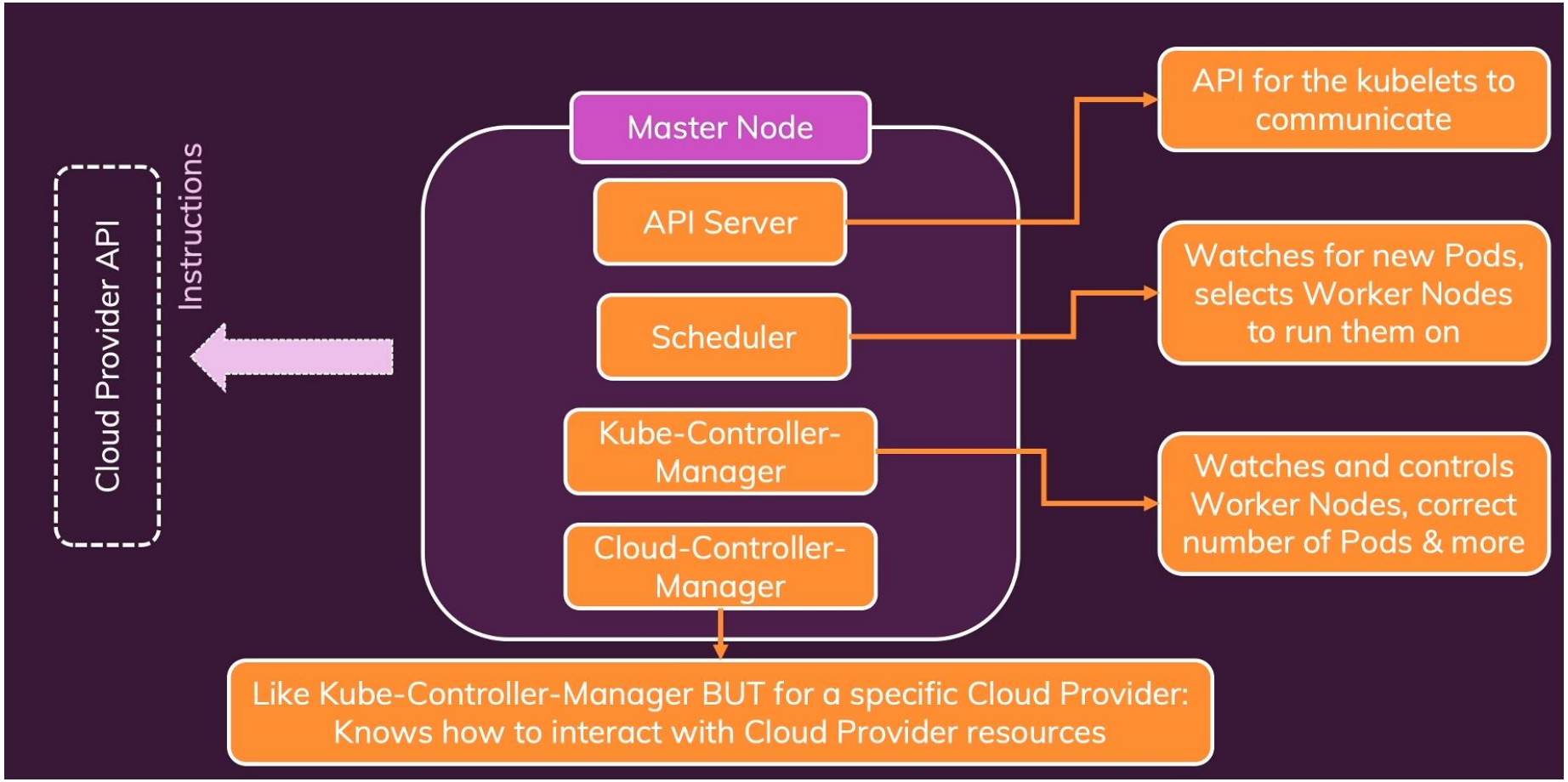



Worker nodes geta verið að keyra marga container-a eða bara einn

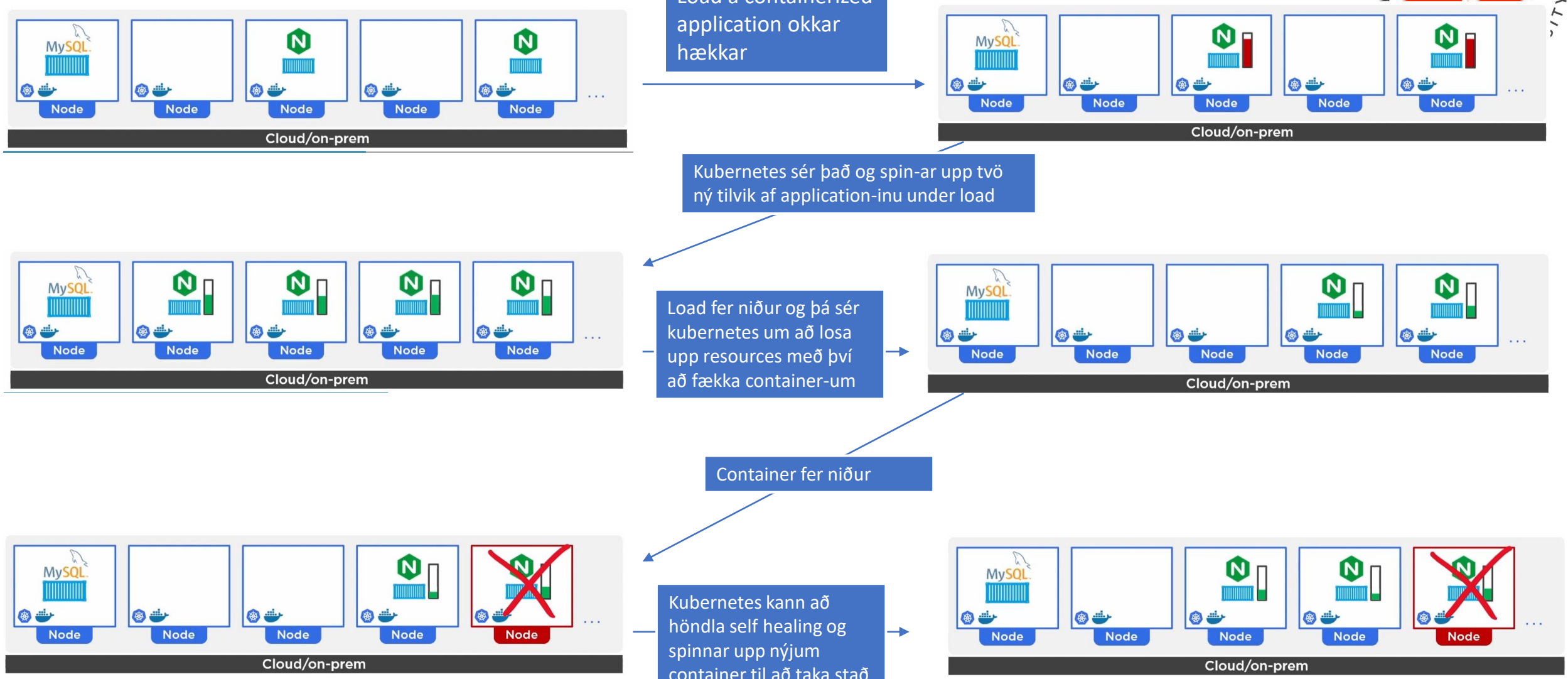
Worker nodes keyra container-ana sem application-ið okkar samanstendur af

Worker nodes eru vélarnar eða í það minnsta virtual instances af vélum.


Worker Node: Think of it as one computer / machine / virtual instance
Communication between Master and Worker Node
The Worker Node is managed by the Master Node
Worker Node
kubelet
Docker
kube-proxy
Pod
Volumes
Container
Container
Pod
Volumes
Container
Hosts one or more application containers and their resources (volumes, IP, run config)
Managed Node and Pod network communication
Pods are created and managed by Kubernetes


Master Node
API Server
Scheduler
Kube-Controller-Manager
Cloud-Controller-Manager
Cloud Provider API
Instructions
API for the kubelets to communicate
Watches for new Pods, selects Worker Nodes to run them on
Watches and controls Worker Nodes, correct number of Pods & more
Like Kube-Controller-Manager BUT for a specific Cloud Provider: Knows how to interact with Cloud Provider resources


What Kubernetes Will Do
What You Need To Do / Setup
(i.e. what Kubernetes requires)
Create your objects (e.g. Pods) and manage them
Create the Cluster and the Node Instances (Worker + Master Nodes)
Monitor Pods and re-create them, scale Pods etc.
Setup API Server, kubelet and other Kubernetes services / software on Nodes
Kubernetes utilizes the provided (cloud) resources to apply your configuration / goals
Create other (cloud) provider resources that might be needed (e.g. Load Balancer, Filesystems)


Load á containerized application okkar hækkar
Kubernetes sér það og spin-ar upp tvö ný tilvik af application-inu under load
Load fer niður og þá sér kubernetes um að losa upp resources með því að fækka container-um
Container fer niður
Kubernetes kann að höndla self healing og spinnar upp nýjum container til að taka stað þanns sem fór niður
MySQL
Node
Cloud/on-prem

Með kubernetes er auðvelt að migrate-a frá einni skýjaþjónustu til annara eða á on-prem server-a