Docker

A history of several fails and one win

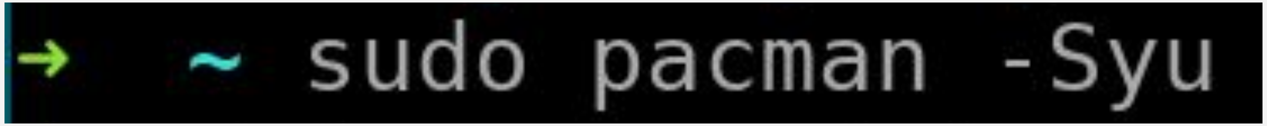
pictec

Why?

```
> one beautiful day OP wanted to install nvtop
```

Why?

> one beautiful day OP wanted to install nvtop

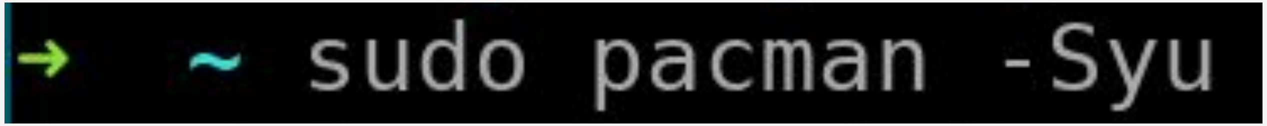


```
➜  ~ sudo pacman -Syu
```

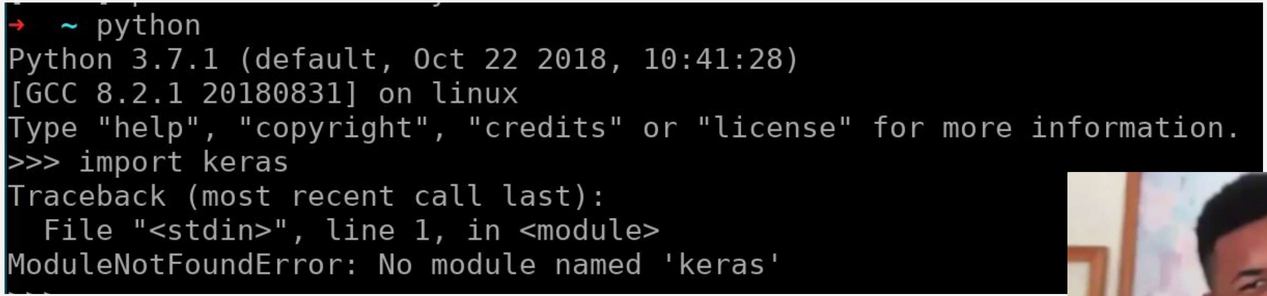
> multiple fails ensured

Why?

> one beautiful day OP wanted to install nvtop



```
➜  ~ sudo pacman -Syu
```



```
➜  ~ python
Python 3.7.1 (default, Oct 22 2018, 10:41:28)
[GCC 8.2.1 20180831] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import keras
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'keras'
```

But why not virtualenv?

Package manager bumps every software by a version.
In normal cases virtualenv works, but...
... my OS of choice decided to wipe all data related to old Python version

But why not virtualenv?

Package manager bumps every software by a version.
In normal cases virtualenv works, but...
... my OS of choice decided to wipe all data related to old Python version

Always use virtualenv before you actually encounter a problem

Docker is virtualenv for OS

Docker is virtualenv for OS

Well, not really, but there are some analogies...

`requirements.txt`	`Dockerfile`
`pip install -r requirements.txt`	`docker build -t my-image .`
`source bin/activate`	`docker run my-container`
`python`	`docker container exec -it my-container python3`

Basic docker terminology

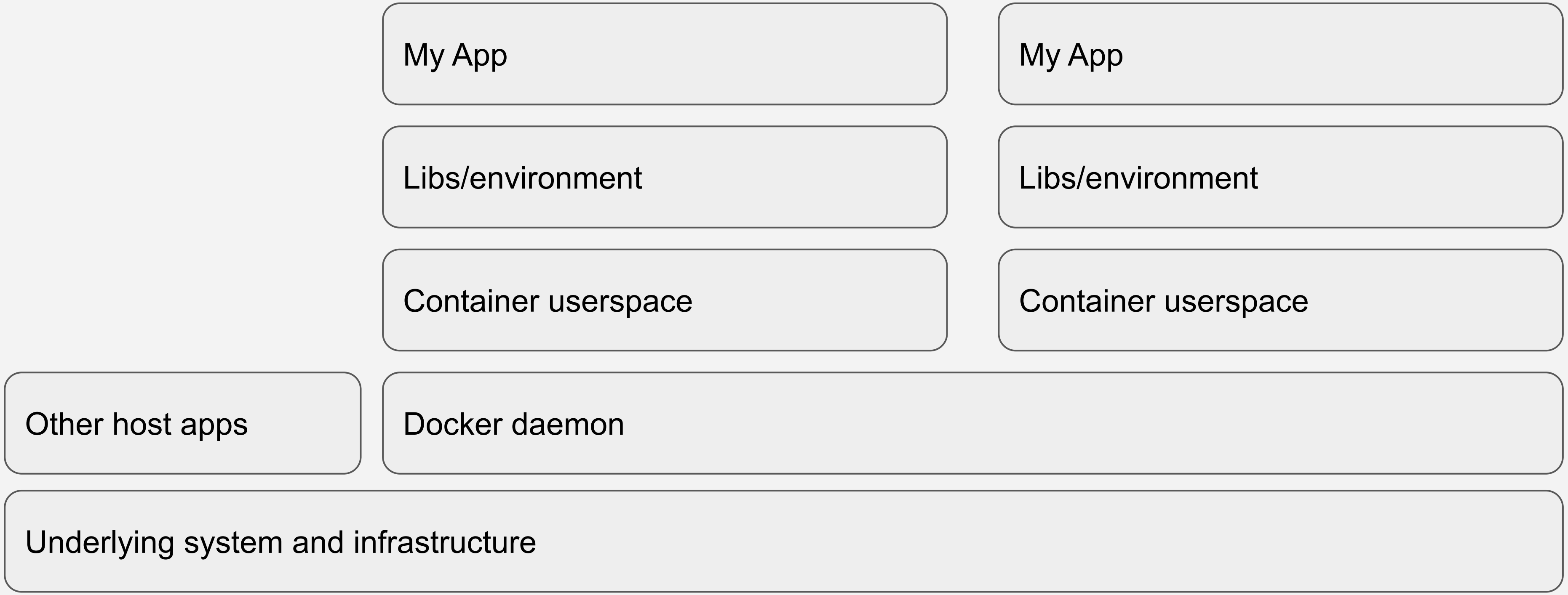
Image - a frozen state of system which is loaded upon docker construction
Container - an environment that’s separated from host OS that forms a base for our application
Volume - a persistent storage than can be appended to container similarly to a drive
Dockerfile - a configuration file that is used to build an image

Run vs start

How does it really work?

1. Choose system image and build Dockerfile upon that config
2. When you build an image, docker pull base images, merges them and then sequentially adds differential images that are results of running such command on the system (each stage is cached). At the end you get a frozen system state.
3. When you run container, all ENTRYPOINT processes occur and then CMD is called; the container is not interactive by default; the container expects the CMD to be run indefinitely
4. If CMD somehow dies, if it was started, the container is loaded anew (meaning all non-persistent changes are gone) and restarted

How does it really work?



Nvidia Runtime

Downloaded from any reasonable OS repository

Doesn’t require CUDA on host OS - only proper nvidia devices should be exposed (drivers). Actually runtime may be not needed and the devices can be added manually with `--device` flag which is a solution for non-standard systems (like Tegra).

`tensorflow:tensorflow` runs just fine with this

Example of a simple environment

1. Pull your favourite docker image

```
$ docker pull tensorflow/tensorflow
```

2. Statically link your codebase in a Dockerfile

```
FROM tensorflow:tensorflow
COPY myapp /myapp
```

3. Link volume for persistence

```
VOLUME /persistent
WORKDIR /myapp
CMD python ./myapp.py
```

4. Build and run container

```
$ docker build -t myname .
$ docker run myname /persistent:/var/somestoragedir
```

Demonstration

Advantages (according to world DevOpses)

> deployment regardless of the operating system of the host
-> as long as docker runtime is installed, it should work
-> for CPU it is available on any
-> for GPU?
> the development environment is static and is not ruined by updates
> we may all have our favourite OS of choice on our machines but still develop under one platform
> we may absolutely demolish the development environment but our system is relatively safe; this also work for webapps
> auto wake-up of services - reduces administrative work which we won't like to have
> it probably is available on Tegra (some workaround for GPU is required)
> composability - images are built on stable configurations present formerly on the system

Why PICTEC should adopt it?

- Because we won’t have dedicated sysadmins to maintain the configuration
- Client will ask for development under Windows, client will ask for installation on non-normal environments, slack with git - it’s easier to ask for one safe environment than for admin rights on even the whole VM
- Because we may prepare code for actual deployments and reuse libraries/code we use for deployment (common codebase)
- Generally current best practice™: creation of microservice for an application
- When Dockerfiles are created and runtime configured - little to zero work on tuning the app’s environment

For more curious

There are more advanced DevOps tools

- Docker-compose
- Kubernetes
- Terraform
- Ansible
- ...and maybe several dozens of more

But we don’t need it, as we manage simple prototypes. Docker seems to be a good balance between industrial IT and research environments.