

Containers and Orchestration: Docker and Kubernetes

Python Girona - March 2020

Jordi Bagot (<https://github.com/jbagot>), Xavi Torelló (<https://github.com/XaviTorello>).

Python Girona



- [Meetup \(https://www.meetup.com/es-ES/PythonGirona/\)](https://www.meetup.com/es-ES/PythonGirona/) JOIN US!
- Share knowledge
- Have fun
- Eat pizzas

General Agenda

- 1) Explanation of Docker and examples
- 2) Explanation of docker-compose and examples
- 3) Create a Django + React + Postgres application using docker and docker-compose! (Workshop)
- 4) Introduction to Kubernetes

Let's start!



Docker Agenda

- 1) What's docker?
- 2) How to create our images?
- 3) How to build our images?
- 4) How to create containers?
- 5) How to manipulate containers?

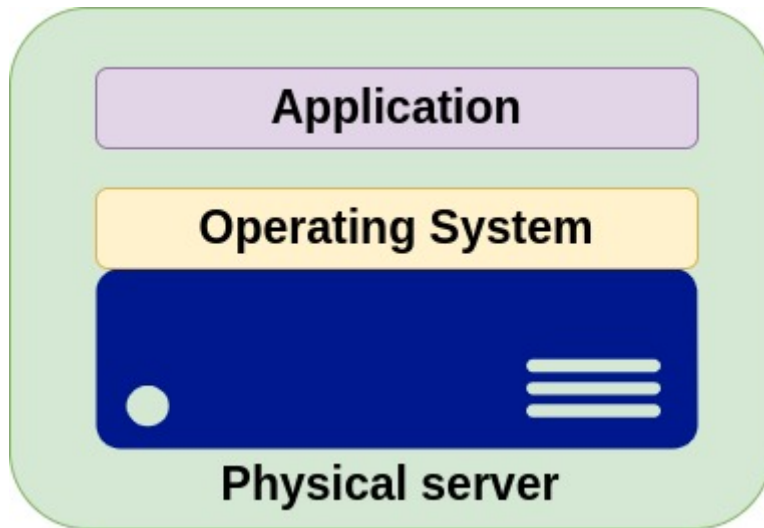
1) What's docker?

Docker is a platform for developers and sysadmins to develop, ship, and run applications

- Docker Engine: open source containerization technology
- Docker Hub: SaaS service for sharing and managing app stacks

Wait... let me go back, some years ago...

One application in one physical server

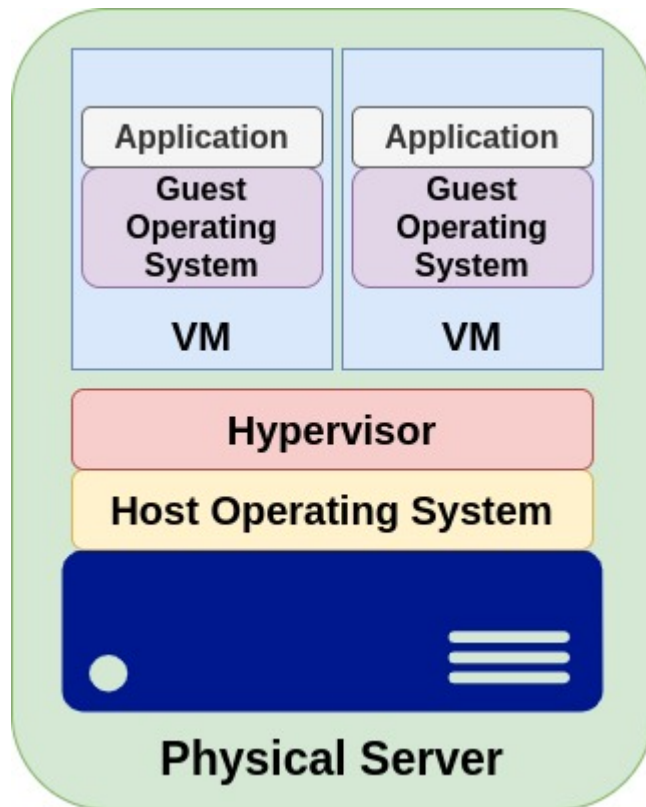


Problems:

- Slow deployments
- Complicate to scale
- Expensive
- Wasted resources

Little bit after...

Hypervisor virtualization



Advantages:

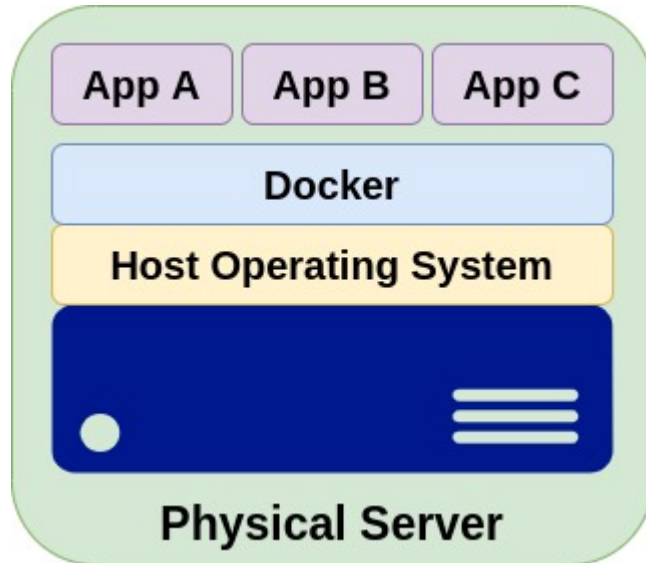
- One server have multiple applications
- Better resources
- VMs in the cloud as AWS, Digital Ocean, Azure...
- Easier to scale

Disadvantages:

- Each VM has an OS
- Each VM has hardware limitations: memory, CPU, ...
- Application portability

and now...

Docker!



Advantages:

- Speed, there is no OS to boot
- Portability
- Efficiency
- Scalability

What is a container?

- It is the unit where the application is embeded with its dependencies
- It is the result when an image is builded and ran
- Isolation
- Ready to run
- Portable, run everywhere

Docker basics



Image

The basis of a Docker container. The content at rest.



Container

The image when it is 'running.' The standard unit for app service



Engine

The software that executes commands for containers. Networking and volumes are part of Engine. Can be clustered together.



Registry

Stores, distributes and manages Docker images

2) How to create our images?

- With a file called `Dockerfile`
- It is like a recipe, with the needed ingredients (dependencies) and steps to create a dish (image)
- Defines the behaviour of the image

```
FROM alpine:latest
ADD . /app
RUN make app
CMD python /app/app.py
```


Docker image

- Made by multiple layers
- The containers are created from the images

```
$ docker images
```

REPOSITORY	SIZE	TAG	IMAGE ID	CREATED
shodan/scanner-scanner	551MB	latest	82daf18d5d92	11 da
ys ago				
empireproject/empire	1.19GB	latest	527d5d78e7fc	3 mon
th ago				
redis	30MB	alpine	05097a3a0549	12 da
ys ago				
redis	186MB	2.8	481995377a04	2 yea
rs ago				
elpaso/qgis-testing-environment	3.39GB	master	334775a61a4f	2 wee
ks ago				
docker_erp	1.05GB	latest	285af92a3352	4 wee
ks ago				
ubuntu	115MB	16.04	b9e15a5d1e1a	5 wee
ks ago				
python	908MB	2.7	4ee4ea2f0113	5 wee
ks ago				
mongo	232MB	3.0	fdab8031e252	5 mon
th ago				


3) How to build our images?

- Build an image means create an image from the Dockerfile.
- `docker build .` (in the Dockerfile path)
- This will download all the layers, for example an alpine, python and so on and will build all of them in one image, ready to be run

4) How to create containers?

- When we have the image built, execute: `docker run <image_name>`

```
~/projects/pygrn/docker_k8s_OlotTech/pictures master
$ docker run --rm -it python:3.8 python
Unable to find image 'python:3.8' locally
3.8: Pulling from library/python
50e431f79093: Pull complete
dd8c6d374ea5: Pull complete
c85513200d84: Pull complete
55769680e827: Downloading [=====] 43.62MB/51.79MB
f5e195d50b88: Downloading [=====] 142.9MB/192.2MB
94cdd3612287: Download complete
3b37b69935d4: Downloading [=====] 3.672MB/29.6MB
b9add85f08c4: Waiting
aalf4a29beac: Pulling fs layer
```

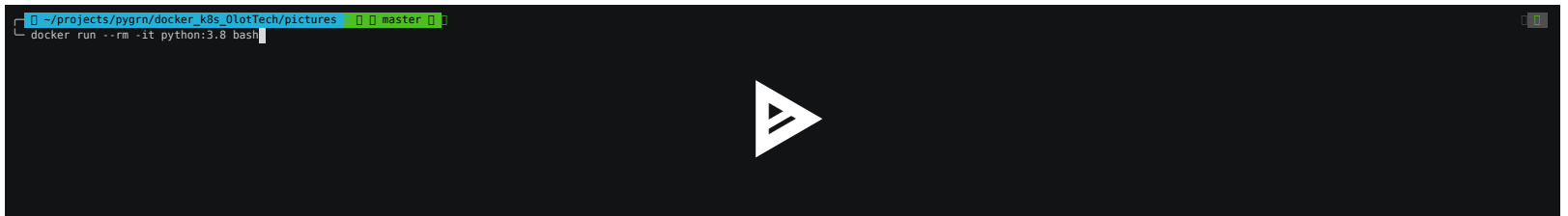


(<https://asciinema.org/a/309102?t=9>)

Container characteristics

- **NOT** persists (normally)
- Does **not expose** any container **port** to the host by default
- Does **not map** any host **resource** to the container by default

5) How to manipulate containers?



<https://asciinema.org/a/309106>

Reviewing existing containers:

```
docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED        STATUS        PORTS
S              NAMES
c06ea563da0e  python:3.8 "bash"                 3 seconds ago Up 2 seconds
upbeat_taussig
```

Use -a flag to see all (not just the started ones)

```
$ docker ps -a
```

CONTAINER ID	IMAGE	COMMAND
CREATED	STATUS	PORTS
NAMES		
c06ea563da0e	python:3.8	"bash"
2 minutes ago	Up 2 minutes	
upbeat_taussig		
47cf55fd3aac	nginx	"nginx -g
'daemon of...' 7 weeks ago	Exited (137) 4 weeks ago	
k8s_workshop_nginx_1		
285fa7d421c2	pentux/pygrn_k8s_workshop:latest	
"/entrypoint /start" 7 weeks ago	Exited (137) 4 weeks ago	
k8s_workshop_django_1		
...		
...		

Start a container

```
$ docker start <container_name>
```

Stop it!

```
$ docker stop <container_name>
```

Delete it!

```
$ docker rm docker_erp
```

But I want to communicate with my container!!!!

- **Expose** ports with
 - - p `$HOST_PORT:$CONTAINER_PORT` at run time
 - i.e -p `8081:8080` to expose the 8080 container port to 8081's host
 - use the `EXPOSE $PORT` command in your `Dockerfile`
- **Mount** paths
 - - v `$HOST_PATH:$CONTAINER_PATH` at run time
 - see the difference mount vs `ADD Dockerfile` command

But my application needs more than one service...

docker-compose is your friend!



docker-compose Agenda

- 1) What's docker-compose?
- 2) How to create our compositions?
- 3) How to run our compositions?

1) What's docker-compose?

- With a single file we can manage multiple images, our images or external ones.
Building or pulling them
- Run multiple containers at same time
- The containers will be connected with an internal network
- You can define how many replicas of each image you want

2) How to create our compositions?

An example of the YAML file:

```
version: '3'
services:
  web:
    build: .
    ports:
      - "5000:5000"
  redis:
    image: "redis:alpine"
```

this will provide two containers

- web: that uses local Dockerfile definition and binds the TCP#5000
- redis: that runs an alpine tagged redis image

3) How to run our compositions?

`docker-compose up` in the directory where you have the `docker-compose.yml` to run the composition.

In the previous example:

- 1) The web service have to build the docker file
- 2) The redis service have to be pulled from Docker Hub
- 3) Run both containers
- 4) Create a network to interconnect both services

`docker-compose down` to stop all the containers

Review logs

```
$ docker-compose logs -f [$service]
```

Rescale service

```
$ docker-compose scale $service=4
```

Stream container events

```
$ docker-compose events --json
```

Drop an interactive shell

```
$ docker-compose -it exec $service bash
```

Docker Hub

- Repository with a lot of images ready to use
- Pull images from Docker Hub with Dockerfile
- Create your own repository. Ex: Gitlab registry
- Push your images to your repository or to Docker Hub with `docker push`

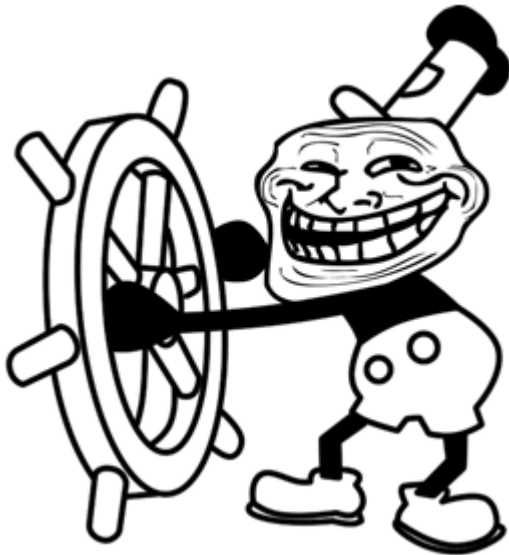
Share your images!!! Share your knowledge!!!



Workshop time!

Now we're going to extend a real and **very, very very important project** with a Composition :

https://github.com/pygrn/todos_django (https://github.com/pygrn/todos_django).



This is a Django project that serves an example TODOS API created by [@manelclos](https://github.com/manelclos) (<https://github.com/manelclos>).

Prepare the repo

```
$ git clone https://github.com/pygrn/todos_django.git .
```

Create our build file

- Create a new file named Dockerfile

```
FROM python:3.6
ENV PYTHONUNBUFFERED 1
COPY . /code/
WORKDIR /code
RUN pip install -r requirements.txt
```

,that means:

- use python:3.6 public image, and extend it with
- export PYTHONUNBUFFERED=1
- copy all the repo code at /code
- assume that current directory will be /code
- process and install all project requirements

- Create the image*

```
docker build -t todos_django:latest .
```

, this will tag the resultant image as `todos_django:latest`

- Run the image, just to review what it contains*

```
docker run --rm -it todos_django:latest bash
```

, this will provide an interactive temporal container that uses our image and drops a shell

Create our Composition

- Create a new file named `docker-compose.yml`

```
version: '3'
services:
  api:
    build: .
    volumes:
      - ./code
    ports:
      - "81:8000"
    command: ["python", "manage.py", "runserver"]
```

, that means:

- hey "mrs compose", this is a version3 composition that should deploy a container serving the api
- the image should be builded using our Dockerfile
- at run time, host project directory should be mounted inside container /code folder
- container 8000/tcp will be exposed to host at 81/tcp
- once everything is ready the command `python manage.py runserver` will be executed to run our django app

, it works, but we should add a DB to our composition!

```
...
api_1 | File "/usr/local/lib/python3.6/site-packages/django/db/backends/postgresql/base.py", line 176, in get_new_connection
api_1 |         connection = Database.connect(**conn_params)
api_1 | File "/usr/local/lib/python3.6/site-packages/psycopg2/__init__.py", line 130, in connect
api_1 |         conn = _connect(dsn, connection_factory=connection_factory, **kwargs)
api_1 | django.db.utils.OperationalError: could not connect to server: No such file or directory
api_1 | Is the server running locally and accepting
api_1 | connections on Unix domain socket "/var/run/postgresql/.s.PGSQL.5432"?
api_1 |
```

```
version: '3'
services:

  db:
    image: kartoza/postgis:latest
    environment:
      - POSTGRES_DB=a_database
      - POSTGRES_USER=a_user
      - POSTGRES_PASS=a_password
      - ALLOW_IP_RANGE=0.0.0.0/0
    ports:
      - 35432:5432
```

, this will

- provide a new service named db that will start a PostgreSQL with PostGIS extensions ready
- creating a new database named a_database
- granting access for a_user:a_password
- allowing connections from any IP
- exposing the container's psql port 5432 as host 35432

WTF?? Both containers are correctly defined, but the DB is not ready for our web

```
db_1 | 2019-03-05 15:37:23.354 UTC [40] LOG: database system was shut down a
t 2019-02-01 14:24:17 UTC
db_1 | 2019-03-05 15:37:23.388 UTC [27] LOG: database system is ready to acc
ept connections
api_1 | Try to load extra settings: settings-production.py
api_1 | Performing system checks...
api_1 |
api_1 | System check identified no issues (0 silenced).
api_1 | Unhandled exception in thread started by <function check_errors.<local
s>.wrapper at 0x7fdc5d447268>
api_1 | Traceback (most recent call last):
api_1 |   File "/usr/local/lib/python3.6/site-packages/django/db/backends/bas
e/base.py", line 213, in ensure_connection
api_1 |     self.connect()
api_1 |   File "/usr/local/lib/python3.6/site-packages/django/db/backends/bas
e/base.py", line 189, in connect
api_1 |     self.connection = self.get_new_connection(conn_params)
api_1 |   File "/usr/local/lib/python3.6/site-packages/django/db/backends/post
gresql/base.py", line 176, in get_new_connection
api_1 |     connection = Database.connect(**conn_params)
api_1 |   File "/usr/local/lib/python3.6/site-packages/psycopg2/__init__.py",
```

Solution: Use wait scripts!

<https://github.com/vishnubob/wait-for-it> (<https://github.com/vishnubob/wait-for-it>).

- Fetch the `wait-for-it.sh` script and save it at `utils/wait-for-it.sh`
//ensure that is executable!

```
$ mkdir -p utils && curl https://raw.githubusercontent.com/vishnubob/wait-for-it/master/wait-for-it.sh -o utils/wait-for-it.sh && chmod +x utils/wait-for-it.sh
```

- Prepare an start script! `utils/start-server.sh` //it should be executable!

```
pip install -r requirements.txt  
python manage.py migrate  
python manage.py runserver 0.0.0.0:8000
```

, this will ensure to review requirements, apply latest pending migrations and start Django!

- Improve our composition to change web start command and define a dependency to db:

```
api:
  build: .
  volumes:
    - ./code
  ports:
    - "81:8000"
  command: ["bash", "./utils/wait-for-it.sh", "db:5432", "--", "bash", "./utils/start-server.sh"]
  depends_on:
    - db
```

, this will start our Django once the 5432/tcp@db is ready to accept connections!

OK! Now our Django is waiting for the DB, but still breaking!

We should review our Django config, it needs some ENV vars to point to our backend

```
$ vi todos_project/settings-production.py
```

```
DATABASES = {
    'default': {
        # 'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': os.environ.get('DB_NAME'),
        'USER': os.environ.get('DB_USER'),
        'PASSWORD': os.environ.get('DB_PASSWORD'),
        'HOST': os.environ.get('DB_HOST'),
        'PORT': os.environ.get('DB_PORT'),
    },
}
ALLOWED_HOSTS = ['*']
```

Config the environment vars in the docker-compose file

...

api:

environment:

- DB_HOST=\${DB_HOST}
- DB_PORT=\${DB_PORT}
- DB_NAME=\${DB_NAME}
- DB_USER=\${DB_USER}
- DB_PASSWORD=\${DB_PASSWORD}

...

db:

environment:

- POSTGRES_DB=\${DB_NAME}
- POSTGRES_USER=\${DB_USER}
- POSTGRES_PASS=\${DB_PASSWORD}
- ALLOW_IP_RANGE=0.0.0.0/0

- Create an .env file

```
DB_HOST=db  
DB_PORT=5432  
DB_NAME=todos  
DB_USER=todos  
DB_PASSWORD=this_is_not_a_secure_password
```


It's magic!! It works!!!

<http://0.0.0.0:81/api/v1/> (<http://0.0.0.0:81/api/v1/>).

Django REST framework

Api Root

Api Root

OPTIONS GET

The default basic root view for DefaultRouter

GET /api/v1/

HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

```
{  
  "todos": "http://0.0.0.0:81/api/v1/todos/"  
}
```

It can be more intense...

We'll try to integrate the React frontend created by [@francescarpi](https://github.com/francescarpi)
(<http://github.com/francescarpi>):

https://github.com/pygrn/todos_react (https://github.com/pygrn/todos_react).

The idea is to show alternative ways to provide a container as a service

- Create another build script named `Dockerfile_frontend`

Use an alpine-based (ver small base) node image

FROM node:alpine

RUN apk update && apk upgrade && \
apk add --no-cache bash git openssh

Prepare our project and their dependencies

RUN mkdir /code

WORKDIR /code

RUN git clone https://github.com/pygrn/todos_react .

RUN sed -i 's;https://server3.microdiseny.com/apps/todos;http://localhost:81;
g' src/lib/apiclient.js

RUN yarn install

Directly upload the wait-for-it script to the image

ADD utils/wait-for-it.sh ./

RUN chmod +x wait-for-it.sh

- Add the new service!

...

```
web:
  build:
    context: ./
    dockerfile: Dockerfile_frontend
  command: ["bash", "./wait-for-it.sh", "api:8000", "--", "yarn", "start"]
  ports:
    - "80:3000"
  depends_on:
    - api
  restart: always
```

Now, you can check <http://localhost> (<http://localhost>) ...



Kubernetes Agenda

- 1) What's Kubernetes?
- 2) Kubernetes structure

1) What's Kubernetes?

- Container orchestrator
- The most famous one
- Layer to manage a cluster of containers
- Auto-scaling

Container orchestrator

- Easy deploy, gracefully, stateless
- Replication: run X copies
- Built-in load balancers
- Auto-scaling: better resource use

Layer to manage a cluster

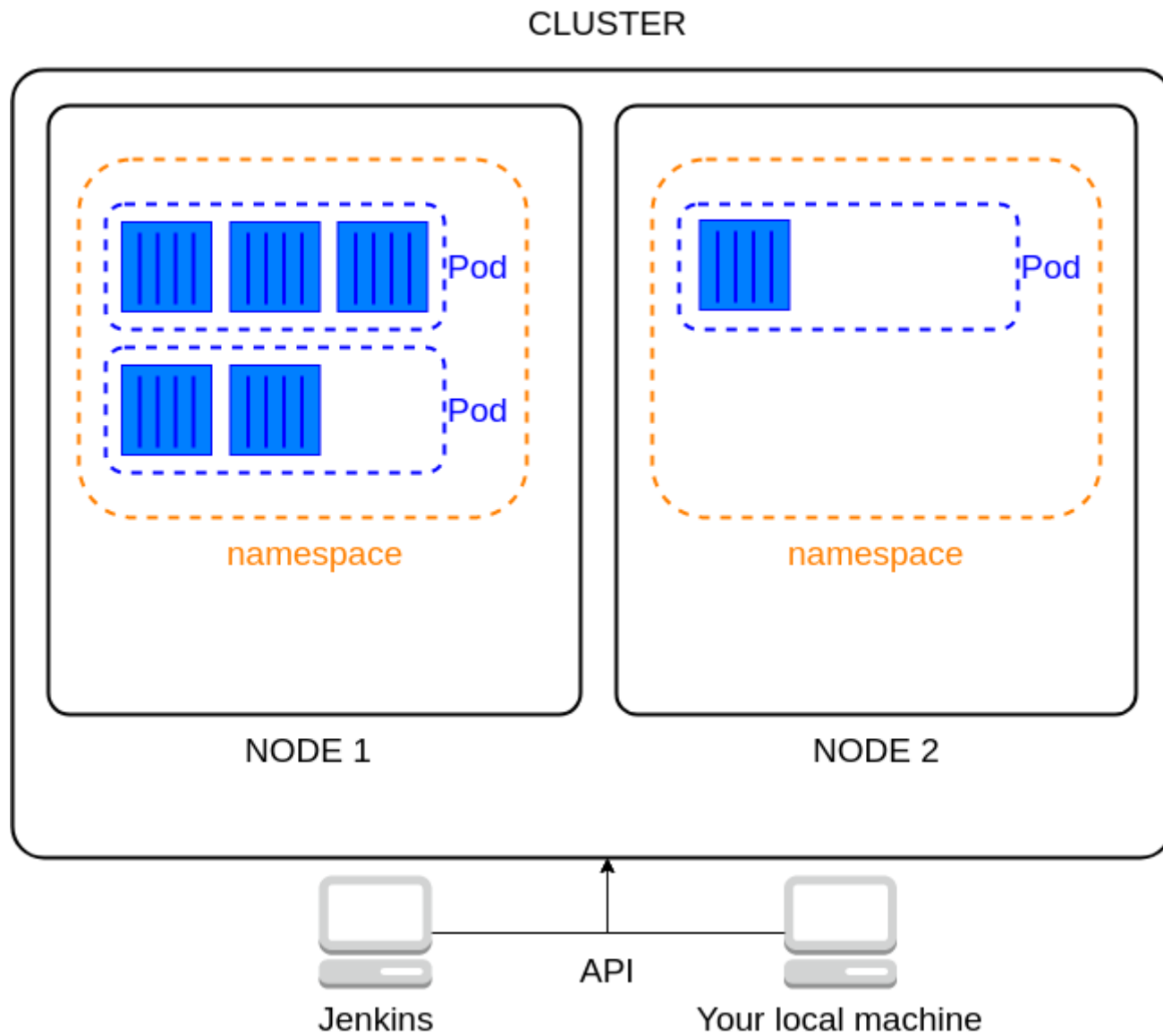
- Manifest: one file to define all your cluster
- Automate cluster maintenance
- Load balancers
- Manage secrets

Scalability

- Specify number of container replicas
- Auto pod scaling, based on CPU, memory or custom metrics
- It's a cluster, add more nodes

2) Kubernetes structure

- Pod
- Service
- Namespace
- Node



And **much much much** more...

If you want more we can do another session

Resources:

- <https://www.slideshare.net/Docker/introduction-to-docker-2017>
(<https://www.slideshare.net/Docker/introduction-to-docker-2017>).

It's all!



Questions?