Docker, Backends & Microservices

Dr. Jose L. Muñoz-Tapia
Information Security Group (ISG)
Universitat Politècnica de Catalunya (UPC)

Docker

Outline

Motivation

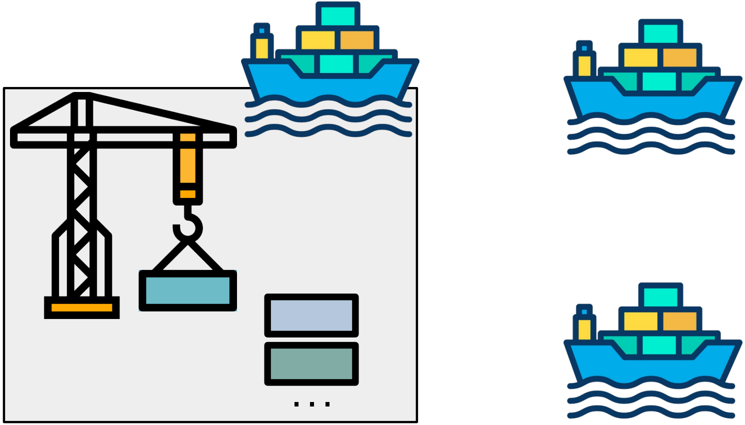
Practical Basics

Container Orchestrators

Docker Motivation: Portable Applications i

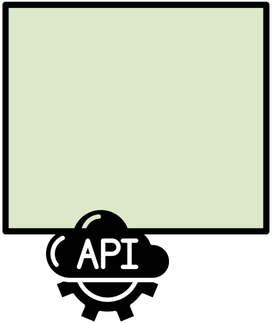
- Applications are not only executables or scripts.
- They also need resources in the computer they execute:
    - If our application is a script we need its interpreter.
    - Dynamic libraries.
    - Directories for data, configuration files and so on.
    - Network resources (e.g. transport ports).
- Portable applications can be easily deployed and moved to computers.

Docker Motivation: Portable Applications ii

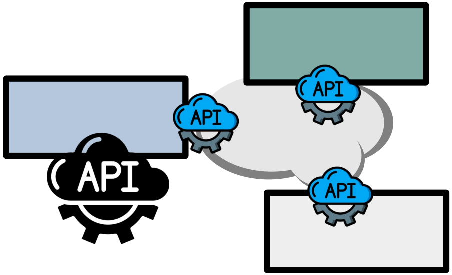


Docker Motivation: Microservices architectures i

Monolitic



Microservices



Docker Motivation: Microservices architectures ii

- In a **monolithic architecture**:
  - All the soft written in a single language and running as a single component in a single container.
  - A well-designed monolith would factor these components into separate libraries and use existing libraries where possible.
- In a **microservices architecture**:
  - The service is composed of multiple small and independent programs or ”microservices”.
  - The idea is ”do one thing and do it well”.
  - We can use different types of implementations (e.g. different languages).
  - There programs use network communication with well defined APIs.
  - Containers offer a great way to wrap microservices.

Outline

Motivation

Practical Basics

Container Orchestrators

Install (Ubuntu)

- Install docker and docker-compose soft:

```
$ sudo apt install docker.io docker-compose
```

- For being able to run docker without sudo (docker binary is priviledged):

```
$ sudo usermod -aG docker $USER
```

- Then, logout and login in your Linux system.
- If needed, you can restart the docker daemon as follows:

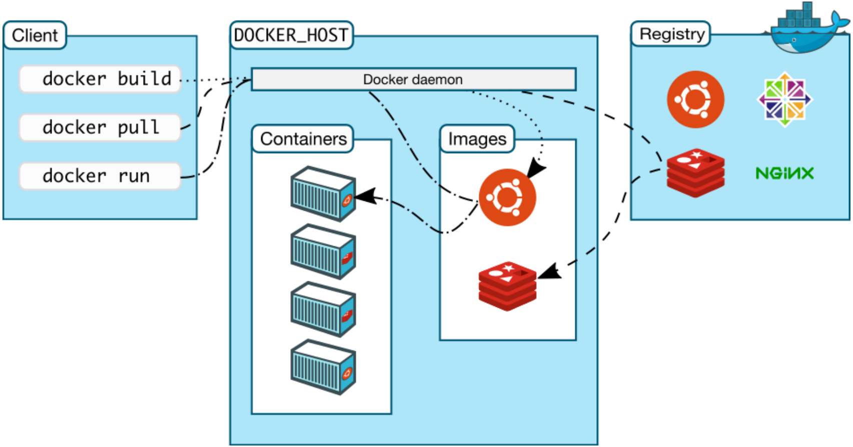
```
$ sudo systemctl restart docker.service
```

- To test that the installation[1], run the following command which downloads the Debian image, creates and executes the container:

```
$ docker run debian echo "Hello World"
Unable to find image 'debian:latest' locally
...
Hello World
```

[1] You might get errors if running behind a proxy, see: https://docs.docker.com/engine/admin/systemd/#http-proxy

Docker Architecture i



Docker Architecture ii

1. Docker daemon:
   - Distributed as a single binary and started normally by the host OS.
   - It is responsible for creating, running, and monitoring containers, as well as building and storing images.
2. Docker client:
   - Distributed as a single binary.
   - Talks to the Docker daemon (via HTTP, Unix Socket or TCP connection).
3. Docker registries:
   - They store and distribute images.
   - The default registry is ”Docker Hub”.
   - Docker daemon will download images from registries if they are not available locally.

Container Life-cycle: Stopped Containers

- However, we don’t see any container running:

```
$ docker ps
CONTAINER ID  IMAGE   COMMAND  CREATED             STATUS   PORTS   NAMES
```

- This is because containers run until its last process exits.
- In our example, the container is not executing any process so it exits immediately.
- When the last process exits, the container is said to be **exited** or **stopped**.
- We can see stopped/exited containers with:

```
$ docker ps -a
CONTAINER ID  IMAGE   COMMAND  CREATED             STATUS   PORTS   NAMES
1e48454d644c  debian  "bash"   About a minute ago  Exited           nifty_noether
400845a9040b  debian  "bash"   28 minutes ago      Exited           epic_sinoussi
```

Container with a Running Process

- To create a container that has a running process we can use the nginx image:

```
$ docker run nginx
```

- In another terminal, we can see it running:

```
$ docker ps
CONTAINER ID        IMAGE   COMMAND  ....
66e18bf5ebbd        nginx   .....
```

- The process running in the container is in foreground and we can kill it with `CRL+c`.
- We can start the container again with:

```
$ docker start ID
```

- Notice that containers started with ”start” are started in background.
- We can use run in background (daemon) with -d:

```
$ docker run -d nginx
```

Container Life-cycle: `create`, `start` and `run`

- We can create a container as follows:

```
$ docker create nginx
400845a9040b8499fcf3e75653f76cc2d0f579560a0497b3e1933a22c951ce23
```

- Then, we can start the container using its hash identifier:

```
$ docker start 400845a9040b84
```

- Note. We can take as ID any number of digits that uniquely identify the container.
- Notice that the **container starts and dies** because there is not any running process.
- The `run` command does the two previous steps: `create + start`.

Container Life-cycle: Interactive Containers i

- We need to create a process that does **not** end to keep our container running.
- We make our first try with a bash:

```
$ docker run debian /bin/bash
```

- This still does not work, bash exits because it will not be able to interact.
- We have to create an interactive container using `-it`:

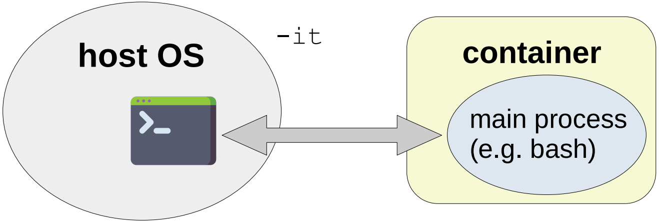
```
$ docker run -it debian /bin/bash
```

- Now you will see that the container is running with `docker ps`.

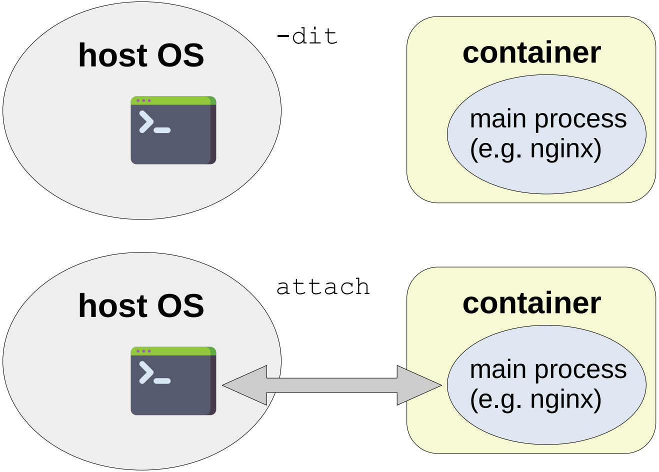
Container Life-cycle: Interactive Containers ii

`--interactive` , `-i` keeps STDIN open even if not attached (input).

`--tty` , -t allocates a pseudo-TTY (output).



Container Life-cycle: attach i



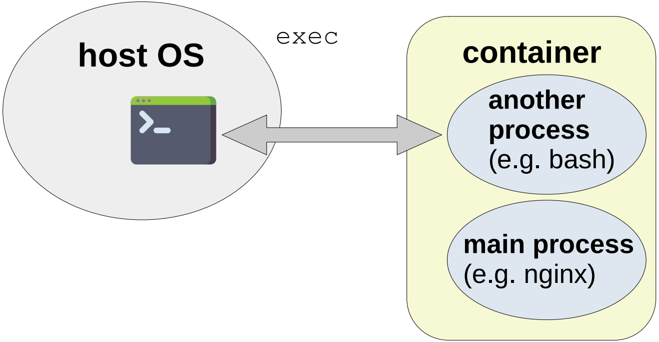
Container Life-cycle: attach ii

- You can run a container as ”detached” (we also say ”as a daemon”) with `-d`.
- You can also later attach local standard input, output, and error streams to a running container with `-it`.
- All together we can use `-dit`:

```
$ docker run -dit nginx
$ docker attach ID
```

- When attached, if you type `ctrl+c` you will stop the container (because it stops the main process).
- To detach without kill, if you started the container in interactive mode, you can detach typing `ctrl+p` and `ctrl+q`.

Container Life-cycle: exec



- We can execute a `bash` interpreter in the container with:

```
$ docker exec -it ID /bin/bash
```

- The previous command executes a bash in the container and attaches its streams to the standard input, output, and error of the local console.

ID, name and hostname

- We can see that Docker gives a hash ID and a name to each container:

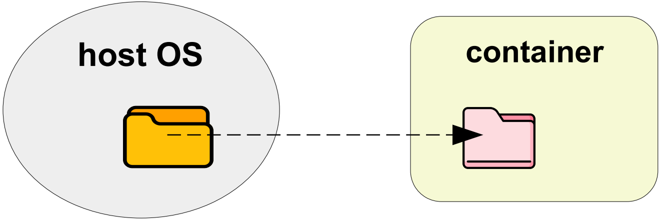
```
$ docker ps -a
CONTAINER ID  IMAGE   COMMAND  CREATED             STATUS   PORTS   NAMES
1e48454d644c  debian  "bash"   About a minute ago  Exited           nifty_noether
400845a9040b  debian  "bash"   28 minutes ago      Exited           epic_sinoussi
```

- We can select the Docker name (`--name`) and the hostname (`-h` or `--hostname`):

```
$ docker create --name mycont --hostname mycont debian
```

- The name is used by Docker.
- The hostname is used by the operating system (`hostname` command).

Copy Files to a Running Container



- We can copy files from the host to a container using `docker cp`:

```
docker cp index.html mycont:/usr/share/nginx/html/index.html
```

Container Life-cycle: Exiting and Re-starting

- To stop/exit a container, type “exit” or `Control+D` when inside a console of an interactive container.
- You can also stop a running container as follows:

```
$ docker stop mycont
```

- You can start again a stopped container:

```
$ docker start mycont
```

- You can use the container name or its hash id and it will be started with the same parameters as the first time.
- With `-i` you can start the container in interactive mode:

```
$ docker start mycont -i
```

Container Life-cycle: Removing

- Unlike processes, stopped Docker containers do not disappear from the system.
- To really get rid of a container:

```
$ docker rm mycont
```

- To get rid of all stopped containers:

```
$ docker rm -v $(docker ps -aq -f status=exited)
```

- You can also use this newer command:

```
$ docker container prune
```

Getting Information: `inspect`, `diff` & `logs`

- `inspect` gives us information about the container:

```
$ docker inspect mycont
$ docker inspect mycont | grep IPAddress
$ docker inspect mycont --format {{.NetworkSettings.IPAddress}} mycont
```

- `diff` lists the changed files with respect the image:

```
$ docker diff mycont
```

- `logs` tells us what has happened inside the container:

```
$ docker logs mycont
```

Summary of Container's Life-cycle Commands

Command	Description
docker create IMG	Creates (not start) a container from an image IMG
docker create -it debian /bin/bash	Creates an interactive container from debian
docker start ID/name	Starts a previous created container
docker run IMG	Runs (creates and starts) a container from an image
docker run -dit debian /bin/bash	Runs an interactive container in background
docker stop ID/name	Stops a running container
docker ps	Lists running containers
docker ps -a	Lists running and stopped containers
docker exec -it mycont /bin/bash	Executes a bash in the container and attaches it locally
docker attach mycont	Attaches locally the streams of a running container
docker rm mycont	Removes mycont
docker container prune	Removes all stopped containers
docker inspect ID/name	Provides information about the container
docker diff ID/name	Lists the filesystem changes regarding the image
docker logs ID/name	Lists the commands executed

Dockerfiles

- We can use a Dockerfile to create an automated build for the image:

```
$ mkdir cowsay ; cd cowsay
cowsay$ touch Dockerfile
```

- Insert the following contents into Dockerfile:

```
FROM ubuntu
RUN apt update
RUN apt install -y cowsay fortune
```

- The FROM instruction specifies the base image to use.
- We can create and commit the image using (do this in the directory of the Dockerfile):

```
cowsay$ docker build -t myuser/cowsay-dockerfile .
```

- The option `-t` or `--tag` creates a tag for the image.

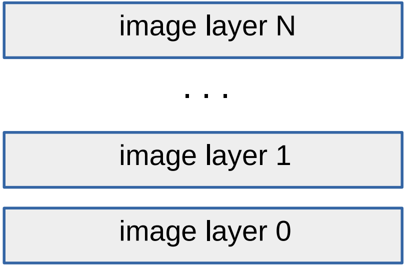
Build Context

- The docker build command requires a Dockerfile and a build context (which may be empty).
- The build context is the set of local files and directories that can be referenced from ADD or COPY instructions in the Dockerfile.
- The context is normally specified as a path to a directory.
- In our previous example, we used the following build command:

```
$ docker build -t myuser/cowsay-dockerfile .
```

- The last dot is the context.
- We can give other type of contexts like git repositories.

Layers of an Image i



A layer can use 0 bytes.
You can see the layers with:

```
$ docker history IMG
```

- Docker images are made up of multiple layers.
- Each of these layers is a read-only fileystem.
- A layer is created for **each instruction in a Dockerfile** and sits on top of the previous layers.
- Many Dockerfiles try to minimize the number of layers by specifying several UNIX commands in a single RUN instruction.
- To create layers, Docker uses **union mounts** which allow multiple file systems to be overlaid, appearing to the user as a single filesytem.

Layers of an Image ii

```c
// hello.c
#include<stdio.h>
void main()
{
  printf("Hello World\n");
}
```

Example (Bad):

```dockerfile
FROM ubuntu

COPY hello.c /
RUN apt update
RUN apt install -y gcc
RUN gcc /hello.c -o /hello
RUN apt remove -y gcc
RUN apt autoclean -y
RUN apt autoremove -y
```

Example (Better):

```dockerfile
FROM ubuntu

COPY hello.c /
RUN apt update \
&& apt install -y gcc \
&& gcc /hello.c -o /hello \
&& apt remove -y gcc \
&& apt autoclean -y \
&& apt autoremove -y
```

Publish & Expose Ports i

- Publishing a port (or ports) makes a service running in a container available outside world.
- Publish means forwarding ports on the host to the container.
- This can be done using the `-p` option, example:

```
$ docker run -d -p 8080:80 nginx
af9038e18360002ef3f3658f16094dadd4928c4b3e88e347c9a746b131db5444
```

- Test this nginx webserver:

```
$ curl localhost:8080
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
```

But...

Publish & Expose Ports ii

1. **When running a container from an image, how do we know which ports use?**
2. **What if the port on the host is occupied?**

To solve the first issue:

- We can use the **EXPOSE** instruction in the docker file.
- **EXPOSE** is used as a way of document the ports used.
- **EXPOSE** indicates to Docker that the container will have a process listening on the given port or ports:

```
EXPOSE 80
```

Publish & Expose Ports iii

To solve the second issue:

- We use the option `-P`.
- This option lets the docker daemon to select available public ports at the host for the exposed ports.

You can check the exposed/public ports with:

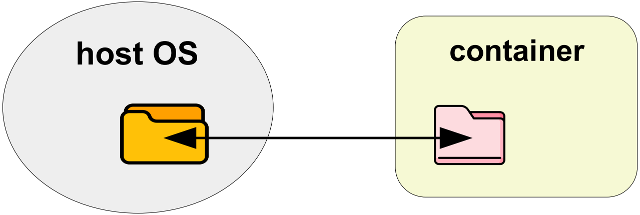
```
$ docker port container_ID
```

Volumes

Volumes

Volumes are files or directories that are directly mounted on the host and not part of the normal union file system.

Volumes are a way of sharing directories between a container and the host and between containers.



There are two types of volumes:

1. **Bind mounts.**
2. Host-independent volumes.

Bind Mounts

Bind Mounts

With a **bind mount** you can specify in the docker command any directory in the host to be mounted.

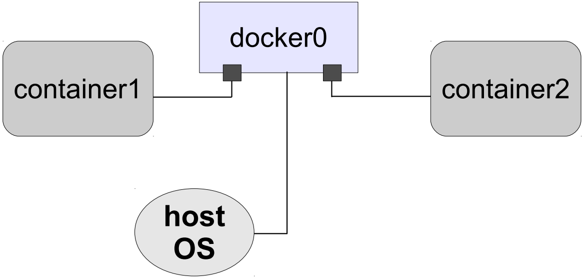
```
docker run -d -v /host/dir:/container/dir IMG
```

- Example:

```
$ docker run -d -p 8081:80 -v /home/myuser/mynginx/data:/usr/share/nginx/html nginx
```

- Notes:
  - No files from the image are copied into the volume.
  - The volume will not be deleted by Docker (`docker rm -v` will not remove the volume).
  - It is not possible to specify a host directory inside a Dockerfile for reasons of portability and security (as shown, this is only possible with a command).

Basic Docker Networking



- Docker creates a virtual switch for connecting the containers.
- Creates a DHCP server to assign IP addresses.
- Configures SNAT so containers can access the Internet.

Default & User-defined Bridges

- Default bridge:
    - When you start Docker, a default bridge network (also called bridge) is created automatically.
    - Newly-started containers connect to it unless otherwise specified.
    - Containers on the default bridge network can only access each other by IP addresses (not by names).
- You can also use a user-defined bridge:
    - Containers connected to the same user-defined bridge network automatically expose all ports to each other (but no ports to the outside world).
    - Containers can resolve each other by name or alias.

Networking in Practice

- To create, remove, list and inspect user-defined bridges:

```
$ docker network create --driver bridge my-net
$ docker network rm my-net
$ docker network ls
$ docker network inspect my-net
```

- To create and connect a container to the user-defined bridge:

```
$ docker run --name my-nginx --network my-net nginx
```

- With a user-defined bridge we can use names to connect containers:

```
$ docker run -it --network my-net debian /bin/bash
# ping my-nginx
```

- To connect/disconnect a running container:

```
$ docker network connect my-net my-nginx
$ docker network connect bridge my-nginx  # the default bridge
$ docker network disconnect my-net my-nginx
```

Example: Redis

- Example with redis, we start the server:

```
$ docker network create my-net
$ docker create --name myredis-server --network my-net redis:alpine
$ docker container start myredis-server
```

- We test with a client (redis-cli)

```
$ docker run -it --rm --network my-net redis:alpine /bin/sh
> redis-cli -h myredis-server
myredis-server:6379> ping
PONG
redis:6379> set "abc" 123
OK
myredis-server:6379> get "abc"
"123"
myredis-server:6379> exit
# exit
exit
```

- The `--rm` option tells Docker to remove the container when exiting.

Outline

Motivation

Practical Basics

Container Orchestrators

Container Orchestration

- Container orchestration is all about managing the lifecycles of containers.
- Software teams use container orchestration to control and automate many tasks:
  - Provisioning, deployment and resource allocation for containers.
  - Health monitoring of containers and hosts.
  - Scaling up or removing containers.
  - Moving containers from one host to another (in case of shortage or fail).
  - Externally exposing services running in a container.

Most Used Orchestration Tools

- **Docker Compose**: from the same team as Docker, mainly for development.
- **Docker Swarm**: from the same team as Docker, suitable for production.
- **Kubernetes**: from Google, suitable for production.
- **Mesos and Marathon**: from apache foundation, suitable for production.

Docker Compose

- Compose is designed to quickly get Docker development environments up and running.
- Uses YAML files to store the configuration for sets of containers.
- Using Compose is basically a three-step process:
    1. Define your app's environment with a `Dockerfile` so it can be reproduced anywhere.
    2. Define the services (containers) that make up your app in `docker-compose.yml` so they can be run together in an isolated environment.
    3. Run `docker-compose up` and compose starts and runs your entire app.

Using Compose

- Create the following file (be careful with indentations):

```yaml
version: '3'
services:
  redis-cli:
    image: "redis:alpine"
  redis-server:
    image: "redis:alpine"
```

  `https://docs.docker.com/compose/compose-file/`

- Run compose:

```
$ docker-compose up
```

- Then, you can exec a shell in the **`redis-cli`** container as follows:

```
$ docker-compose exec redis-cli /bin/sh
```

- You can use the name **`redis-server`** to connect to the server.
- The **`exec`** subcommand will allow you to use a container that is already running.

Networks in Compose

The previous compose creates a network:

```
$ docker networks ls
NETWORK ID          NAME                DRIVER              SCOPE
...
3696be1307c6        mycompose_default   bridge              local
...
```

```yaml
version: '3'
services:
  redis-cli:
    image: "redis:alpine"
    networks:
      - mynet
  redis-server:
    image: "redis:alpine"
    networks:
      - mynet
networks:
  mynet: # If not specified a docker-compose creates a user-network with a default name.
    driver: bridge
```

```
3696be1307c6        mycompose_mynet     bridge              local
```

Building Images with Compose

- We can define to build the image also from the `docker-compose.yaml` file:

```yaml
version: '3.5'
services:
  myservice:
      build: .
      image: myimage
```

- As a simple example, we use the following `Dockerfile`:

```dockerfile
FROM nginx
```

- If we omit the image field, the image will take the name of the service.
- Now, we can build the image with `docker-compose`:

```
$ docker-compose build
```

Typical Compose Workflow

1. Run the app (in background):
   ```
   docker-compose up -d
   ```
2. Verify status and debugging:
   ```
   docker-compose logs
   docker-compose ps
   ```
3. After changes in the image/code:
   ```
   docker-compose build
   ```
4. Use down+up to run the app again:
   ```
   docker-compose down
   docker-compose up -d
   ```
5. To start/stop the same containers:
   ```
   docker-compose stop
   docker-compose start
   ```
6. To remove the containers:
   ```
   docker-compose rm
   ```

Environment Variables i

- We can create environment variables that will be available inside the container:

```yaml
version: '3.5'
services:
  mycontainer:
    image: "ubuntu"
    entrypoint: /bin/bash
    stdin_open: true
    tty: true
    environment:
      - MYVAR=hello world
```

- How to manage secrets?
  - Compose supports declaring default environment variables in an environment file named `.env`
  - The `.env` file must be placed in the folder where the `docker-compose` command is executed (current working directory).

Environment Variables ii

```yaml
version: '3.5'
services:
  mycontainer:
    image: "ubuntu"
    entrypoint: /bin/bash
    stdin_open: true
    tty: true
    environment:
      - MYVAR=hello world
      - MYSECRET
```

- The `.env` file:

```
MYSECRET=super secret
```

- Syntax rules of the `.env` file:
  - Compose expects each line in an env file to be in `VAR=VAL` format.
  - Lines beginning with # are processed as comments and ignored.
  - Blank lines are ignored.
  - There is no special handling of quotation marks (this means that they are part of the VAL).

Environment Variables iii

- We can also tell `docker-compose` to use host environment variables:

```yaml
version: '3.5'
services:
  mycontainer:
    image: "ubuntu"
    entrypoint: /bin/bash
    stdin_open: true
    tty: true
    environment:
      - MYVAR=${MYVAR}
      - MYOTHERVAR=${MYOTHERVAR:-hello world}
      - MYSECRET
```

- We can try the config as follows:

```
$ export MYOTHERVAR=hi
$ docker-compose up
```