

Docker Avançado

Aula Teórica nº9

2020/2021

Docker Images

- Every Docker container is based on an image
- To launch a container, you must either download a public image or create your own
- You can think of the image as the filesystem for the container
- Every Docker image consists of one or more filesystem layers that generally have a direct one-to-one mapping to each individual build step used to create that image

Agenda

- Building images
- Uploading (pushing) images to an image registry
- Downloading (pulling) images from an image registry
- Creating and running containers from an image

Anatomy of a Dockerfile

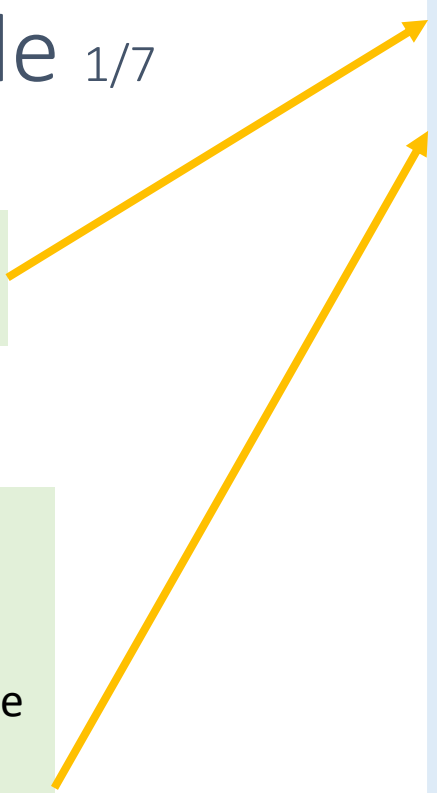
- This file describes all the steps that are required to create an image
- Would usually be contained within the root directory of the source code repository for your application
- Each line in a *Dockerfile* creates a new image layer that is stored by Docker
- This layer contains all of the changes that are a result of that command being issued
- Docker will only need to build layers that deviate from previous builds

Example: Container for a Node.js-based application

- We could build a Node instance from a plain, base Linux image
- We can also explore the **Docker Hub** for official images for Node
- The **Node.js community** maintains a series of **Docker images** and tags that allows you to quickly determine what versions are available
- If we want to lock the image to a specific point release of Node, you could point it at something like `node:11.11.0`
- The base image that follows will provide you with an **Ubuntu Linux image running Node 11.11.x**

Example Dockerfile 1/7

Base image: Ubuntu Linux image running Node 11.11.x



Labels: Applying labels to images and containers allows you to add metadata via key/value pairs that can later be used to search for and identify Docker images and containers.

You can see the labels applied to any image using the `docker inspect` command

```
FROM node:11.11.0

LABEL "maintainer"="anna@example.com"
LABEL "rating"="Five Stars" "class"="First Class"

USER root

ENV AP /data/app
ENV SCPATH /etc/supervisor/conf.d

RUN apt-get -y update

# The daemons
RUN apt-get -y install supervisor
RUN mkdir -p /var/log/supervisor

# Supervisor Configuration
ADD ./supervisord/conf.d/* $SCPATH/

# Application Code
ADD *.js* $AP/


WORKDIR $AP

RUN npm install

CMD ["supervisord", "-n"]
```

Example Dockerfile 2/7

User: By default, Docker runs all processes as root within the container, but you can use the USER instruction to change this



Caution! Even though containers provide some isolation from the underlying operating system, they still run on the host kernel. Due to potential security risks, production containers should almost always be run under the context of a nonprivileged user.

```
FROM node:11.11.0

LABEL "maintainer"="anna@example.com"
LABEL "rating"="Five Stars" "class"="First Class"

USER root

ENV AP /data/app
ENV SCPATH /etc/supervisor/conf.d

RUN apt-get -y update

# The daemons
RUN apt-get -y install supervisor
RUN mkdir -p /var/log/supervisor

# Supervisor Configuration
ADD ./supervisord/conf.d/* $SCPATH/

# Application Code
ADD *.js* $AP/


WORKDIR $AP

RUN npm install

CMD ["supervisord", "-n"]
```

Example Dockerfile 3/7

The **ENV** instruction allows you to set shell variables that can be used by your running application for configuration and during the build process to simplify the *Dockerfile*



```
FROM node:11.11.0

LABEL "maintainer"="anna@example.com"
LABEL "rating"="Five Stars" "class"="First Class"

USER root

ENV AP /data/app
ENV SCPATH /etc/supervisor/conf.d

RUN apt-get -y update

# The daemons
RUN apt-get -y install supervisor
RUN mkdir -p /var/log/supervisor

# Supervisor Configuration
ADD ./supervisord/conf.d/* $SCPATH/

# Application Code
ADD *.js* $AP/

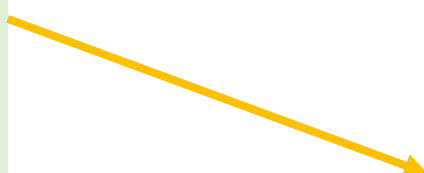
WORKDIR $AP

RUN npm install

CMD ["supervisord", "-n"]
```


Example Dockerfile 4/7

RUN instructions to start and create the required file structure that you need, and install some required software dependencies



Warning! While we're demonstrating it here for simplicity, it is not recommended that you run commands like `apt-get -y update` or `yum -y update` in your application's *Dockerfile*. This is because it requires crawling the repository index each time you run a build, and means that your build is not guaranteed to be repeatable since package versions might change between builds. Instead, consider basing your application image on another image that already has these updates applied to it and where the versions are in a known state. It will be faster and more repeatable.

```
FROM node:11.11.0

LABEL "maintainer"="anna@example.com"
LABEL "rating"="Five Stars" "class"="First Class"

USER root

ENV AP /data/app
ENV SCPATH /etc/supervisor/conf.d

RUN apt-get -y update

# The daemons
RUN apt-get -y install supervisor
RUN mkdir -p /var/log/supervisor

# Supervisor Configuration
ADD ./supervisord/conf.d/* $SCPATH/

# Application Code
ADD *.js* $AP/

WORKDIR $AP

RUN npm install

CMD ["supervisord", "-n"]
```

Example Dockerfile 5/7

The **ADD** instruction is used to copy files from either the local filesystem or a remote URL into your image. Most often this will include your application code and any required support files.

Note: Because ADD actually copies the files into the image, you no longer need access to the local filesystem to access them once the image is built. You'll also start to use the build variables you defined in the previous section to save you a bit of work and help protect you from typos.



```
FROM node:11.11.0

LABEL "maintainer"="anna@example.com"
LABEL "rating"="Five Stars" "class"="First Class"

USER root

ENV AP /data/app
ENV SCPATH /etc/supervisor/conf.d

RUN apt-get -y update

# The daemons
RUN apt-get -y install supervisor
RUN mkdir -p /var/log/supervisor

# Supervisor Configuration
ADD ./supervisord/conf.d/* $SCPATH/

# Application Code
ADD *.js* $AP/

WORKDIR $AP

RUN npm install

CMD ["supervisord", "-n"]
```

Tips

- Remember that every instruction creates a new Docker image layer, so it often makes sense to combine a few logically grouped commands onto a single line. It is even possible to use the **ADD** instruction in combination with the **RUN** instruction to copy a complex script to your image and then execute that script with only two commands in the *Dockerfile*.
- The order of commands in a *Dockerfile* can have a very significant impact on ongoing build times. You should try to order commands so that things that change between every single build are closer to the bottom. This means that adding your code and similar steps should be held off until the end. When you rebuild an image, every single layer after the first introduced change will need to be rebuilt.

Example Dockerfile 6/7

With the **WORKDIR** instruction, you change the working directory in the image for the remaining build instructions and the default process that launches with any resulting containers

And finally you end with the **CMD** instruction, which defines the command that launches the process that you want to run within the container

```
FROM node:11.11.0

LABEL "maintainer"="anna@example.com"
LABEL "rating"="Five Stars" "class"="First Class"

USER root

ENV AP /data/app
ENV SCPATH /etc/supervisor/conf.d

RUN apt-get -y update

# The daemons
RUN apt-get -y install supervisor
RUN mkdir -p /var/log/supervisor

# Supervisor Configuration
ADD ./supervisord/conf.d/* $SCPATH/

# Application Code
ADD *.js* $AP/

WORKDIR $AP

RUN npm install

CMD ["supervisord", "-n"]
```

Example Dockerfile 7/7

Though not a hard and fast rule, it is generally considered a best practice to try to run only a single process within a container. The core idea is that a container should provide a single function so that it remains easy to horizontally scale individual functions within your architecture.

In the example, we are using supervisord as a process manager to help improve the resiliency of the node application within the container and ensure that it stays running

```
FROM node:11.11.0

LABEL "maintainer"="anna@example.com"
LABEL "rating"="Five Stars" "class"="First Class"

USER root

ENV AP /data/app
ENV SCPATH /etc/supervisor/conf.d

RUN apt-get -y update

# The daemons
RUN apt-get -y install supervisor
RUN mkdir -p /var/log/supervisor

# Supervisor Configuration
ADD ./supervisord/conf.d/* $SCPATH/

# Application Code
ADD *.js* $AP/

WORKDIR $AP

RUN npm install

CMD ["supervisord", "-n"]
```

Building an Image

- To build an example image, clone a Git repo that contains an example application called *docker-node-hello*:

```
git clone https://github.com/spkane/docker-node-hello.git \  
--config core.autocrlf=input
```

- You can configure Git to handle line endings automatically so you can collaborate effectively with people who use different operating systems using the `--config core.autocrlf=input`

What we got

- If we look at the contents while ignoring the Git repo directory, we should see the following:

```
PAVALENT-M-P201:docker-node-hello pavalent$ tree -a -I .git
```

```
.
├── .dockerignore
├── .gitignore
├── Dockerfile
├── Makefile
├── README.md
├── Vagrantfile
├── index.js
├── package.json
└── supervisord
    └── conf.d
        ├── node.conf
        └── supervisord.conf
```

Most relevant files

- The *Dockerfile* should be exactly the same as the one you just reviewed.
- The *.dockerignore* file allows you to define files and directories that you do not want uploaded to the Docker host when you are building the image. In this instance, the *.dockerignore* file contains the following line: `.git`
- You don't need the contents of the *.git* directory to build the Docker image, and since it can grow quite large over time, you don't want to waste time copying it every time you do a build.
- *package.json* defines the Node.js application and lists any dependencies that it relies on.
- *index.js* is the main source code for the application.
- The *supervisord* directory contains the configuration files for `supervisord` that you will use to start and monitor the application.

Troubleshooting Broken Builds

- We normally expect builds to just work, especially when we've scripted them, but in the real world things go wrong.
- Let's create a failing build. To do that, edit the *Dockerfile* so that the line that reads:

```
RUN apt-get -y update
```

Now reads:

```
RUN apt-get -y update-all
```

Docker build with the error...

```
PAVALENT-M-P201:docker-node-hello pavalent$ docker build -t docker-node-hello:latest .
Sending build context to Docker daemon 15.87kB
...
---> 02eb7fa25ff1
Step 6/14 : ENV SCPATH /etc/supervisor/conf.d
---> Running in 732f949423a0
Removing intermediate container 732f949423a0
---> 8564b1b8179b
Step 7/14 : RUN apt-get -y update-all
---> Running in 3d89de2fce6a
E: Invalid operation update-all
The command '/bin/sh -c apt-get -y update-all' returned a non-zero code: 100
```

→ This line that reads **Running in 732f949423a0** is telling you that the build process has started a new container, based on the image created in step 5

The next line, which reads **Removing intermediate container 732f949423a0**, is telling you that Docker is now removing the container, after having altered it based on the instruction in step 6. In this case, it was simply adding a default environment variable via `ENV SCPATH /etc/supervisor/conf.d`.

The final line, which reads → **8564b1b8179b**, is the one we really care about, because this is giving us the image ID for the image that was generated by step 6.

We need this to troubleshoot the build, because it is the image from the last successful step in the build.

Troubleshoot

Container ID if the container created from the image

```
PAVALENT-M-P201:docker-node-hello pavalent$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
<none>	<none>	8564b1b8179b	2 hours ago	904MB
node	11.11.0	9ff38e3a6d9d	13 months ago	904MB
hello-world	latest	fce289e99eb9	16 months ago	1.84kB

```
PAVALENT-M-P201:docker-node-hello pavalent$ sudo docker run --rm -ti 8564b1b8179b /bin/bash
```

```
root@a3c8c01e68d9:/# apt-get -y update-all
```

```
E: Invalid operation update-all
```

```
root@a3c8c01e68d9:/# apt-get --help
```

```
apt 1.4.9 (amd64)
```

```
Usage: apt-get [options] command
```

```
...
```

```
Most used commands:
```

```
  update - Retrieve new lists of packages
```

```
  upgrade - Perform an upgrade
```

```
...
```

```
root@a3c8c01e68d9:/# apt-get -y update
```

```
Ign:1 http://deb.debian.org/debian stretch InRelease
```

```
Get:2 http://deb.debian.org/debian stretch-updates InRelease [91.0 kB]
```

```
...
```

```
Reading package lists... Done
```

```
root@a3c8c01e68d9:/#
```

```
root@a3c8c01e68d9:/# exit
```

We can see there is no option such as **update-all**

The option were looking for was **update**

We can try it on the running container

Worked ok!

Let's go back and re-edit the *Dockerfile* so that the instruction gets back to `apt-get -y update`

Build the image and run it (i.e., create a container)

```
PAVALENT-M-P201:docker-node-hello pavalent$ docker build -t docker-node-hello:latest .
Sending build context to Docker daemon 15.87kB
Step 1/14 : FROM node:11.11.0
...
Step 14/14 : CMD ["supervisord", "-n"]
---> Running in 8ac517b422da
Removing intermediate container 8ac517b422da
---> 52f525131fd0
Successfully built 52f525131fd0
Successfully tagged docker-node-hello:latest
```

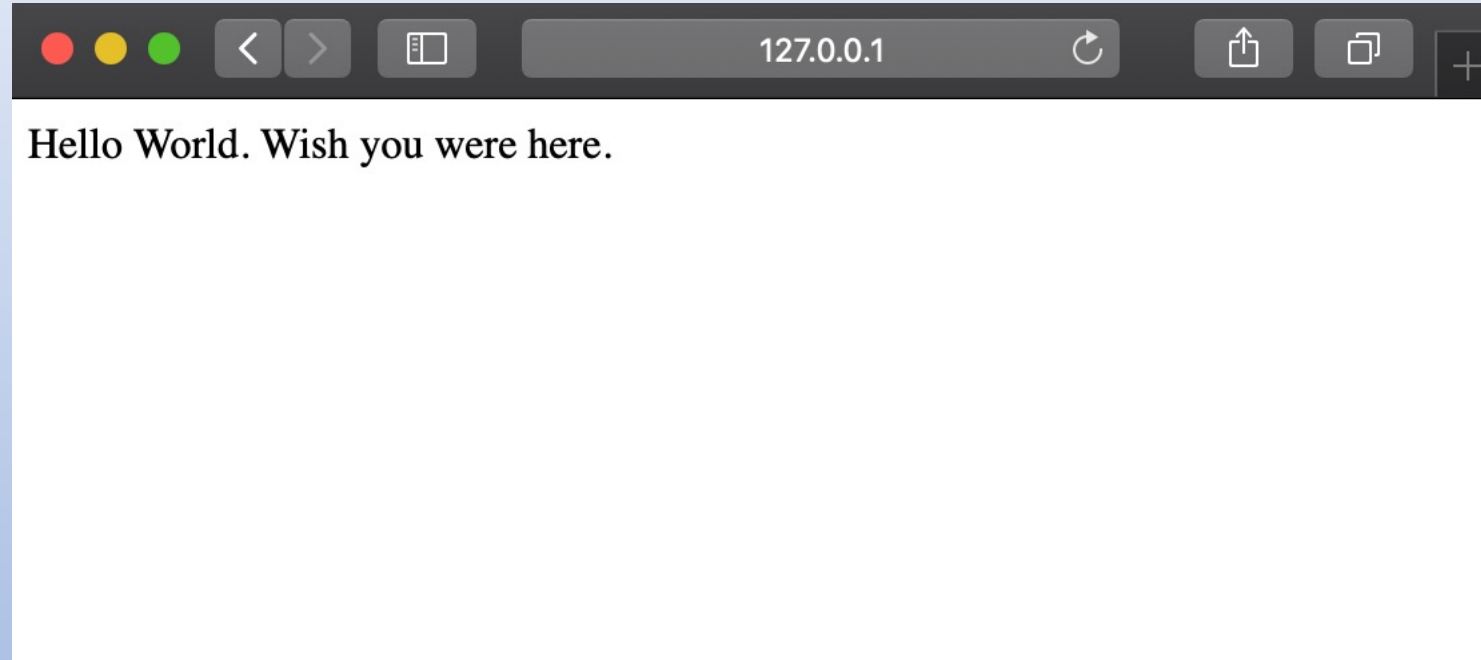
```
PAVALENT-M-P201:docker-node-hello pavalent$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
docker-node-hello	latest	52f525131fd0	41 seconds ago	928MB
node	11.11.0	9ff38e3a6d9d	13 months ago	904MB
hello-world	latest	fce289e99eb9	16 months ago	1.84kB

```
PAVALENT-M-P201:docker-node-hello pavalent$ docker run -d -p 8080:8080 docker-node-hello:latest
fee641ec1d855f166f083b56b5847e2e138b49fb9fa08ba8a2f4edb46de662c4
```

```
PAVALENT-M-P201:docker-node-hello pavalent$ docker ps --format "table {{.ID}}\t{{.Image}}\t{{.Status}}"
CONTAINER ID          IMAGE                  STATUS
fee641ec1d85          docker-node-hello:latest Up About a minute
```

We can now test the app



Environment Variables ^{1/2}

If we read the *index.js* file, we will notice that part of the file refers to the variable **\$WHO**, which the application uses to determine who the application is going to say Hello to:

```
var DEFAULT_PORT = 8080;
var DEFAULT_WHO = "World";
var PORT = process.env.PORT || DEFAULT_PORT;
var WHO = process.env.WHO || DEFAULT_WHO;

// App
var app = express();
app.get('/', function (req, res) {
  res.send('Hello ' + WHO + '. Wish you were here.\n');
});

app.listen(PORT)
console.log('Running on http://localhost:' + PORT);
```

Environment Variables 2/2

```
PAVALENT-M-P201:docker-node-hello pavalent$ docker ps --format "table {{.ID}}\t{{.Image}}\t{{.Status}}"
CONTAINER ID          IMAGE                STATUS
fee641ec1d85         docker-node-hello:latest Up About a minute
```

Identify the container...

```
PAVALENT-M-P201:docker-node-hello pavalent$ docker stop fee641ec1d85
fee641ec1d85
```

... and stop it

```
PAVALENT-M-P201:docker-node-hello pavalent$ docker run -d -p 8080:8080 -e WHO="Sean and Karl" docker-
node-hello:latest
Ee02f82f08bb45d215b744d0e5bfc030403153d7ac1e25a883cfa4e21b242984
```

Create a new container
passing the variable **WHO**

```
PAVALENT-M-P201:docker-node-hello pavalent$ docker ps --format "table
{{.ID}}\t{{.Image}}\t{{.Status}}\t{{.Ports}}"
```

CONTAINER ID	IMAGE	STATUS	PORTS
ee02f82f08bb	docker-node-hello:latest	Up 4 seconds	0.0.0.0:8080->8080/tcp

Container name

- By default, Docker randomly names your container by combining an adjective with the name of a famous person.
- If you want to give your container a specific name, you can use the `--name` argument.

```
PAVALENT-M-P201:~ pavalent$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
docker-node-hello	latest	52f525131fd0	About an hour ago	928MB
node	11.11.0	9ff38e3a6d9d	13 months ago	904MB
hello-world	latest	fce289e99eb9	16 months ago	1.84kB

```
PAVALENT-M-P201:~ pavalent$ docker create --name="awesome-service" node:11.11.0 sleep 120
4aa1ffdec316e4dc6cc798c3e7a4d69efd7b45b892fd6a68bb9c94f310fe9431
```

```
PAVALENT-M-P201:~ pavalent$ docker ps --format "table
```

```
{{.ID}}\t{{.Image}}\t{{.Status}}\t{{.Ports}}\t{{.Names}}" -a
```

CONTAINER ID	IMAGE	STATUS	PORTS	NAMES
4aa1ffdec316	node:11.11.0	Created		awesome-service
ee02f82f08bb	docker-node-hello:latest	Up 57 minutes	0.0.0.0:8080->8080/tcp	competent_poincare
fee641ec1d85	docker-node-hello:latest	Exited (0) 58 minutes ago		zealous_chatterjee
3d89de2fce6a	8564b1b8179b	Exited (100) 4 hours ago		sleepy_bell

Labels 1/2

- When new Docker containers are created, they automatically inherit all the labels from their parent image. It is also possible to add new labels to the containers so that you can apply metadata that might be specific to that single container:

```
PAVALENT-M-P201:~ pavalent$ docker run -d --name has-some-labels -l deployer=Ahmed -l tester=Asako \
    hello-world
8fb919b4f67ab5763058f6a2dd2ec823b3d66cd21fe0d81dfbd63ebb610fa2dd
```

- We can then search for and filter containers based on this metadata, using commands like `docker ps`.

```
PAVALENT-M-P201:~ pavalent$ docker ps -a -f label=deployer=Ahmed
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
8fb919b4f67a	hello-world	"/hello"	11 seconds ago	Exited ...		has-some-labels

Labels 2/2

- We can use the `docker inspect` command on the container to see all the labels that a container has:

```
PAVALENT-M-P201:~ pavalent$ docker inspect 8fb919b4f67a
  "Labels": {
    "deployer": "Ahmed",
    "tester": "Asako"
  }
```

Hostname

- By default `/etc/hostname` contains the container's ID and is not fully qualified with a domain name:

```
PAVALENT-M-P201:~ pavalent$ docker run --rm -ti ubuntu:latest /bin/bash
root@1bacc2f1486:/# hostname -f
1bacc2f1486
```

```
PAVALENT-M-P201:~ pavalent$ docker ps --format "table {{.ID}}\t{{.Image}}\t{{.Status}}"
CONTAINER ID      IMAGE               STATUS
1bacc2f1486       ubuntu:latest      Up About a minute
```

To set the hostname specifically, we can use the `--hostname` argument to pass in a more specific value:

```
PAVALENT-M-P201:~ pavalent$ docker run --rm -ti --hostname="mycontainer.example.com" ubuntu /bin/bash
root@mycontainer:/# hostname -f
mycontainer.example.com
```

Domain Name Service

- Just like `/etc/hostname`, the `/etc/resolv.conf` file that configures the Domain Name Service (DNS) resolution is managed via a bind mount between the host and container.
- We could use a combination of the `--dns` and `--dns-search` arguments to override this behavior in the container:

```
PAVALENT-M-P201:~ pavalent$ docker run --rm -ti --dns=8.8.8.8 --dns=8.8.4.4 --dns-search=example1.com \
    --dns-search=example2.com ubuntu:latest /bin/bash
root@25db2b196793:/# more /etc/resolv.conf
search example1.com example2.com
nameserver 8.8.8.8
nameserver 8.8.4.4
```

Storage Volumes ^{1/2}

- There are times when the default disk space allocated to a container, or the container's ephemeral nature, is not appropriate for the job at hand, so you'll need storage that can persist between container deployments.
- For times like this, you can leverage the `-v` command to mount directories and individual files from the host server into the container. The following example mounts `/mnt/session_data` to `/data` within the container:

```
$ docker run --rm -ti -v /mnt/session_data:/data ubuntu:latest /bin/bash
root@0f887071000a:/# mount | grep data
/dev/sda9 on /data type ext4 (rw,relatime,data=ordered)
root@0f887071000a:/# exit
```

- Neither the host mount point nor the mount point in the container needs to preexist for this command to work properly. If the host mount point does not exist already, then it will be created as a directory. This could cause you some issues if you were trying to point to a file instead of a directory.

Storage Volumes 2/2

- If the container application is designed to write into /data, then this data will be visible on the host filesystem in /mnt/session_data and will remain available when this container stops and a new container starts with the same volume mounted
- It is possible to tell Docker that the root volume of your container should be mounted read-only so that processes within the container cannot write anything to the root filesystem. This prevents things like logfiles, which a developer may be unaware of, from filling up the container's allocated disk in production. When it's used in conjunction with a mounted volume, you can ensure that data is written only into expected locations:

```
$ docker run --rm -ti --read-only=true -v /mnt/session_data:/data ubuntu:latest /bin/bash
root@df542767bc17:/# mount | grep " / "
overlay on / type overlay (ro,relatime,lowerdir=...,upperdir=...,workdir=...)
root@df542767bc17:/# mount | grep data
/dev/sda9 on /data type ext4 (rw,relatime,data=ordered)
root@df542767bc17:/# exit
```

Containers should be designed to be stateless whenever possible. Managing storage creates undesirable dependencies and can easily make deployment scenarios much more complicated.

Resource Quotas

- Virtual machines have the advantage that you can easily and very tightly control how much memory and CPU, among other resources, are allocated to the virtual machine.
- When using Docker, we must instead leverage the `cgroup` functionality in the Linux kernel to control the resources that are available to a Docker container.
- The `docker create` and `docker run` commands directly support configuring `CPU, memory, swap, and storage I/O` restrictions when you create a container.
- Constraints are normally applied at the time of container creation. If we need to change them, we can use the `docker container update` command or deploy a new container with the adjustments.

While Docker supports various resource limits, we must have these capabilities enabled in the host kernel in order for Docker to take advantage of them. We might need to add these as command-line parameters to the host kernel on startup. To figure out if kernel supports these limits, we can run `docker info`. If we are missing any support, we will get warning messages at the bottom, like:

```
WARNING: No swap limit support
```

Quotas Examples: CPU

- While CPU shares were the original mechanism in Docker for managing CPU limits, today we can now simply tell Docker how much CPU we would like to be available to a container, and it will do the math required to set the underlying cgroups correctly:

```
$ docker run -d --cpus=".25" progrium/stress --cpu 2 --io 1 --vm 2 --vm-bytes 128M --timeout 60s
```

- The `docker update` command can be used to dynamically adjust the resource limits of one of more containers. We could adjust the CPU allocation on two containers simultaneously, for example, like so:

```
docker update --cpus="1.5" 6b785f78b75e 92b797f12af1
```


Quotas Examples: Memory

- We can control how much memory a container can access in a manner similar to constraining the CPU. There is, however, one fundamental difference: while constraining the CPU only impacts the application's priority for CPU time, the memory limit is a hard limit.
- Because of the way the virtual memory system works on Linux, it's possible to allocate more memory to a container than the system has actual RAM. In this case, the container will resort to using swap, just like a normal Linux process.

```
$ docker run --rm -ti --memory 512m progrium/stress \
    --cpu 2 --io 1 --vm 2 --vm-bytes 128M --timeout 120s
```

- When we use the `--memory` option alone, you are setting both the amount of RAM and the amount of swap that the container will have access to. So by using `--memory 512m` here, we've constrained the container to 512 MB of RAM and 512 MB of additional swap space. If we want those values to be different we can use the `--memory-swap` option:

```
$ docker run --rm -ti --memory 512m --memory-swap=768m progrium/stress \
    --cpu 2 --io 1 --vm 2 --vm-bytes 128M --timeout 120s
```

Quotas: Block I/O

- Many containers are just stateless applications and won't have a need for I/O restrictions. But Docker also supports limiting block I/O in a few different ways via the cgroups mechanism.
- Today we can implement it by limiting the maximum number of bytes or operations per second that are available to a container via its cgroup. The following settings let us control that:

```
--device-read-bps Limit read rate (bytes per second) from a device
--device-read-iops Limit read rate (IO per second) from a device
--device-write-bps Limit write rate (bytes per second) to a device
--device-write-iops Limit write rate (IO per second) to a device
```

- We can test how these impact the performance of a container by running:

```
$ time docker run -ti --rm spkane/train-os:latest bonnie++ -u 500:500 -d /tmp -r 1024 -s 2048 -x 1
```

```
...
real 0m27.715s
user 0m0.027s
sys 0m0.030s
```

```
$ time docker run -ti --rm --device-write-iops /dev/sda:256
\spkane/train-os:latest bonnie++ -u 500:500 -d /tmp -r 1024 -s 2048 -x 1
```

```
...
real 0m58.765s
user 0m0.028s
sys 0m0.029s
```