# Creación de imaxes

Tal e como vimos nesta unidade, as imaxes son:

- A parte **estática** da containerización.
- O **conxunto de software** (sistema, utilidades e aplicacións) no que se basan os containers para face-lo seu traballo.
- Dende un punto de vista físico, son un **ficheiro comprimido**, onde están empaquetadas, tódalas dependencias da imaxe.
- A súa estructura está formada por **capas**.

O docente pode comprender que, a creación e mantemento de imaxes, é unha das tarefas básicas na containerización.

Docker ofrece unha serie de ferramentas e utilidades para poder construir e manter imaxes:

- O conxunto **docker-commit** e **docker-push**
- O Dockerfile

Ámba-las dúas serán obxecto de estudio no presente tema.

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0]

# Obxectivos

## Objetivos

- Entende-lo ciclo de construcción dunha imaxe
- Familiarizarse cos comandos de evolución das imaxes
- Entrar en contacto co DSL do Dockerfile
- Subir imaxes a repos

Obra publicada con Licencia Creative Commons Reconocimiento Compartir igual 4.0

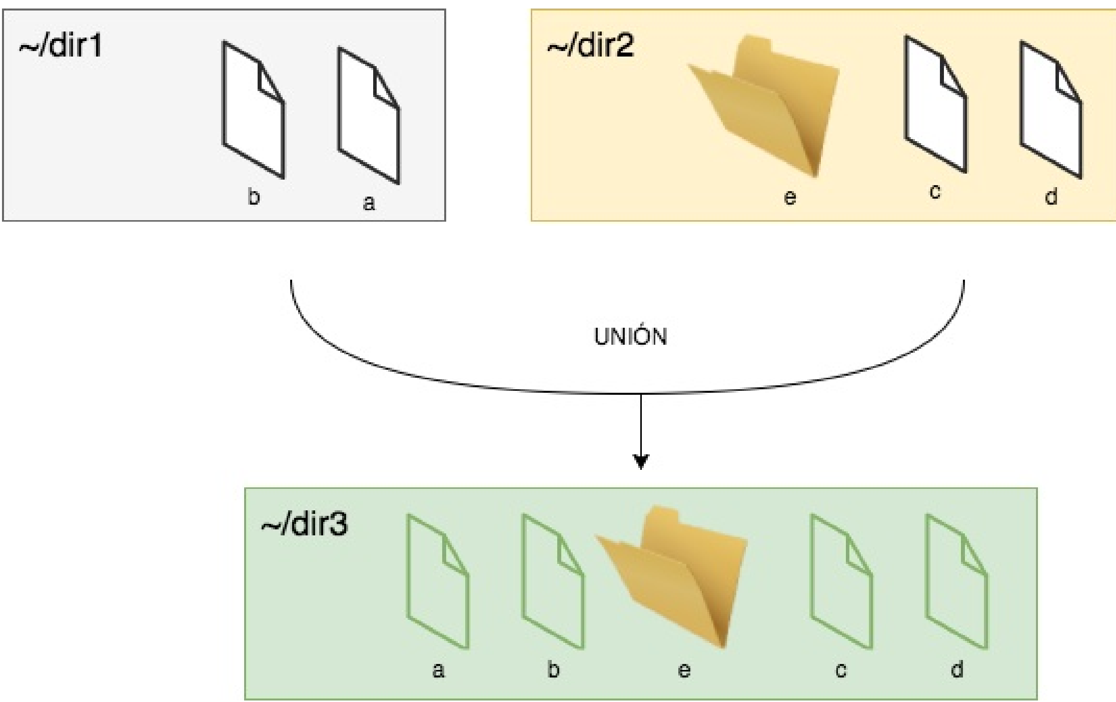
## Revisitando as imaxes: o overlayfs

No tema anterior vimos o concepto de imaxe e cómo se empregan para acada-los obxectivos da containerización.

Nembargantes, nesta sección imos profundizar no concepto de imaxe botando unha ollada ás tecnoloxías que as fan posibles: referímonos ós **union mounts**.

# 1. Os unions mounts

Un union mount é un sistema que permite combinar diferentes directorios nun só que se ve como unha mestura dos mesmos: de ahí o seu nome.




## 1.A) Xogando cos union mounts

Para facer esta práctica guiada compre un sistema linux cun kernel >= 3.18. Non se pode realizar dentro dun contedor!

A tecnoloxía empregada hoxe en día por Docker é o OverlayFS, incluído no propio kernel dende a 3.18.

Imos crear dous directorios:

```
~# mkdir a b
```

Dentro de o directorio **~/a** imos crear dous ficheiros baleiros:

```
~/a# touch a b
```

Dentro do directorio **~/b** imos crear dous ficheiros baleiros (c e d) e un novo directorio e

```
~/b# touch c d

~/b# mkdir e
```

Agora creamos un directorio c ó mesmo nivel que **~/a** e **~/b**

```
~# mkdir c
```

Lanzando o seguinte comando:

```
~# mount -t overlay overlay -o lowerdir=./a,upperdir=./b,workdir=./c ./c
```

Se listamos os contidos do **~/c** veremos que temos unha estrutura como a do diagrama.

Qué fixemos con este comando?

- Montamos unha unidade nun directorio ~/c
- De tipo **overlay**
- Especificamos como obxectivo: un directorio de só lectura (~/a) un directorio de escritura-lectura (~/b) e un directorio especial de trasvase (~/c)

## 1.B) O mecanismo COW e o copy-up

Por suposto, xorden preguntas:

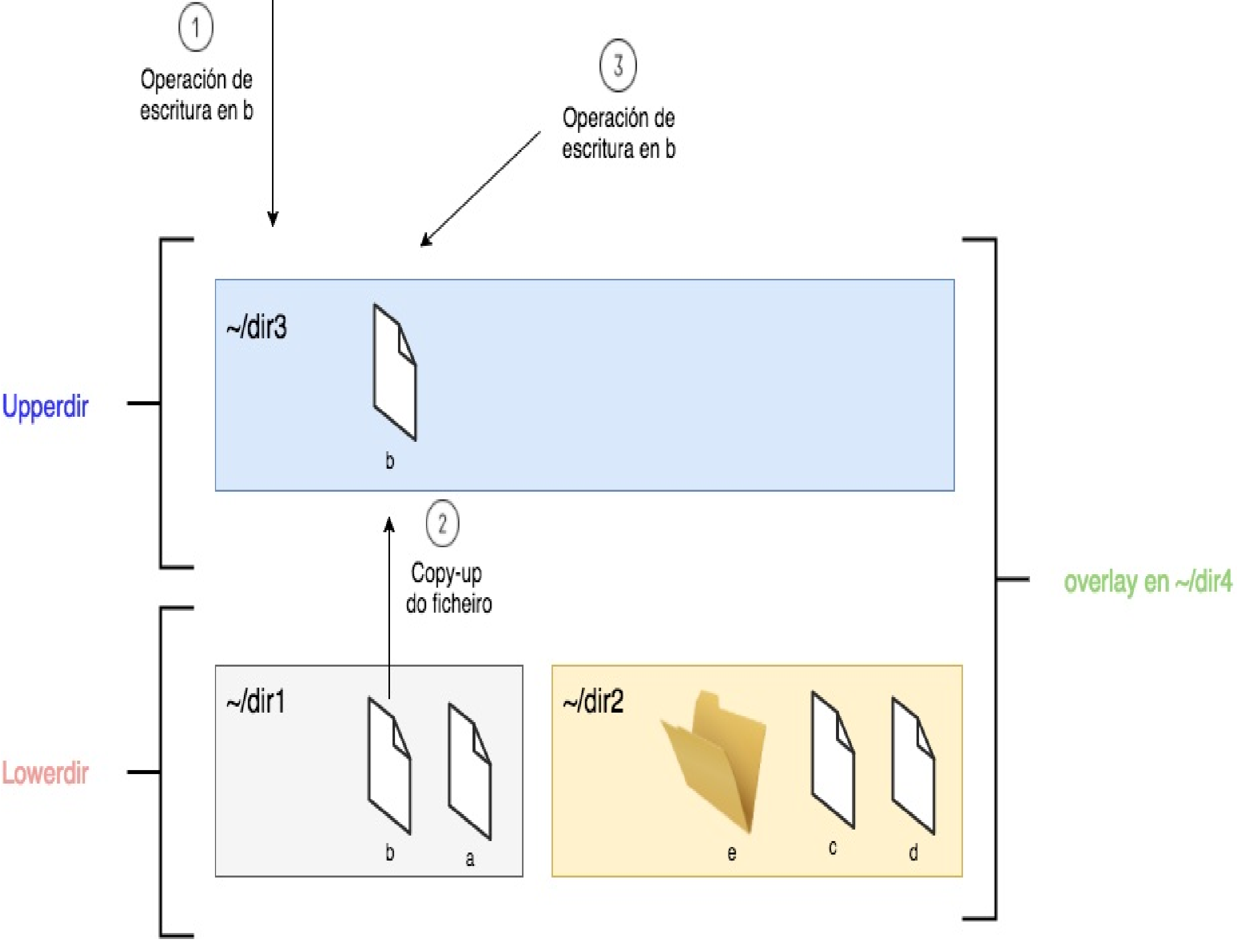
- Qué pasa se modifico un ficheiro?
- Qué pasa se elimino un directorio ou ficheiro?
- Podo establecer elementos de solo lectura?

Cando falamos de lowerdir e upperdir estámonos a referir a dous niveles de montaxe no overlayfs.

## i) O Lowerdir

As capas montadas neste nivel non poder ser modificadas ou eliminadas.

En caso de modificar un ficheiro pertencente a esta capa, o que se vai facer e copialo (COW - copy on write) a unha capa superior, ó upperdir.




O copy-up recibe o seu nome de esa operación de copia-lo ficheiro dunha capa de lowerdir a unha de upperdir antes de modificala. Esto permite dúas cousas:

a. Sólo se copian os ficheiros que son realmente modificados (Copy

On Write -> COW)

b. Pódense manter capas de só lectura mesturadas con capas de escritura.
c. Todas estas operacións son transparentes para o proceso que corre no volume de overlayfs.

## ii) O upperdir

Trátase dos directorios de escritura/lectura. Os copy-up realízanse nestas capas.

**Actividad**

Probemos a crear un volume con OverlayFS:

- Os lowerdir será /tmp/a
- Crear tres ficheiros baleiros (f1, f2 e f3) en /tmp/a
- O upperdir será /tmp/b
- O workdir será /tmp/c
- Montar todo en /tmp/d

Hai que ter creados tódolos directorios antes de facé-lo mount!!

1. Dende /tmp/d vemos todos os contidos de /tmp/a?
2. Podemos introducir contidos en /tmp/d/f1?
3. Qué hai no directorio /tmp/b? Por qué?

# 2. Imaxes e OverlayFS

Como contaramos, o contedor créase a partir dunha imaxe que é un sistema de ficheiros de só lectura.

Nembargantes, dentro de contedor podemos instalar, borrar e modificar todo o que queiramos.

Agora, entendemos que tódolos cambios se están a guardar na capa do contedor que é un upperdir montado precisamente por enriba da imaxe que é capa de lowerdir.

E os volumes? Os volumes, dependendo de se os montamos de só lectura ou non, estarán en lowerdir ou en upperdir.

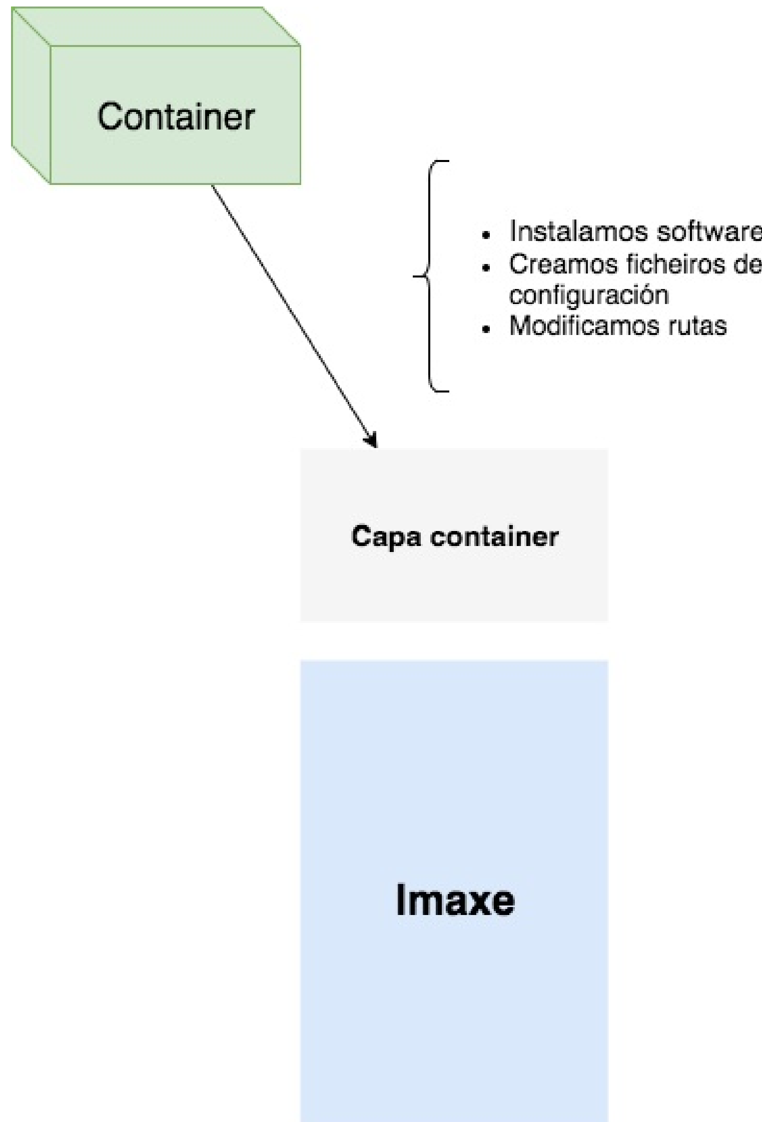
Obra publicada con Licencia Creative Commons Reconocimiento Compartir igual 4.0

## Emprego de comandos

No tema anterior falamos de que container sempre parte dunha **imaxe** e ten asociada unha capa de **container,** de tal xeito que, mediante o mecanismo de **copy-on-write** (COW) os cambios que faga no sistema de ficheiros quedan reflectidos nesa capa e non na imaxe que, dende o punto de vista do container, é algo **inmutable**.

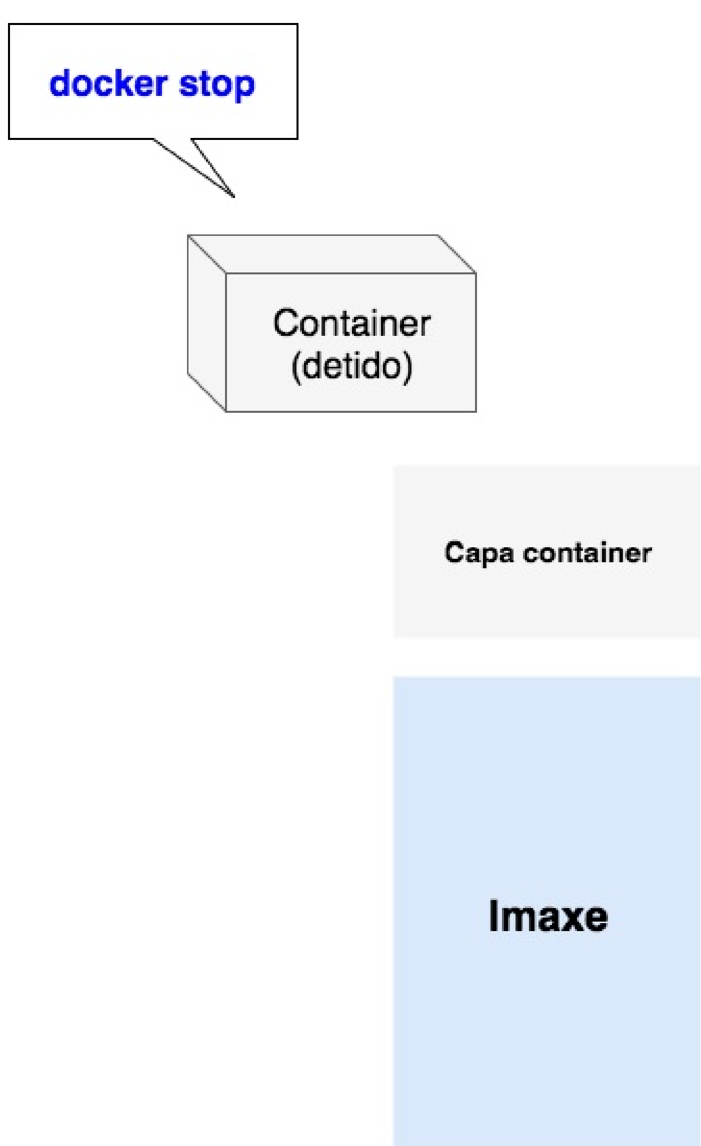
Non obstante, as imaxes pódense evolucionar. Para facelo a clave está, precisamente, nesa capa de container.

Partamos dun container que fai cambios no seu sistema de ficheiros.



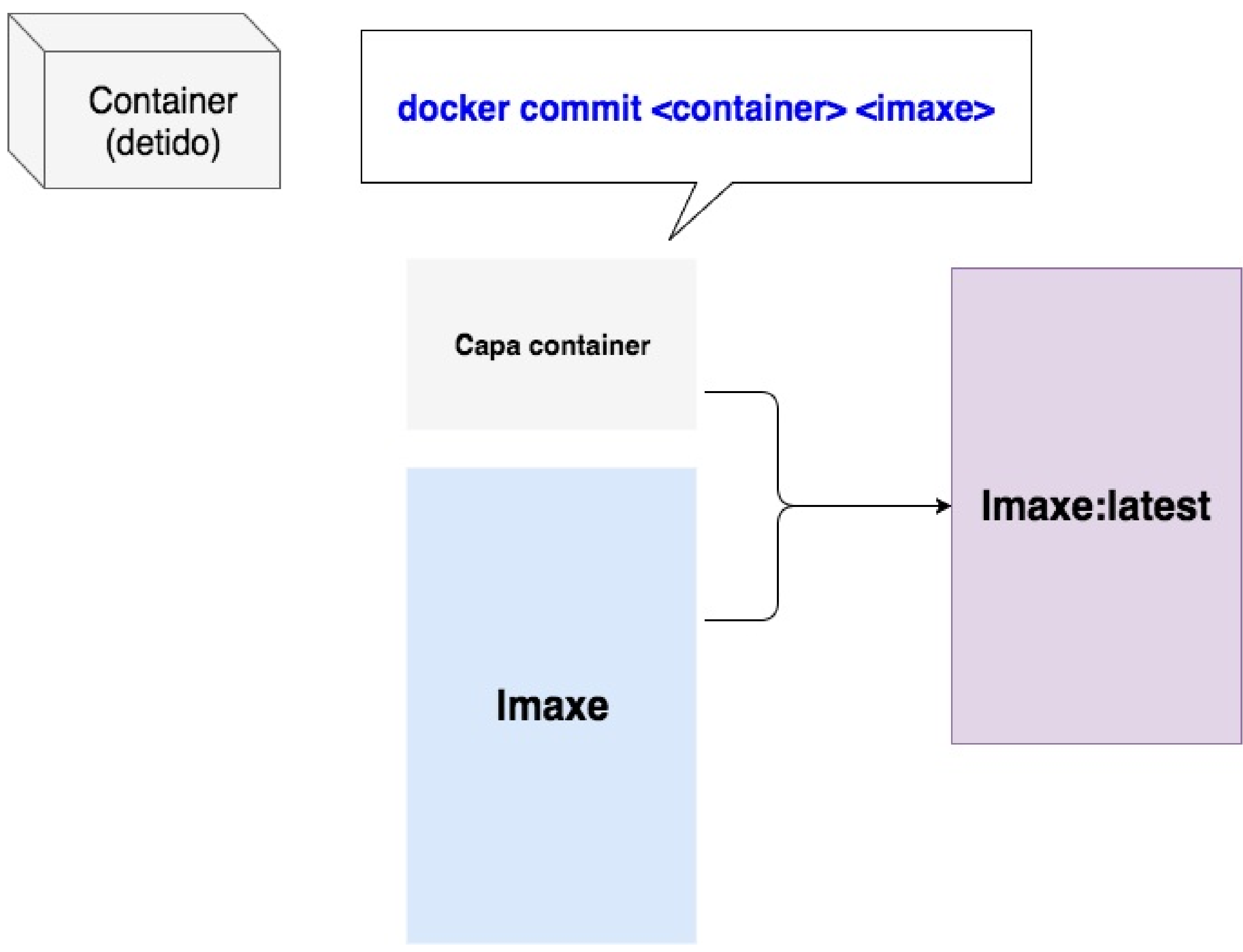

Como sabemos, esos cambios quedan reflectidos no súa capa de container.

Se detemos agora o container, de tal xeito que non poida facer máis cambios:



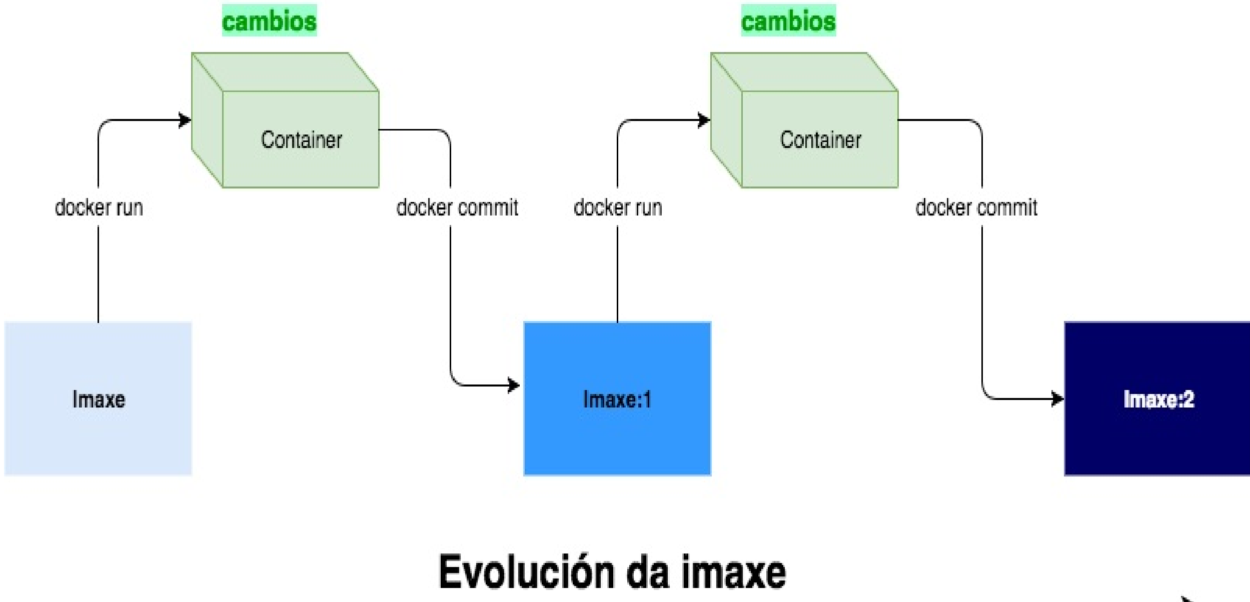

Nótese que o container está **detido**, non **destruido**, polo tanto o container non está a correr pero está presente no motor de Docker, e polo tanto tamén a súa capa de datos.

Se agora collemos esa capa de datos propia do container e facemos un **commit**, o que estamos a facer e producir unha nova imaxe que sí que incorpora os cambios da capa de container á súa propia estrutura interna.




En definitiva, **acabamos de evoluciona-la imaxe**. E os novos containers baseados nesa nova imaxe sí verán os cambios que fixeramos no container orixinal.

Este precisamente, é o ciclo de evolución das imaxes en Docker.




Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0]

## Exemplo: creando imaxe mediante comandos

# O noso "Hello World"

Imos ilustrar o proceso de creación dunha imaxe a través dun Hello World!, pero á galega.

Animamos ó docente a que probe a facer este exemplo na súa máquina porque será relevante para despoís face-la práctica de final da sección.

Imos construir unha imaxe que conteña unha sinxela aplicación baseada en apache2 que exporta unha páxina web cun "Hello Wolrd" á galega.

O plan de traballo é o seguinte:

1. Partiremos dunha Debian:Jessie
2. Instalaremos un apache
3. Introduciremos unha configuración
4. Copiaremos o noso Hello World!!
5. Probaremo-la nosa imaxe lanzando un container

# I. Poñendo as cousas en orde

O primeiro que precisamos é a imaxe de Debian que imos a empregar. Ímola descargar dende o Dockerhub:

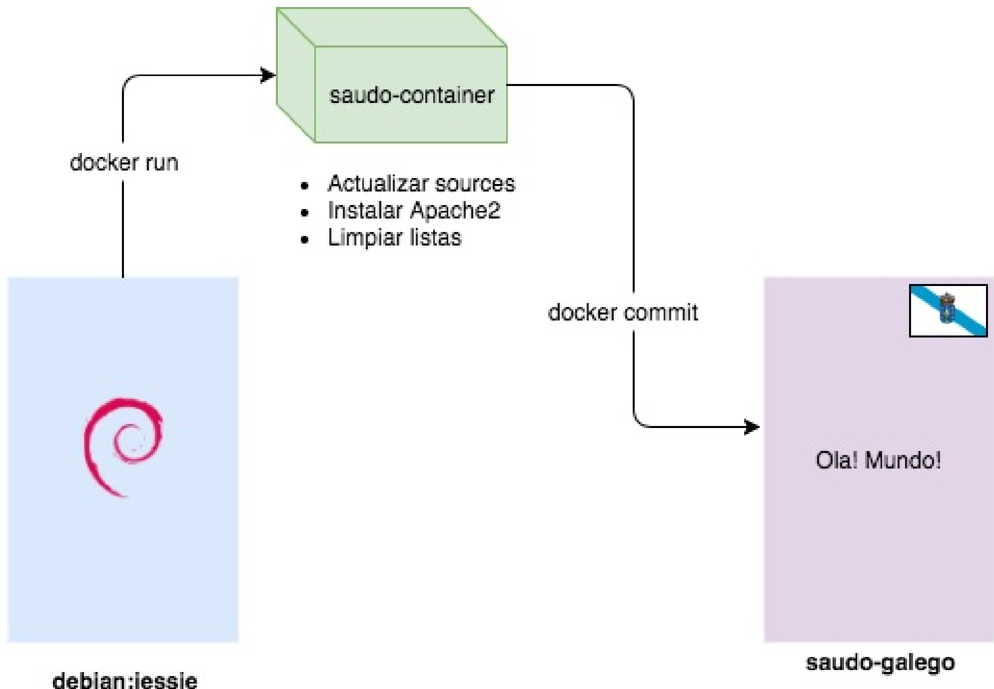
```
~$ docker pull debian:jessie
```

Agora xa temos a imaxe de Debian na nosa máquina local. Podemos empezar a traballar!

Para segui-lo noso ciclo de evolución, imos a empregar o container **saudo-container,** e a imaxe a producir vaise chamar **saudo-galego**.

# II. Instalando Apache2

Compre que instalemos o apache2 no noso container e que despois fagamos un commit dos cambios á nosa imaxe. Nun esquema:




Arrincamos un container para face-lo traballo de instalar o apache2:

- O container vai ter o nome **saudo-container**
- Ten que estar en modo **interactivo** para poder empregalo e que, unha vez que saigamos do container, este último quede **detido**.
- Ademáis, queremos que o **punto de entrada** ó container sexa o

bash

```
~$ docker run --name saudo-container -ti debian:jessie bash
```

Unha vez executado, estaremos dentro do container, e facemo-lo noso traballo:

- Actualizamos fontes de software

```
root@23b874a05279:/# apt-get update
```

- Instalamos o apache

```
root@23b874a05279:/# apt-get install apache2
```

- Limpiamos as fontes

```
root@23b874a05279:/# rm -r /var/lib/apt/lists/*
```

Agora temo-lo noso container co estado que queremos. Salimos mediante exit ou CTRL+D

Xa na máquina anfitrión, facemo-lo **commit** á nova imaxe.

```
~$ docker commit saudo-container saudo-galego
```

Temos unha nova imaxe co noso software instalado.

Podemos prescindir do container saudo-container posto que os cambios xa están na nova imaxe. Borrámolo:

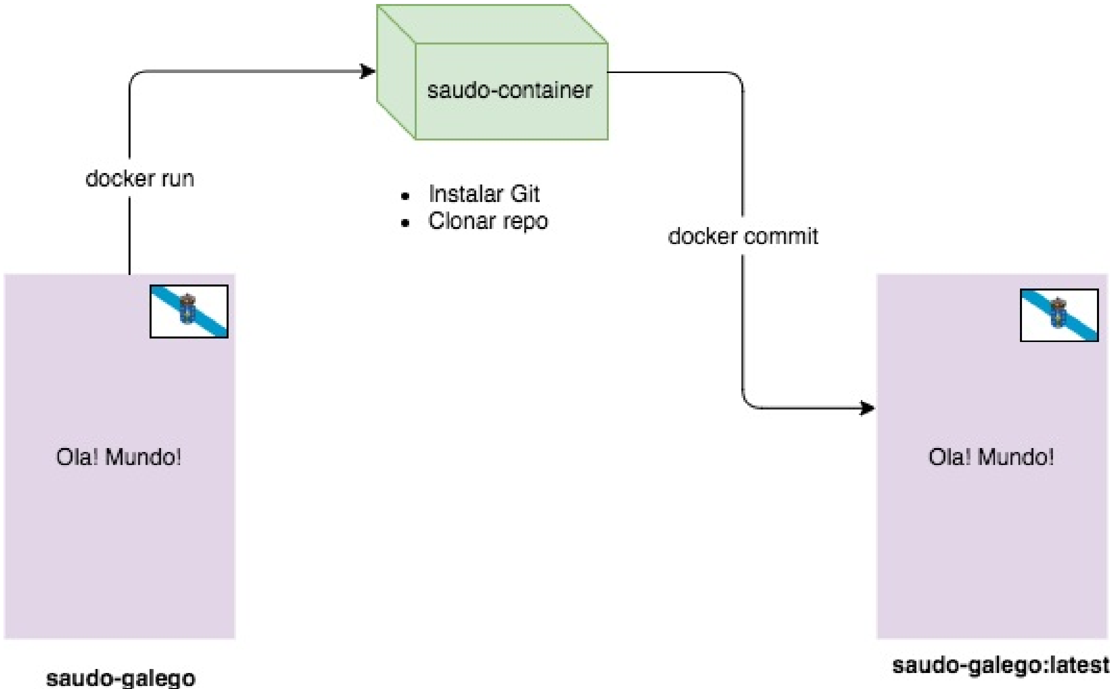
```
~$ docker rm -v saudo-container
```

# III. Clonando o repo do Saudo Galego

Coa base que temos do paso anterior, imos arrincar de novo un container (xa baseado a nosa imaxe) e face-las seguintes tarefas:

1. Instalar o git como VCS
2. Clonar o repo da páxina de saúdo
3. Face-lo commit á imaxe

Nun esquema, quedaría:




Para comezar, arrincamos un novo container baseado na imaxe de saudo-galego que xa producimos no paso anterior:

- O container se chamará tamén **saudo-container**
- Necesitamos que sexa interactivo para poder traballar dentro do container
- O comando a lanzar é o bash

Na nosa máquina, tecleamos:

```
~$ docker run -ti --name saudo-container saudo-galego bash
```

Imos empregar o software de [Git] polo que instalamos o paquete:

```
root@dd8ea1731cf3:/# apt-get install -y git
```

Agora clonamos o repo do proxecto saudo-gl nunha ruta do noso container:

```
root@dd8ea1731cf3:/# cd /opt && git clone https://github.com/prefapp/saudo-gl.git
```

Xa estamos dentro dun container baseado na imaxe do paso anterior. Polo tanto ten xa instalado un Apache2. O Apache2, por defecto, sirve os contidos dende **/var/www/html**

Movémonos a esa ruta:

```
root@dd8ea1731cf3:/# cd /var/www/html/
```

E copiamos os contidos do proxecto saudo-gl na ruta onde o Apache2 serve ficheiros:

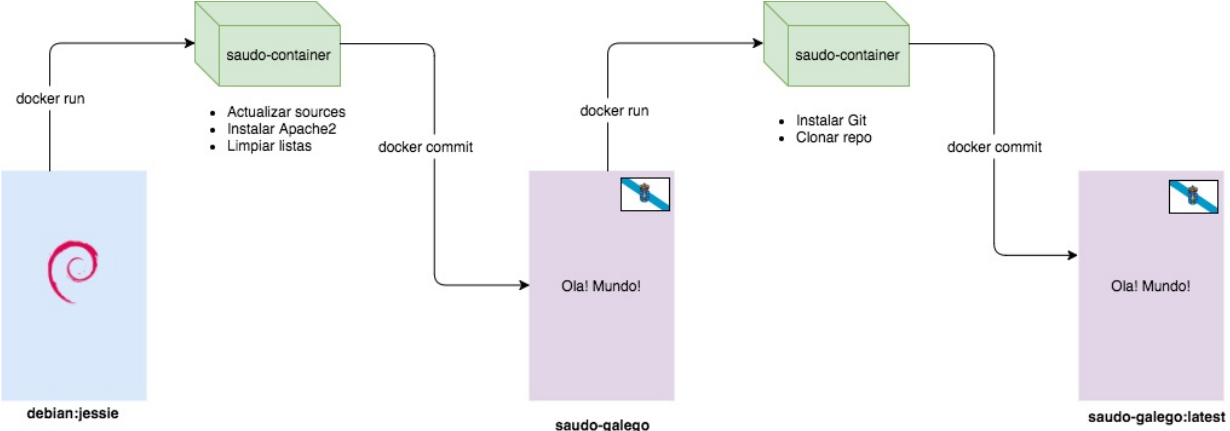
```
root@dd8ea1731cf3:/var/www/html# cp -r /opt/saudo-gl/* .
```

Et voilà! Saimos do container con exit ou CTRL+D e facemos un commit dos novos cambios á nosa imaxe:

```
~$ docker commit saudo-container saudo-galego
```

Coma sempre, borramos o container de traballo porque non o precisamos máis. É listo!

O proceso completo quedaría como segue:




# IV. Empregando a imaxe de saudo-galego

Para lanzar un container coa nosa aplicación, basta lembrar dúas cousas:

- Compre establecer o **entrypoint** declarando que debe ser arrinca-lo Apache2
- É convinte asociarlle un porto diferente do 80 para evitar colisións con outros servicios que o docente poida ter levantados na súa máquina

Introducindo isto na nosa máquina:

```
~$ docker run --rm -p 8000:80 -d saudo-galego apachectl -DFOREGROUND
```

Teríamos un container correndo coa nosa imaxe preparada e escoitando no porto 8000. Se vamos ó noso navegador e introducimos [http://localhost:8000](http://localhost:8000) a aplicación nos saludará.

## Actividad

Probe a lanzar 10 instancias da mesma aplicación en 10 portos consecutivos (por exemplo dende o 8000 ó 8010)

Comprobe que os 10 apaches están a funcionar correctamente.

Obra publicada con Licencia Creative Commons Reconocimiento Compartir igual 4.0

Tal e como vimos na sección do saudo-gl, a confección dunha imaxe pode resultar un proceso tedioso e complexo que implica o lanzamento de moitos comandos e que pode resultar moi difícil de replicar.

Por sorte, Docker aporta unha ferramenta que nos permite traballar de xeito moito máis sinxelo: o Dockerfile.

# 1. O Dockerfile

O Dockerfile é un ficheiro de texto que contén a receita de cómo construir unha imaxe.

O conxunto de instruccións do Dockerfile constitúen un DSL que nos vai permitir expresar dun xeito razoablemente doado os pasos que hai que dar para producir-la imaxe.

Unha vez redactado o Dockerfile, basta con empregar o comando docker build para producir unha imaxe con él. Docker creará (e destruirá) os containers de traballo que precise para construir a nosa imaxe mantendo únicamente o resultado final: a imaxe que queremos.

## 1.a) A sintaxe dun Dockerfile

O Dockerfile agrupa un conxunto de sentencias nun ficheiro de texto. Cada unha das sentencias son interpretadas e executadas por orde polo Docker producindo unha imaxe de saída.

Cada sentencia implica unha nova capa na imaxe.

Nun exemplo:

```
# imaxe da que se parte
FROM ubuntu

# mantedor do Dockerfile
LABEL maintainer=fmaseda@4eixos.com

# actualizamos as sources
RUN apt-get update

# instalamos o nginx
RUN apt-get install -y nginx

# declaramos que o punto de entrada ó container e un nginx en foreground
ENTRYPOINT ["/usr/sbin/nginx","-g","daemon off;"]

# o porto para conectar o container co mundo exterior e o 80
EXPOSE 80
```

Se poñemos ese contido nun ficheiro que se chame Dockerfile e tecleamos:

```
~$ docker build -t imaxe_propia .
```

O Docker vainos a producir unha imaxe de nome **imaxe_propia** que terá un nginx correndo no porto 80 e baseada na imaxe de Ubuntu.

O DSL do Dockerfile é sinxelo pero moi completo. [Aquí](#) pode ver unha relación dos comandos.

## Actividad

Probe a construir a imaxe do nginx con este Dockerfile.

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0]

# O Dockerfile: construíndo a imaxe do "gatiño do día"

## 1.b) Empregando a sintaxe de Dockerfile para construí-la imaxe de "gatiño do día"

Neste exemplo imos construir unha imaxe que teña Python e corra unha aplicación feita en Flask que é un microframework de aplicacións e páxinas web feito en Python.

Precisamos:

1. Partir dunha ubuntu
2. Instalar o software necesario de python
3. Instalar o git para clonar un repositorio
4. Clonar un repositorio de código: https://github.com/prefapp/catweb.git
5. Establecer como punto de entrada por defecto dos contedores baseados nesta imaxe a execución do app.py

Creamos un ficheiro baleiro co nome Dockerfile e imos paso a paso:

```
FROM ubuntu
```

O primeiro que declaramos é que a nosa imaxe vaise basear nunha ubuntu. A sentencia FROM establece unha imaxe de base, por suposto poderíamos incluir imaxes dun registry privado.

Imos establecer unha ruta de traballo:

```
WORKDIR /home
```

Nesta sentencia establecemos o directorio de traballo actual. O WORKDIR establece ónde está o pwd dunha serie de comandos do propio Dockerfile.

Agora instalamos o software que precisamos:

```
RUN apt-get update -y && apt-get install -y python-pip python-dev build-essential git
```

A sentencia RUN corre un comando dentro dun contedor e almacena os resultados (a capa de contedor) na imaxe que se está a construir.

Agora clonamos o repositorio de código da nosa aplicación (en /home)

```
RUN git clone https://github.com/prefapp/catweb.git
```

Poñemos os directorio de traballo no repo clonado:

```
WORKDIR /home/catweb
```

Esta aplicación ten un ficheiro **requirements.txt** que nos indica qué dependencias de Python se precisan para que funcione. Imos executa-lo

pip para instalar o necesario:

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

Por último imos establecer o comando de arranque:

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

A sentencia CMD permítenos establecer un **único comando de arranque por Dockerfile**.

Este comando de arranque implica que, se no docker-run ou no docker-create non expresamos outra cousa, o contedor baseado nesta imaxe vai lanzar o comando que establezcamos nesta sentencia. Digamos que é un init por defecto.

⚠ Distinto do CMD é o ENTRYPOINT. Unha discusión das diferencias pódese ver aquí.

Con isto, temos xa o noso Dockerfile:

```
FROM ubuntu

WORKDIR /home

RUN apt-get update -y && apt-get install -y python-pip python-dev build-essential git

RUN git clone https://github.com/prefapp/catweb.git

WORKDIR /home/catweb

RUN pip install -r requirements.txt

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

Poñéndonos no directorio do Dockerfile, construímo-la imaxe:

```
~# docker build . -t web-gatinhos
```

E agora lanzamos a nosa web para o gato do día:

```
~# docker run -d -p 5000:5000 web-gatinhos
```

Se imos a un navegador e miramos no localhost:5000, veremos unha foto dun gatiño. Cada vez que recarguemo-la páxina teremos unha nova foto.

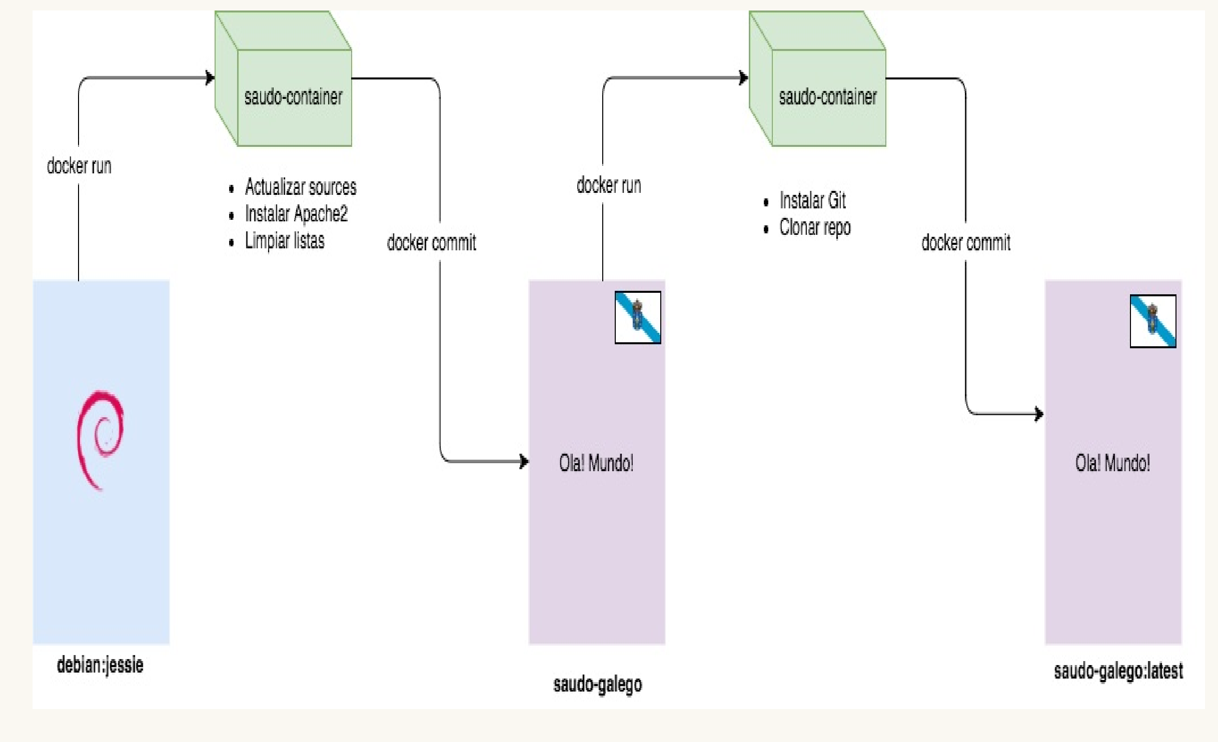
Obra publicada con Licencia Creative Commons Reconocimiento Compartir igual 4.0

# Actividade: Dockerfile para saudo-galego

## Actividad

Empregando a sintáxe aprendida nesta sección, cree e testee un **Dockerfile** que permita construir a mesma imaxe do proxecto saudo-galego tal e como vimos na sección anterior.

Lembre que o proceso quedaría como segue:





Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0]

## Mellorando o Dockerfile da web "gatiño do día"

# 1. Alixeirando a imaxe

Se construímo-la imaxe do gatinho web da actividade anterior, podemos sabe-lo tamaño da mesma.

Basta con facer:

```
docker images web-gatinhos
```

Para obter o tamaño da imaxe. Veremos que a imaxe pesa 491MB. Este peso é excesivo segundo os estándares actuais. Pensemos que estas imaxes:

- Son susceptibles de instalarse en moitos nodos.
- A velocidade de disposición delas depende, en grande medida, do seu tamaño posto que compre descargalas ó host para poder empregalas.
- Hoxe en día a tendencia é a de construir as imaxes máis pequenas posibles.

Afortunadamente, o Docker e a comunidade van poñer á nosa disposición unha serie de mecanismos para facer as nosas imaxes máis pequenas.

## A) A estratexia do multi-stage

Repasemos o proceso de construcción da imaxe do "gatiño do día":

1. Partimos dunha Ubuntu.
2. Instalamos git e as dependencias básicas de python.
3. Clonamos o repo de gatiño do día.
4. Facemos a instalación das librarías de python do proxecto.
5. Definimos arranque do sistema.

Se o pensamos, o git emprégamolo para clonar o repo, pero unha vez feito este paso, qué utilidade ten?

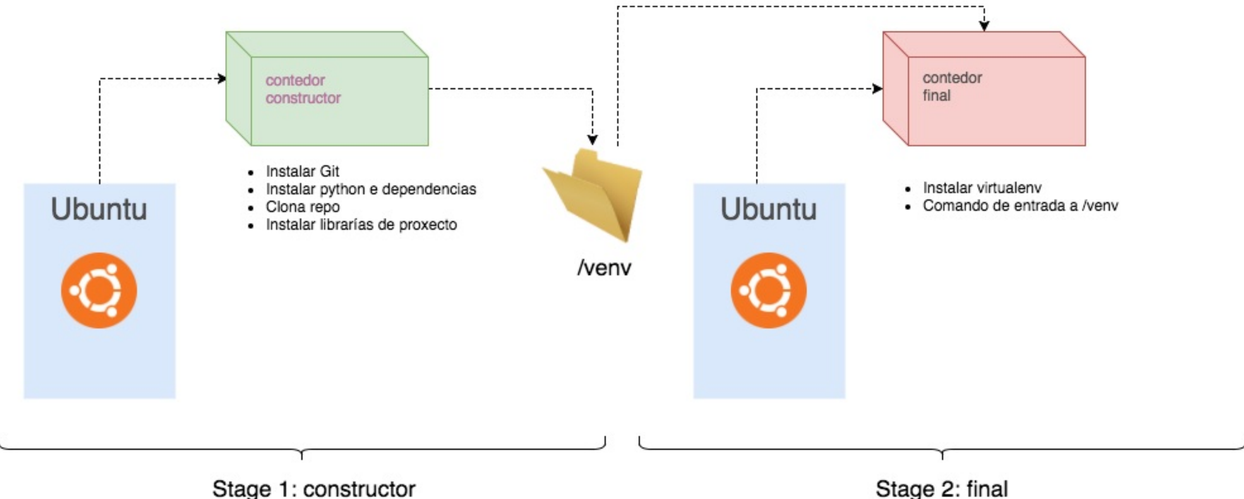
A idea do multi-stage consiste en ir creando contedores que fan un traballo e o deixan en directorios que poden ser empregados por outros contedores. O contedor que fixo un traballo, pode ser desbotado e con él tódalas súas dependencias. Co cal quedamos co resultado do traballo e non con todo o software auxiliar que compre empregar para obtelo.

No noso caso, podemos crear un contedor, instala-lo git, clonar o repo e despois desbotar o git e tódalas súas dependencias. O que nos interesa é ter o noso repo nalgures para o seguinte contedor que faga o seguinte traballo.

Para facer unha imaxe da web "gatiño do día" imos dividi-lo traballo en dúas fases (stages):

1. **Constructor**: imos instalar todo o software necesario para clona-lo repo e instalar as librarías de proxecto. Todo o software resultante e estrictamente necesario vaise poñer nun directorio: o **/venv.** O resto do software do contedor vaise desbotar.
2. **Final**: imos montar o directorio resultante da fase anterior (o /venv)

nun novo contedor que terá o mínimo necesario para move-lo python (o [pipenv](pipenv))




Se o vemos nun Dockerfile:

```
1
2  #
3  # constructor: stage de producción do /venv con todo o necesario para
4  # o noso proxecto
5  #
6  FROM ubuntu:16.04 as builder
7
8  # instalamos o software necesario
9  RUN apt-get update && \
10     apt-get install -y libssl-dev libffi-dev \
11     git python-dev build-essential \
12     python-pip python-virtualenv
13
14 # montamos o pipenv e clonamos o repo dos gatiños
15 RUN bash -c "virtualenv /venv && \
16     source /venv/bin/activate && \
17     git clone https://github.com/prefapp/catweb.git /venv/catweb && \
18     pip install -r /venv/catweb/requirements.txt"
19
20 ##
21 #final: stage final (todo o stage anterior desaparece, salvo o directorio /venv)
22 #
23 FROM ubuntu:16.04
24
25 COPY --from=builder /venv /venv
26
27 RUN apt-get update && \
28     apt-get install -y python-virtualenv && \
29     apt-get clean &&
30     rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*
31
32 ENV PATH="/venv/bin:${PATH}" \
33     VIRTUAL_ENV="/venv"
34
35
36 EXPOSE 5000
37
38 # comando de arranque do contedor
39 CMD ["python", "/venv/catweb/app.py"]
40
```

Como podemos ver:

- Hai dúas fases que comenzan en dous from distintos (liñas 6 e 23).
- O traballo do constructor vólcase no **/venv** (liña 15)
- O contedor final "importa" o /venv do contedor constructor (liña 23)

> Para evitar ter moitos RUN fanse cadeas de comandos con && e se corta a liña para facelo máis lexíbel.

Se executamos este Dockerfile e creamos outra imaxe, veremos que a imaxe nova ten un tamaño de 205MB, producimos un aforro de 286MB!!!

> Mostramos neste exemplo dous stages, pero poderían ser moitos máis segundo as nosas necesidades.

> Onde se tira tamén moito partido desta técnica é nas aplicacións compilables con linguaxes como C, C++ ou Go.

> Para unha discusión relativa ás construcción multi-stage ou builds de tipo mono-stage pódese consultar este artigo.

## B) Todavía máis.... Xogando con alpine

As distros de linux como Alpine, están deseñadas para ser moi lixeiras e seguras, pero mantendo unha orientación hacia o propósito xeral.

Alpine destaca especialmente polo seu reducido tamaño (5MB) e por ser unha das distros estrela no Dockerhub.

Empregando micro-distros, podemos reducir ainda máis o tamaño das nosas imaxes.

```
FROM alpine:latest

RUN apk add --update py-pip

RUN pip install --upgrade pip

COPY requirements.txt /usr/src/app/
RUN pip install --no-cache-dir -r /usr/src/app/requirements.txt

COPY app.py /usr/src/app/
COPY templates/index.html /usr/src/app/templates/

EXPOSE 5000

CMD ["python", "/usr/src/app/app.py"]
```

Con este Dockerfile e clonando o repo en local, podemos ter unha imaxe de "gatiño do día" de 60MB!!!!

### Actividade

Probemos a construí-la imaxe do "gatiño do día" co multi-stage e vexamos cal é o seu tamaño.

# 2. Construindo distintas versións da web

No caso de ter distintas versións da web, sempre que o proceso de

construcción é o mesmo, parece útil poder empregar o mesmo Dockerfile para ambas construccións.

Imaxinemos que temos un repo de git (como o dos gatiños) con dúas ramas:

- *master*: ten a versión 1 da web
- *v2*: ten a versión 2 da web

Interesaría ter un Dockerfile que, segundo lle pasemos un argumento ou outro poida construir versións diferentes.

Para acadar isto temos os ARGS do Dockerfile. Os args son valores que podemos pasarlle ó **docker-build** no momento da construcción da imaxe.

Estos ARGS pódense interpolar nas diversas instruccións do Dockerfile.

Imaxinemos que temos dúas ramas no noso repo da web "gatiños do día" (master, v2), podemos modificar o noso Dockerfile orixinal para que solo baixe unha das ramas no clone.

É dicir, no canto de empregar isto:

```
RUN git clone https://github.com/prefapp/catweb.git
```

Imos utiliza-la seguinte sentencia:

```
RUN git clone https://github.com/prefapp/catweb.git --single-branch -b $branch
```

Deste xeito o git sólo baixara unha das ramas do proxecto (master, v2, v3...)

O problema. obviamente, é como establecer o valor da variable $branch.

Para iso, imos a empregar un ARG.

```
ARG branch=master

RUN git clone https://github.com/prefapp/catweb.git --single-branch -b $branch
```

Agora, o noso Dockerfile agarda un argumento branch cando se invoque. No caso de non recibir tal argumento, o arg inicialízase a un valor por defecto "master".

Se agora facemos:

```
docker build --build-arg branch=master . -t web-gatinhos:v1
```

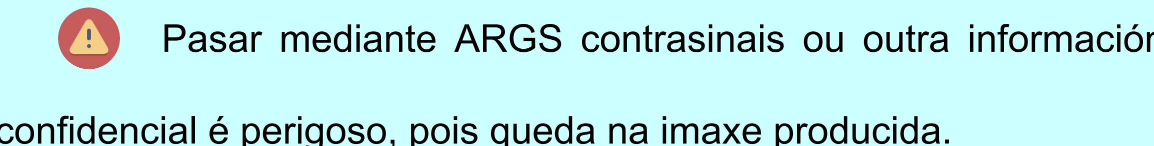
Estamos a dicirlle ó Docker:

- Constrúe unha imaxe (docker build)
- Pásalle un arg "branch" con valor "master" (--build-arg branch=master)
- O dockerfile está nesta ruta (.)
- A imaxe resultante ten que ser web-gatinhos:v1

Se agora facemos o mesmo pero para v2:

```
docker build --build-arg branch=v2 . -t web-gatinhos:v2
```

Teremos unha imaxe web-gatinhos:v2 cos novos cambios.



Pasar mediante ARGS contrasinais ou outra información confidencial é perigoso, pois queda na imaxe producida.

## Actividade

Probemos a lanzar contedores coas imaxes de v1 e v2. Vense diferencias?

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0]