Mondragon Unibertsitatea

Dockers: Contenedorizando una aplicación

Docker: Contenerizando una aplicación

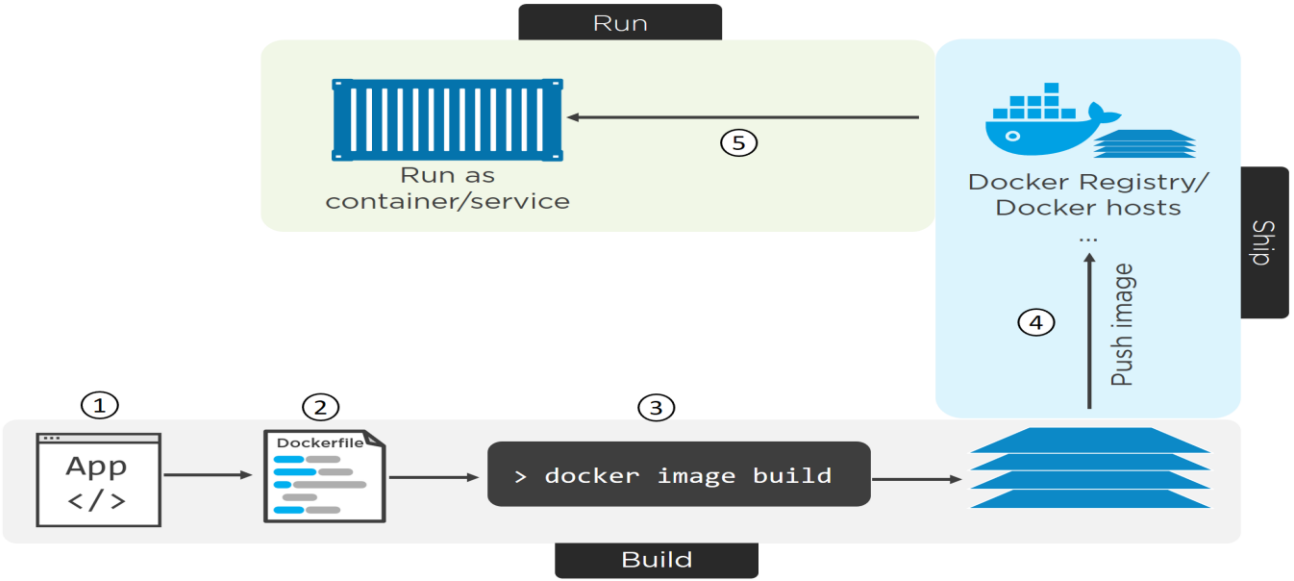
Mondragon Unibertsitatea
Goi Eskola Politeknikoa

- Definición: Al proceso de coger una aplicación y configurarlo para que funcione en un contenedor se conoce como “contenedorizar”. A veces se le denomina como “Dockerizar”.
- Objectivos:
  - Aprender el proceso de contenedorizar una aplicación Linux simple.

Contenedorizando una aplicación: Idea general

Mondragon Unibertsitatea
Goi Eskola Politeknikoa

- El proceso de contenedorizar una aplicación se ve así:
  1. Comience con el código de su aplicación.
  2. Crea un Dockerfile que describa tu aplicación, sus dependencias y cómo ejecutarla.
  3. Introduzca este archivo Docker en el comando de generación de imágenes de docker.
  4. Espere mientras Docker construye su aplicación en una imagen de Docker
- Una vez que la aplicación se ha contenedorizado (convertido en una imagen de Docker), está listo para enviarla y ejecutarla como contenedor.



Contenedorizando una aplicación: Detalles

Mondragon Unibertsitatea
Goi Eskola Politeknikoa

Contenedorizando una aplicación en un solo contenedor

Vamos a ver el proceso de contenedorización de una simple aplicación web Node.js de un solo contenedor.

El ejemplo sigue los siguientes pasos generales:

- Obtener el código de la aplicación
- Inspeccionar el archivo Dockerfile
- Contenedorizar la aplicación
- Ejecutar la aplicación
- Probar la aplicación
- Chequear algunos detalles

Contenedorizando una aplicación en un solo contenedor

- Obtener el código de la aplicación

Clonar el ejemplo desde Gitlab:

```
$ git clone https://gitlab.danz.eus/macc/cloud-computing/platforms-and-infrastructures/pi_docker/pi-simplewebserver-example.git
$ cd SimpleWebServerExample
$ ls -l
```

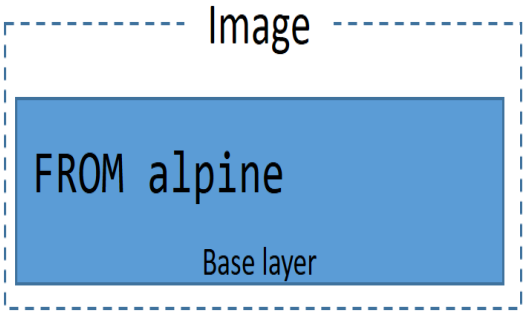
Contenedorizando una aplicación en un solo contenedor

- Analizando el Dockerfile

```
$ cat Dockerfile
FROM alpine
LABEL maintainer=“millarramendi@mondragon.edu"
RUN apk add --update nodejs npm
COPY . /src
WORKDIR /src
RUN npm install
EXPOSE 8080
ENTRYPOINT ["node", "./app.js"]
```

Contenedorizando una aplicación en un solo contenedor

- Todos los Dockerfiles empiezan con la instrucción FROM.



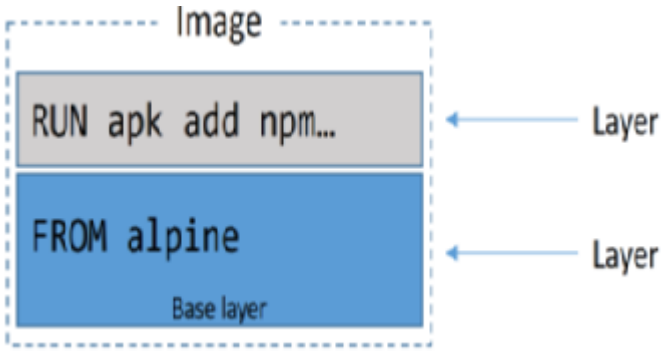
- Después, el Dockerfile crea un LABEL para especificar a millarramendi@Mondragon.edu como mantenedor de la imagen.

Contenedorizando una aplicación: Detalles

Mondragon Unibertsitatea
Goi Eskola Politeknikoa

Contenedorizando una aplicación en un solo contenedor

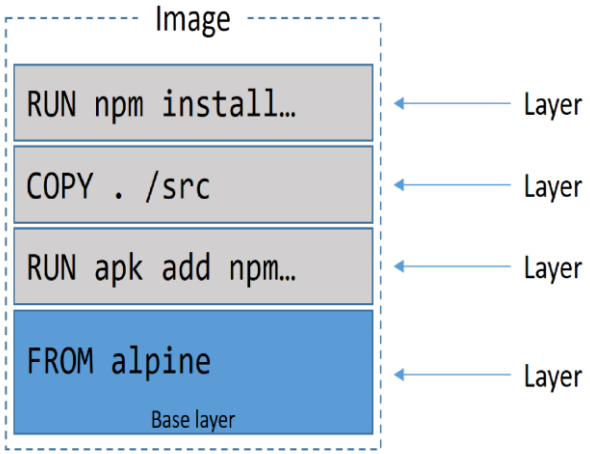
- RUN apk add --update nodejs nodejs-npm.



- COPY . /src

Contenedorizando una aplicación en un solo contenedor

- **WORKDIR** ➔ definer directorio de trabajo
- **RUN npm install**



- **EXPOSE 8080**
- **ENTRYPOINT**

Contenedorizando una aplicación: Detalles

Mondragon Unibertsitatea
Goi Eskola Politeknikoa

Contenedoriza una aplicación / crear una imagen

```
$ docker image build -t web:latest .
```

```
$ docker image ls
```

Ya hemos contenedorizado la aplicación!

```
$ docker image inspect web:latest
```

Contenedorizando una aplicación: Detalles

Mondragon Unibertsitatea
Goi Eskola Politeknikoa

Haciendo push de la imagen

```
$ docker image tag <current-tag> <new-tag>/web:latest
$ docker image tag web:latest username/web:latest

$ docker image ls

$ docker image push millarramendi/web:latest
```

Contenedorizando una aplicación: Detalles

Mondragon Unibertsitatea
Goi Eskola Politeknikoa

Poner en marcha la aplicación

```
$ docker container run -d --name c1 -p 80:8080 web:latest
$ docker container ls
```

Testear la aplicación

- Abra un navegador web y apúntelo al nombre DNS o dirección IP del host en el que se está ejecutando el contenedor.

- Si la prueba no funciona, intente lo siguiente:
  1. *$ docker container ls*. El nombre del contenedor es c1 y debería ver la asignación de puertos como 0.0.0.0:80->8080/tcp.
  2. Comprueba que el cortafuegos/firewall y otras configuraciones de seguridad de red no bloquean el tráfico en el puerto 80 del host Docker.
  3. Entrar en el docker (/bash) y ver la estructura y lo que tiene dentro. Chequear con lo que se ha escrito en el Dockerfile

Contenedorizando una aplicación: Detalles

Mondragon Unibertsitatea
Goi Eskola Politeknikoa

Resumen contenido Dockerfile

- ¿Cómo funciona? Vamos a analizar el Dockerfile:
  - Las líneas de comentarios empiezan con #
  - Todas las líneas no comentadas son **Instrucciones**.
  - Formato:INSTRUCCIÓN argumento
  - Los nombres de las instrucciones no distinguen entre mayúsculas y minúsculas, pero es normal escribirlas en MAYÚSCULAS.
  - El comando "`$ docker image build`" parsea el fichero Dockerfile línea por línea (una línea cada vez) empezando desde la primera arriba del todo.
  - Algunas instrucciones crean nuevas capas y otras solo añaden metadata a la imagen.
    - FROM, RUN y COPY: crean nuevas capas➔ La instrucción está añadiento contenido como ficheros y programas a la imagen, crearán una nueva capa.
    - LABEL, EXPOSE, WORKDIR, ENV y ENTRYPOINT: crean metadata.➔ La instrucción está añadiendo información sobre cómo crear la imagen y ejecutar la aplicación.

Contenedorizando una aplicación: Detalles

Mondragon Unibertsitatea
Goi Eskola Politeknikoa

Detalles

```
$ docker image history web:latest
```

Cada línea corresponde a una instrucción en el Dockerfile (empezando desde abajo hacia arriba). La columna `CREATED BY` lista el comando que se ejecutó en cada caso.

Sólo 4 de las líneas crean una nueva capa (las que tienen valor diferente a 0 en la columna SIZE).

Contenedorizando una aplicación: Detalles

Mondragon Unibertsitatea
Goi Eskola Politeknikoa

Detalles

```
$ docker image inspect web:latest
```

➔ nos confirma que hemos generado 4 capas.

```
<Snip>
},
"RootFS": {
"Type": "layers",
"Layers": [
"sha256:cd7100...1882bd56d263e02b6215",
"sha256:b3f88e...cae0e290980576e24885",
"sha256:3cfa21...cc819ef5e3246ec4fe16",
"sha256:4408b4...d52c731ba0b205392567"
]
}
```

Contenedorizando una aplicación: Detalles

Mondragon Unibertsitatea
Goi Eskola Politeknikoa

Pasar a la producción con construcciones de varias etapas

- Objetivo: imágenes pequeñas.
- Imágenes en producción: tienen que contener justo y lo necesario para poder ejecutar en operación.
- La forma en el que se escribe el Dockerfile impacta mucho en el tamaño de la imagen.
  - Cada instrucción RUN añade una nueva capa➔ Buena práctica: incluir varios comandos como parte de una única instrucción RUN (encolando las diferentes instrucciones con && y \ saltos de línea)
  - Patron del “Builder”: dos diferentes Dockerfiles.
    - Etapa de desarrollo: Dockerfile.dev
    - Producción: Dockerfile.prod

Contenedorizando una aplicación: Detalles

Mondragon Unibertsitatea
Goi Eskola Politeknikoa

Pasar a la producción con construcciones de varias etapas

- Solución: Construcciones en varias etapas.
  - Único Dockerfile: multiples instrucciones FROM. Cada instruccion FROM es una nueva etapa de construcción que puede copiar (COPY) artefactos de etapas anteriores.

Contenedorizando una aplicación: Detalles

Mondragon Unibertsitatea
Goi Eskola Politeknikoa

Dockerfile (PI/Multistage)

```
FROM node:latest AS storefront   (Stage 0)
WORKDIR /usr/src/mgep/app/react-app
COPY react-app .
RUN npm install
RUN npm run build

FROM maven:latest AS appserver (Stage 1)
WORKDIR /usr/src/mgep
COPY pom.xml .
RUN mvn -B -f pom.xml -s /usr/share/maven/ref/settings-docker.xml dependency\
:resolve
COPY . .
RUN mvn -B -s /usr/share/maven/ref/settings-docker.xml package –DskipTests

FROM java:8-jdk-alpine AS production (Stage 2)
RUN adduser -Dh /home/pi_student pi_student
WORKDIR /static
COPY --from=storefront /usr/src/mgep/app/react-app/build/ .
WORKDIR /app
COPY --from=appserver /usr/src/atsea/target/AtSea-0.0.1-SNAPSHOT.jar .
ENTRYPOINT ["java", "-jar", "/app/AtSea-0.0.1-SNAPSHOT.jar"]
CMD ["--spring.profiles.active=postgres"]
```

Contenedorizando una aplicación: Detalles

Mondragon Unibertsitatea
Goi Eskola Politeknikoa

Pasar a la producción con construcciones de varias etapas

```
FROM node:latest AS storefront   (Stage 0)
WORKDIR /usr/src/mgep/app/react-app
COPY react-app .
RUN npm install
RUN npm run build
```

- La etapa 0 hace pull de la imagen node: latest (unos 600 MB).
- Configura el directorio de trabajo
- Copia código de la aplicación
- Usa dos instrucciones RUN para efectuar el npm
- Resultado:
  - Añade tres capas con un tamaño considerable
  - Grande imagen que contiene un montón de contenido relativo a la generación de la imagen, pero no mucho código de aplicación

Contenedorizando una aplicación: Detalles

Mondragon Unibertsitatea
Goi Eskola Politeknikoa

Pasar a la producción con construcciones de varias etapas

```
FROM maven:latest AS appserver (Stage 1)
WORKDIR /usr/src/mgep
COPY pom.xml .
RUN mvn -B -f pom.xml -s /usr/share/maven/ref/settings-docker.xml
dependency\
:resolve
COPY . .
RUN mvn -B -s /usr/share/maven/ref/settings-docker.xml package –
DskipTests
```

- La etapa 1 hace pull de la imagen maven: latest (unos 700 MB).
- Añade cuatro capas de contenido mediante dos instrucciones COPY
- Usa dos instrucciones RUN para ejecutar mvn
- Resultado:
  - Otra grande imagen con un montón de herramientas para la generación, y poco código para la aplicación de código

Contenedorizando una aplicación: Detalles

Mondragon Unibertsitatea
Goi Eskola Politeknikoa

Pasar a la producción con construcciones de varias etapas

```
FROM java:8-jdk-alpine AS production (Stage 2)
RUN adduser -Dh /home/pi_student pi_student
WORKDIR /static
COPY --from=storefront /usr/src/mgep/app/react-app/build/ .
WORKDIR /app
COPY --from=appserver /usr/src/atsea/target/AtSea-0.0.1-SNAPSHOT.jar .
ENTRYPOINT ["java", "-jar", "/app/AtSea-0.0.1-SNAPSHOT.jar"]
CMD ["--spring.profiles.active=postgres"]
```

- La etapa 2 hace pull de la imagen java:8-jdk-alpine (150 MB aprox).
- Añade usuario
- Configura el directorio de trabajo
- Copia código de la aplicación desde la imagen producida en la etapa `storefront`.
- Luego, configura un nuevo directorio de trabajo y copia el código de aplicación generada en la imagen producida en la etapa `appserver`.
- Finalmente, configure la aplicación principal para la imagen que va a poner a correr cuando se inicie como contenedor.
- Para tener en cuenta: COPY `--from`: se usa solo para copiar el código de la aplicación de producción desde las imágenes generadas en las etapas anteriores.

Contenedorizando una aplicación: Detalles

Mondragon Unibertsitatea
Goi Eskola Politeknikoa

Pasar a la producción con construcciones de varias etapas

Ejecución del ejemplo:

```
$ git clone https://gitlab.danz.eus/millarramendi/piwebdockerfile.git
o
$ git clone https://gitlab.danz.eus/macc/cloud-computing/platforms-and-infrastructures/pi_docker/multistage.git
$ cd piwebdockerfile/app
$ ls –l
$ docker image build –t multi:stage  .
$ docker image ls
```

Contenedorizando una aplicación: Detalles

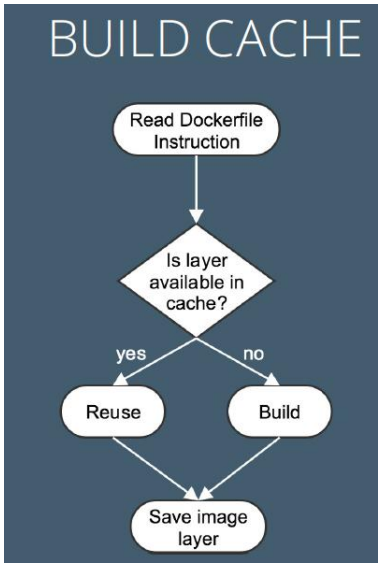
Mondragon Unibertsitatea
Goi Eskola Politeknikoa

Buenas prácticas: Aprovechar la cache de creacion

```
$docker image build
```

Para cada instrucción, el Docker verifica si existe esa imagen en el cache.

- Si está: “cache hit” (usa la capa existente)➔ Acelera el proceso de creación
- Si no está: “cache miss” (generar nueva capa)



Contenedorizando una aplicación: Detalles

Mondragon Unibertsitatea
Goi Eskola Politeknikoa

Buenas prácticas: Aprovechando la cache de creación

- Es importante entender unas cosas.
  - Cuando una instrucción resulta en una cache-miss (no se encuentra ninguna capa para esa instrucción), la cache no se utiliza más durante todo el proceso de generación. Esto tiene un impacto importante en cómo escribimos el Dockerfile. Intenta ir generando capas mediante instrucciones que tengan que hacer cambios al final del fichero. Esto significa que el cache-miss ocurrirá al final de la generación. La idea es sacar el mayor beneficio posible a la cache.
  - Puedes forzar que el proceso de generacion no haga uso de la cache mediante el flag `--nocache=true` al generar la imagen.
  - Es importante entender que las instrucciones COPY y ADD incluyen pasos para asegurar que el contenido que se copia en la imagen no haya cambiado desde la última generación. Por ejemplo, es posible que la instrucción COPY . /src en el Dockerfile no haya cambiado desde la última vez, pero el contenido del directorio que se ha copiado en la imagen haya cambiado.
    - Para proteger esto, Docker realiza un checksum a cada fichero a copiar y compara este checksum del mismo fichero en la capa cacheada. Si no coinciden, invalida la cache y genera una nueva imagen.

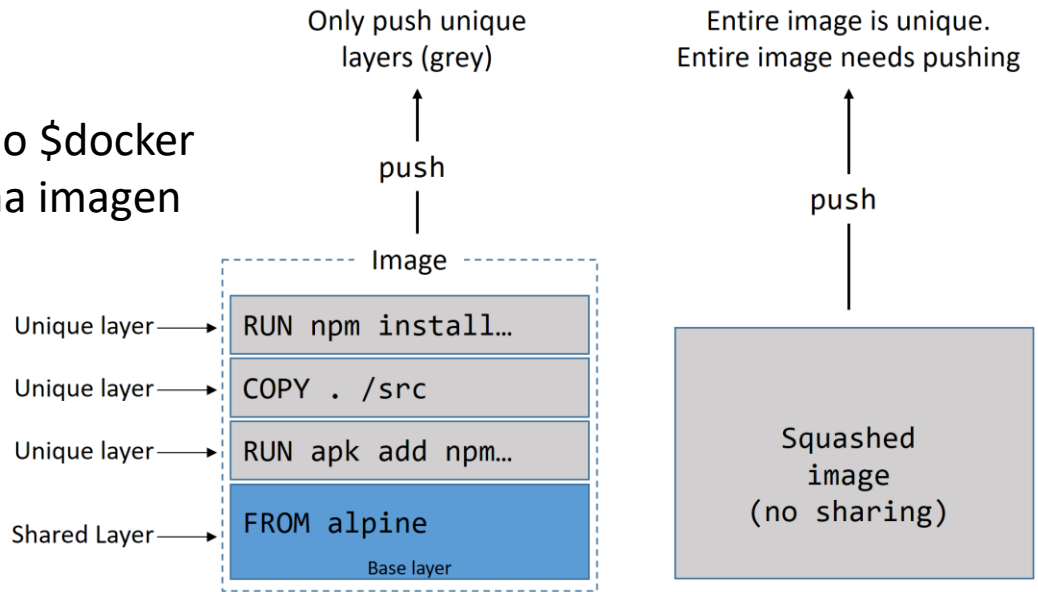
Contenedorizando una aplicación: Detalles

Mondragon Unibertsitatea
Goi Eskola Politeknikoa

Buenas prácticas: Comprimir la imagen fusionando capas

- La imagen no comprimida comparte las capas con otras imágenes del host (ahorrando espacio en el disco) pero las imágenes comprimidas no.
- La imagen comprimida necesita mandar todos los bytes al Docker Hub al ejecutar el comando $ docker image push, mientras que las imágenes no comprimidas pueden mandar solo las capas necesarias.

Añade el **flag --squash** al comando $docker image build si queremos crear una imagen comprimida.



Contenedorizando una aplicación: Detalles

Mondragon Unibertsitatea
Goi Eskola Politeknikoa

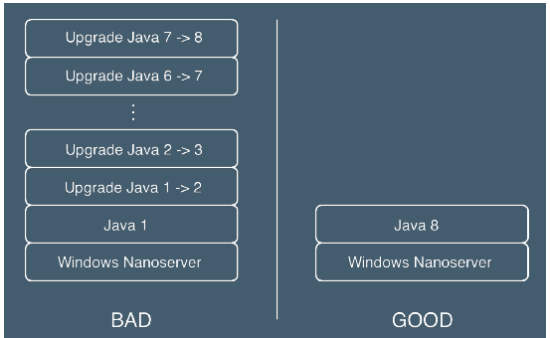
Buenas prácticas: Utiliza la opción no-install-recommends

- Linux: usando el gestor apt package, deberías usar el flag `no-install-recommends` con el comando `apt-get install` ➔ así, apt solo instala las dependencias principales (no los paquetes recomendados o propuestos) reduciendo el tamaño e incrementando la eficiencia.

Buenas prácticas: No instalar los paquetes MSI

- Windows: intenta no utilizar el gestor de paquetes MSI : no hacen un uso eficiente del espacio.

Buenas prácticas : tener la imagen lo más limpia posible



Contenedores Docker: Comandos

Mondragon Unibertsitatea
Goi Eskola Politeknikoa

- **`$ docker image build:`** lee el Dockerfile y contenedoriza una aplicación.
  - -t: etiqueta la imagen
  - -f: te deja especificar el nombre y la localización del Dockerfile
- **`FROM in a Dockerfile:`** especifica la imagen base para la nueva imagen que vas a generar (normalmente la primera instrucción en el Dockerfile).
- **`RUN in a Dockerfile:`** te permite ejecutar comandos dentro de la imagen, Cada instrucción RUN crea una nueva capa.
- **`COPY in a Dockerfile:`** añade ficheros dentro de la imagen como una nueva capa. Es muy usual usar la instrucción COPY para copiar tu código de la aplicación en la imagen.
- **`ADD in a Dockerfile:`** misma funcionalidad que COPY pero además ofrece otras dos funciones. La primera, poder usar una URL en vez de un fichero local o directorio. Segunda, permite extraer un fichero .tar desde la el fichero origen al destino.
- **`EXPOSE in a Dockerfile:`** documenta el puerto de red que usa la aplicación.
- **`ENTRYPOINT y CMD in a Dockerfile:`**
  - ENTRYPOINT: configura la **aplicación** por defecto a poner en marcha cuando la imagen se pone en marcha en el contenedor.
  - CMD: configura el **comando** y/o parámetros por defecto que se ejecutarán al iniciar el contenedor
  - CMD+ENTRYPOINT: el ENTRYPOINT proporciona el comando y CMD los parámetros por defecto.
- **`LABEL:`** permite añadir cualquier tipo de información en formato de key-value.
- **`ENV:`** permite configurar las variables de entorno dentro del contenedor
- **`ARG:`** define los argumentos que pueden ser pasados desde la línea de commandos durante la generación
- **`ONBUILD`**: añade a la imagen una instrucción de trigger que se ejecutará posteriormente, cuando la imagen se utilice como base para otra generación.
- **`HEALTHCHECK`**: le dice a Docker cómo probar un contenedor para comprobar que sigue funcionando.

Contenedores Docker: Práctica

Mondragon Unibertsitatea
Goi Eskola Politeknikoa

1. Multi-Stage Builds

```
FROM alpine:3.5 AS build
RUN apk update && \
apk add --update alpine-sdk
RUN mkdir /app
WORKDIR /app
ADD hello.c /app
RUN mkdir bin
RUN gcc -Wall hello.c -o bin/hello
# Lightweight image returned as final product
FROM alpine:3.5
COPY --from=build /app/bin/hello /app/hello
CMD /app/hello
```

Comprobar la imagen final mediante docker image ls (comprobar su tamaño) y describir el proceso. ¿Cuantos Stages tiene?

Contenedores Docker: Práctica

Mondragon Unibertsitatea
Goi Eskola Politeknikoa

2.Escribir y generar el Dockerfile

- Crea una carpeta “myImage” y crea un fichero con el nombre Dockerfile dentro de esta carpeta.
- Contenido del Dockerfile:

```
FROM centos:7
RUN yum update –y
RUN yum install -y wget
```

- Esto sirve como receta para una imagen basada en centos:7, que tiene todos sus paquetes por defecto actualizados y wget instalado
- Genera la imagen con el nombre myimage
- Compruebe que su nueva imagen existe (docker image ls)
- Ejecutar la imagen en un contenedor y haga wget en ese contenedor.

Contenedores Docker: Práctica

Mondragon Unibertsitatea
Goi Eskola Politeknikoa

3.Build Caché

1. Abra su Dockerfile y agregue otro paso RUN al final para instalar vim.

RUN yum install vim

1. Construya la imagen de nuevo como en el ejemplo anterior; ¿para qué pasos se utiliza la caché?
2. Construya la imagen de nuevo; ¿qué pasos utiliza esta vez la caché?
3. Cambia el orden de los dos comandos RUN para instalar wget y vim en el Dockerfile, y construye una vez más la imagen. ¿Qué pasos se almacenan en la memoria caché esta vez?

4.History Command

- Ejecuta docker image history a la imagen creada en el paso anterior y observa la información devuelta
- Cambia las instrucciones de instalación de wget y vim para que se instalen con un único comando: **RUN yum install -y wget vim**
- Genera la imagen de nuevo y llama al comando history. ¿Qué ha cambiado?

Contenedores Docker: Práctica

Mondragon Unibertsitatea
Goi Eskola Politeknikoa

5.Configurar comandos por defecto

- Añade la siguiente línea al Dockerfile de la práctica anterior:

CMD ["ping", "127.0.0.1", "-c", "5"]

- Genera de nuevo la imagen y pon en marcha el contenedor. ¿Que obtienes?
- Ahora ejecuta el siguiente comando al poner en marcha el contenedor:

$docker container run myimage echo "hello world“

- ¿Qué ha pasado esta vez?
- Cambia la última línea del Dockerfile por: ENTRYPOINT ["ping"]
- Genera y pon en marcha la nueva imagen en un nuevo contenedor. ¿Qué pasa?
- Prueba a poner en marcha mendiante este comando:

$docker container run myimage 127.0.0.1

- ¿Puedes describir las diferencias entre ENTRYPOINT y CMD?
- Cambia la última línea del Dockerfile por:

ENTRYPOINT ["ping", "-c", "3"]

CMD ["127.0.0.1"]

- Genera la imagen y pon en marcha el contenedor (sin pasarle comandos)
- ¿Te ha dado error? ¿Cómo interpreta el valor de CMD?
- Pon en marcha el contenedor:

$docker container run myimage 8.8.8.8

- ¿Qué ha pasado esta vez?

Contenedores Docker: Práctica

Mondragon Unibertsitatea
Goi Eskola Politeknikoa

Prácticas extra (opcional):

- 2.3.1 Create a Python Flask app that displays random cat pix➔

https://github.com/docker/labs/blob/master/beginner/chapters/webapps.md

- Create your first image using Java

https://github.com/docker/labs/blob/master/developer-tools/java/chapters/ch03-build-image.adoc#create-your-first-image

Mondragon Unibertsitatea

Goi Eskola Politeknikoa

Eskerrik asko

Muchas gracias

Thank you