**Trabajo Práctico 6 - Construcción de Imágenes de Docker**

**1- Objetivos de Aprendizaje**

- Adquirir conocimientos para construir y publicar imágenes de Docker.

- Familiarizarse con el vocabulario.

**2- Unidad temática que incluye este trabajo práctico**

Este trabajo práctico corresponde a la unidad Nº: 3

**3- Consignas a desarrollar en el trabajo práctico:**

- En los puntos en los que se pida alguna descripción, realizarlo de la manera más clara posible.

**4- Desarrollo:**

*1- Conceptos de Dockerfiles*

- Leer https://docs.docker.com/engine/reference/builder/ (tiempo estimado 2 horas)

- Describir las instrucciones

- FROM: inicializa una nueva etapa de construcción y establece la imagen base para las instrucciones posteriores. Un Dickerfile válido debe comenzar con ésta instrucción. La imagen puede ser cualquier imagen válida. Es fácil extraer una imagen de un repo público.

- RUN: va a ejecutar cualquier comando en una nueva capa encima de la imagen actual y confirmará los resultados. La imagen confirmada resultante se usará para el siguiente paso en el Dockerfile.

- ADD: copia nuevos archivos, directorios o direcciones URL de archivos remotos desde <scr> y los agrega al sistema de archivos de la imagen en la ruta <dest>

- COPY: copia nuevo archivos o directorios desde el <src> y los agrega al sistema de archviso del contenedor en la ruta <dest>

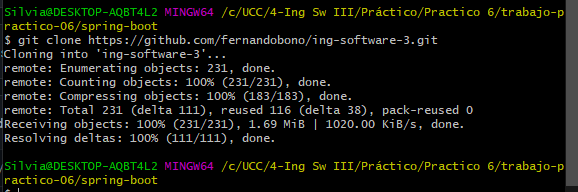
- EXPOSE: informa a Docker que el contenedor escucha en los puertos de red especificados en tiempo de ejecución. Puede especificar si el puerto escucha en TCP o UDP, y el valor predeterminado es TCP, si no se especifica el protocolo. Ésta instrucción en realidad no publica el puerto, funciona como un tip ode documentación entre la persona que construye la imagen y la persona que ejecuta el contenedor sobre qué puertos se pretende publicar.

- CMD: solo puede haber una instrucción CMD en un Docker. Si incluye más de un CMD, solo tendrá efecto el último CMD. El objetivo principal de esta instrucción es proporcionar valores predeterminados para un contenedor en ejecución. Estos valores predeterminados pueden incluir un ejecutable o pueden omitirlo, en cuyo caso, también debe especificar una instrucción ENTRYPOINT.

- ENTRYPOINT: permite configurar un contenedor que correrá como un ejecutable. Solo tendrá efecto la última instrucción en el Dockerfile.

*2- Generar imagen de docker*

- Clonar/Actualizar el repositorio de <https://github.com/fernandobono/ing-software-3>



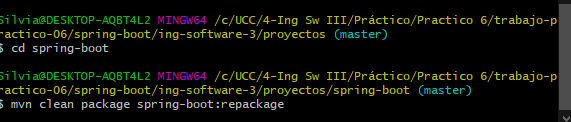
- El código se encuentra en la carpeta “./proyectos/spring-boot”

- Se puede copiar al repositorio personal en una carpeta “trabajo-practico-06/spring-boot”

- Compilar la salida con:

cd proyectos/spring-boot

mvn clean package spring-boot:repackage



Al dar error en un test, agregué unos comandos a la sentencia para compilar. Ese comando hace que se buildee ignorando los errores en los tests.



- Agregar un archivo llamado \*\*Dockerfile\*\* (en el directorio donde se corrió el comando mvn)

FROM java:8-jre-alpine

RUN apk add --no-cache bash

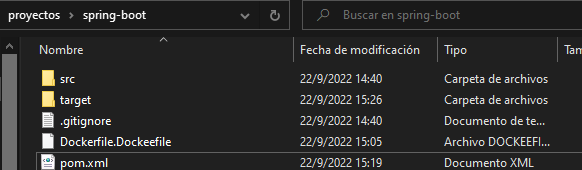
WORKDIR /app

COPY target/\*.jar ./spring-boot-application.jar

ENV JAVA\_OPTS="-Xms32m -Xmx128m"

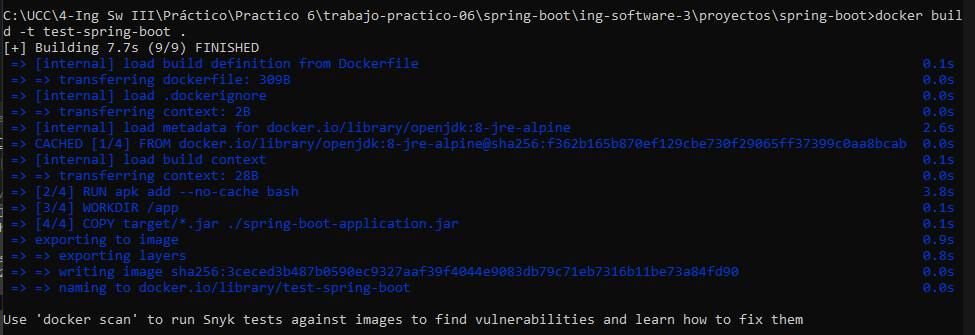
EXPOSE 8080

ENTRYPOINT exec java $JAVA\_OPTS -Djava.security.egd=file:/dev/./urandom -jar spring-boot-application.jar



- Generar la imagen de docker con el comando build

docker build -t test-spring-boot .



- Ejecutar el contenedor

docker run -p 8080:8080 test-spring-boot

- Capturar y mostrar la salida.

- Verificar si retorna un mensaje (correr en otro terminal o browser)

curl -v localhost:8080

*3- Dockerfiles Multi Etapas*

Se recomienda crear compilaciones de varias etapas para todas las aplicaciones (incluso las heredadas). En resumen, las compilaciones de múltiples etapas:

- Son independientes y auto descriptibles

- Resultan en una imagen de Docker muy pequeña

- Puede ser construido fácilmente por todas las partes interesadas del proyecto (incluso los no desarrolladores)

- Son muy fáciles de entender y mantener.

- No requiere un entorno de desarrollo (aparte del código fuente en sí)

- Se puede empaquetar con pipelines muy simples

Las compilaciones de múltiples etapas también son esenciales en organizaciones que emplean múltiples lenguajes de programación. La facilidad de crear una imagen de Docker por cualquier persona sin la necesidad de JDK / Node / Python / etc. no puede ser sobrestimado.

- Modificar el dockerfile para el proyecto Java anterior de la siguiente forma

FROM maven:3.5.2-jdk-8-alpine AS MAVEN\_TOOL\_CHAIN

COPY pom.xml /tmp/

RUN mvn -B dependency:go-offline -f /tmp/pom.xml -s /usr/share/maven/ref/settings-docker.xml

COPY src /tmp/src/

WORKDIR /tmp/

RUN mvn -B -s /usr/share/maven/ref/settings-docker.xml package

FROM java:8-jre-alpine

EXPOSE 8080

RUN mkdir /app

COPY --from=MAVEN\_TOOL\_CHAIN /tmp/target/\*.jar /app/spring-boot-application.jar

ENV JAVA\_OPTS="-Xms32m -Xmx128m"

ENTRYPOINT exec java $JAVA\_OPTS -Djava.security.egd=file:/dev/./urandom -jar /app/spring-boot-application.jar

HEALTHCHECK --interval=1m --timeout=3s CMD wget -q -T 3 -s http://localhost:8080/actuator/health/ || exit 1

- Construir nuevamente la imagen

docker build -t test-spring-boot .

- Analizar y explicar el nuevo Dockerfile, incluyendo las nuevas instrucciones.

A partir de este nuevo Dockerfile, se tiene la creación de una imagen maven, en donde:

* con el **From**, se trae la imagen de maven.
* En **copy**, va a copiar el archivo pom.xml a la carpeta /tmp.
* El primer **run**, ejecuta todas las dependencias,
* Con **copy**, va a copiar el archivo src a la carpeta /tmp/src
* **Workdir** especifica el directorio de trabajo del contenedor que, en este caso, es /tmp
* **Run** ejecuta el comando mvn package para compilar el poryecto

La segunda parte es para la creación de la imagen de java, en donde:

* From trae la imagen de java
* Expose nos dice que se expone en el puerto 8080
* Run crea y corre un subdirectorio /app
* Copy copia la imagen de maven anterior los ejecutables especificados
* Env setea la variable de entorno que se observa
* Entrypoint indica que cada vez que corra el contenedor, se ejecute el archivo jar especificado
* Heathcheck realiza chequeos en el contenedor para saber si la app es saludable

Como conclusión, el archivo Dockerfile anterior, la primera parte se parte de la imagen de maven donde se compila el código y genera los archivos .jar necesarios para poder ejecutarlo. La segunda parte, se parte de la imagen java. Esto reduce el tamaño de la imagen considerablemente, copiando de la etapa anterior los archivos .jar generados que son necesarios para ejecutar la app y genera una imagen similar a la del ej 2, en donde, al final, se permite crear un contenedor con la app spring-boot en funcionamiento.

*4- Python Flask*

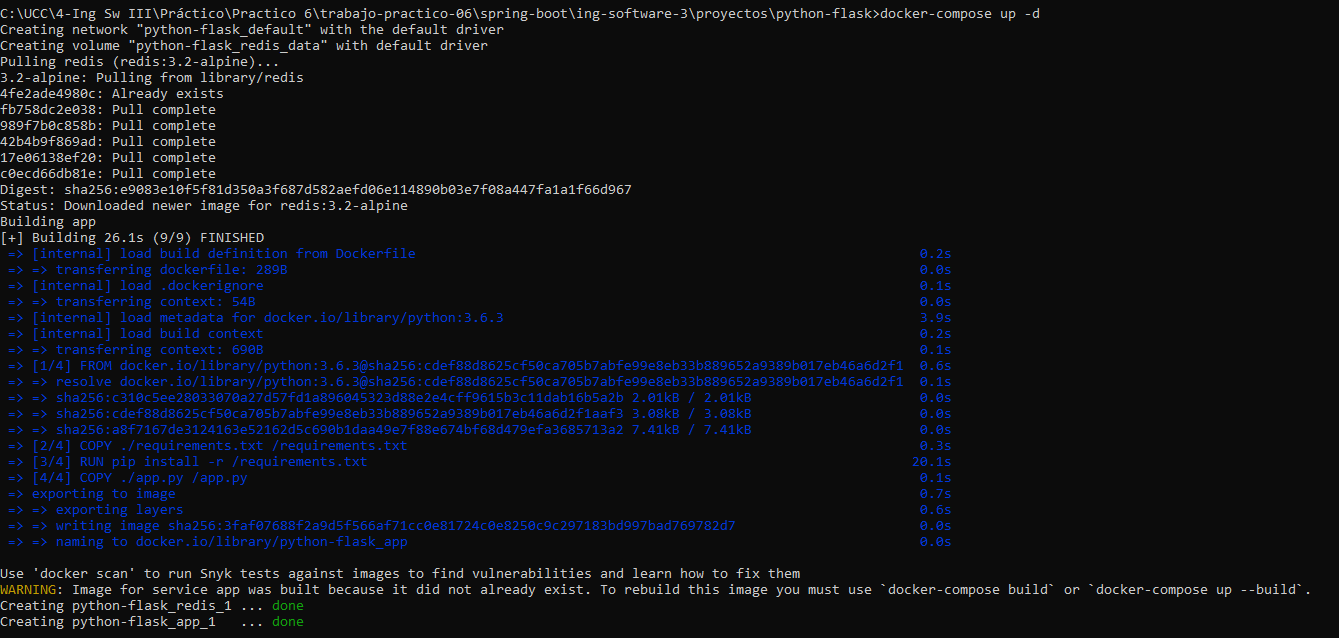
- Utilizar el código que se encuentra en la carpeta “./proyectos/python-flask”

- Se puede copiar al repositorio personal en una carpeta “trabajo-practico-06/python-flask”

- Correr el comando

cd ./proyectos/python-flask

docker-compose up –d

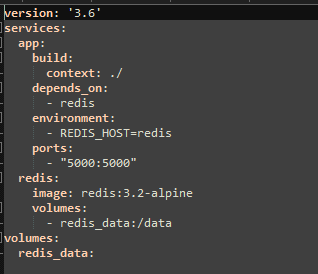


- Explicar que sucedió!

Al ejecutar el comando se crearon dos contendores, app y redis tal como indica el archivo docker-compose.yml. Aquí se descargaron y se iniciaron los servicios de esas imágenes.

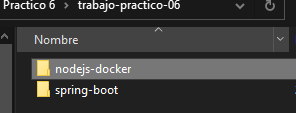
- ¿Para qué está la key “build.context” en el docker-compose.yml?

Usando el archivo docker-compose.yml en el biuld, se definen las opciones de configuración que serán aplicadas al momento de crear el contenedor. El context se refiere a un set de archivos que se encuentran en un PATH o URL. Por lo tanto, el build usará los archivos de ese contexto para saber donde debe buscar las imágenes y el directorio será también el contexto de creación enviado al deamon de docker. El contenedor app se creó a partir de una imagen a su vez creada de un archivo Dockerfile que se encuentra en la raíz del directorio del proyecto. Por ende, docker-compose permite esto gracias a que se especifica la ubicación del archivo Dockerfile mediante la key build.context.

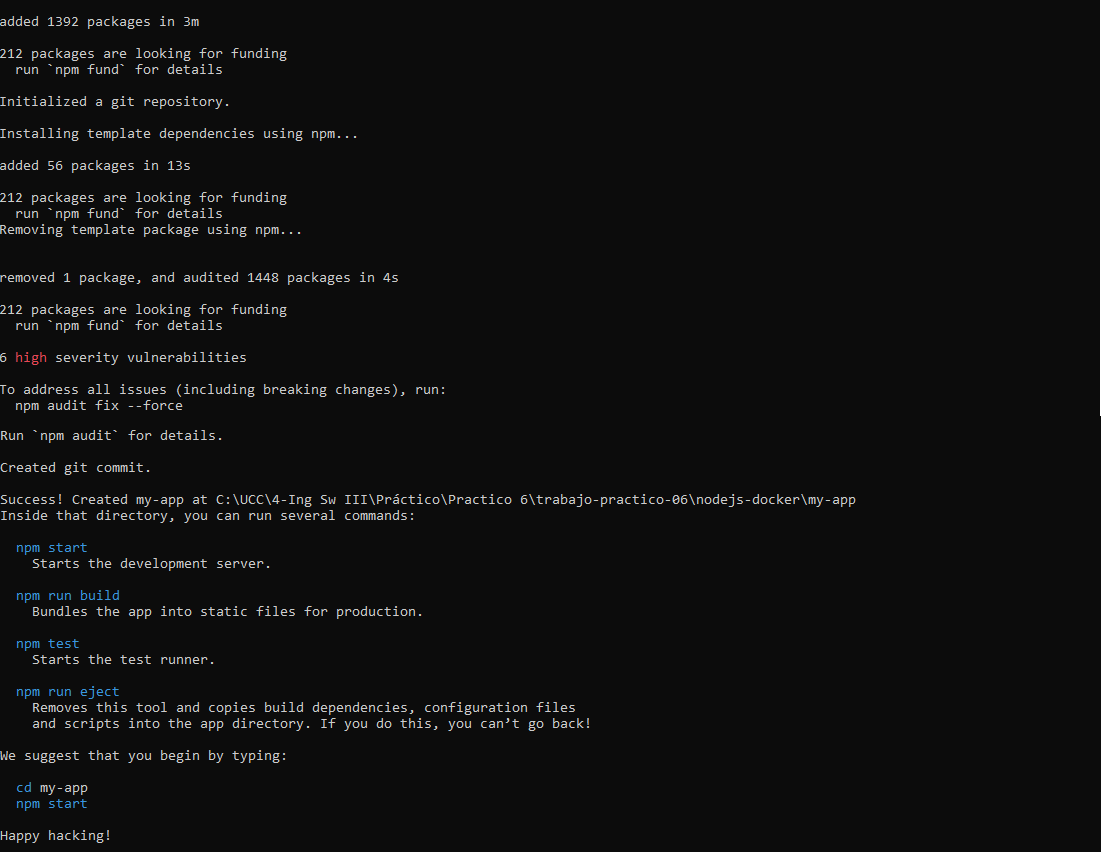


*5- Imagen para aplicación web en Nodejs*

- Crear una la carpeta `trabajo-practico-06/nodejs-docker`



- Generar un proyecto siguiendo los pasos descriptos en el trabajo práctico 5 para Nodejs



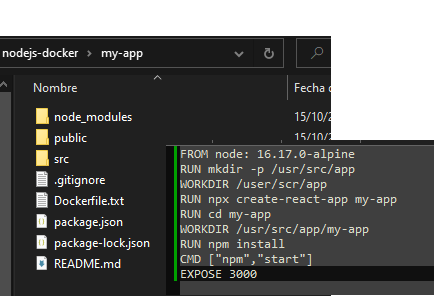
- Escribir un Dockerfile para ejecutar la aplicación web localizada en ese directorio

- Idealmente que sea multistage, con una imagen de build y otra de producción.

- Usar como imagen base \*\*node:13.12.0-alpine\*\*

- Ejecutar \*\*npm install\*\* dentro durante el build.

- Exponer el puerto 3000



- Hacer un build de la imagen, nombrar la imagen \*\*test-node\*\*.

docker build -t test-node .

pendiente

- Ejecutar la imagen \*\*test-node\*\* publicando el puerto 3000.

Docker run –p 3000:3000 test-node

pendiente

- Verificar en http://localhost:3000 que la aplicación está funcionando.

pendiente

- Proveer el Dockerfile y los comandos ejecutados como resultado de este ejercicio.

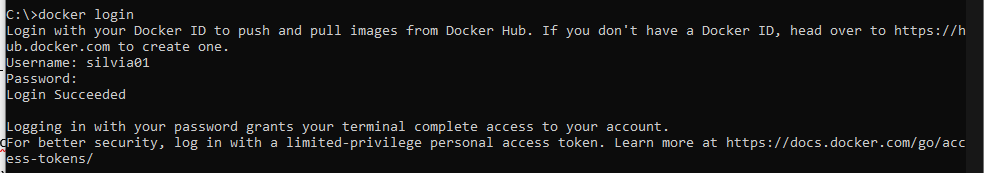
**6- Publicar la imagen en Docker Hub.**

- Crear una cuenta en Docker Hub si no se dispone de una.

- Registrase localmente a la cuenta de Docker Hub:

docker login

Ya tengo cuenta.



- Crear un tag de la imagen generada en el ejercicio 3. Reemplazar <mi\_usuario> por el creado en el punto anterior.

docker tag test-node <mi\_usuario>/test-node:latest

pendiente

- Subir la imagen a Docker Hub con el comando

docker push <mi\_usuario>/test-node:latest

pendiente

- Como resultado de este ejercicio mostrar la salida de consola, o una captura de pantalla de la imagen disponible en Docker Hub.

pendiente