

## Table of Contents

PRIMERA PART: Introducció i que necessitem.....	1
Pas 1. Donar-se d'alta al Docker Hub.....	2
Pas 2: Instalar Docker.....	2
Pas 3: Clonar la imatge amb el nostre exemple desde el git.....	3
Pas 4: Obrir el projecte en IntelliJ i fer el building i executar la aplicació sense docker.....	4
Pas 5: Containeritzar la aplicació.....	5
Pas 5.1 Construir la imatge (docker build).....	5
Pas 5.2: Tornar a fer el build fent canvis en la aplicació.....	6
Pas 5.3 Pujar la imatge al nostre repositori remot Docker Hub (docker push).....	7
Pas 6. Executar la aplicació dins del contenidor (docker run i docker stop).....	8
Pas 6.1. Executar la imatge amb un profile de Spring (test, dev, prod...).....	9
Pas 6.2. Debuggar.....	10
Pas 6.3. Imatges vs. Contenidors.....	10
Imatges.....	10
Contenidors.....	10
Pas 7. Versionar imatges (docker tag).....	12
Pas 8. Actualitzar la versió JDK del projecte i de la imatge.....	13
Pas 8.1 Actualitzar el JDK del projecte.....	13
Pas 8.1 Actualitzar el JDK de la imatge Docker.....	13
Pas 9. Actualitzar la distribució linux de la imatge.....	15
SEGONA PART: Ampliar la nostra aplicació Spring.....	16
Afegir el model de dades i Spring data JPA.....	16
Dubtes (fins a la data).....	16
Referències.....	17

## PRIMERA PART: Introducció i que necessitem

Docker és un conjunt d'eines de gestió de contenidors Linux, que permet als usuaris publicar imatges de contenidors i consumir les publicades per altres. Una imatge de Docker és una “recepta” per executar un procés containeritzat.

En els primers passos d'aquest document el que estarem fent és seguir exemple que podeu veure a: <https://spring.io/guides/gs/spring-boot-docker/> i en aquesta guia en construirem una per a una aplicació senzilla d'arrencada Spring.

Que necessitem:

- SO Linux (corrent sobre una màquina de 64 bits o no anirà el Docker)
- Un IDE qualsevol instal·lat
- JDK 1.8 o superior instal·lat
- Gradle 4+ o Maven 3.2+ instal·lat
- GIT instal·lat (i smartgit + compte al github si voleu pujar el codi a alguna banda)

En aquesta primera part del treball, anirem seguint i aimpliant l'exemple, per entendre com crear la nostra primera aplicació Spring, crear una imatge Docker que la executi i fer-la correr. També tractarem temes bàsics com, com fer una nova versió i crear una nova imatge, o com fer canvis bàsics com canviar la distribució de Linux sobre la que s'assenta la imatge o canviar la versió del JDK que estem utilitzant.

En la segona part del treball, el que farem és fer i documentar modificacions sobre la aplicació Spring.

El resultat de tot plegat es podrà descarregar de:

- Imatge docker: <https://hub.docker.com/repository/docker/mbatet/gs-spring-boot-docker>
- Codi: <https://github.com/mbatet/gs-spring-boot-docker>

## Pas 1. Donar-se d'alta al Docker Hub

Si encara no tenim un compte al docker hub, anar a <https://hub.docker.com/> > **Signup** I crear un nou compte. El camp "Docker ID" és el que hauré de fer servir per fer push i pulls de les nostres imatges, per tant, guardem aquest "Docker ID" per futures referències. En el meu cas és "mbatet", I podeu veure el resultat de la autoformació a:

<https://hub.docker.com/u/mbatet>

## Pas 2: Instalar Docker

Instalem Docker seguint la guia: <https://docs.docker.com/installation/#installation>. En el meu cas, seguim la **guia de instal·lació per Ubuntu**: <https://docs.docker.com/engine/install/ubuntu/>.

Resumint les comandes executades i que fan cadascuna:

```
# Desinstalar si tenim alguna versio anterior
> sudo apt-get remove docker docker-engine docker.io containerd runc

#SET UP THE REPOSITORY
> sudo apt-get update

#intalar les utilitats que encara no tinguem i necessitem:
> sudo apt-get install apt-transport-https ca-certificates curl gnupg-agent
software-properties-common

# Afegir repo
```

```

> curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
> sudo apt-key fingerprint 0EBFCD88
> sudo add-apt-repository "deb [arch=amd64]
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"

#INSTALL DOCKER ENGINE

> sudo apt-get update
> sudo apt-get install docker-ce docker-ce-cli containerd.io

# Provar que ha funcionat

> sudo docker run hello-world

#Si ens apareix el missatge següent és que tot s'ha instalat bé

#Hello from Docker!
#This message shows that your installation appears to be working correctly.

#La comanda "docker" ens mostrarà totes les opcions:

> docker
> sudo docker info

#A partir d'aquí, si volem actualitzar el Docker Engine, executem primer...
> sudo apt-get update

# ...i després tornem a seguir aquestes instruccions d'instalació

```

Necessitarem utilitzar sudo per executar les comandes de Docker.

## Pas 3: Clonar la imatge amb el nostre exemple desde el git

El següent pas serà descarregar-nos del github el exemple proporcionat a la guia i que es troba a <https://github.com/spring-guides/gs-spring-boot-docker.git>.

```

# Anem a la carpeta on volguem crear el directori i clonem el repo
> cd myworkspace
> git clone https://github.com/spring-guides/gs-spring-boot-docker.git

# Anem a la la carpeta que se'ns acaba de crear i comprovem que s'ha clonat
bé...

> cd ./gs-spring-boot-docker/complete

# La carpeta /initial conté la base de l'exemple a desenvolupar. La carpeta
/complete conté el exemple ja fet. Quan acabem l'exemple, podem comparar el que

```

```
obtenim amb el que hauria de ser si ho hem fet bé a la carpeta /complete.
```

```
#Com que utilitzarem gradle, podem eliminar els fitxers maven de l'arrel del projecte (mvnw, mvnw.cmd, pom.xml) per no liar-nos
```

```
> rm mvnw
```

```
# etc...
```

Això ens ha clonat una carpeta amb dos projectes “/initial” i “/complete”. Ens centrarem directament amb la aplicació “/complete”.

## Pas 4: Obrir el projecte en IntelliJ i fer el building i executar la aplicació sense docker

A la guia (<https://spring.io/guides/gs/spring-boot-docker/#initial>) explica com fer el building directament amb maven o gradle. El que nosaltres hem fet és (i el que expliquem aquí) es com **fer el building Gradle desde el IntelliJ**:

Podriem ara obrir només un dels dos projectes (initial o complete)

- *Obrir el IntelliJ > Open or import project > seleccionar la carpeta “gs-spring-boot-docker/complete” > Seleccionar “Gradle project”*
- *Anar a la pestanya Gradle > complete > Tasks > build > Clicar a “build”*

Si tot ha anat bé, ens apareixerà el missatge de “*BUILD SUCCESSFUL*”.

(\*) A la guia també explica com fer la aplicació Spring d'exemple desde zero, que es tractaria simplement de crear la estructura de fitxers `src/main/java/hello`, un fitxer `build.gradle` i una classe java `Application.java`.

La aplicació /complete conté el directori `resources/main` amb un fitxer `application.yml`, on a banda d'uns altres pocs paràmentres hi podem veure el port on s'aixecarà l'aplicació (8080). Aquesta aplicació és limitada a ser una aplicació web que davant del request `http://localhost:8080/` retorna la cadena “Hello Docker World”.

Ara anem a executar la aplicació localment (en el nostre propi host), sense el contenidor Docker:

- *Anem a la pastanya Gradle del IntelliJ > Complete > Tasks > application > bootRun*

Quan aparegui el missatge “*Started Application ...* “ ja podem anar al navegador a [localhost:8080](http://localhost:8080) i veure el missatge “Hello Docker World”. La nostra aplicació spring boot ja està funcionant.

Abans de crear la imatge Docker , i cada cop que fem canvis en l'aplicació, hem de refer el nostre el jar de la aplicació, que és el que s'executarà al Docker:

- Anar a la pestanya Gradle > complete > Tasks > build > Clicar a “clean”
- Anar a la pestanya Gradle > complete > Tasks > build > Clicar a “assemble”

Això ens deixarà un fitxer `«gs-spring-boot-docker-0.1.0.jar»` a la carpeta `complete/build/libs`.

## Pas 5: Containeritzar la aplicació

Per containeritzar la aplicació farem servir les comandes **build** i **push**:

- **docker build**: Build an image from a Dockerfile
- **docker push**: Push an image or a repository to a registry

### Pas 5.1 Construir la imatge (docker build)

Docker té un simple fitxer `Dockerfile` a l'arrel del projecte que utilitza un format específica per a especificar les capes de la nostra imatge. Alguna cosa similar al que veiem al nostre exemple:

```
FROM openjdk:8-jdk-alpine
RUN addgroup -S spring && adduser -S spring -G spring
USER spring:spring
ARG DEPENDENCY=target/dependency
COPY ${DEPENDENCY}/BOOT-INF/lib /app/lib
COPY ${DEPENDENCY}/META-INF /app/META-INF
COPY ${DEPENDENCY}/BOOT-INF/classes /app
ENTRYPOINT ["java", "-cp", "app:app/lib/*", "hello.Application"]
```

Anem al projecte i busquem aquest fitxer a l'arrel. Podem fer el building de la nostra imatge en Gradle tal com:

```
> cd gs-spring-boot-docker/complete

#Aquest fitxer Dockerfile té un paràmentre DEPENDENCY apuntant a un directori
#on se suposa que hi haurà desempaquetat el “fat jar”, pel que abans de
#executar-lo, això és el que haurem de fer. La comanda següent crea el
#directori build/dependency i en aquest directori fa un extract de tots els
#fitxers jar que trobi al lib (en concret extreu el contingut de /libs/gs-
spring-boot-docker-0.1.0.jar I ho copia tot al build/dependency.

> mkdir -p build/dependency && (cd build/dependency; jar -xf ../libs/*.jar)

# Ara ja podem fer el building de la imatge docker

> sudo docker build --build-arg DEPENDENCY=build/dependency -t springio/gs-
spring-boot-docker .
```

```
#Aquesta darrera comanda construeix una imatge i la tagged com a "pringio/g-spring-boot-docker". Per sapiguer que tot ha anat bé, ens ha d'aparèixer el missatge
```

```
#Successfully built ....
```

```
#Successfully tagged springio/g-spring-boot-docker:latest
```

```
##(*) També podríem simplement modificar el fitxer Dockerfile per fer el valor per defecte del paràmetre DEPENDENCY coincidir amb la localització del "fat jar" sense empaquetar, i aleshores la imatge la construiríem simplement amb la línia:
```

```
> docker build -t springio/g-spring-boot-docker .
```

En aquest punt ja tindrem una imatge Docker construïda (taggejada com a "springio/g-spring-boot-docker"). Per veure que ha passat fins al moment al nostre repository docker podem fer servir les comandes docker que ens ofereix el CLI:

```
#Llistar les imatges que tenim al nostre repositori:
```

```
> sudo docker images
```

```
#Aquí veiem llistada la nostra imatge recent creada: springio/g-spring-boot-docker.
```

```
#Si volem saber tota la història d'una imatge determinada podem fer:
```

```
> sudo docker history springio/g-spring-boot-docker
```

```
#Per veure tota la info de la imatge podem fer:
```

```
> sudo docker inspect springio/g-spring-boot-docker
```

```
# ... per llistar tots les comandes que tenim amb el CLI, només cal que escrivim:
```

```
> docker
```

## Pas 5.2: Tornar a fer el build fent canvis en la aplicació

Anem al fitxer build.gradle que es troba a l'arrel del projecte i canviem la línia `group = 'springio'` per `group = 'mbatet'`, d'aquesta manera el group el farem coincidir amb el nostre Docker ID del Docker Hub (no és realment imprescindible, però així ja deixem l'exemple net).

També podem fer algun canvi al missatge que mostrem a la classe Application.java, per assegurar que els nostres canvis es veuen correctament a la nova versió de la imatge.

Abans de crear la nova imatge Docker cada cop que fem canvis a l'aplicació, hem de refer el nostre fitxer `complete/build/libs/g-spring-boot-docker-0.1.0.jar`, que es troba a el que executarà el Docker.

- Anar a la pestanya Gradle > complete > Tasks > build > Clicar a “clean”
- Anar a la pestanya Gradle > complete > Tasks > build > Clicar a “assemble”
- 

```
# Un cop fet el canvi tornem a fer el mkdir i copiar els jars a la carpeta que toca:
> mkdir -p build/dependency && (cd build/dependency; jar -xf ../libs/*.jar)

#Ara ja podem fer el build, taggejant la imatge amb el nostre Docker ID:
> sudo docker build --build-arg DEPENDENCY=build/dependency -t mbatet/gspring-boot-docker .

#I després, llistem les imatges del registre:
> sudo docker images

#... i veurem com n'ha aparegut una de nova, que es la mateixa pero que està tagged com a "mbatet/gspring-boot-docker"
```

Abans de containeritzar la aplicació, cada cop que fem algun canvi a al app, haurem de fer el jar de la aplicació, que és el que s'executarà al Docker

- Anar a la pestanya Gradle > complete > Tasks > build > Clicar a “clean”
- Anar a la pestanya Gradle > complete > Tasks > build > Clicar a “assemble”
- `mkdir -p build/dependency && (cd build/dependency; jar -xf ../libs/*.jar)`

### Pas 5.3 Pujar la imatge al nostre repositori remot Docker Hub (docker push)

Un Docker registry és un repositori on s'emmagatzemen imatges de Docker. Docker Hub és un registre public, i **Docker està configurat per pujar i buscar imatges al Docker Hub per defecte.**

Quan s'utilitzen les comandes **docker pull** o **docker run**, les imatges requesrides es van a buscar al registre que tinguem configurat. Quan utilitzes la comanda docker **push command**, es fa push de la imatge al registre configurat. i com diem, per defecte aquests és Docker Hub.

Així doncs, la comanda docker build ens ha posat la imatge al nostre repositori local. Ara el volem pujar al Docker Hub.

```
#Fem un login al docker hub
> sudo docker login -username=mbatet

#Amb la comanda docker login, ja estem fent un login al registre Docker pe
```

```
#defecte, que es Docker Hub. Ens demanarà el passwod. El nostre username
#(Docker ID) i contrasenya són els entrats al pas 1, quan ens hem donat d'alta.

#Ara pujem aquesta imatge al Docker Hub, utilitzant la comanda  docker push

> sudo docker push mbatet/gs-spring-boot-docker

#Ojo, si amb el primer building que havíem fet de l'exemple, que ens havia fet
#una imatge taggeda com a "springio/gs-spring-boot-docker" intentem fer un

> sudo docker push springio/gs-spring-boot-docker

#ens donarà un error amb les credencials, perquè nosaltres no podem fer login
#al compte de  springio
```

Amb això acabem de pujar la nostra imatge del Docker Hub. Anem a <https://hub.docker.com/> I veiem com la nostra imatge s'ha pujat. La podeu veure a:

<https://hub.docker.com/repository/docker/mbatet/gs-spring-boot-docker>

De la mateixa manera, si ens volguessim fer un pull de qualsevol de les imatges públiques de DockerHub, només caldria que fèssim el següent:

```
#Fem un pull de la imatge springio/spring-ci-base del Docker Hub

> sudo docker pull  springio/spring-ci-base

#Llistem les imatges el nostre repos local per veure si  ja la tenim

> sudo docker images
```

No cal registrar-se al Docker Hub per fer un run d'una imatge docker construïda localment . Si fas un buid amb docker, tindràs igualment una imatge local taggejda i que podem fer correr amb la comanda docker run.

## Pas 6. Executar la aplicació dins del contenidor (docker run i docker stop)

La comanda **docker run** primer crea una capa “modificable” sobre la imatge especificada, i després la fa córrer.

```
#Fem córrer la nostra aplicació desde la imatge Docker
# La opció "- p" (publish) el que fa és publicar la imatge i fa un mapeig del
port del host I del contenidor.

> sudo docker run -p 8080:8080 -t mbatet/gs-spring-boot-docker
```



Ara ja podem tornar a entrar a:

<http://localhost:8080/>

I veure el nostre missatge “Hello Docker World”, que aquest cop s’està executant a dins d’un container Docker, i no directament a dins del nostre host.

Per a fer un “shut down” del procés que està corrent, necessitem executar la comanda **docker stop**, però abans, necessitem saber el id del contenidor, amb la comanda **docker ps**, de la forma següent:

```
# Obtenir els ids dels contenidors que s'estan executant
```

```
> sudo docker ps
```

CONTAINER ID		IMAGE	COMMAND	
CREATED	STATUS		PORTS	NAMES
536b381fe23d	mbatet/gs-spring-boot-docker		"java -cp app:app/li..."	
About a minute ago	Up	About a minute	0.0.0.0:8080->8080/tcp	stoic_pasteur

```
#Aturar el nostre contenidor
```

```
> sudo docker stop 536b381fe23d
```

```
#Alternativament, també podem utilitzar el seu "alias":
```

```
> sudo docker stop stoic_pasteur
```

```
#...si tornem a fer un ps, veurem com primer la imatge està fent un "shutting down" I després ja no apareix...
```

Com funciona el mapeig de ports entre el port de la nostra màquina i el port de la aplicació que està al contenidor? Ho veiem amb un exemple:

```
#Si ara fessim:
```

```
> sudo docker run -p 8081:8080 -t mbatet/gs-spring-boot-docker
```

Hauriem d’anar localment a <http://localhost:8081/>, tot i que el port de la nostra aplicació dins del container seguirà sent el 8080, però aquest cop l’hem mapejat cap al port 8081 del nostre host. Si intentem anar a <http://localhost:8080/> no hi tindrem cap servei aixecat.

### Pas 6.1. Executar la imatge amb un profile de Spring (test, dev, prod...)

Executar la nostra imatge Docker amb perfils Spring és tan fàcil com passar una variable d’entorn a la comanda docker run:

```
#Executar a un perfil definit com a "dev"
```

```
>sudo docker run -e "SPRING_PROFILES_ACTIVE=dev" -p 8080:8080 -t mbatet/gs-spring-boot-docker

#Executar a un perfil definit com a "prod"
>sudo docker run -e "SPRING_PROFILES_ACTIVE=prod" -p 8080:8080 -t mbatet/gs-spring--boot-docker
```

## Pas 6.2. Debuggar

Per debuggar l'aplicació utilitzarem JPDA Transport. Així doncs, tractarem el contenidor com un servidor remot, de la forma següent:

```
#Debuggar la nostra aplicació

> docker run -e "JAVA_TOOL_OPTIONS=-agentlib:jdwp=transport=dt_socket,address=5005,server=y,suspend=n" -p 8080:8080 -p 5005:5005 -t springio/gs-spring-boot-docker
```

*(PENDENT ACABAR COM S'ENLLAÇA AMB EL TEU IDE!!!)*

## Pas 6.3. Imatges vs. Contenidors

### Imatges

En altres entorns de màquines virtuals, les imatges s'anomenen a "snapshots". Són una «fotografia» d'una màquina virtual Docker en un moment concret del temps. Però les imatges de Docker són una mica diferents d'un snapshot de màquina virtual. Per començar, les imatges de Docker no es poden canviar mai. Una imatge és pot eliminar però no la podem modificar. Si necessitem una nova versió del «snapshot», crearem una imatge completament nova.

Aquesta incapacitat de canvi (anomenada "immutabilitat") és una eina potent, ja que ens dóna la seguretat que una imatge no pot canviar mai. Així doncs, si aconseguim que la nostra màquina virtual Docker funcioni en un moment donat del temps i en creem una imatge, ja sabem que aquesta imatge funcionarà i funcionarà de la mateixa manera per sempre més. Això facilita la prova de les addicions al nostre entorn, proporcionant-nos una línia base que sempre funciona (i que sempre funciona igual).

### Contenidors

S'utilitza freqüentment el concepte de "màquina virtual Docker", però la millor manera de d'anomenar-ho hauria de ser "contenidor Docker".



Si una imatge Docker és una fotografia en un moment del temps, un contenidor Docker és com una impressió d'aquesta fotografia, una "instància" de la imatge.

Cada contenidor Docker corre per separat, i podem modificar un contenidor mentre està corrent. Les modificacions d'un contenidor Docker, però, no es desaran a no ser que creem una nova imatge, tal com hem comentat abans.

La majoria de les imatges Docker inclouen sistemes operatius complets que ens permetren fer tot el que necessitem. Això fa que sigui fàcil descarregar-nos o executar un programa, per exemple, des de línia de comandes, al contenidor que s'està executant.

Dins d'aquesta línia de comandes, podeu instal·lar un nou paquet de programari o configurar la seguretat del sistema. Aleshores, podem crear a partir d'aquí una altra imatge i penjar-la al nostre repositori.

A vegades, hem de fer coses en un contenidor que necessitem desar, però que no volem que formin part de la imatge Docker. Un exemple és crear una aplicació web, on probablement tindrem un contenidor Docker que conté la nostra base de dades. Les bases de dades han de poder escriure dades al disc dur per recuperar-les més tard. Docker permet configurar contenidors i "compartir" les carpetes entre el contenidor i el host on corre el contenidor. Un dels casos d'ús més comuns per a això és compartir un directori que contingui el codi de la nostra aplicació o modificar el codi de l'aplicació al nostre host i que ho detecti el servidor d'aplicacions que corre dins del contenidor Docker.

```
#llista les imatges que tenim construïdes i emmagatzemades en el nostre repos
> sudo docker images

#dues opcions per llistar els contenidors que tenim corrents

> sudo docker ps
> sudo docker container ls

#llista tots els contenidors que tenim:
> sudo docker container ls -a

#elimina un contenidor
> sudo docker rm 3b9d94c3b03c

# elimina una imatge
> sudo docker rmi f20414cb136b

#si intentem eliminar una imatge que està corrent un contenidor, ens donarà un
error
```

## Pas 7. Versionar imatges (docker tag)

Quan hem fet un canvi al nostre projecte, a la nostra aplicació ... per fer una nova versió de la imatge, simplement cal que tornem a fer el build afegint al final del nom de la imatge “:tag”. Si no ho fem, el tag per defecte que posarà és “:latest”.

També podem taggejar una imatge ja creada amb la comanda **docker tag**.

```
#Creem i versionem una imatge:
```

```
> cd ~/git/gs-spring-boot-docker/complete
> sudo docker build --build-arg DEPENDENCY=build/dependency -t mbatet/gsspring-boot-docker:version1.0 .
```

```
#Taggejem o versionem una imatge existent:
```

```
#Tag an image referenced by ID
```

```
> sudo docker tag 0e5574283393 fedora/httpd:version1.0
```

```
#Tag an image referenced by Name
```

```
#To tag a local image with name “gs-spring-boot-docker” into the “mbatet” repository with “version1.0” (Note that since the tag name is not specified, the alias is created for an existing local version httpd:latest)
```

```
> docker tag gs-spring-boot-docker mbatet/gs-spring-boot-docker:version1.0
```

```
#Tag an image referenced by Name and Tag
```

```
#To tag a local image with name “gs-spring-boot-docker” and tag “test” into the “mbatet” repository with “version1.0.test”:
```

```
> docker tag gs-spring-boot-docker:test mbatet/gs-spring-boot-docker:version1.0.test
```

## Pas 8. Actualitzar la versió JDK del projecte i de la imatge

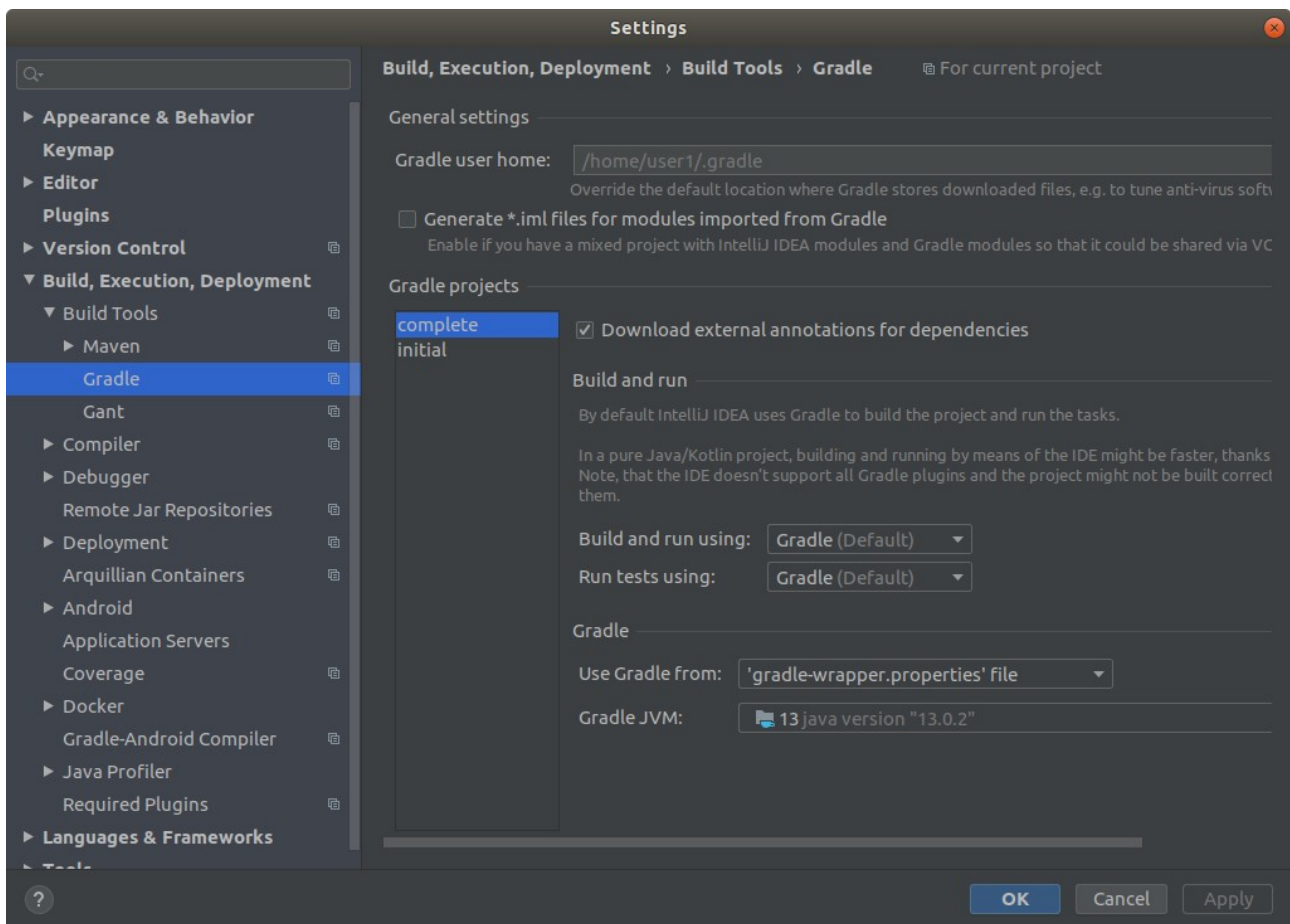
### Pas 8.1 Actualitzar el JDK del projecte

Haurem d'anar al `build.gradle` i referenciar la versió de Java que ens interressi:

```
sourceCompatibility = 13
```

```
targetCompatibility = 13
```

(\*) També hem hagut de canviar al IntelliJ, a `File > Settings > Gradle projectes > Seleccionar projecte "complete"> Gradle JVM > Seleccionar Java 13`



Ara ja podem fer un bootRun i provem que poguem accedir a la aplicació (executada desde el host local) a <http://localhost:8080>. Si veiem el missatge «Hello Docker World» és que ha anat bé.

### Pas 8.1 Actualitzar el JDK de la imatge Docker

El següent pas és canviar la versió del Java definida a la imatge Docker.

Anem al `build.gradle` i comentem la línia

```
#FROM openjdk:8-jdk-alpine
```

I en el seu lloc, afegim:

```
FROM azul/zulu-openjdk-alpine:13 as jdk
```

Després, tornem a fer el **docker build**, pujant la versió de la nostra imatge i **docker run** d'aquesta nova versió.

```
#Fem un building de la imatge en una nova versió. Veurem com es descarrega el  
JDK afegit al Dockerfile
```

```
> cd git/gs-spring-boot-docker/complete/  
> sudo docker build --build-arg DEPENDENCY=build/dependency -t mbatet/gs-  
spring-boot-docker:version1.1 .
```

```
#Fem un run de la nova versió
```

```
> sudo docker run -p 8080:8080 -t mbatet/gs-spring-boot-docker:version1.1
```

```
#Si tot ha funcionat OK, recordem de pujar la nova versió al docker hub:
```

```
>sudo docker push mbatet/gs-spring-boot-docker:version1.1
```

Ara tornem a <http://localhost:8080> i aquí podrem veurem la aplicació executant-se desde el contenidor.

(\*) Si no volem conflictes de ports entre executar la aplicació en local i la executada en el contenidor:

```
> cd git/gs-spring-boot-docker/complete/  
> sudo docker build --build-arg DEPENDENCY=build/dependency -t mbatet/gs-  
spring-boot-docker:version1.1 .  
> sudo docker run -p 8080:8080 -t mbatet/gs-spring-boot-docker:version1.1
```

```
user1@Ubuntu18-4-LTR:~/git/gs-spring-boot-docker/complete$ sudo docker run -p  
8080:8080 -t mbatet/gs-spring-boot-docker:version1.1  
docker: Error response from daemon: driver failed programming external  
connectivity on endpoint competent_boyd  
(5cd4049231ac0513b6cdd8721aba0b40de75e126684f58322aeb943049ac110): Error  
starting userland proxy: listen tcp 0.0.0.0:8080: bind: address already in use.  
ERR0[0000] error waiting for container: context canceled
```

Ens haurem d'acostumar a fer correr la versió del contenidor en un altre port, a partir d'ara:

```
#Fem un run de la versió 1.1, però mapejada al port 8081
```

```
> sudo docker run -p 8081:8080 -t mbatet/gs-spring-boot-docker:version1.1
```

## Pas 9. Actualitzar la distribució linux de la imatge

La nostra imatge està utilitzant una distribució Linux Alpine. Per canviar la nostra imatge de Alpine a Ubuntu. Al docker file cal comentar la línia:

```
FROM azul/zulu-openjdk-alpine:13 as jdk
```

i descomentar la línia:

```
FROM azul/zulu-openjdk:13 as jdk
```

Al intentar fer el build de la imatge, però, ens donarà un error a la comanda «**addgroup**», del Dockerfile:

```
> sudo docker build --build-arg DEPENDENCY=build/dependency -t mbatet/gs-spring-boot-docker:version1.2 .
```

```
Step 2/8 : RUN addgroup -S spring && adduser -S spring -G spring
---> Running in 96e6f5fafeeb
Option s is ambiguous (shell, system)
```

Pel que caldrà canviar la sintaxis de la comanda, queden tel Dockerfile com:

**(PENDENT)**

Per saber quina distribució de Linux està corrent una determinada imatge ho podem fer executant la comanda següent:

```
#La línia següent ens donarà la distribució sobre la que està construïda la nostra imatge. Podem buscar-la a través del container ID o del se nom, que haurem obtingut fent un > docker ps
```

```
> sudo docker exec 6894c9216ad7 cat /etc/os-release
```

```
NAME="Alpine Linux"
ID=alpine
VERSION_ID=3.10.4
PRETTY_NAME="Alpine Linux v3.10"
HOME_URL="https://alpinelinux.org/"
BUG_REPORT_URL="https://bugs.alpinelinux.org/"
```



## SEGONA PART: Amplair la nostra aplicació Spring

En aquesta part ens centrarem més en la aplicació Spring que en el tema Docker, pel que les proves les farem directament en local sense fer el building de la imatge.

### Afegir el model de dades i Spring data JPA

(pendent)

```
>
```

### Dubtes (fins a la data)

- Com s'enllaça una imatge corrent en debug amb el nostre IDE?
- On s'està executant la imatge que està corrent?
- On van a parar els logs de la aplicació? Com consultar-los?
- Si hem de comparar, i per estandaritzar, necessitem utilitzar tota la mateixa distribució de linux + java
- Es pot exposar més d'un port?
- Com configurar-li el proxy a la imatge, quan l'estiguem executant allí?
- Es pot accedir al filesystem? Es poden llegir / copiar dades de / desde a un filesistem extern? Tenir configuracions externes?
- Obre una interfície de xarxa per cada instància de imatge que estigui corrent
- Com fer per no executar totes les comandes amb el sudo?
- Com fer per poder consultar els fitxers de var/lib/docker/containers/xxxxx.../ sense fer un sudo su – root?
- Com "compartir" les carpetes entre el contenidor i el host on corre el contenidor
- Com em puc estalviar d'afegir la DEPENDENCY i de fer el copiat de fitxers?
- Com s'entra al container??

## Referències

- <https://spring.io/guides/gs/spring-boot-docker/>
- <https://www.baeldung.com/dockerizing-spring-boot-application>
- <https://docs.docker.com/engine/reference/commandline/cli/>
- <https://spring.io/guides/topicals/spring-boot-docker>
- <https://hub.docker.com/r/azul/zulu-openjdk>
- <https://docs.docker.com/engine/reference/builder/>
- [https://docs.docker.com/develop/develop-images/dockerfile\\_best-practices/](https://docs.docker.com/develop/develop-images/dockerfile_best-practices/)
- <https://stackify.com/docker-image-vs-container-everything-you-need-to-know/>