# Table of Contents



# 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 esl 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 instalat
- JDK 1.8 o superior instalat
- Gradle 4+ o Maven 3.2+ instalat
- GIT instalat (i smartgit + compte al github si voleu puajr 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 haurem 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 instalació 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 seguent 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 comporvem 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 aplicacio "/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.grade 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 limita 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` 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 aplicció 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  Constuir 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
#dicretori 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/gs-
spring-boot-docker”. Per sapiguer que tot ha anat bé, ens ha d’aparèixer el
missatge

#Successfully built …..
#Successfully tagged springio/gs-spring-boot-docker:latest

#(*) També podriem simplment modificar el fitxer Dockerfileper fer el valor per
#defecte delparàmentr DEPENDENCY coincidir amb la localització del “fat jar”
#sense empaquetar, I aleshores la imatge la construiriem simplement amb la
#linia:

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

En aquest punt ja tindrem una imatge Docker construida (taggejada com a “springio/gs-spring-boot-docker”). Per verue 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/gs-spring-boot-
docker.

#Si volem saber tota la historia d’una imatge determinada podem fer:

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

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

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

# … per llistar tots les comandes que tenim amb el CLI, només cal que escribim:

> 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 linia `group = 'springio'` per `group = 'mbatet'`, d'aquesta manera el group el farem coincidir amb el nostre Docker ID del Dokcer Hub (no és realment imprescindible, però així ja deixem l'exemple net).

També podem fer algún 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/gs-spring-boot-docker-0.1.0.jar`, que és 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/gs-spring-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/gs-spring-boot-docker"
```

Abans de conteinaritzar la aplciació, cada cop que fem algun canvi a al app, haurem de fer el jar de la aplciació, 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 haviem 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 donarpa un error amb les credencials, perqué 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 publiques de DockerHub, nomś 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 construiida localment . Si fas un buid amb  docker, tindras 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 còrrer.

```
#Fem còrrer 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 p**s, 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 mappeig 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 apliació 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», creearem 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 corent. 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 imategs que tenim construides 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/gs-spring-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
```

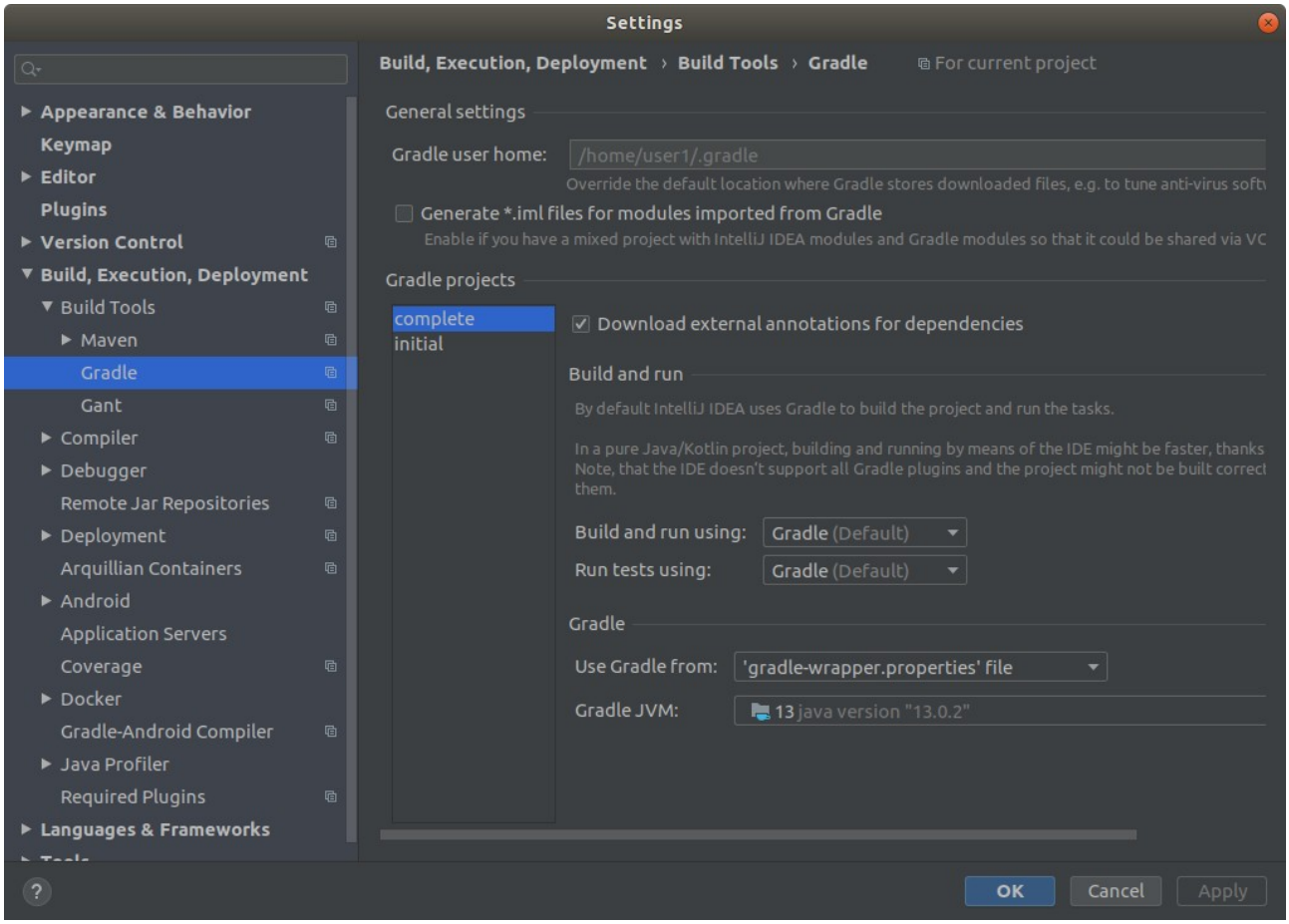
# Pas 8. Actualitzar la versió JDK del projecte i de la imatge

## Pas 8.1 Actualitzar el JDK del projecte

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

```
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 linia

```
#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 nostre 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 (5cd4049231ac0513b6cdd8721aba0b40de75e126684f58322aebe943049ac110): Error starting userland proxy: listen tcp 0.0.0.0:8080: bind: address already in use.
ERRO[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 mapejjada 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 linia:

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

i descomentar la linia:

`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, quedan tel Dockerfile com:

(PENDENT)

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

```
#La linia 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/"
```

# Dubtes (fins a la data)

- Com s’enllaça una imateg corrent en debug amb el nostre IDE?
- On s’està executan la imatge que està corrent?
- On van a parar els logs de la aplicació? Com consultar-los?
- Si hem de comparar, i per estandaritzar, necessitarem utilitzar tots 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 alli?
- Es pot accedir al filesystem? Es poden llegir / copiar dades de / desde a un filesistem extern? Tenir configuracions externes?
- Obre una interifice de xarxa per cada instància de imatge que estas 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 (primera part)


- 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://stackify.com/docker-image-vs-container-everything-you-need-to-know/

# SEGONA PART: Ampliar 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, que ja farem al final de tot.

# Afegir el model de dades i Spring data JPA

La aplicació més bàsica contindrà Spring data JPA si pretén accedir a base de dades.  Amb spring data JPA tenim el query language de JPA, però a més ens proporciona les dynamic querys basades en el nom del métode (the JPA module supports defining a query manually as String or have it being derived from the method name.... coneixeu GORM, oi? Doncs això...).

Spring data també ens oferrià altres funcionailitats com al paginació, el query by example, etc...

Comencem amb el nostre exemple:

## Dependències al build.gradle

Afegim les dependències necessàries al build.gradle:

```
compile ("org.springframework.boot:spring-boot-starter-validation")
compile ("org.springframework.data:spring-data-jpa:2.1.6.RELEASE")
compile ("org.springframework.boot:spring-boot-starter-data-jpa:2.1.3.RELEASE")
compile ("com.h2database:h2:1.4.197")
```

## Model (@Entity) i bean validation

Afegim les classes del model, anotades com a @Entity. A diferència de Grails, al nostre package «model» (o com el volguem anomenar), hi podem barrejar POJOS que es corresponguin amb objectes de domini de la base de dades (els que aniran anotats com a @Entity), i POJOS que representin objectes de domini no mappejats a la base de dades.

L'únic requeriment que tenim és que una classe @Entity ha de tenir per força un camp anotat com a @Id que indiqui quina és la clau primària a la taula de la base de dades. Per exemple:

```
@Entity
public class Book {
```

```
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Long id;

    @NotNull
    @Column(name = "name", nullable = false)
    private String name;

    @NotNull
    @Column(name = "isbn", nullable = false)
    private String isbn;

    @ManyToOne
    private Genre genre;

    public Book() {}

    public Book(@NotNull String name, @NotNull String isbn, Genre genre) {
        this.isbn = isbn;
        this.name = name;
        this.genre = genre;
    }


    @Override
    public String toString() {
        return "Book{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", isbn='" + isbn + '\'' +
                ", genre=" + genre +
                '}';
    }
}
```

Els camps que siguin camps de base de dades aniran anotats com a @Column. També podem tenir anotacions que ens fan validacions de beans. D'questa manera, l'objecte ja serà validat en la seva creació, abans de ser persistit.

Les validacions bàsiques de beans que tenim dins del paquest org.hibernate.validators són:

- **@NotNull** – validates that the annotated property value is not null
- **@AssertTrue** – validates that the annotated property value is true
- **@Size** – validates that the annotated property value has a size between the attributes min and max; can be applied to String, Collection, Map, and array properties
- **@Min** – Validates that the annotated property has a value no smaller than the value attribute
- **@Max** – validates that the annotated property has a value no larger than the value attribute
- **@Email** – validates that the annotated property is a valid email address
- **@NotEmpty** – validates that the property is not null or empty; can be applied to String, Collection, Map or Array values
- **@NotBlank** – can be applied only to text values and validated that the property is not null or whitespace

- **@Positive** and **@PositiveOrZero** – apply to numeric values and validate that they are strictly positive, or positive including 0
- **@Negative** and **@NegativeOrZero** – apply to numeric values and validate that they are strictly negative, or negative including 0
- **@Past** and **@PastOrPresent** – validate that a date value is in the past or the past including the present; can be applied to date types including those added in Java 8
- **@Future** and **@FutureOrPresent** – validates that a date value is in the future, or in the future including the present

Algunes anotacions accepten atributs addicionals. Un atribut habitual és el missatge per defecte que serà renderitzat quan la validació falli. Per exemple:

- @Size(min = 10, max = 200, message = "About Me must be between 10 and 200 characters")
- @Min(value = 18, message = "Age should not be less than 18")
- @Max(value = 150, message = "Age should not be greater than 150")
- @Email(message = "Email should be valid")

## ApplicationConfig (definició del dataSource)

Ara creem la classe ApplicationConfig on cal que hi hagi la definició del nostre dataSource. La creació i inicialització d'aquests beans, que amb versions antigues de spring es solia fer sempre declarativament (xml-based injection), en versions actuals és troba habitualment de forma programàtica. Tot i això, es pot continuar fent servir la definició declarativa via XML, creant el bean del dataSource tal com  podeu veure explicat a: [https://www.baeldung.com/spring-xml-injection](https://www.baeldung.com/spring-xml-injection)

```
@Configuration
@EnableJpaRepositories
@EnableTransactionManagement
class ApplicationConfig {


    @Bean
    public DataSource dataSource() {

        //si cal, aquí recuperariem les propietats de la aplciació desde un obejcte
Environtment
        EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
        return builder.setType(EmbeddedDatabaseType.H2).build();


    }

    @Bean
    public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
```

```
        HibernateJpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
        vendorAdapter.setGenerateDdl(true);


        LocalContainerEntityManagerFactoryBean factory = new
LocalContainerEntityManagerFactoryBean();
        factory.setJpaVendorAdapter(vendorAdapter);
        factory.setPackagesToScan("hello.model");
        factory.setDataSource(dataSource());
        return factory;
    }

    @Bean
    public PlatformTransactionManager transactionManager(EntityManagerFactory
entityManagerFactory) {

        JpaTransactionManager txManager = new JpaTransactionManager();
        txManager.setEntityManagerFactory(entityManagerFactory);
        return txManager;
    }
}
```

## Repository

El patró de desenvolupament «repository» ja el coneixem, és el que ens permet  abstraure l'emmagatzematge de dades proporcionant un conjunt de mètodes per llegir, persistir, actualitzar i eliminar  entitats del repositori de dades subjacent.

Amb Spring Data JPA definim una interfície de repositori per a cada entitat de domini de l'aplicació, on s'hi defineixen les operacions CRUD que realitzarem sobre la entitat de domini, ordenacions i paginacions de dades.

Els repositoris de dades de spring ens permeten  eliminar molt de codi «boilerplate» mitjançant funcionalitats genèriques i mètodes de consulta genèrics CRUD.

El repositori serà una interficie definida mitjançant l'anotació @Repository, que indica que la classe proporciona el mecanisme per a l'operació d'emmagatzematge, recuperació, cerca, actualització i eliminació dels objectes.

Val la pena llegir la documentació de spring per entendre la potència que ens proporcionen els repositoris de Spring Data JPA i el que podem fer amb ells:
https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories

Pel moment, seguim amb l'exemple: Creem la interficie **CustomerRrepository** que exten de **CrudRepository** amb el qual ja en proporcionarà, per defecte, de les operacions:

- **count**(): Returns the number of entities available.
- **delete**(T entity): Deletes a given entity.
- **deleteAll**(): Deletes all entities managed by the repository.
- **deleteAll**(Iterable<? extends T> entities): Deletes the given entities.

- **deleteById**(ID id): Deletes the entity with the given id.
- **existsById**(ID id): Returns whether an entity with the given id exists.
- **findAll**(): Returns all instances of the type.
- **findAllById**(Iterable<ID> ids): Returns all instances of the type T with the given IDs.
- **findById**(ID id): Retrieves an entity by its id.
- **save**(S entity): Saves a given entity.
- **saveAll**(Iterable<S> entities): Saves all given entities.

**Dynamic query derivated from repository method name (query a partir del nom del métode)**

En aquesta classe només cal que definim els métodes que no tenim, però no ens cal implementar-los, ja que habitualment, per a consultes senzilles, LA CREACIÓ DE LA QUERY DEL MÉTODE ES FA A PARTIR DEL NOM DEL MÉTODE. Per exemple:

```
public interface CustomerRepository extends CrudRepository<Customer, Long> {
    List<Customer> findByLastName(String lastName);
}
```

D’aquesta forma podem declarar métodes, sense necessitat d’implementar-los, al nostre repositori, tal com:

- findByLastnameAndFirstname
- findByLastnameOrFirstname
- findByFirstname(Is)
- findByStartDateBetween
- findByAgeLessThan
- findByAgeLessThanEqual
- findByAgeGreaterThan
- findByAgeGreaterThanEqual
- findByStartDateAfter
- findByStartDateBefore
- findByAge(Is)Null
- findByAge(Is)NotNull

- findByFirstnameLike
- findByFirstnameNotLike
- findByFirstnameStartingWith
- findByFirstnameEndingWith
- findByFirstnameContaining
- findByAgeOrderByLastnameDesc
- findByLastnameNot
- findByAgeIn(Collection<Age> ages)
- findByAgeNotIn(Collection<Age> ages)
- findByActiveTrue
- findByActiveFalse()
- findByFirstnameIgnoreCase

**Consultes que no poden ser definides segons el nom del mètode: @Query**

Per a la resta de querys que tinguin massa complexitat per a ser definides mitjançant el nom del métode, podem definir querys a dins del repositori mitjançant l'anotació @Query. D'aquesta manera, el nostre repositori podria quedar com a:

```
public interface CustomerRepository extends CrudRepository<Customer, Long> {

    List<Customer> findByLastName(String lastName);

    @Query("SELECT c.firstName FROM Customer c WHERE c.firstName LIKE
CONCAT('%',:username,'%')")
    List<String> findUsersWithPartOfName(@Param("username") String username);

}
```

**NamedQuerys**

També es poden definir Named querys a dins de la entitat:

```
@Entity
@NamedQuery(name = "User.findByEmailAddress",
  query = "select u from User u where u.emailAddress = ?1")
public class User {
...
```

```
}
```

## Guardar / recuperar dades desde controladors o services

Ara ja només ens queda accedir a aquests métodes des de el controlador o el service. Seguint amb el nostre exemple:

```
@RestController
@RequestMapping(value = "/customer")
public class CustomerController {

    @Autowired
    private CustomerRepository repository;

    private static final Logger log =
LoggerFactory.getLogger(CustomerController.class);


    //http://localhost:8080/customer/insertData
    @RequestMapping("/insertData")
    public String insertData()
    {
        //Moure tot això a al controller & service de Customer
        repository.save(new Customer("Jack", "Bauer"));
        repository.save(new Customer("Chloe", "O'Brian"));
        repository.save(new Customer("Kim", "Bauer"));
        repository.save(new Customer("David", "Palmer"));
        repository.save(new Customer("Michelle", "Dessler"));


        //Genre(@NotNull String name)
        //Book(@NotNull String name, @NotNull String isbn, Genre genre)

        return "Congratulations! You have inserted a few rows!";

    }



    //http://localhost:8080/customer/retrieveData
    @RequestMapping("/retrieveData")
    public String retrieveData()
    {

        List<Customer> customers = new ArrayList<Customer>();
        Iterable<Customer> iterator = repository.findAll();  //no ha calgut
definir el metode findAll
        iterator.forEach(customers::add);
        Optional<Customer> customer = repository.findById(Long.valueOf(1));
//no ha calgut definir el metode findById
        List<Customer> someCustomers = repository.findByLastName("Dessler"); //
no ha calgut definir el metode findByLastName, només declarar-lo
        List<String> customersWithPartOfName =
repository.findUsersWithPartOfName("Ki");  //aquest metode si te una query jpa
```

```
        String result =" repository.findAll(): " + customers.size()  + "
customer rows";
        result += "<br/>Customers: " + customers;
        result += "<br/>repository.findById(1): " +  (customer.isPresent()?
customer.get().toString():"");
        result += "<br/>repository.findByLastName('Dessler'): " +
someCustomers.size();
        result += "<br/>repository.findUsersWithPartOfName('Ki')): "
+customersWithPartOfName;


        return result;

    }
}
```

## Service Layer

Crar una capa de serveis és tan senzill com anotar la classe de servei com a @Component i definir el seu scope, amb l'anotació @Scope.

Seguint amb l'exemple, moure'm elcodi que teniem al controllador a la nostra classe de Service, tal com:

```
@Scope(value = "singleton")
@Component(value = "customerService")
public class CustomerService {


    @Autowired
    private CustomerRepository repository;

    public String insertData()
    {
        repository.save(new Customer("Jack", "Bauer"));
        repository.save(new Customer("Chloe", "O'Brian"));
            ...

    }


    public String retrieveData()
    {


        Optional<Customer> customer = repository.findById(Long.valueOf(1));
      ...
    }

}
```

**Scopes**

Escollirem un scope o un altre depenent del que necessitem. Els valors possibles dels scopes són

- singleton
- prototype
- request
- session
- application
- websocket

## Logging

Spring porta per defecte logback, pel que per a fer funcioanr els logs només cal afegir a la carpeta /resources algun d'auqests fitxers:

- logback-spring.xml
- logback.xml
- logback-spring.groovy
- logback.groovy

(https://www.baeldung.com/spring-boot-logging)

# Referències


- https://blog.marcnuri.com/field-injection-is-not-recommended/
- https://www.baeldung.com/spring-xml-injection
- https://www.baeldung.com/javax-validation
- https://www.baeldung.com/the-persistence-layer-with-spring-data-jpa
- https://www.baeldung.com/spring-data-jpa-query
- https://www.baeldung.com/jsf-spring-boot-controller-service-dao
- https://www.baeldung.com/spring-bean-scopes
- https://www.baeldung.com/jackson
- https://www.baeldung.com/spring-boot-logging
- https://github.com/eugenp/tutorials/blob/master/persistence-modules/spring-persistence-simple
- https://github.com/spring-guides
- https://spring.io/guides/gs/accessing-data-jpa/
- https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#reference
- https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories
- https://docs.spring.io/spring-data/data-jpa/docs/1.5.x/reference/html/index.html
- https://docs.spring.io/spring-data/data-jpa/docs/1.5.x/reference/html/jpa.repositories.html

```
>
```