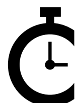
Casos de uso PaaS y de automatización completa

PID_00241998

Joan Caparrós Ramírez
Lorenzo Cubero Luque
Jordi Guijarro Olivares

Tiempo mínimo de dedicación recomendado: 5 horas



Universitat Oberta de Catalunya

Ninguna parte de esta publicación, incluido el diseño general y la cubierta, puede ser copiada, reproducida, almacenada o transmitida de ninguna forma, ni por ningún medio, sea éste eléctrico, químico, mecánico, óptico, grabación, fotocopia, o cualquier otro, sin la previa autorización escrita de los titulares del copyright.

Índice

Objetivos.......... 5

1. Caso de uso 1: Docker Machine y OpenNebula.......... 7
- 1.1. Introducción a la Docker Machine 7
- 1.2. Instalar docker-machine 7
- 1.3. Instalar el *driver* para OpenNebula de docker-machine 8
  - 1.3.1. Preparación del sistema 8
  - 1.3.2. Instalación del *driver*.......... 8
- 1.4. Instalar docker-engine 9
- 1.5. Cómo utilizar el *driver* de OpenNebula con docker-machine 9
  - 1.5.1. Crear nuestro Docker Engine 9
  - 1.5.2. Enlazar la *shell* local con la de la máquina virtual 10
- 1.6. Ejemplo de uso 10

2. Caso de uso 2: Docker Swarm Cluster con Consul sobre OpenNebula.......... 12
- 2.1. Consul 13
- 2.2. Docker Swarm 14
  - 2.2.1. Swarm-master 14
  - 2.2.2. Swarm-nodo 14
- 2.3. Conexión con el Swarm-master 14
- 2.4. Red 14
- 2.5. Ejemplo de uso 15

3. Caso de uso 3: Balanceo transparente con Docker Swarm, Compose y Consul.......... 16
- 3.1. Consul 17
- 3.2. Docker Swarm 18
  - 3.2.1. Swarm-master 18
  - 3.2.2. Swarm-nodo 18
  - 3.2.3. Desplegar contenedores Registrator 18
- 3.3. Load Balancer 19
- 3.4. Dockerfile 20
- 3.5. Docker Compose 21
  - 3.5.1. Gestionar Docker Compose 22
- 3.6. Ejemplo de uso 22

4. Herramientas de automatización completa y *testing*: Jenkins.......... 28
- 4.1. Servlet Demo 29
- 4.2. Ejecución de test del proyecto 30

4.3. Construcción del proyecto 32
4.3.1. Despliegue del proyecto 32
4.3.2. Página inicial 32
4.3.3. Página resultado 32
4.4. Dockerización del proyecto 33
4.4.1. Construcción de la imagen docker 33
4.4.2. Fichero Dockerfile 36
4.4.3. Ejecución de la aplicación mediante contenedor Docker 37
4.4.4. Detención de la aplicación mediante contenedor Docker 38
4.5. Jenkins 38
4.5.1. Instalación mediante imagen dockerizada 38
4.5.2. Instalación mediante fichero WAR oficial 39
4.5.3. Configuración de Jenkins 39
4.6. Registry Docker Hub 43
4.6.1. Registro mediante imagen dockerizada 44
4.6.2. Contribuir con nuestra imagen a la comunidad 46
4.7. Proceso de automatización completo (*pipeline*) 47
4.7.1. Compilar el código cada vez que este sufra un cambio 49
4.7.2. Construcción y publicación de la imagen 51
4.7.3. Notificación de errores durante el proceso de automatización 53

Objetivos

Introducir al estudiante varios casos de uso 100 % prácticos sobre escenarios reales con aplicación directa de los conocimientos cubiertos en el módulo didáctico «Infraestructura DevOps».

1. Caso de uso 1: Docker Machine y OpenNebula

Para poder crear máquinas Docker Engine en la OpenNebula podemos usar el *driver* docker-machine; una vez instalado, podremos crear e instanciar nuestros contenedores de Docker.

Requerimientos:

- **Docker Engine**: Nos permite crear los contenedores, gestionarlos e instanciarlos.
- **Docker Machine**: Nos permite crear y gestionar las máquinas virtuales.
- ***Driver* Docker Machine OpenNebula**: Nos permite utilizar el docker-machine en el *cloud* del OpenNebula.

1.1. Introducción a la Docker Machine

Con docker-machine podemos aprovisionar y administrar múltiples Dockers remotos, además de aprovisionar clústeres Swarm en entornos Windows, Mac y Linux.

Es una herramienta que nos permite instalar el demonio Docker en *hosts* virtuales y administrar dichos *hosts* con el comando docker-machine. Además, podemos hacerlo en distintos proveedores (VirtualBox, OpenStack, OpenNebula, VmWare, etc.).

Usando el comando docker-machine podemos iniciar, inspeccionar, parar y reiniciar los *hosts* administrados, actualizar el cliente y demonio Docker, y configurar un cliente Docker para que interactúe con el *host*. Con el cliente podemos correr dicho comando directamente en el *host*.

1.2. Instalar docker-machine

Como usuario privilegiado, nos descargamos el binario y ejecutamos:

```
curl -L https://github.com/docker/machine/releases/download/v0.6.0/docker-machine-`uname -s`-
`uname -m` > /usr/local/bin/docker-machine && \
chmod +x /usr/local/bin/docker-machine
```

Una vez instalado, para comprobar que podéis ejecutar docker-machine, ejecutad desde vuestro usuario:

```
docker-machine version
```

Si os dijera que no encuentra el binario, ejecutad:

```
export PATH=$PATH:/usr/local/bin
```

Enlace de interés

Podéis encontrar más información en: <https://docs.docker.com/machine/install-machine/>.

1.3. Instalar el *driver* para OpenNebula de docker-machine

Antes de instalar el *driver*, tenemos que preparar el equipo para la instalación.

1.3.1. Preparación del sistema

1) Paquete GO

```
wget https://storage.googleapis.com/golang/go1.6.linux-amd64.tar.gz
tar -C /usr/local -xvzf go1.6.linux-amd64.tar.gz
export GOROOT=/usr/local/go
export PATH=$PATH:$GOROOT/bin
```

Creamos el directorio work en nuestro espacio de trabajo, por ejemplo, en el $HOME:

```
mkdir $HOME/work
```

Indicamos las variables de entorno necesarias:

```
export GOPATH=$HOME/work
export PATH=$PATH:$GOPATH/bin
```

2) Paquete GIT, BZR

```
sudo apt-get install git bzr
```

3) Paquete GODEP

```
go get github.com/tools/godep
```

1.3.2. Instalación del *driver*

Le damos la siguiente orden:

```
go get github.com/opennebula/docker-machine-opennebula
cd $GOPATH/src/github.com/opennebula/docker-machine-opennebula
```

```
make build
make install
```

1) ONE_AUTH y ONE_XMLRPC

Nos creamos un fichero donde indicaremos nuestro usuario y contraseña del OpenNebula:

```
mkdir -p $HOME/.one
echo usuario:password > $HOME/.one/one_auth
```

Creamos las variables de entorno necesarias:

```
export ONE_AUTH=$HOME/.one/one_auth
export ONE_XMLRPC=http://iaas.csuc.cat:2633/RPC2
```

Enlace de interés

Podéis encontrar más información en: <https://github.com/opennebula/docker-machine-opennebula>.

1.4. Instalar docker-engine

Tal como se ha podido ver en el módulo «Infraestructura DevOps» es necesario disponer del componente docker-engine para disponer de los comandos de control de contenedores de manera local.

Para instalar el docker-engine, se recomienda visitar la página siguiente: <https://docs.docker.com/engine/installation/linux/>.

1.5. Cómo utilizar el *driver* de OpenNebula con docker-machine

1.5.1. Crear nuestro Docker Engine

Existe una imagen en la OpenNebula de un sistema operativo donde ya está instalado el docker y donde podemos lanzar contenedores; se llama boot2docker. También existe una versión de Ubuntu.

Si no está disponible, desde el *marketplace* del OpenNebula la descargamos. Una vez tenemos descargada la imagen, creamos la máquina virtual:

```
docker-machine create --driver opennebula --opennebula-image-id [num_imagen_boot2docker]
--opennebula-network-id [num_red] --opennebula-b2d-size [medida_en_MB] [nombre_máquina_virtual]
```

Más opciones:

- --opennebula-cpu: CPU value for the VM.

- --opennebula-vcpu: VCPUs for the VM.
- --opennebula-memory: Size of memory for VM in MB.
- --opennebula-template-id: Template ID to use.
- --opennebula-template-name: Template to use.
- --opennebula-network-id: Network ID to connect the machine to.
- --opennebula-network-name: Network to connect the machine to.
- --opennebula-network-owner: User ID of the Network to connect the machine to.
- --opennebula-image-id: Image ID to use as the OS.
- --opennebula-image-name: Image to use as the OS.
- --opennebula-image-owner: Owner of the image to use as the OS.
- --opennebula-dev-prefixe: Dev prefixe to use for the images: 'vd', 'sd', 'hd', etc.
- --opennebula-disk-resize: Size of disk for VM in MB.
- --opennebula-b2d-size: Size of the Volatile disk in MB (only for b2d).
- --opennebula-ssh-user: Set the name of the SSH user.
- --opennebula-disable-vnc: VNC is enabled by default. Disable it with this flag.

1.5.2. Enlazar la *shell* local con la de la máquina virtual

Ahora conectaremos nuestra *shell* local a la de la máquina virtual para que cada vez que ejecutemos docker, este pueda dar órdenes al docker instalado en la máquina virtual.

Obtenemos las variables de entorno en la máquina virtual:

```
docker-machine env [nombre_máquina_virtual]
```

Enlazamos la *shell*:

```
eval "$(docker-machine env [nombre_máquina_virtual])"
```

Ahora ya podemos utilizar docker normalmente.

Variables de entorno

Para mantener las nuevas variables de entorno de manera persistente, añadidlas a vuestro .profile ($HOME/.profile).

1.6. Ejemplo de uso

Actualmente disponéis de dos imágenes, ya precreadas con el Docker Engine instalado, que podéis usar: Boot2docker y Docker-Machine-Ubuntu-14.04.

Crear un contenedor con nginx en una máquina Docker con el *driver* de Docker Machine en el OpenNebula

Crear una máquina en el OpenNebula con el Docker Engine y un disco volátil de 10 GB con el comando:

```
docker-machine create --driver opennebula --opennebula-network-id $NETWORK_ID --opennebula-image
-name boot2docker --opennebula-image-owner oneadmin --opennebula-b2d-size 10240 mydockerengine
```

Donde $NETWORK_ID será el ID de la red donde crearemos la máquina con Docker Engine.

Figura 1. Detalle del *dashboard* de OpenNebula

Fuente: Jordi Guijarro.

Podemos comprobar que todo ha funcionado bien con el comando:

```
docker-machine ls
```

Los siguientes comandos nos muestran las variables de entorno necesarias para acceder a la *shell*:

```
docker-machine env mydockerengine
eval $(docker-machine env mydockerengine)
```

Ahora podemos comenzar a usar docker sobre la máquina Docker Engine que acabamos de crear:

```
docker pull nginx
docker run --name mynginx -d -p 80:80 nginx
```

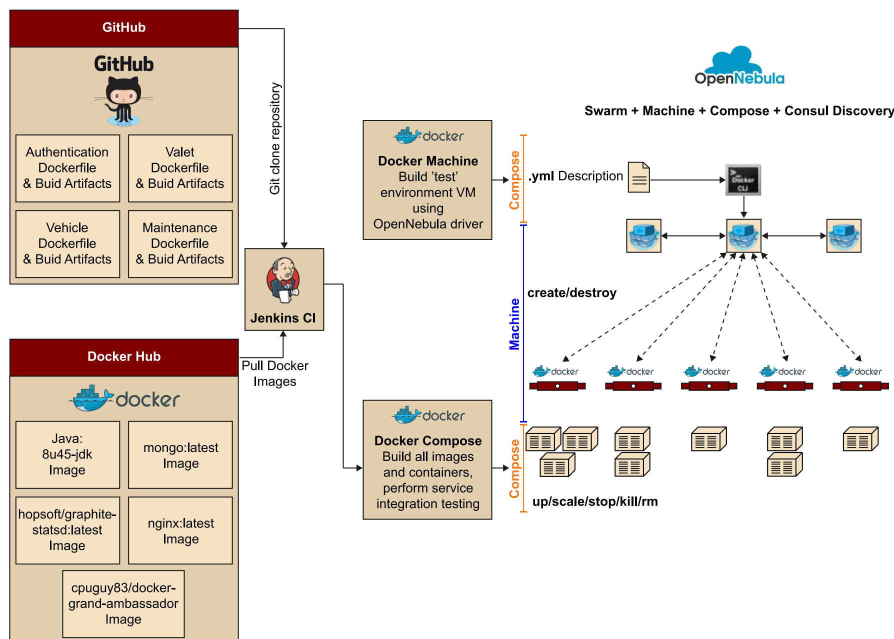
Finalmente, comprobamos que podemos acceder al servidor web que acabamos de crear.

2. Caso de uso 2: Docker Swarm Cluster con Consul sobre OpenNebula

En este apartado se tratará de la instalación de los siguientes requerimientos:

- Consul.
- Docker Swarm.
- Swarm-master.
- Swarm-nodo.
- Conexión con el Swarm-master.

Figura 2. Diagrama relacional de Git, los diferentes componentes de Docker y OpenNebula



La idea es disponer de un clúster con Docker Engine y poder instanciar en él nuestros contenedores Docker.

- **Docker Swarm:** Nos permite crear un clúster Docker. El nodo máster recibirá y distribuirá los contenedores a los nodos secundarios donde se ejecutarán.

- **Consul:** Nos proporcionará el servicio de *discovery* de los nuevos nodos que añadimos al clúster swarm.

1) Requerimientos previos

Los requerimientos previos son los siguientes:

- **Docker Engine:** Nos permite crear los contenedores, gestionarlos e instanciarlos.

- **Docker Machine:** Nos permite crear y gestionar las máquinas virtuales.

- ***Driver* Docker Machine OpenNebula:** Nos permite utilizar el docker-machine en el *cloud* del OpenNebula.

2) Instalar los requerimientos

Para poder implementar el **Docker Swarm Cluster** con el OpenNebula tenemos que instalar: **Docker Engine**, **Docker Machine** y el ***driver* de Docker Machine por el OpenNebula**.

Ved también

En los subapartados siguientes disponéis de un tutorial de cómo hacerlo.

2.1. Consul

Dispondremos de una máquina virtual con Docker Engine que nos proporcionará el servicio de *discovery* con Consul.

Usaremos el *driver* de Docker Machine por el OpenNebula, y crearemos una máquina virtual llamada Consul, donde después ejecutaremos un contenedor con el propio servicio de Consul dentro de esta:

```
docker-machine create -d opennebula --opennebula-network-id [network_id] --opennebula-image-id
[boot2docker_image_id] --opennebula-b2d-size [volatile_disk_size] consul
docker $(docker-machine config consul) run -d -p "8500:8500" -h "consul" progrium/consul -server
-bootstrap
CONSUL_IP=$(docker-machine ip consul)
```

Podemos verificar que se ha instalado correctamente el servicio de Consul accediendo a la IP de la máquina, que acabamos de crear, por el puerto 8500.

Figura 3. Confirmación de la correcta instalación de Consul

Fuente: Jordi Guijarro.

2.2. Docker Swarm

Dispondremos de dos tipos de máquinas virtuales con Swarm:

- **Swarm Master:** Esta máquina será la encargada de distribuir las instancias de los contenedores a los diferentes nodos que tengamos en el clúster.

- **Swarm Nodo:** Esta o estas máquinas serán las que ejecutarán los contenedores Docker.

2.2.1. Swarm-master

Creamos la máquina virtual, con el *driver* de Docker Machine, por el nodo máster de los clústeres:

```
docker-machine create -d opennebula --opennebula-network-id [network_id] --opennebula-image-id
[boot2docker_image_id] --opennebula-b2d-size [volatile_disk_size] --swarm --swarm-master
--swarm-discovery="consul://$CONSUL_IP:8500" --engine-opt cluster-store=consul://$CONSUL_IP:8500
--engine-opt cluster-advertise="eth0:2376" swarm-master
```

2.2.2. Swarm-nodo

Creamos la máquina virtual con el *driver* de Docker Machine, por los diferentes nodos de los clústeres:

```
docker-machine create -d opennebula --opennebula-network-id [network_id] --opennebula-image-id
[boot2docker_image_id] --opennebula-b2d-size [volatile_disk_size] --swarm
--swarm-discovery="consul://$CONSUL_IP:8500" --engine-opt cluster-store=consul://$CONSUL_IP:8500
--engine-opt cluster-advertise="eth0:2376" swarm-nodo-01
```

Podemos crear tantos nodos como queramos. Solo tenemos que ir modificando el nombre de la máquina virtual.

2.3. Conexión con el Swarm-master

Una vez hemos creado las diferentes máquinas virtuales que componen el clúster, podemos conectarnos al nodo máster con el siguiente comando:

```
eval $(docker-machine env --swarm swarm-master)
```

2.4. Red

Una vez tenemos creado nuestro clúster con Swarm, también tenemos la opción de poder crear redes privadas dentro del clúster:

```
docker network create --driver overlay --subnet=10.0.1.0/24 overlay_limpio
```

```
docker network ls
```

Figura 4. Detalles de las redes disponibles en Docker

```
NETWORK ID          NAME                    DRIVER
6968b8228eb9        swarm-master/none       null
04168ccd6192        swarm-master/host       host
a52245a565f3        overlay_net             overlay
3fd4a80e457b        swarm-master/bridge     bridge
f79e429ab6f1        swarm-node-01/none      null
2a5cd0d44a16        swarm-node-01/host      host
08957e6ef5a6        swarm-node-01/bridge    bridge
```

Fuente: Jordi Guijarro.

Enlace de interés

Para más información, podéis consultar el siguiente enlace: <http://opennebula.org/docker-swarm-with-opennebula/>.

2.5. Ejemplo de uso

Ejecutar contenedores Docker en el clúster creado

Ahora podemos empezar a usar Docker sobre el clúster que acabamos de crear. Utilizaremos un contenedor con nginx, por ejemplo:

```
docker pull nginx
docker run --name mynginx -d -p 80:80 nginx
```

Podemos comprobar cómo el contenedor se está ejecutando en un nodo del clúster. Para ello, podemos observar que el contenedor con nuestro nginx está corriendo en la máquina swarm-nodo-01:

```
docker ps
```

Figura 5. Detalle del contenedor corriendo en Swarm

```
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
                    NAMES
a4aa9d56ff83        nginx               "nginx -g 'daemon off"   12 minutes ago      Up 17 minutes       :80
->80/tcp, 443/tcp   swarm-node-01/mynginx
```

Fuente: Jordi Guijarro.

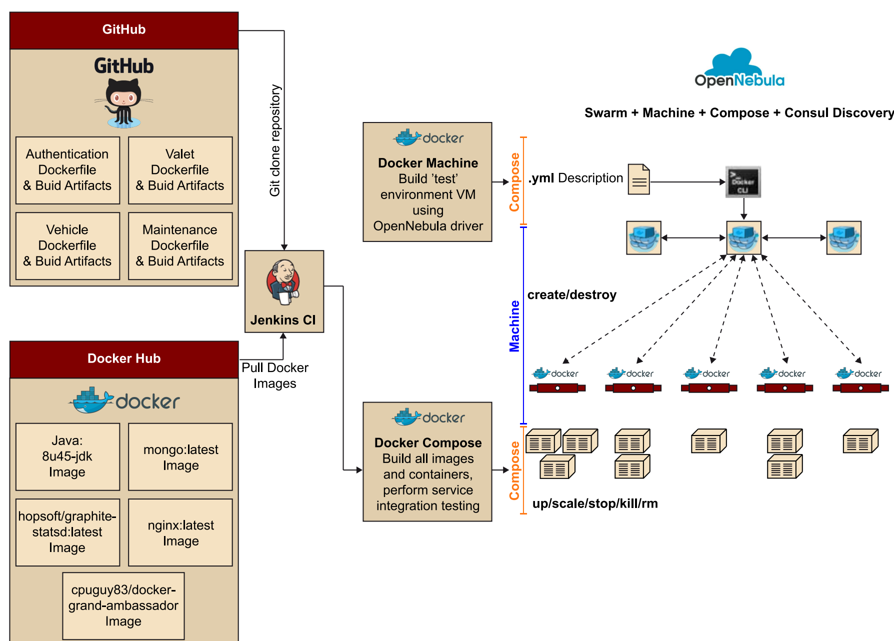
Para acceder al servicio del contenedor, tenemos que poner la dirección IP del nodo del clúster en el navegador.

3. Caso de uso 3: Balanceo transparente con Docker Swarm, Compose y Consul

En este apartado se tratará la instalación de los siguientes requerimientos previos:

- Consul.
- Docker Swarm.
- Load Balancer.
- Docker Compose.

Figura 6. Diagrama relacional de Git, los diferentes componentes de Docker y OpenNebula



A partir de la ilustración y para ir cubriendo todo el *pipeline* en este caso de uso, crearemos un clúster de docker-engine e instanciaremos allá nuestros contenedores docker. Además, configuraremos un servicio balanceado de alta disponibilidad escalable.

- **Docker Swarm:** Nos permite crear un clúster docker. El nodo máster recibirá y distribuirá los contenedores a los nodos secundarios donde se ejecutarán.

- **Consul:** Nos proporcionará el servicio de *discovery* de los nuevos nodos y servicios que añadimos al clúster swarm.

- **Registrator:** Este servicio monitorizará la información como por ejemplo IP y PUERTO de cada servicio activado en los *hosts* y la guardará en el consul.

- **Docker Compose:** Nos permite configurar y crear grupos de contenedores. Podemos escalar el servicio ofrecido por un contenedor aumentando o disminuyendo el número de contenedores.

1) Requerimientos previos

Los requerimientos previos son los siguientes:

- **Docker Engine:** Nos permite crear los contenedores, gestionarlos e instanciarlos.

- **Docker Machine:** Nos permite crear y gestionar las máquinas virtuales.

- ***Driver* Docker Machine OpenNebula:** Nos permite utilizar el docker-machine en el *cloud* del OpenNebula.

- **Docker Compose.**

2) Instalar los requerimientos

Para poder implementar el **docker swarm cluster** con el OpenNebula tenemos que instalar el **Docker Engine**, **Docker Machine** y el ***driver* Docker Machine OpenNebula**.

Ved también

En el apartado 1 de este módulo disponéis de un tutorial de cómo hacerlo.

Para instalar Docker Compose:

```
sudo -y curl
-L https://github.com/docker/compose/releases/download/1.6.2/docker-compose-`uname -s`-`uname -m`
> /usr/local/bin/docker-compose
chmod +x /usr/local/bin/docker-compose
exit
```

3.1. Consul

Dispondremos de una máquina virtual que nos proporcionará el servicio de *discovery* con consul.

Creamos una máquina virtual en la OpenNebula llamada consul y después ejecutamos un contenedor docker de consul dentro de ella.

```
docker-machine create -d opennebula --opennebula-network-id [network_id] --opennebula-image-id
[boot2docker_image_id] --opennebula-b2d-size [volatile_disk_size] consul
```

```
docker $(docker-machine config consul) run -d -p "8500:8500" -h "consul" progrium/consul
-server -bootstrap
```

3.2. Docker Swarm

Dispondremos de dos tipos de máquinas virtuales con swarm:

- **Swarm-master:** Esta máquina será la encargada de distribuir las instancias de los contenedores a los diferentes nodos que tengamos en el clúster.

- **Swarm-nodo:** Esta o estas máquinas serán las que ejecutarán los contenedores docker.

3.2.1. Swarm-master

Creamos la máquina virtual del swarm-master:

```
export CONSUL_IP=$(docker-machine ip consul)
docker-machine create -d opennebula --opennebula-network-id [network_id] --opennebula-image-id
[boot2docker_image_id] --opennebula-b2d-size [volatile_disk_size] --swarm --swarm-master
--swarm-discovery="consul://$CONSUL_IP:8500" --engine-opt cluster-store=consul://$CONSUL_IP:8500
--engine-opt cluster-advertise="eth0:2376" swarm-master
```

3.2.2. Swarm-nodo

Creamos la máquina virtual del swarm-nodo:

```
docker-machine create -d opennebula --opennebula-network-id [network_id] --opennebula-image-id
[boot2docker_image_id] --opennebula-b2d-size [volatile_disk_size] --swarm
--swarm-discovery="consul://$CONSUL_IP:8500" --engine-opt cluster-store=consul://$CONSUL_IP:8500
--engine-opt cluster-advertise="eth0:2376" swarm-nodo-1
```

Podemos crear los nodos que queramos. Solo tenemos que cambiar el nombre de la máquina virtual.

3.2.3. Desplegar contenedores Registrator

Una vez hemos creado el máster y los diferentes nodos, instanciaremos los contenedores con el registrator a todos ellos. Utilizaremos la imagen de gliderlabs/registrator.

Necesitamos tener un servicio Registrator corriendo en cada *host* para monitorizar todos los servicios que corre cada *host*.

```
export MASTER_IP=$(docker-machine ip swarm-master)
```

```
export NODO_IP=$(docker-machine ip swarm-nodo-1)
```

```
eval $(docker-machine env --swarm swarm-master)
docker run -d --name=registrator -h ${MASTER_IP} --volume=/var/run/docker.sock:/tmp/docker.sock
gliderlabs/registrator:latest consul://${CONSUL_IP}:8500

eval $(docker-machine env swarm-nodo-1)
docker run -d --name=registrator -h ${NODO_IP} --volume=/var/run/docker.sock:/tmp/docker.sock
gliderlabs/registrator:latest consul://${CONSUL_IP}:8500
```

3.3. Load Balancer

Ahora tenemos que implementar un balanceador de carga que pueda distribuir las peticiones por las diferentes instancias del servicio. Como aumentamos y disminuimos las instancias del servicio, necesitamos que el balanceador de carga se actualice automáticamente.

Utilizaremos nginx para el Load Balancer y consul-template para la configuración del nginx.

1) default.ctmpl

Antes de nada, necesitamos crear un *template* para la configuración del nginx. Consul-template automáticamente llenará este fichero con la información en lo referente al servicio instanciado y creará la configuración para el nginx.

El fichero default.ctmpl tiene que ser así:

```
{{$app := env "APP_NAME"}}

upstream {{printf $app}} {
    least_conn;
    {{range service $app}}
    server {{.Address}}:{{.Puerto}} max_fails=3 fail_timeout=60 weight=1;{{end}}
}

server {
    listen 80 default;

    location / {
        proxy_pass http://{{printf $app}};
    }
}
```

Crearemos la variable **app** con el valor de la variable de entorno APP_NAME (definida más adelante en el fichero docker-compose.yml). También crearemos un upstream con el nombre de la variable **app**. La línea least_conn hace que nginx encamine el tráfico hacia la instancia con menos conexiones.

Por cada instancia del servicio corriendo crearemos una línea con la dirección del nodo donde se ejecuta ({{.Address}}) y el puerto por donde está escuchando ({{.Puerto}}).

Finalmente está la zona de configuración del servidor. Aquí definimos el puerto de escucha del load balancer y creamos una reverse proxy en la upstream que hemos creado.

2) start.sh

Necesitamos un *script* que actúe como *entry point* para esta imagen de docker.

El fichero start.sh tiene que ser así:

```
#!/bin/bash
service nginx start
consul-template -consul=$CONSUL_URL -template="/templates/default.ctmpl:/etc/nginx/conf.d/
default.conf:service nginx reload"
```

Este *script* pone en marcha primero el servicio nginx. Después pone en marcha consul-template pasándole dos parámetros:

- -consul: Le pasamos la variable de entorno $CONSURL_URL definida en el docker-compose.yml.

- -template: Le pasamos tres parámetros: el *path* donde está el *template* que hemos creado anteriormente (dentro del contenedor), el *path* donde se guardará el fichero de configuración generado y el comando que se ejecutará una vez que se genere una nueva configuración.

El consul-template creará un nuevo fichero de configuración cada vez que un servicio se ponga en marcha o se pare. La información sobre estos servicios es recogida por el servicio **registrator** de cada nodo swarm y es almacenada en el consul.

Los ficheros **start.sh** y default.ctmpl los guardaremos en un directorio denominado «**files**». En el directorio guardaremos los ficheros **Dockerfile** y docker-compose.yml.

3.4. Dockerfile

```
FROM nginx:latest
```

```
RUN apt-get update \
  && apt-get install -y unzip

ADD filas/start.sh /bin/start.sh
RUN chmod +x /bin/start.sh
ADD filas/default.ctmpl /templates/default.ctmpl

ADD https://releases.hashicorp.com/consul-template/0.12.2/consul-template_0.12.2_linux_amd64.zip
/usr/bin/
RUN unzip /usr/bin/consul-template_0.12.2_linux_amd64.zip -d /usr/local/bin

EXPONGO 80
ENTRYPOINT ["/bin/start.sh"]
```

Este fichero utiliza nginx como imagen base e instala consul-template encima. Después copia el *script* start.sh y el *template* default.ctmpl (que antes hemos creado) dentro del contenedor. Finalmente, expone el puerto 80 y define el *script* start.sh como *entry point* de la imagen.

3.5. Docker Compose

Docker Compose nos permite escribir la configuración que queremos que tengan los contenedores que desplegar. Utilizaremos Docker Compose File version 2, que nos permite definir la configuración en lo referente a la red, volúmenes, puertos, variables de entorno.

Enlace de interés

Para saber más sobre la versión 2 del fichero de docker-compose, podéis consultar el siguiente enlace: <https://docs.docker.com/compose/compose-file/compose-file-v2/>.

1) docker-compose.yml

Ejemplo de docker-compose.yml:

```
version: '2'
services:
  lb:
    build: .
    container_name: lb
    puertos:
      - "80:80"
    environment:
      - APP_NAME=[Nombre_de el_servicio]
      - CONSUL_URL=${CONSUL_IP}:8500
    depends_on:
      - web
    networks:
      - frente-tier
  web:
    image: [imagen_docker]
```

```
      puertos:
        - "[Puerto_de el_servicio]"
      environment:
        - SERVICE_NAME=[Nombre_de el_servicio]
      networks:
        - frente-tier
networks:
  frente-tier:
    driver: overlay
```

3.5.1. Gestionar Docker Compose

Para arrancar todos los servicios, ejecutamos lo siguiente:

```
eval $(docker-machine env -swarm swarm-master)
docker-compose up -d
```

Para ver los detalles de los servicios corriendo podemos utilizar el comando:

```
docker-compose ps
```

Para parar y eliminar los servicios que están corriendo actualmente:

```
docker-compose stop; docker-compose rm -f
```

Ahora mismo solo tenemos una instancia del servicio. Si queremos aumentar o disminuir:

```
docker-compose scale [nombre servicio]=[número de instancias]
```

Enlace de interés

Si queréis obtener más información, visitad el siguiente enlace: <https://botleg.com/stories/load-balancing-with-docker-swarm/>.

3.6. Ejemplo de uso

Queremos crear un clúster de docker y crear un servicio web balanceado con nginx y escalable con Docker Compose.

Actualmente disponéis de dos imágenes, ya precreadas con el Docker Engine instalado, que podéis usar: Boot2docker o Docker-Machine-Ubuntu-14.04.

Creamos las máquinas virtuales de consul, swarm-master y un swarm-nodo:

```
docker-machine create -d opennebula --opennebula-network-id $NETWORK_ID --opennebula-image-name
boot2docker --opennebula-b2d-size 10240 consul
docker $(docker-machine config consul) run -d -p "8500:8500" -h "consul" progrium/consul -server
```

```
-bootstrap
export CONSUL_IP=$(docker-machine ip consul)

docker-machine create -d opennebula --opennebula-network-id $NETWORK_ID --opennebula-image-name
boot2docker --opennebula-b2d-size 10240 --swarm --swarm-master --swarm-discovery="consul:
//$CONSUL_IP:8500" --engine-opt cluster-store=consul://$CONSUL_IP:8500 --engine-opt
cluster-advertise="eth0:2376" swarm-master

docker-machine create -d opennebula --opennebula-network-id $NETWORK_ID --opennebula-image-name
boot2docker --opennebula-b2d-size 10240 --swarm --swarm-discovery="consul://$CONSUL_IP:8500"
--engine-opt cluster-store=consul://$CONSUL_IP:8500 --engine-opt cluster-advertise="eth0:2376"
swarm-nodo-1
```

Nota

Con --opennebula-b2d-size 10240 creamos discos volátiles de 10 GB.

Donde $NETWORK_ID será el ID de la red donde crearemos el clúster.

Figura 7. Detalle del *dashboard* de OpenNebula

Virtual Machines

ID	Owner	Group	Name	Status	Host	IPs
2265		Docker	swarm-node-01	RUNNING	cluster06	
2264		Docker	swarm-master	RUNNING	cluster04	
2261		Docker	consul	RUNNING	cluster04	

Showing 1 to 3 of 3 entries

3 TOTAL 3 ACTIVE 0 OFF 0 PENDING 0 FAILED

Fuente: Jordi Guijarro.

Desplegamos Registrator en todos los nodos:

```
export MASTER_IP=$(docker-machine ip swarm-master)
export NODO_IP=$(docker-machine ip swarm-nodo-1)

eval $(docker-machine env --swarm swarm-master)
docker run -d --name=registrator -h ${MASTER_IP} --volume=/var/run/docker.sock:/tmp/docker.sock
gliderlabs/registrator:latest consul://${CONSUL_IP}:8500

eval $(docker-machine env swarm-nodo-1)
docker run -d --name=registrator -h ${NODO_IP} --volume=/var/run/docker.sock:/tmp/docker.sock
gliderlabs/registrator:latest consul://${CONSUL_IP}:8500
```

Creamos los ficheros de configuración necesarios:

1) files/default.ctmpl

```
{{$app := env "APP_NAME"}}
```

```
upstream {{printf $app}} {
    least_conn;
    {{range service $app}}
    server {{.Address}}:{{.Puerto}} max_fails=3 fail_timeout=60 weight=1;{{end}}
}

server {
    listen 80 default;

    location / {
        proxy_pass http://{{printf $app}};
    }
}
```

2) files/start.sh

```
#!/bin/bash
service nginx start
consul-template -consul=$CONSUL_URL -template="/templates/default.ctmpl:/etc/nginx/conf.d/
default.conf:service nginx reload"
```

3) Dockerfile

```
FROM nginx:latest

RUN apt-get update \
  && apt-get install -y unzip

ADD files/start.sh /bin/start.sh
RUN chmod +x /bin/start.sh
ADD filas/default.ctmpl /templates/default.ctmpl

ADD https://releases.hashicorp.com/consul-template/0.12.2/consul-template_0.12.2_linux_amd64.zip
/usr/bin/
RUN unzip /usr/bin/consul-template_0.12.2_linux_amd64.zip -d /usr/local/bin

EXPOXE 80
ENTRYPOINT ["/bin/start.sh"]
```

4) docker-compose.yml

```
version: '2'
services:
  lb:
    build: .
    container_name: lb
```

```
      puertos:
        - "80:80"
      environment:
        - APP_NAME=web_nginx
        - CONSUL_URL=${CONSUL_IP}:8500
      depends_on:
        - web
    web:
      image: nginx
      puertos:
        - "80"
      environment:
        - SERVICE_NAME=web_nginx
networks:
  default:
    driver: overlay
```

Activamos el servicio. Desde el directorio donde tenemos el docker-compose.yml, ejecutamos:

```
eval $(docker-machine env -swarm swarm-master)
docker-compose up -d
docker-compose ps
```

Figura 8. Detalle de la activación del servicio

```
Creating network "swarmcompose_default" with driver "overlay"
Pulling web (nginx:latest)...
swarm-node-1: Pulling nginx:latest... : downloaded
swarm-master: Pulling nginx:latest... : downloaded
Building lb
Step 1 : FROM nginx:latest
 ---> eb4a127a1188
Step 2 : RUN apt-get update   && apt-get install -y unzip
 ---> Running in 3d6a67072f9b
Get:1 http://security.debian.org jessie/updates InRelease [63.1 kB]
Ign http://nginx.org jessie InRelease
Ign http://httpredir.debian.org jessie InRelease
Get:2 http://security.debian.org jessie/updates/main amd64 Packages [291 kB]
Get:3 http://nginx.org jessie Release.gpg [287 B]
Get:4 http://httpredir.debian.org jessie-updates InRelease [142 kB]
Get:5 http://nginx.org jessie Release [2325 B]
Get:6 http://nginx.org jessie/nginx amd64 Packages [7377 B]
Get:7 http://httpredir.debian.org jessie Release.gpg [2373 B]
Get:8 http://httpredir.debian.org jessie Release [148 kB]
Get:9 http://httpredir.debian.org jessie-updates/main amd64 Packages [7407 B]
Get:10 http://httpredir.debian.org jessie/main amd64 Packages [9034 kB]
Fetched 9698 kB in 20s (480 kB/s)
Reading package lists...
Reading package lists...
Building dependency tree...
Reading state information...
Suggested packages:
  zip
The following NEW packages will be installed:
  unzip
0 upgraded, 1 newly installed, 0 to remove and 0 not upgraded.
Need to get 162 kB of archives.
After this operation, 347 kB of additional disk space will be used.
Get:1 http://httpredir.debian.org/debian/ jessie/main unzip amd64 6.0-16+deb8u2 [162 kB]
debconf: delaying package configuration, since apt-utils is not installed
Fetched 162 kB in 1s (97.2 kB/s)
Selecting previously unselected package unzip.
(Reading database ... 9733 files and directories currently installed.)
Preparing to unpack .../unzip_6.0-16+deb8u2_amd64.deb ...
Unpacking unzip (6.0-16+deb8u2) ...


:~/Escriptori/swarm+compose$ docker-compose ps
      Name               Command          State                 Ports
--------------------------------------------------------------------------------------
lb                   /bin/start.sh        Up      443/tcp,              :80->80/tcp
swarmcompose_web_1   nginx -g daemon off;   Up      443/tcp,              :32768->80/tcp
```

Fuente: Jordi Guijarro.

Para aumentar o disminuir las instancias del servicio, utilizamos el siguiente comando:

```
docker-compose scale web=4
```

Con este comando aumentaremos en 4 las instancias desplegadas del servicio web.

El nombre que indicamos tiene que ser el nombre que hemos indicado en el fichero docker-compose.yml. Nosotros en el fichero docker-compose.yml hemos indicado que el nombre del servicio era **web**.

Ejecutamos **docker-compose ps** para saber el estado de las instancias. Hemos borrado las direcciones IP. Aquí se vería cómo las instancias están repartidas por los diferentes nodos del clúster swarm.

Figura 9. Detalle de las múltiples instancias del servicio

```
:~/Escriptori/swarm+compose$ docker-compose ps
      Name               Command          State                 Ports
--------------------------------------------------------------------------------------
lb                   /bin/start.sh        Up      443/tcp,              :80->80/tcp
swarmcompose_web_1   nginx -g daemon off;   Up      443/tcp,              :32768->80/tcp
swarmcompose_web_2   nginx -g daemon off;   Up      443/tcp,              :32769->80/tcp
swarmcompose_web_3   nginx -g daemon off;   Up      443/tcp,              :32768->80/tcp
swarmcompose_web_4   nginx -g daemon off;   Up      443/tcp,              :32769->80/tcp
```

Fuente: Jordi Guijarro.

Para acceder al servicio, tenemos que acceder con la IP del **load balancer (lb)**.

Para finalizar, podemos optar por parar y/o borrar los servicios:

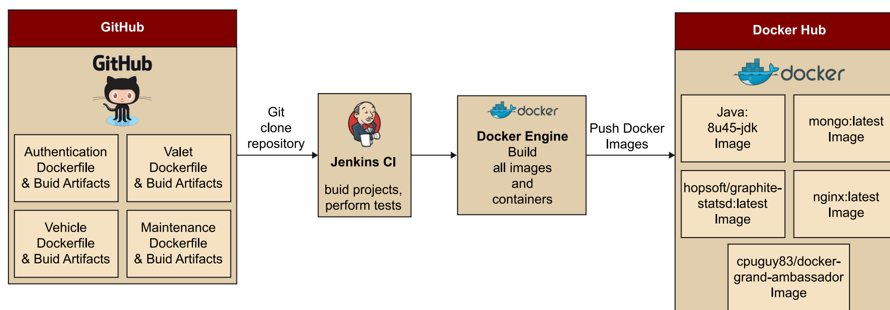
```
docker-compose stop; docker-compose rm -f
```

4. Herramientas de automatización completa y *testing*: Jenkins

Requerimientos previos:

- **Java:** Lenguaje de programación multiplataforma.
- **Maven:** Herramienta de gestión y construcción de software, encargada de proveer vía online de las dependencias necesarias.
- **Git:** Software de control de versiones
- **Tomcat:** Funciona como un contenedor de servlets que implementa las especificaciones de Java Servlet y JavaServer Pages (JSP).
- **Docker Engine:** Nos permite crear los contenedores, gestionarlos e instanciar-los.
- **Jenkins:** Software dedicado a la integración continua que nos permitirá automatizar los procesos de test, construcción y despliegue entre otros.
- **Registry Docker Hub:** Nos permite crear, compartir y utilizar imágenes creadas por nosotros o por terceros.

Figura 10. Diagrama relacional de Git, los diferentes componentes de Docker y OpenNebula



En este caso de uso partiremos de un proyecto web desarrollado en Java cumpliendo con las especificaciones de JavaServlet Pages. Por ello, se deberá instalar **Java Development Kit** (JDK) y descargar la librería **JUnit 4** para ejecutar los test; por último, solo quedará instalar el contenedor de servlets **Apache Tomcat**.

4.1. Servlet Demo

El servlet utilizado durante la fase de automatización es un ejemplo sencillo de una aplicación web Java ubicada en un sistema de versionado público como GitHub.

Para trabajar con el proyecto de forma individual se deberá hacer un «*fork*» del repositorio, lo que permitirá a cada alumno poder tener una copia remota con permisos para hacer cualquier cambio requerido.

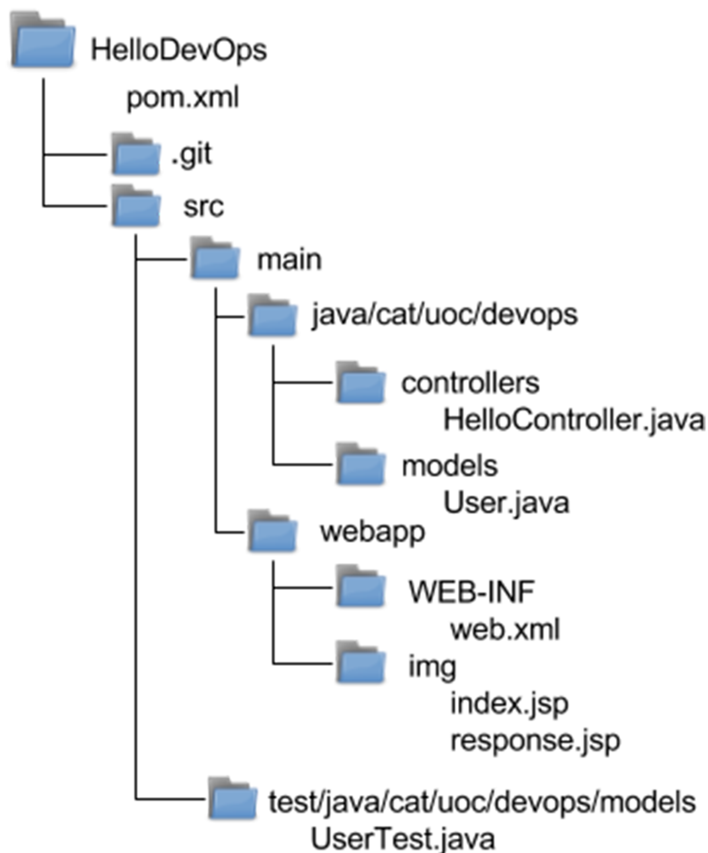
Una vez disponemos del *fork* del repositorio en nuestra cuenta de GitHub, procederemos al clonado en local mediante el comando:

```
git clone https://github.com/<usuario>/HelloDevOps.git
```

El proyecto demo está basado en un servlet JSP con una clase principal User junto con su controlador y los test JUnit, que verificarán el correcto funcionamiento de este.

La estructura completa del proyecto será la siguiente:

Figura 11. Detalle de las estructura del proyecto



Fuente: Joan Caparrós.

El proyecto HelloDevOps contiene los siguientes subdirectorios:

- **.git:** Esta carpeta se crea automáticamente en proyectos versionados bajo Git.

- **src/main:** Contendrá los modelos y los controladores junto con los jsp que construirán las páginas web de la aplicación.

- **src/test:** Contendrá los diferentes test desarrollados bajo la librería JUnit.

4.2. Ejecución de test del proyecto

Para ejecutar las pruebas unitarias, no es necesario configurar nada. La configuración predeterminada de Maven escaneará el directorio ${basedir}/src/test/java en busca de ficheros con el patrón *Test.java.

Para efectuar las pruebas unitarias desarrolladas, ejecutaremos:

```
$ mvn test
```

En el caso de superar los test, verificaremos que todos los test se han superado y que el resultado completo de la ejecución de las pruebas unitarias ha sido satisfactorio.

Figura 12. Detalle de una ejecución satisfactoria de los test

```
...
-------------------------------------------------------
T E S T S
-------------------------------------------------------
Running cat.uoc.devops.models.UserTest
Testing getName
Testing setName
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.044 sec

Results :

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
...
```

Fuente: Joan Caparrós.

Existen otros tipos de test, llamados **test de integración continua**, que se diferencian de los unitarios en que no necesitan disponer de todo el entorno de la aplicación para ejecutarse, sino que analizan las aplicaciones a partir de sus contextos (partes independientes en las que se divide una aplicación).

Maven incorpora la posibilidad de poder realizar estos test independientes sin que afecten a la continuidad de la construcción del proyecto. Imaginad que uno de los contextos de nuestra aplicación formase parte de un sistema externo susceptible a sufrir inestabilidades, los test pararían el proceso de construcción cada vez que este no respondiera de forma adecuada. Para estos casos, Maven –mediante el uso de su *plugin* Failsafe– introduce la posibilidad de incorporar en el ciclo de vida de construcción las fases de «pre-integration-test», «integration-test» y «post-integration-test».

Para el uso del *plugin* Failsafe se deberá añadir al fichero pom.xml el siguiente código:

```
<project>
  [...]
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-failsafe-plugin</artifactId>
        <version>2.19.1</version>
        <configuration>
            <testFailureIgnore>false</testFailureIgnore>
       <configuration>
        <executions>
          <execution>
            <goals>
              <goal>integration-test</goal>
              <goal>verify</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
  [...]
</project>
```

Según convención, los test que queramos integrar en los test de integración continua deberán incorporar «IT» o «IntegrationTest» como sufijo en el nombre del fichero.

Definir si el proceso de *build* debe o no pararse al fallar un test de integración continua vendrá indicado por la variable *testFailureIgnore* de nuestro fichero pom.xml, «false» (valor por defecto) para detener el proceso ante fallos de test de integración continua y «true» para continuar pese el fallo.

4.3. Construcción del proyecto

Para compilar el proyecto de forma manual tendremos que ejecutar el siguiente comando:

```
$ mvn package
```

Mediante el uso de Maven y dado el contenido del fichero principal **pom.xml**, el comando resolverá todas las dependencias, procederá a la ejecución de los test desarrollados para finalmente empaquetar el proyecto dentro de un fichero WAR en el directorio target.

Los archivos WAR (Web Application Archive) son ficheros utilizados para la distribución de JavaServer Pages, servlets, clases Java, archivos XML, librerías y otros ficheros necesarios para el despliegue de aplicaciones en contenedores web como el proporcionado por Tomcat.

4.3.1. Despliegue del proyecto

Una vez realizada la construcción del proyecto y para hacer el servlet accesible, deberemos:

- Copiar el archivo WAR generado en la carpeta webapps de nuestro tomcat.
- Reiniciar el servicio de tomcat.

A partir de este momento la aplicación estará disponible bajo el subdirectorio con nombre igual al fichero WAR desplegado.

4.3.2. Página inicial

La página inicial se compone de un formulario simple que requiere un nombre en forma de cadena de caracteres.

Figura 13. Detalle de la página inicial

Ejemplo JSP - Automatización y testing

Entre su nombre: DevOp Enviar

Fuente: Joan Caparrós.

4.3.3. Página resultado

La página resultado utiliza la cadena anteriormente introducida para saludar al programador en cuestión.

Figura 14. Detalle de la página resultado

Ejemplo JSP - Automatización y testing

Hola DevOp

Fuente: Joan Caparrós.

4.4. Dockerización del proyecto

En este momento disponemos de una aplicación web funcional, que requiere un contenedor de aplicaciones Apache Tomcat para hacerla accesible.

¿Cómo hacerla portable y que pueda ser ejecutada en cualquier máquina con docker? La respuesta es la dockerización de la aplicación.

Docker nos permitirá introducir en una «caja» todas aquellas cosas que la aplicación necesita para ser ejecutada, sin tener que preocuparse por la versión de software que la máquina *host* tiene instalada, por si tiene instalados todos los módulos necesarios o por si son compatibles o no con nuestra aplicación.

El despliegue de la imagen dockerizada resultante ofrecerá a los desarrolladores, *testers* y administradores de sistemas confianza y seguridad en el producto resultante, ya que asegurará el entorno de ejecución, permitiendo a los desarrolladores centrarse en el código sin preocuparse de posibles fallos dados por las máquinas donde se ejecutarán.

4.4.1. Construcción de la imagen docker

Para construir una imagen docker podemos partir de cualquier contenedor disponible en el registro de imágenes de Docker (Docker Hub Registry), donde se encuentran disponibles aplicaciones funcionales, tales como bases de datos, servidores de aplicaciones, servidores web, etc.; lo único necesario será construir un archivo Dockerfile que contenga la composición de nuestro contenedor.

Nota

Docker Hub dispone de más de cien mil imágenes disponibles de forma pública y listas para ser usadas.

Nuestra imagen partirá de la imagen oficial dockerizada del contenedor de aplicaciones web tomcat en su versión 8.5. El funcionamiento del contenedor de aplicaciones es sencillo y solo requerirá copiar en su carpeta webapps el fichero WAR resultante de la construcción del proyecto para que de esta forma al ejecutar la aplicación se realice el despliegue de la aplicación de manera automática.

El fichero Dockerfile pondrá por escrito todo aquello necesario para el despliegue de nuestra aplicación, ejecutando todas aquellas instrucciones para preparar el entorno encapsulado. Este fichero puede construirse en cualquier directorio de nuestra máquina.

Las instrucciones más importantes disponibles para la construcción de un fichero Dockerfile son:

1) FROM: establece la imagen base a partir de la cual crearemos la imagen que construirá el Dockerfile. Un Dockerfile válido debe situar la instrucción FROM como primera línea (excluyendo comentarios).

```
#Inclusión de la última versión de la imagen (por defecto tag:latest).
FROM <imagen>

#Definición de la versión específica de la imagen a utilizar mediante tag.
FROM <imagen>:<tag>

#Definición de la versión específica de la imagen a utilizar mediante el uso del código sha256
generado durante la construcción de la imagen.
FROM <imagen>@<digest>
```

2) MAINTAINER: permite establecer el campo autor de la imagen generada.

```
MAINTAINER <autor>
```

3) ENV: establece variables de entorno que podrán ser interpretadas por las instrucciones mediante el formato $variable o ${variable}.

```
ENV <variable> <valor>
ENV <variable>=<valor>
```

4) RUN: permite ejecutar una instrucción en el contenedor, para instalar o persistir cambios en el contenedor. Resulta útil para la instalación de paquetería mediante gestores de paquetes o ejecución de *scripts* que afecten a la construcción de la imagen.

```
#Ejecución del comando en una shell, por defecto /bin/sh -c en Linux y cmd /S /C em Windows
RUN <command>

#Ejecución del comando en una shell en formato parametrizado (formato exec)
RUN ["ejecutable", "param1", "param2"]
```

5) ADD: permite añadir ficheros, directorios o ficheros remotos al sistema de ficheros del contenedor.

```
#Copia ficheros con origen local o remoto en un destino dentro del contenedor
ADD <origen>... <destino>

#Versión parametrizada de la instrucción
ADD ["<origen>",... "<destino>"]
```

6) COPY: similar a la instrucción ADD, permite añadir ficheros y directorios pero no permite la inclusión de ficheros remotos.

```
COPY <origen>... <destino>
COPY ["<origen>",... "<destino>"]
```

7) VOLUME: crea un punto de montaje en el sistema de ficheros del contenedor. Los volúmenes permiten externalizar un determinado directorio y así proporcionar persistencia a los datos depositados (las imágenes de docker no almacenan datos en el contenedor entre diferentes ejecuciones), lo que permite a su vez que estos directorios sean compartidos por otros contenedores o por la máquina anfitrión.

```
VOLUME ["/data"]
```

8) EXPOSE: permite exponer los puertos TCP/IP por los que se pueden acceder a los servicios del contenedor. Por ejemplo: puerto 22 para SSH, 80 para HTTP o 3306 para MySQL. Esta instrucción expone los puertos para ser usados por el propio contenedor. En el caso de querer hacerlos accesibles desde fuera del contenedor, se deberá utilizar el *flag* -p (publish) durante la ejecución.

```
EXPOSE <puerto> [<puerto>...]
```

9) CMD: similar al RUN, permite la ejecución de instrucciones pero con la diferencia de que este no se ejecuta al construir la imagen, sino que lo hace en el momento de ejecución. Para que el fichero Dockerfile sea válido, solo puede contener una única instrucción CMD.

```
#Formato exec parametrizado
CMD ["ejecutable","param1","param2"]

#Definición de los parámetros utilizados por ENTRYPOINT
CMD ["param1","param2"]

#Formato shell
CMD comando param1 param2
```

10) ENTRYPOINT: permite sobreescribir el binario por defecto (/bin/sh -c) que ejecutará CMD.

```
#Formato exec parametrizado
ENTRYPOINT ["ejecutable", "param1", "param2"]

#Formato shell
ENTRYPOINT comando param1 param2
```

4.4.2. Fichero Dockerfile

Volviendo al ejemplo, la definición del fichero Dockerfile se ha establecido dentro de la carpeta principal **HelloDevOps** del proyecto, por tanto, las referencias a ficheros partirán de esta carpeta, y quedarán de la siguiente forma:

```
#Imagen de partida tomcat en su versión 8.5
FROM tomcat:8.5-jre8

#Copiamos el contenedor WAR generado de nuestra aplicación dentro la carpeta webapps.
COPY target/hellodevops.war  /usr/local/tomcat/webapps/
```

En este momento tenemos todo lo necesario para la creación de una imagen docker.

Para crear una imagen a partir del fichero Dockerfile, nos situaremos en el directorio donde se encuentra el dichero y ejecutaremos:

```
#Instrucción de creación de imágenes:
#docker build -t <nombre_de_la_imagen> <ruta_al_fichero_Dockerfile>

$ docker build -t hellodevops:v1.
```

- **docker build:** Es el comando utilizado para la creación de una imagen docker.
- **-t hellodevops:v1:** Indicamos mediante el *flag* **-t** el nombre de la imagen y opcionalmente un *tag* en formato «nombre:tag».
- .: Indica que el fichero Dockerfile se encuentra en el directorio donde nos encontramos.

Al finalizar la construcción de nuestra imagen docker se mostrará el resultado del proceso, y en el caso de ser exitoso acompañará con el identificador de nuestra nueva imagen.

```
Successfully built 94af2f7c9151
```

Para verificar que la imagen se ha creado correctamente, usaremos la instrucción **docker images**:

```
$ docker images
REPOSITORY   TAG     IMAGE ID     CREATED      SIZE
hellodevops  v1   94af2f7c9151  4 minutes ago 332.8 MB
tomcat  8.5-jre8  fd9c13f14ae6  11 hours ago  332.6 MB
```

Dentro del resultado deberíamos encontrar la imagen creada junto con la dependencia de la imagen del Apache Tomcat 8.5, que podremos incluir en futuros proyectos sin la necesidad de volver a descargarlo.

4.4.3. Ejecución de la aplicación mediante contenedor Docker

Para iniciar un contenedor es tan sencillo como usar el comando **docker run**. Si la imagen no existe en el equipo local, Docker intentará buscarla en el registro público de Docker. Es importante tener en cuenta que los contenedores están diseñados para detenerse cuando el comando ejecutado en su interior termina.

```
Instrucción para la ejecución de contenedores Docker:
#docker run [OPCIONES] IMAGEN[:TAG|@DIGEST] [COMANDO] [ARG...]

$ docker run -p 8080:8080 -d hellodevops:v1
```

- **docker run:** Comando utilizado para la ejecución de contenedores a partir de una imagen.

- **-p 8080:8080:** Indicamos mediante el *flag* -p <puerto_local>:<puerto_en_el_contenedor> la relación de puertos que se establecerá para que nuestra aplicación sea accesible desde la máquina anfitrión. En este caso, la relación será directa y el puerto 8080 local referenciará al puerto 8080 dentro del Apache Tomcat del contenedor docker.

- **-d:** Indicamos mediante el *flag* -d que el contenedor se ejecutará en segundo plano; de no ser así, la duración de la aplicación estaría unida a la sesión de nuestro terminal.

- **hellodevops:v1:** Es el nombre de la imagen con la cual queremos crear el contenedor. También podemos usar el id pero el nombre es más fácil de recordar.

En este momento, mediante el comando docker ps verificaremos que nuestro contenedor está activo, con la asignación de puertos establecida.

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
f37315e8cf14        hellodevops:v1      "catalina.sh run"   9 minutes ago       Up 9 minutes

PORTS                    NAMES
0.0.0.0:8080->8080/tcp   pensive_jennings
```

Comprobamos el correcto funcionamiento de la aplicación web dockerizada mediante la URL local: http://<my.servlet.host>:8080/hellodevops/.

4.4.4. Detención de la aplicación mediante contenedor Docker

Para detener un contenedor docker disponemos de dos maneras distintas:

1) docker stop: detiene un contenedor en ejecución enviando primero una señal SIGTERM a la aplicación principal dentro del contenedor para luego, después de un tiempo de gracia, enviar una señal SIGKILL:

```
$ docker stop f37315e8cf14
f37315e8cf14
```

2) docker kill: detiene un contenedor en ejecución enviando directamente la señal SIGKILL o la señal indicada mediante **--signal** a la aplicación principal dentro del contenedor.

```
$ docker kill f37315e8cf14
f11f5cc90e84
```

La ejecución de docker stop intenta acabar con la aplicación de manera amistosa mediante una señal estándar POSIX, mientras que docker kill mata el proceso. La recomendación de uso será siempre dentro de lo posible la utilización del comando docker stop.

4.5. Jenkins

Esta herramienta nos ofrece un componente fiable para ejercer de servidor de integración continua. El propósito principal para su uso no es solo por su capacidad de automatización de procesos, sino que ofrece una plataforma orientada al seguimiento del proceso de ejecución de todas estas tareas.

4.5.1. Instalación mediante imagen dockerizada

Conociendo el funcionamiento de los contenedores, podemos optar por la utilización de la imagen oficial dockerizada de Jenkins suministrada por Docker Hub; este contenedor nos proporcionará el servicio de automatización de una manera sencilla y rápida.

```
#docker run [OPCIONES] IMAGEN [COMANDO] [ARGUMENTOS...]
$ docker run -p 8090:8080 -p 50000:50000   \
-v <ubicación_fichero_docker.sock>:/var/run/docker.sock -v $(which docker):/usr/bin/docker jenkins
```

- **docker run:** Comando utilizado para la ejecución de contenedores a partir de una imagen.

- **-p 8090:8080 -p 50000:50000:** Esta aplicación será accesible a través de los puertos 8090 ofreciendo la interfaz gráfica web y 50000 exponiendo la API, para ejecución de operaciones remotas.

- **-v <ubicación_fichero_docker.sock>:/var/run/docker.sock -v $(which docker):/usr/bin/docker:** Dado que este contenedor requiere el uso de los comandos docker, mapearemos el socket y el propio binario docker para que este pueda ser usado por el contenedor.

- **jenkins:** Nombre de la imagen del contenedor en Docker Hub.

En el caso de precisar persistencia de nuestras configuraciones, mapearemos mediante la opción «-v» el *path* del *home* de Jenkins dentro del contenedor con una ubicación conocida de nuestro ordenador.

```
$ docker run -p 8090:8080 -p 50000:50000 -v <ubicación_local>:/var/jenkins_home  \
-v <ubicación_fichero_docker.sock>:/var/run/docker.sock -v $(which docker):/usr/bin/docker jenkins
```

Nota

La ubicación del fichero docker.sock en la mayoría de las distribuciones Linux es /var/run/docker.sock.

4.5.2. Instalación mediante fichero WAR oficial

La instalación de Jenkins puede realizarse mediante el despliegue del fichero WAR que encontraremos disponible para su descarga en la página oficial de Jenkins. Para ello, disponemos de dos formas:

- Ubicando el fichero WAR en el directorio webapps de nuestro tomcat.

- Ejecutando **java -jar jenkins.war--httpPort=8090 start** donde hayamos guardado nuestro fichero oficial.

En ambos casos la aplicación será accesible mediante la URL http:/<my.jenkins.host>:8090.

4.5.3. Configuración de Jenkins

A la hora de montar un entorno de integración continua se requerirá dotar a la plataforma de las herramientas necesarias para la ejecución de todos los procesos automatizados. Jenkins ya incorpora muchos mecanismos que tan solo deberán ser configurados, como el servidor de correo, pero otros deberán instalarse (Maven y *plugins*).

Configuración servidor de correo saliente

Jenkins notificará a los desarrolladores los sucesos importantes producidos durante cada una de sus operaciones.

Para configurar el servidor de correo saliente a través de la interfaz gráfica de Jenkins, nos dirigiremos utilizando los diferentes menús a *Administrar Jenkins* -> *Configurar el Sistema* y al apartado dedicado a la Notificación por correo electrónico.

Figura 15. Detalle de menú de configuración de Jenkins

Fuente: Joan Caparrós.

El servidor utilizado puede ser local o hacer referencia a un servidor smtp ofrecido por terceros. A continuación mostramos un ejemplo de configuración de la plataforma Jenkins para la utilización de Gmail como plataforma de envío de correos.

Figura 16. Detalle de la configuración del servidor de correo saliente de Jenkins

Notificación por correo electrónico

Servidor de correo saliente (SMTP)	smtp.gmail.com
sufijo de email por defecto	@gmail.com
☑ Usar autenticación SMTP	
Nombre de usuario	<Nombre Usuario/Servidor>
Contraseña	••
Usar seguridad SSL	☑
Puerto de SMTP	465
Dirección para la respuesta	<email respuesta>
Juego de caracteres	UTF-8

☐ Probar la configuración enviando un correo de prueba

Fuente: Joan Caparrós.

Una vez configurado, podremos enviar un correo de prueba para asegurar la correcta configuración del servidor.

Instalación Maven en Jenkins

Debido a que Jenkins utilizará los comandos mvn para la compilación y construcción de nuevas imágenes, será un requisito que Maven esté instalado dentro de nuestro contenedor Jenkins.

Para instalar Maven a través de la interfaz gráfica de Jenkins, nos dirigiremos utilizando los diferentes menús a *Administrar Jenkins -> Administrar Plugins* y en el apartado dedicado a la configuración de Maven pondremos un nombre para la versión elegida (por ejemplo: «Maven 3.3.9») y clicaremos en *Instalar automáticamente*.

Figura 17. Detalle de la configuración de Maven en Jenkins

Fuente: Joan Caparrós.

A partir de este momento, Maven ya estará disponible para ser incorporado en la *pipeline* del proceso de construcción de nuestros proyectos.

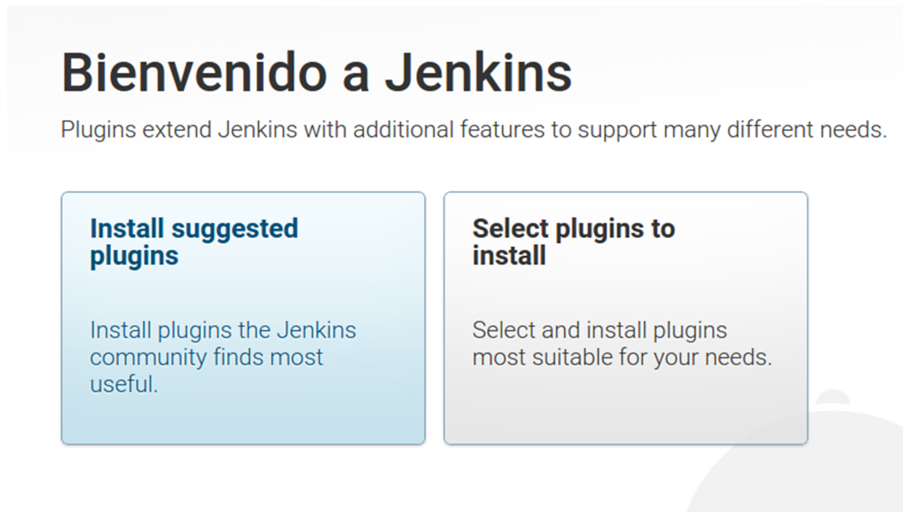
Instalación de *plugins* necesarios

Para poder realizar las acciones propuestas en este caso de uso, se requerirá la instalación de distintos *plugins* de Jenkins.

Jenkins pone a disposición la capacidad de instalar los *plugins* más usados por la comunidad durante la fase de configuración de la aplicación.

Así pues, podremos adquirir la mayoría de los *plugins* necesarios clicando sobre «Install suggested plugins» dentro de la página de bienvenida de Jenkins.

Figura 18. Página de bienvenida de Jenkins



Fuente: Joan Caparrós.

En el caso de poseer una instalación previa de Jenkins o de no querer instalar la totalidad de los *plugins* sugeridos, se deberán instalar como mínimo los siguientes:

- **Maven Plugin:** Necesario para dotar a nuestro servidor de integración continua de las herramientas para construir y testar nuestro proyecto.

- **Git Plugin:** Permite la interacción con Git, por tanto, con plataformas como GitHub.

- **Mailer Plugin:** Permite configurar las notificaciones de correo electrónico que la plataforma enviará con los resultados de la compilación y testeo.

- **Workspace Cleanup Plugin:** Elimina el espacio de trabajo antes de la construcción asegurando que no haya ninguna interferencia de archivos entre construcciones.

Instalación del *plugin* CloudBees Docker Build and Publish en Jenkins

Construir una imagen docker en Jenkins requiere el uso de los binarios de docker. Hemos aprendido cómo construir la imagen durante la dockerización de la aplicación, y estos podrían ser perfectamente incluidos en una de las diferentes etapas que podríamos definir en un *pipeline* de nuestro proceso completo de automatización, pero Jenkins ofrece herramientas para facilitar la construcción y publicación de imágenes docker en un Registry Docker Hub, sin tener que escribir ninguna línea de código. En este caso de uso utilizaremos el *plugin* CloudBees Docker Build and Publish.

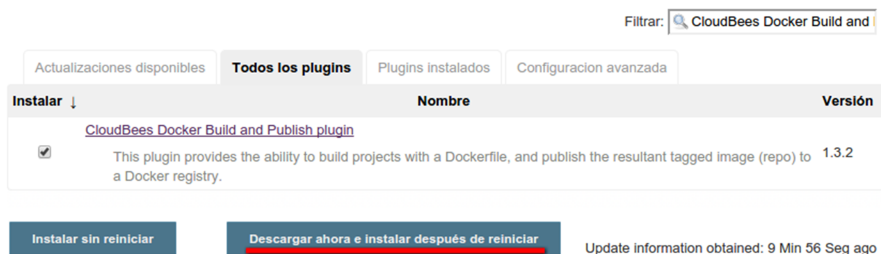
Para instalar el *plugin* nos dirigiremos utilizando los diferentes menús a *Administrar Jenkins -> Administrar Plugins*.

Figura 19. Detalle del menú de administración de Jenkins

Fuente: Joan Caparrós.

Dentro de la administración de *plugins* nos situaremos en la pestaña *Todos los plugins*, utilizaremos el filtro para indicar el *plugin* en cuestión, *CloudBees Docker Build and Publish*, marcaremos el *checkbox* de instalación y procederemos clicando sobre *Descargar ahora e instalar después de reiniciar*.

Figura 20. Detalle de la instalación y configuración del *plugin* CloudBees Docker Build and Publish



Fuente: Joan Caparrós.

4.6. Registry Docker Hub

Para completar el ciclo de nuestras imágenes docker, construiremos nuestro propio registro de imágenes de Docker. Así, seremos capaces de enviar nuestras aplicaciones que se encuentran en contenedores a la nube, teniendo un punto donde almacenar y compartir estas aplicaciones con otros usuarios.

Docker ofrece dentro de sus productos un punto centralizado de almacenamiento seguro de imágenes, que hasta el momento lo hemos usado para construir nuestras imágenes, Docker Hub.

Aunque Docker Hub nos permite disponer de solo un repositorio privado y debe ser considerado un servicio de pago para el almacenamiento de aplicaciones privadas, como sucede con GitHub, Docker cubre las necesidades de los desarrolladores entregando las herramientas necesarias para que cualquiera pueda construirse su propio repositorio privado en la nube.

4.6.1. Registro mediante imagen dockerizada

Podemos optar por la utilización de la imagen oficial dockerizada de Registry Docker Hub suministrada por Docker Hub. Este contenedor nos proporcionará el registro de nuestras propias imágenes generadas mediante docker.

Iniciaremos nuestro *registry*:

```
$ docker run -p 5000:5000 -v <ubicación_local>:/tmp/registry-dev registry
```

- **docker run:** Comando utilizado para la ejecución de contenedores a partir de una imagen.

- **-p 5000:5000:** Esta aplicación será accesible a través del puerto 5000, no dispone de interfaz gráfica web.

- **-v <ubicación_local>:/tmp/registry-dev:** Con el uso del *flag* «-v» dotamos de persistencia al contenedor ubicando de forma local las imágenes subidas a nuestra aplicación.

- **registry:** Nombre de la imagen del contenedor en Docker Hub.

Para probar y entender el funcionamiento del registro de imágenes de Docker, realizaremos una pequeña prueba donde descargaremos una imagen desde Docker Hub a nuestra máquina local para luego enviarla a nuestro propio Registry Docker Hub.

Descargamos la imagen hello-world https://hub.docker.com/_/hello-world/ a nuestra máquina:

```
#docker pull [OPCIONES] NOMBRE[:ETIQUETA|@DIGEST]
$ docker pull hello-world
```

- **docker pull:** Comando utilizado para la descarga de imágenes desde un registro de imágenes de Docker.

- **hello-world:** Nombre de la imagen del contenedor en Docker Hub.

Para que nuestra imagen local pueda ser subida a otro registro de imágenes, deberemos etiquetarla.

```
#docker tag IMAGEN[:TAG] IMAGEN[:ETIQUETA]
$ docker tag hello-world <my.registry.host>:5000/myhello-world
```

- **docker tag:** Comando utilizado para etiquetar aplicaciones en contenedores indicando el nuevo registro de imágenes, nombre y etiqueta del propio contenedor.

- **hello-world:** Nombre de la imagen del contenedor en Docker Hub.

- **<my.registry.host>:5000/myhello-world:** Asignación del nombre y registro de contenedores para nuestra imagen local hello-world.

Comprobamos que la imagen con su nueva etiqueta aparece dentro de nuestras imágenes locales.

```
$ docker images
REPOSITORY                                TAG              IMAGE ID
hello-world                              latest            c54a2cc56cbb
<my.registry.host>:5000/myhello-world    latest            c54a2cc56cbb
```

Si la imagen aparece, procederemos a subirla hacia nuestro propio registro de imágenes Docker.

```
#docker push [OPCIONES] NOMBRE[:ETIQUETA]
$ docker push <my.registry.host>:5000/myhello-world
The push refers to a repository [<my.registry.host>:5000/myhello-world]
a02596fdd012: Pushed
latest: digest: sha256:a18ed77532f6d6781500db650194e0f9396ba5f05f8b50d4046b294ae5f83aa4 size: 524
```

- **docker push:** Comando utilizado para la subida de imágenes hacia un registro de imágenes de Docker.

- **<my.registry.host>:5000/myhello-world:** Nombre y ubicación de la imagen del contenedor en Docker Hub.

Para asegurar que la imagen se encuentra dentro del registro de imágenes creado, verificaremos que esta se encuentre en el registro y procederemos a descargarla.

```
#Existen distintos comandos para hacer llamadas a la API del registro de imágenes Docker,
dependiendo de la versión de este estarán o no disponibles, es por eso que mostraremos dos
métodos para recuperar las imágenes remotas:
#Mediante instrucciones docker
$ docker search <my.registry.host>:[PUERTO]/library

#Mediante llamadas a la API
$ curl <my.registry.host>:5000/v2/_catalog
{"repositories":["myhello-world"]}
```

```
$ docker pull <my.registry.host>:5000/myhello-world
Using default tag: latest
latest: Pulling from myhello-world
Digest: sha256:a18ed77532f6d6781500db650194e0f9396ba5f05f8b50d4046b294ae5f83aa4
Status: Image is up to date for <my.registry.host>:5000/myhello-world:latest
```

4.6.2. Contribuir con nuestra imagen a la comunidad

Hemos hablado anteriormente de la plataforma Docker Hub Registry como un punto de almacenamiento de imágenes públicas de Docker listas para su uso. Detrás de todas estas imágenes hay un trabajo duro por parte de los desarrolladores para que nosotros podamos disfrutar de todos estos contenedores con tan solo invocar su ejecución.

Para compartir nuestro trabajo con la comunidad, podemos hacer accesible nuestras imágenes dentro de dicha plataforma mediante los siguientes pasos:

1) Visitar la página de Docker Hub.

2) Registrarnos dentro de la plataforma mediante el formulario de ***Sign Up*** y verificar la cuenta.

3) Entrar en la plataforma (***Log In***).

4) Crear nuestro repositorio, indicando nombre, descripción y visibilidad (pública o privada).

5) A partir de este momento tendremos disponible nuestro primer repositorio en la nube.

6) Para subir nuestro contenedor al repositorio creado en Docker Hub, deberemos autenticar nuestra máquina local mediante la instrucción:

```
$ docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID,
head over to https://hub.docker.com to create one.
Username: <usuario>
Password: <contraseña>
Login Succeeded
```

7) Etiquetamos nuestra imagen local *<imagen.local>*, para ser albergada dentro del repositorio con nombre *<usuario>/<nombre.repositorio>*.

```
#docker tag IMAGEN[:TAG] IMAGEN[:ETIQUETA]
$ docker tag <imagen.local> <usuario>/<nombre.repositorio>
```

8) Subimos la imagen al repositorio de Docker Hub:

```
#docker push [OPCIONES] NOMBRE[:ETIQUETA]
$ docker push <usuario>/<nombre.repositorio>
The push refers to a repository [docker.io/<usuario>/<nombre.repositorio>]
a02596fdd012: Mounted from library/<imagen.local>
latest: digest: sha256:a18ed77532f6d6781500db650194e0f9396ba5f05f8b50d4046b294ae5f83aa4 size:
```

En este momento nuestra imagen ya estará disponible para todos los usuarios de la comunidad Docker simplemente con ejecutar la respectiva instrucción *docker pull*.

Vale la pena pararse e investigar todas las opciones que la plataforma Docker Hub nos ofrece, ya que no se limita solo a albergar sino que permite la creación de organizaciones con sus respectivos equipos de trabajo y derivar la automatización de construcción de imágenes mediante cuentas GitHub o Bitbucket relacionadas.

4.7. Proceso de automatización completo (*pipeline*)

El proceso de generación de contenedores funcionales con nuestra aplicación será el objetivo principal de nuestro *pipeline*. Hasta el momento hemos realizado cada uno de los pasos de forma manual y faltaría dotar a nuestro proceso de creación de imágenes de una cierta automatización para realizar cada una de las acciones.

Jenkins aparece en nuestro esquema como plataforma encargada de la automatización y del monitoreo, pasando a ser nuestro **sistema de integración continua**, que se encargará de todas las tareas definidas en nuestro proceso de automatización.

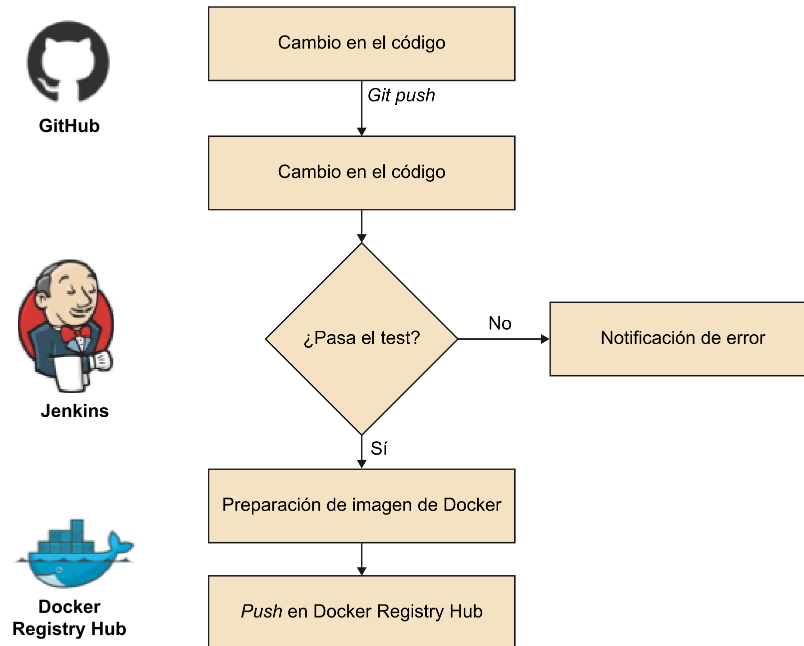
Para nuestro *pipeline* definido detectaremos las siguientes tareas:

- Compilar el código cada vez que este sufra un cambio.
- Realizar las pruebas unitarias definidas.
- Construir mediante Maven nuestra aplicación.
- Realizar la imagen del contenedor con nuestra aplicación.

- Registrar la imagen generada en nuestro registro de imágenes de Docker.

- Avisar a los desarrolladores, en el caso de que alguno de los pasos establecidos no finalice correctamente, para efectuar los cambios en el código que sean necesarios.

Figura 21. Tareas definidas en la Jenkins *pipeline*

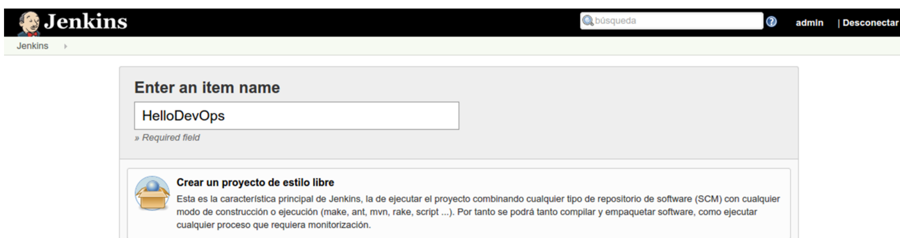


Fuente: Joan Caparrós.

Para iniciar nuestro proyecto en Jenkins, se deberá crear una **Nueva Tarea**, disponible en el menú lateral derecho de la plataforma. En el caso de no disponer de ninguna tarea anterior, Jenkins indicará en su parte central un enlace directo para empezar con la creación de la primera tarea.

Definiremos un nombre de proyecto interno para la plataforma (*"HelloDevOps"*) y elegiremos el método de *Crear un proyecto de estilo libre*, ideal para la ejecución de proyectos provinientes de cualquier tipo de repositorio de software, con cualquier método de construcción o ejecución (make, ant, mvn, scripts...).

Figura 22. Creación de la tarea *HelloDevOps*



Fuente: Joan Caparrós.

Una vez dentro de la pantalla de configuración del proceso de automatización definiremos las propiedades **Generales**, que definirán aquellas propiedades transversales a todas las operaciones realizadas. Para nuestro ejemplo dejaremos todos estos campos en blanco.

A continuación definiremos la configuración para cada una de las acciones descritas durante la definición del *pipeline*.

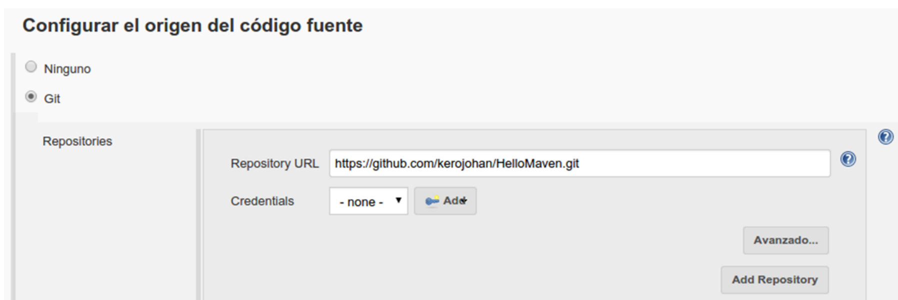
4.7.1. Compilar el código cada vez que este sufra un cambio

Dado que nuestro proyecto debe generar una nueva imagen cada vez que se incorporen cambios a la rama principal (*master*) de nuestro proyecto, deberemos establecer nuestro repositorio Git como fuente de origen.

Configurar el origen del código fuente

- Seleccionar **Git**.
- Introducir en **Repository URL** https://github.com/<usuario>/HelloDevOps.git manteniendo las credenciales en **none** (si vuestro repositorio es público).

Figura 23. Configuración de la tarea HelloDevOps



Fuente: Joan Caparrós.

Existen diferentes acciones que pueden hacer que nuestro proyecto Jenkins se ejecute de manera automática. Estos representan los **Disparadores de ejecuciones**, y son los siguientes:

- **Lanzar ejecuciones remotas:** Permite la ejecución del *pipeline* accediendo a una URL especial (JENKINS_URL/job/HelloDevOps/build?token=TOKEN_NAME o /buildWithParameters?token=TOKEN_NAME), útil para integrar dentro de *scripts*. Un uso habitual es la llamada de esta URL desde los *scripts* (*hooks*) ejecutados cuando se realizan cambios dentro de un repositorio de versiones.
- **Construir después de la construcción de otros proyectos:** Permite definir como disparadores de ejecución la finalización de construcción de otros proyectos.

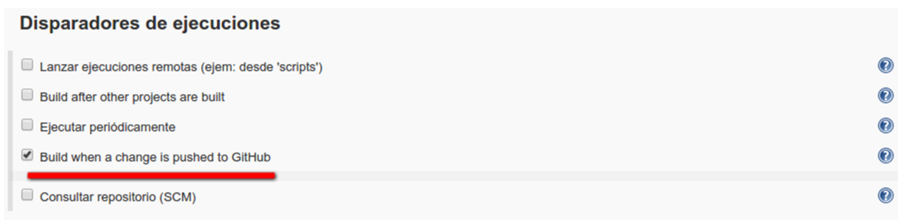
- **Ejecutar periódicamente:** Permite la definición de tiempos de ejecución del proyecto definidos en formato cron con algunas pequeñas adaptaciones (ejemplo: H/15 * * * * - dispara la ejecución del *pipeline* cada 15 minutos).

- **Construir cuando un cambio es subido (*pushed*) al GitHub:** Permite relacionar la ejecución del proyecto Jenkins a cada comando PUSH recibido por el repositorio de versiones GitHub definido en la sección «Configurar el origen del código fuente».

- **Consultar repositorio (SCM):** Esta opción complementa a la anterior y permite la ejecución de comprobaciones del repositorio de versiones en el caso de que no se disponga de *scripts* capaces de disparar ejecuciones. Se definen en el formato cron adaptado de Jenkins.

Para nuestro proyecto estableceremos la opción **Construir cuando un cambio es subido (*pushed*) al GitHub**.

Figura 24. Configuración de la tarea HelloDevOps



Fuente: Joan Caparrós.

Realizar las pruebas unitarias definidas

El lanzamiento de las pruebas unitarias define un proceso de ejecución, que lanzará un comando maven con parámetro «test» (mvn test). Los resultados de la ejecución definirán si el proceso continúa y esto indicará que nuestro proyecto funciona adecuadamente.

Ejecutar:

- En el seleccionable elegir **Ejecutar tareas 'maven' de nivel superior**.

- **Versión de Maven**: Elegiremos mediante el listado desplegable la versión de Maven definida en el apartado Instalación Maven en Jenkins de este caso de uso.

- **Goals**: test.

Figura 25. Configuración de la tarea HelloDevOps

Fuente: Joan Caparrós.

Construir mediante Maven nuestra aplicación

El proceso de construcción de nuestro proyecto es indispensable para la generación del contenedor WAR en la carpeta target, que alimentará las siguientes fases de nuestro *pipeline*. De modo similar a la ejecución de las pruebas unitarias, lanzarán un proceso maven con parámetro «package».

Ejecutar:

- En el seleccionable elegir **Ejecutar tareas 'maven' de nivel superior**.
- **Versión de Maven**: Elegiremos mediante el listado desplegable la versión de Maven definida en el apartado Instalación Maven en Jenkins de este caso de uso.
- **Goals**: package

Figura 26. Configuración de la tarea HelloDevOps

Fuente: Joan Caparrós.

4.7.2. Construcción y publicación de la imagen

Realizar la imagen del contenedor y posterior registro en cualquier registro de imágenes de Docker será una tarea fácil mediante el *plugin* instalado **Docker Build and Publish**. Este ejecutará todos los comandos docker indicados en el apartado de *Dockerización y Registry Docker Hub* simplemente indicando el repositorio destino y la URL del repositorio de imágenes docker.

Para la subida en nuestro repositorio de imágenes Docker, hemos de ejecutar:

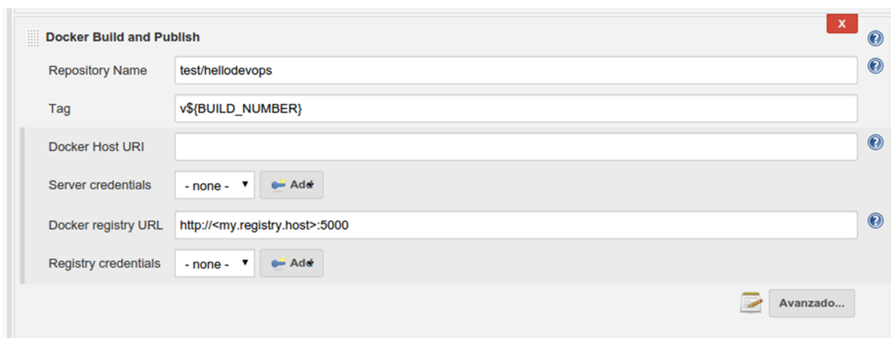
- En el seleccionable elegir **Docker Build and Publish**.
- **Repository Name:** «test/hellodevops», definiendo un registro «test» donde se ubicará nuestra imagen con nombre «hellodevops».

- **Tag:** La etiqueta de nuestra imagen definirá la versión del contenedor; para ello concatenamos «v» con una variable interna con la versión actual de construcción de Jenkins (${BUILD_NUMBER}).

- **Docker Host URI:** Puede dejarse en blanco para utilizar los valores de entorno docker por defecto (normalmente unix:///var/run/docker.sock o tcp://127.0.0.1:2375).

- **Docker registry URL:** http://<my.registry.host>:5000/.

Figura 27. Configuración de la tarea HelloDevOps



Fuente: Joan Caparrós.

Para la subida en la nube (Docker Hub), hemos de ejecutar:

- En el seleccionable elegir **DockerBuild and Publish**.

- **Repository Name:** «<usuario.docker.hub>/hellodevops», definiendo el repositorio remoto que previamente habremos creado.

- **Tag:** La etiqueta de nuestra imagen definirá la versión del contenedor; para ello concatenamos «v» con una variable interna con la versión actual de construcción de Jenkins (${BUILD_NUMBER}).

- **Docker Host URI:** Dejaremos los valores en blanco para el uso de las variables por defecto de la plataforma.

- **Docker registry URL:** Podemos dejar el campo en blanco, ya que por defecto este contendrá el valor que nos interesa https://index.docker.io/v1/.

- **Registry credentials:** Indicaremos un nuevo registro con las credenciales de nuestro usuario dentro de la plataforma Docker Hub.

Figura 28. Configuración de la tarea HelloDevOps

Fuente: Joan Caparrós.

Una vez compilado el proyecto y publicada la imagen en el registro de imágenes, liberaremos el espacio mediante la definición de un último proceso de ejecución.

Acciones para ejecutar después: en el seleccionable hemos de elegir **Delete workspace when build is done**.

Figura 29. Configuración de la tarea HelloDevOps

Fuente: Joan Caparrós.

Esto eliminará la carpeta con el proyecto al terminar el proceso.

4.7.3. Notificación de errores durante el proceso de automatización

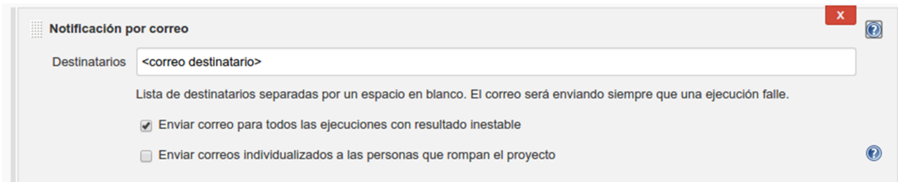
Si se ha configurado, Jenkins enviará un correo electrónico a los destinatarios especificados cuando se produzca un determinado evento importante.

Es importante denotar que cada construcción fallida enviará un nuevo correo electrónico y que también se enviarán en casos de superación de problemas tras una construcción fallida y construcciones inestables.

Acciones para ejecutar después:

- En el seleccionable, elegir **Notificación por correo**.
- **Destinatarios**: Permite especificar múltiples destinatarios a los cuales se les enviará la copia completa de la salida por consola de procesos fallidos. Las direcciones de correo serán separadas por un espacio en blanco.
- Se deberá especificar qué usuarios deberán recibir los errores, o todos los desarrolladores o aquellos que hayan realizado el último commit.

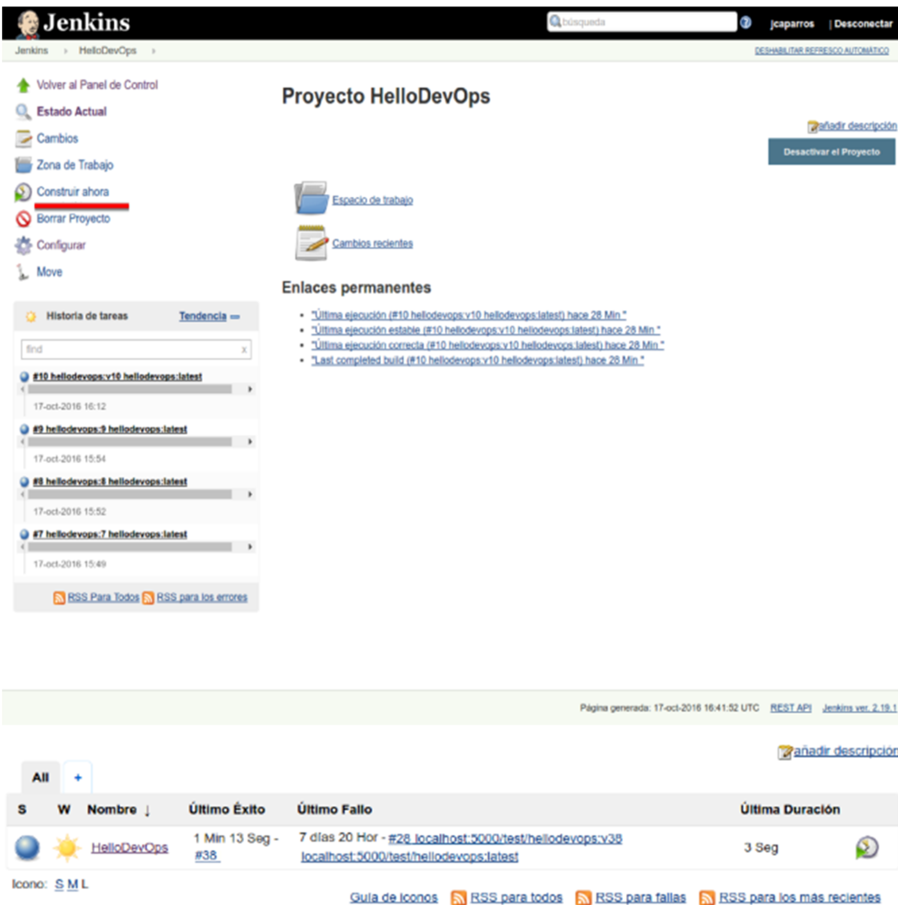
Figura 30. Configuración de la tarea HelloDevOps



Fuente: Joan Caparrós.

Para ejecutar por primera vez nuestro proceso de integración continua, podremos o bien introducir algún cambio en nuestro código y hacer *push* en el repositorio de versiones, o bien clicar en la opción del menú lateral *Construir ahora*.

Figura 31. Ejecución de la tarea HelloDevOps



Fuente: Joan Caparrós.

¡Ya tenemos nuestro servidor de integración continua preparado y funcionando!