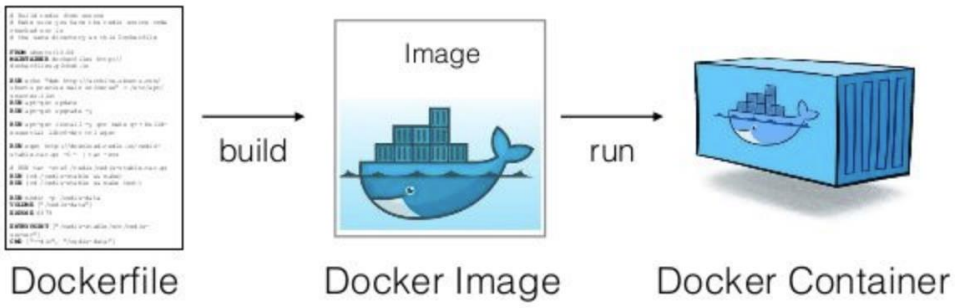
Utilización de Docker y Bash para el despliegue de aplicaciones web.

1. Introducción

La utilización de contenedores de Docker es una buena práctica para agilizar el proceso de despliegue de aplicaciones web, ya sea en los servidores de "*qa"*, "*staging"* o "*production"*; porque que nos permiten tratar a la infraestructura como código.

La conteinerización de una aplicación o servicio se puede dividir en tres partes:

1- Escritura del Dockerfile donde se definen los pasos de construcción y ejecución de un servicio con todas sus dependencias incluidas.
2- Constricción (build) de la imagen a ejecutarse.
3- Ejecución (run) de la imagen que deriva en la creación de un contenedor.



Creación de contenedores de Docker.

2. Alcance

Este procedimiento puede ser aplicado a la mayoría de los proyectos de construcción de software realizados en la empresa; dado que es común la utilización de VPC (virtual private computer) y servidores de Linux para el hosting de las aplicaciones web desarrolladas.

3. Objetivo

El objetivo principal de este procedimiento es agilizar el proceso de despliegue de aplicaciones web. Mediante la utilización de Docker se reduce drásticamente las configuraciones de los servidores y se asegura de que la aplicación se ejecute de igual forma en cualquier servidor utilizado, dado que no es necesario instalar todas las dependencias (programas y lenguajes) de la aplicación a desplegarse en el sistema operativo host, como podrían ser Git, Nginx, Node.js, C# SDK, Python, PostgreSQL, MySql. Solamente mediante la instalación de Docker (ej: https://docs.docker.com/engine/install/ubuntu/) el servidor ya se encuentra listo para servir a la aplicación y desplegar los cambios instantáneamente.

4. Roles y responsabilidades

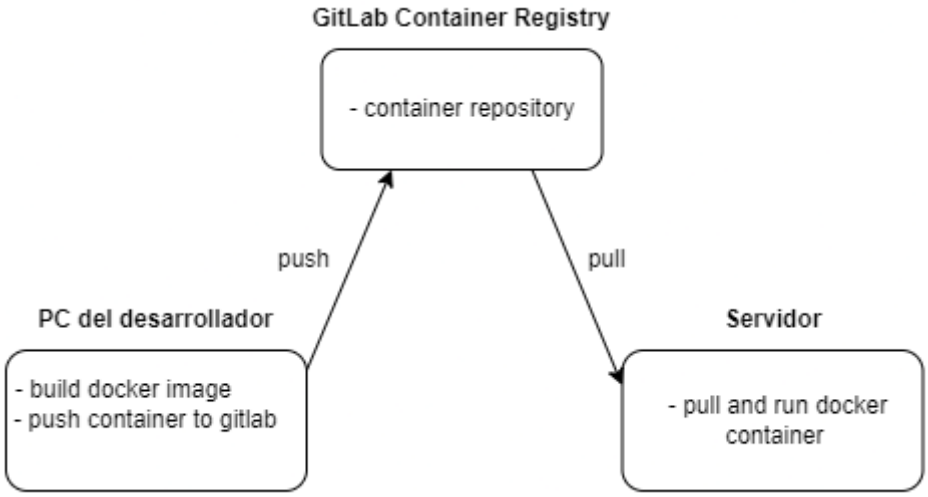
Desarrollador(es) de la aplicación.	● Desarrollo de la aplicación (Backend, Frontend, Database) ● Creación del Dockerfile perteneciente a cada servicio (los servicios pueden ser Backend, Frontend, Database, Web-Server/Proxy) ● Construcción (build) de la imagen de cada servicio y carga (push) al Docker Registry
Encargado del despliegue de la aplicación.	● Puede ser el mismo desarrollador. ● Configuración del servidor (elegir el sistema operativo, instalar docker, crear reglas de firewall, etc.). ● Creación de proyecto en el Docker Registry donde se subirán las imágenes del proyecto. ● Creación del archivo docker-compose.yml utilizado para el despliegue.

5. Actividades

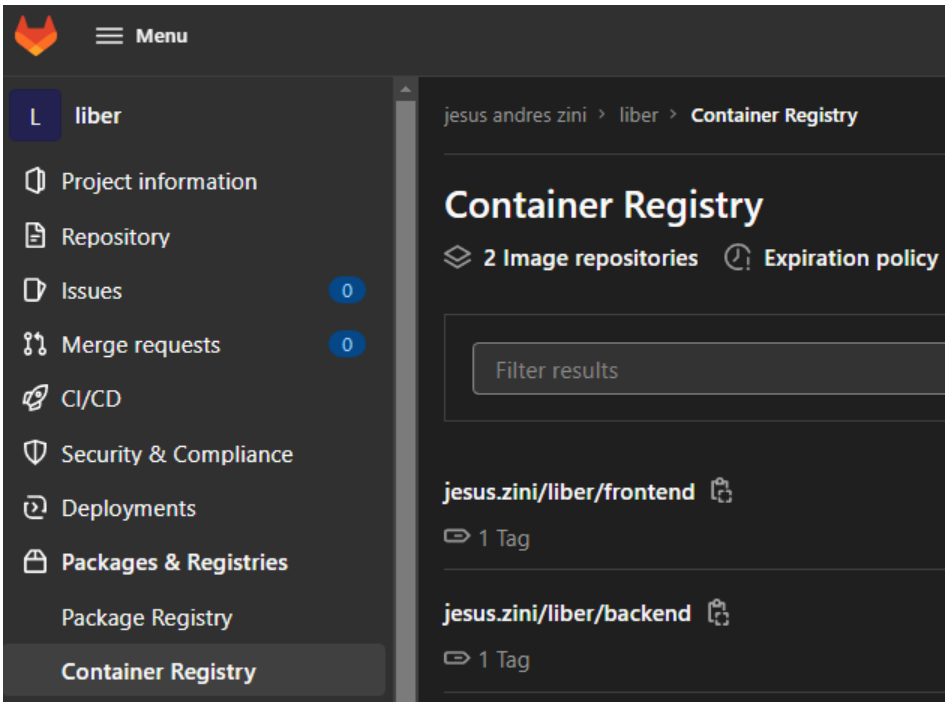
Las actividades del procedimiento son las siguientes: (Se utiliza como ejemplo un proyecto que ya siguió este proceso de despliegue).

5.1 Creación de proyecto en el Container Registry

Necesitamos de un Container Registry donde alojar los contenedores de nuestro proyecto para luego descargarlos desde el servidor y ejecutarlo. El repositorio de contenedores de Docker más utilizado es Docker Hub (container registry oficial de Docker). Una alternativa gratuita podría ser GitLab, además de ser un repositorio para código fuente, esta plataforma también cuenta con un registro de contenedores de Docker; y no hace falta usarlo también como repositorio del código fuente de nuestro proyecto. Por lo tanto, podemos usar por ejemplo GitHub o Bitbucket como repositorio del código del proyecto, y GitLab como repositorio de los contenedores.



Container flow.



Panel del Container Registry de GitLab.

5.2 Creación de archivos Dockerfile

Para cada servicio que compone nuestra aplicación, se debe crear un archivo Dockerfile, el cual va a ser utilizado para construir la imagen de ese servicio específico (back, front, proxy, etc.) y luego hacer un push de esas imágenes al docker registry.

Conjunto de Dockerfiles que compone el proyecto de ejemplo:

```
# Nginx
FROM nginx:1.21-alpine

COPY ./nginx.conf /etc/nginx/conf.d/default.conf
```

Reverse-Proxy Dockerfile.

```
# Node
FROM node:14-alpine

RUN mkdir -p /usr/src/app

WORKDIR /usr/src/app

COPY package*.json ./

RUN npm install

COPY . .

EXPOSE 4000

CMD ["npm", "start"]
```

Backend Dockerfile.

```
# React
FROM node:14-alpine as build

RUN mkdir -p /usr/src/app

WORKDIR /usr/src/app

COPY package*.json ./

RUN npm install

COPY . .

RUN npm run build

# Nginx
FROM nginx:1.21-alpine

COPY ./nginx.conf /etc/nginx/conf.d/default.conf

COPY --from=build /usr/src/app/build /usr/share/nginx/html

EXPOSE 3000
```

Frontend Dockerfile.

No se crea un archivo Dockerfile para la base de datos ya que se hace un pull de la imagen oficial disponibles en el Docker Hub, de la base de datos a utilizarse (en este caso PostgreSQL). En el archivo *docker-compose.yml* se especifica qué imagen se va a utilizar y se configuran los volúmenes para que los datos creados en el contenedor, persistan también en el servidor (favorablemente también se debería configurar un *cronjob* que haga backup de la base de datos cada cierto periodo de tiempo para prevención de contingencias).

5.3 Push de contenedores al Docker Registry

Nos posicionamos en el directorio de cada Dockerfile, construimos una imagen y hacemos push de la misma al Docker Registry, en este caso al de GitLab.

Se pueden ejecutar los comandos a mano o se pueden crear Bash Scripts:

```
#!/bin/sh

REPO_NAME=liber
IMG_NAME=liber-front
IMG_PATH=registry.gitlab.com/jesus.zini/liber/front:latest

# start...
cd ../${IMG_NAME}

# docker image prune -af
docker build -t="${IMG_NAME}" .

# get just created image ID
IMG_ID=$(docker images --format "{{.ID}} {{.Repository}}" | grep ${IMG_NAME} | awk '{ print $1 }')

docker tag ${IMG_ID} ${IMG_PATH}

docker push ${IMG_PATH}

cd ../${REPO_NAME}
```

docker-push-back.sh.

Se utiliza un script similar al anterior para cada imagen que se debe subir al registry (en este caso, uno más para el Backend).

5.4 Creación de archivos docker-compose.yml

Este archivo YAML es el encargado de levantar cada servicio que conformar la aplicación y de crear una network privada y aislada de su entorno para que los contenedores se comuniquen entre sí.

```yaml
version: '3'

services:
  db:
    container_name: liber_db
    image: postgres:12-alpine
    restart: always
    environment:
      POSTGRES_PASSWORD: ${DB_PASSWORD}
      POSTGRES_USER: ${DB_USER}
      POSTGRES_DB: ${DB_DATABASE}
    volumes:
      - ../../pgdata:/var/lib/postgresql/data
    ports:
      - '5555:5432'
    expose:
      - '5555'
  api:
    container_name: liber_api
    restart: always
    image: registry.gitlab.com/jesus.zini/liber/back:latest
    depends_on:
      - db
  app:
    container_name: liber_app
    restart: always
    image: registry.gitlab.com/jesus.zini/liber/front:latest
    depends_on:
      - api
  nginx:
    container_name: liber_nginx
    restart: always
    build: ../nginx
    ports:
      - '8081:80'
    depends_on:
      - api
      - app
```

docker-compose.yml.

5.5 Despliegue de la aplicación

El transporte del archivo docker-compose.yml que vamos a usar para el despliegue de la aplicación puede hacerse de muchas formas. Puede simplemente crease el archivo en el servidor de forma remota, o mediante la utilización de git, crear un repositorio donde se alojarán nuestros scripts, configuraciones del web-server/proxy, archivos docker-compose y todo lo relacionado a las tareas de operaciones en el servidor. Esta segunda alternativa sería la más favorable.

Una vez posicionado en el directorio donde se encuentra el archivo dokcer-compose.yml se puede poner en marcha la aplicación ejecutando "*docker-compose up -d*"

El comando "*up"* levanta todos los contendedores y la red privada donde conviven. Y la flag "*-d"* sirve para especificar que sea en "detached mode" (que los contenedores se ejecuten como background processes).

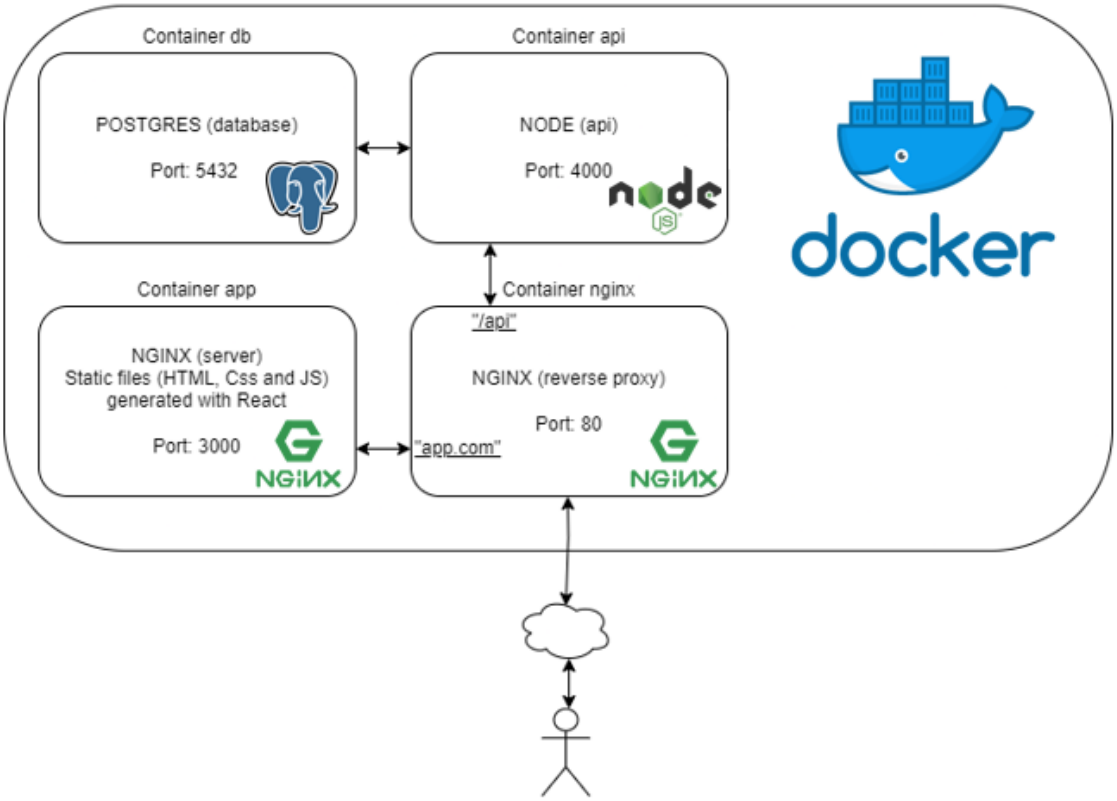
También se puede utilizar un script para facilitar aún más la tarea:

```
#!/bin/sh

echo "Stopping containers..."
sudo docker-compose down

echo "About to pull containers"
sudo docker-compose pull

echo "About to start containers"
sudo docker-compose up -d
```

docker-pull-and-upt.sh.

6. Resultado

Como resultado se obtiene la aplicación desplegada con éxito, un servidor limpio y fácil de administrar (ya que no tuvimos que instalar dependencias de la aplicación ni preocuparnos por compatibilidad de versiones, lenguajes, programas, etc.), además de un uso eficiente de recursos ya que en el mismo servidor pueden coexistir múltiples aplicaciones sin interferir entre ellas.



Arquitectura final de aplicación web conteinerizada y desplegada.