

**ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA**

**GRADO EN GESTION DEL SOFTWARE**

**Curso Académico 20\_\_/20\_\_**

**Trabajo Fin de Grado**

**TÍTULO DEL TRABAJO FIN DE GRADO**

**Autor**: Helena Esteban Illescas

**Normativa TFG ETSII**

Aprobada por Junta de Escuela el 7 de Noviembre de 2017

Aprobada por Junta de Escuela el 27 de Abril de 2017

Modificado en la Comisión de Ordenación Docente el 27 de Enero de 2017

Aprobada por Junta de Escuela el 15 de Diciembre de 2016

**Directores/Tutor**: Carlos Enrique Cuesta

AGRADECIMIENTOS

Quiero agradecer a mi tutor, Carlos Enrique Cuesta, no sólo haberme permitido realizar este Trabajo de Fin de Grado por todo el apoyo que he recibido por su parte y darme la idea para hacer el proyecto sobre Docker y su disponibilidad para ayudarme a afrontar los problemas que hemos ido encontrando.

A esa persona que en todo momento me ha proporcionado incontables apoyos en las circunstancias más adversas.

Por supuesto, agradecimiento a toda mi familia, también por su apoyo y por el ánimo que me han dado y siempre me han animado y estimulado a conseguir las metas que me he propuesto a lo largo mi vida.

A mis compañeros y amigos más cercanos, sin los cuales todo esto no hubiese sido lo mismo.

RESUMEN

Este proyecto analiza el despliegue de aplicaciones usando la tecnología Docker y Docker Swarm. Para ello se ha realizado un breve estudio de las tecnologías actuales que permite el despliegue de aplicación, quedándonos con Docker.

La elección final ha sido Docker por su sencillez como se mostrará a lo largo del presente documento.

Para demostrar cómo realizar el despliegue se ha usado una aplicación obtenida de GitHub “Voting Application”. La he modificado ligeramente para demostrar la sencillez en el despliegue de nuevas aplicaciones.

**ABSTRACT**

This project analyzes the deployment of applications using Docker and Docker Swarm technology. I have done a brief study of the current technologies that allows the deployment of applications, staying with Docker.

The final choice has been Docker for its simplicity as will be showed throughout this document.

To demonstrate how to deploy applications. I have used an application obtained from GitHub "Voting Application". I have modified it slightly to demonstrate the simplicity of its use in the deployment of new applications.

ÍNDICE GENERAL

[1 Introducción 7](#_Toc498857502)

[2 OBJETIVOS 8](#_Toc498857503)

[2.1 Objetivos generales 8](#_Toc498857504)

[2.2 Objetivos específicos 8](#_Toc498857505)

[3 Estado del Arte 9](#_Toc498857506)

[3.1 Que es Docker 9](#_Toc498857507)

[3.2 Que es Máquina Virtual Contenedor vs Maquina Virtual 9](#_Toc498857508)

[4 Plataforma Docker 10](#_Toc498857509)

[4.1 Arquitectura 10](#_Toc498857510)

[4.2 Otros componentes Docker 11](#_Toc498857511)

[4.3 Instalación de Docker 12](#_Toc498857512)

[4.4 Instalación de Docker-Machine 13](#_Toc498857513)

[4.5 Instalación de Docker Compose 14](#_Toc498857514)

[5 Docker en el entorno de Desarrollo y Pruebas 16](#_Toc498857515)

[5.1 Imágenes 17](#_Toc498857516)

[5.1.1 Dockerfile 17](#_Toc498857517)

[5.1.2 Docker Build 19](#_Toc498857518)

[5.1.3 Docker Push 23](#_Toc498857519)

[5.1.4 Docker-Compose 23](#_Toc498857520)

[6 Docker en el Entorno de Producción 27](#_Toc498857521)

[6.1 Docker Swarm 27](#_Toc498857522)

[6.1.1 Nodos 27](#_Toc498857523)

[6.1.2 Servicios y Tareas 28](#_Toc498857524)

[6.2 Creación del Docker Swarm 28](#_Toc498857525)

[7 Experimentos y Validación 34](#_Toc498857526)

[8 Conclusiones 36](#_Toc498857527)

[8.1 Trabajo futuro 36](#_Toc498857534)

[9 REFERENCIAS 37](#_Toc498857535)

ÍNDICE DE FIGURAS

[Figura 3‑1: MotorDocker 10](file:////Users/gillescas/Documents/PFG-Helena/Helena_TFG.v0.5.docx#_Toc498857536)

[Figura 3‑2: Arquitectura Docker 10](#_Toc498857537)

[Figura 3‑3: Comprobación instalación Docker 12](#_Toc498857538)

[Figura 3‑4: Ver versión Docker 12](#_Toc498857539)

[Figura 3‑5: Fichero Dockerfile para creación de imagen vote 13](#_Toc498857540)

[Figura 3‑6: Docker build vote-he:1.0 13](#_Toc498857541)

[Figura 3‑7: Contenedores Docker en ejecución 13](#_Toc498857542)

[Figura 3‑8: Versión de Docker-machine instalada 14](#_Toc498857543)

[Figura 3‑9: Versión docker-compose 14](#_Toc498857544)

[Figura 3‑10: Fichero doocker-compose-vote.yml 15](#_Toc498857545)

[Figura 3‑11: Ejecución docker-compose 15](#_Toc498857546)

[Figura 3‑12: Página web para votación 15](#_Toc498857547)

[Figura 4‑1: arquitectura de la aplicación – Obtenida de Github 16](#_Toc498857548)

[Figura 4‑2: Dockerfile para crear imagen worker\_he 18](#_Toc498857549)

[Figura 4‑3: Dockerfile para crear imagen result\_he 19](#_Toc498857550)

[Figura 4‑4: script para creación de imágenes 19](#_Toc498857551)

[Figura 4‑5: Creación imagen vote\_he:2.0 20](#_Toc498857552)

[Figura 4‑6: Página web – aplicación vote\_he:2.0 21](#_Toc498857553)

[Figura 4‑7: Creación imagen worker\_he:1.0 – parte1 21](#_Toc498857554)

[Figura 4‑8: Creación imagen worker:1.0 – parte2 22](#_Toc498857555)

[Figura 4‑9: Creación imagen result:2.0 22](#_Toc498857556)

[Figura 4‑10: Página web de result:1.0 y página web de result:2.0 23](#_Toc498857557)

[Figura 4‑11: Docker tag y docker push al repositorio “docker hub” 23](#_Toc498857558)

[Figura 4‑12: Ficheros docker-compose 24](#_Toc498857559)

[Figura 4‑13: Aplicación Original - No hay votos 25](#_Toc498857560)

[Figura 4‑14: Aplicación Original - 1 voto emitido 25](#_Toc498857561)

[Figura 4‑15: Aplicación Modificada - 5 votos emitidos 26](#_Toc498857562)

[Figura 5‑1: Creación de host-1 29](#_Toc498857563)

[Figura 5‑2: Lista de hosts 29](#_Toc498857564)

[Figura 5‑3: docker swarm init 29](#_Toc498857565)

[Figura 5‑4: añadir worker al docker swarm 29](#_Toc498857566)

[Figura 5‑5: token para añadir un segundo master 30](#_Toc498857567)

[Figura 5‑6: añadir segundo master 30](#_Toc498857568)

[Figura 5‑7: docker swarm 30](#_Toc498857569)

[Figura 5‑8: fichero docker-compose para Swarm 31](#_Toc498857570)

[Figura 5‑9: Copiar fichero compose a host-1 31](#_Toc498857571)

[Figura 5‑10: docker stack deploy 32](#_Toc498857572)

[Figura 5‑11: estado del despliegue de servicios 32](#_Toc498857573)

[Figura 5‑12: visualizer 32](#_Toc498857574)

# Introducción

El uso de entornos virtualizados está tomando cada vez más importancia, Docker es una de las tecnologías que permite virtualización

Existen varios tipos de virtualización, uno de ellos es la virtualización a nivel de sistema operativo, y uno de los proyectos que más destaca en este sentido es **Docker,**junto con otros proyectos open-source como OpenVZ, LXC/LXD, Linux-VServer, etc….

Otro tipo de virtualización es la virtualización a nivel hardware, como VMware, VirtualBox, etc… que permiten a la máquina física subyacente multiplicarse entre varias máquinas virtuales, cada una ejecutando su propio sistema operativo. A la capa de software que permite la virtualización se la llama monitor de máquina virtual o **hypervisor**.

# OBJETIVOS

## Objetivos generales

Considero que Docker es una tecnología en auge y estoy interesada en su estudio y aprendizaje, pensando que me puede servir profesionalmente para el desarrollo de mi carrera profesional.

## Objetivos específicos

1. Análisis y estudio de la plataforma Docker
2. Consolidar los conocimientos adquiridos durante el estudio de docker. Para este fin una vez analizado y entendido los conceptos Docker (contenedores, imágenes, Dockerfiles, etc.) descargue de la web una aplicación que cumplía los requisitos para poder poner en práctica los conocimientos adquiridos y demostrar la utilidad de Docker en el entorno de empresas de software.
3. Creación de imágenes usando el comando *docker build* y uso de ños ficheros Dockerfiles incluidos. *Esto* me permitió comprobar los conocimientos adquiridos sobre estos dos conceptos.
4. Despliegue de aplicación con *docker-compose*, que me permitió comprobar el despliegue de la aplicación completa.
5. Creación de hosts virtuales con *docker-machine* y configurarlos en cluster con *docker swarm*

# Estado del Arte

## Que es Docker

De acuerdo a la definición de la página oficial de Docker, es una plataforma abierta para desarrollar, enviar y ejecutar aplicaciones.

La plataforma Docker proporciona la capacidad de empaquetar y ejecutar una aplicación en un entorno suelto aislado llamado un contenedor.

Un contenedor es un paquete con todo lo necesario para que una o varias aplicaciones se puedan ejecutar, es decir, similar a una máquina virtual, pero más ligero al depender el host para la ejecución.

## Contenedor vs Máquina Virtual

**Máquina Virtual**

Una máquina virtual consisten en un hypervisor que sobre un hardware físico permite emular varias máquinas virtuales cada uno con su propio sistema operativo, librerías y aplicaciones, con esto en un servidor físico se pueden tener varios sistemas operativos y aplicaciones, consolidando y reduciendo costos, con un aprovisionamiento más rápido y una mejor recuperación del entorno en caso de que ocurra una catástrofe que inhabilite la sede en la que está ubicada la aplicación, como puede ser un terremoto.

Una de las ventajas de las máquinas virtuales es la seguridad que provee la abstracción a nivel de hardware ya que un ataque puede llegar únicamente a afectar la máquina virtual comprometida, aislando las demás que viven en el mismo hardware.

Uno de los inconvenientes de las máquinas virtuales es que agregan gran complejidad al sistema en tiempo de ejecución. Esto tiene como efecto la ralentización del sistema, es decir, el programa no alcanzará la misma velocidad de ejecución que si se instalase directamente en el sistema operativo "anfitrión" (host) o directamente sobre la plataforma de hardware. Sin embargo, a menudo la flexibilidad que ofrecen compensa esta pérdida de eficiencia.

**Contenedor**

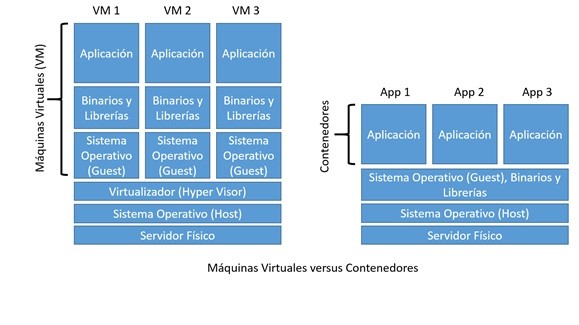
Los contenedores son una tecnología que busca lo mismo, pero con la diferencia que además del servidor físico, se tiene un sistema operativo host compartido para los contenedores. Gracias a esto los contenedores son de menor tamaño, haciéndolos eficientes, más fácil de migrar, iniciar, recuperar y mover entre nube pública y/o privada entre proveedores cloud.

Los contenedores ofrecen una gran portabilidad de aplicaciones que permite pasar entre ambientes de desarrollo, test y producción fácilmente a servidores locales o en la nube.

Si comparamos los contenedores con las máquinas virtuales se podría que los contenedores tienen mucho más potencial que las máquinas virtuales.

* Rapidez: Los contenedores pueden ser más rápidos y consumirán menos recursos que las máquinas virtuales. Una máquina virtual puede tardar hasta varios minutos en crearse y ponerse en marcha mientras que un contenedor puede ser creado y lanzado en unos pocos segundos.
* Portabilidad:
* Seguridad: Una de las ventajas de las máquinas virtuales es la abstracción a nivel de hardware físico que se traduce en kernels individuales, que limitan la superficie de ataque al hipervisor. Los contenedores también ofrecen grupos de control de acceso al demonio que controla la virtualización, restringiéndolo así sobre posibles cambios que puedan hacerse al sistema.

Si nos surge la necesidad de decidir entre usar contenedores o máquinas virtuales hay que fijarse en el alcance de nuestro trabajo, es decir, si queremos múltiples copias de una misma aplicación, debemos usar contenedores. Si queremos flexibilidad o ejecutar múltiples aplicaciones debemos utilizar máquinas virtuales.



Insertar titulo. Sacado de la página oficial de Docker

# Plataforma Docker

Docker permite empaquetar y ejecutar una aplicación en un entorno llamado contenedor. Los contenedores son ligeros porque no necesitan la carga adicional de un hipervisor, sino que se ejecutan directamente dentro del núcleo del servidor. Esto implica que se puedan ejecutar más contenedores en un hardware que si se utilizan máquinas virtuales.

El motor de docker es una aplicación cliente-servidor que se compone principalmente de:

Figura ‑: Motor Docker

Figura ‑: MotorDocker

* Un servidor que es un tipo de programa de larga ejecución llamado proceso daemon (el comando dockerd).
* Una API REST que especifica interfaces que los programas pueden usar para hablar con el demonio Docker e instruirlo qué hacer.
* Un cliente de interfaz de línea de comandos (CLI) (el comando docker).

## Arquitectura



Figura ‑: Arquitectura Docker

Los componentes de la arquitectura docker son:

* Demonio Docker, escucha las solicitudes de la API de Docker y gestiona los objetos de Docker, como imágenes, contenedores, redes y volúmenes. Un demonio Docker también puede comunicarse con otros demonios Docker para administrar los servicios de Docker.
* El cliente Docker es la forma principal de interacción con Docker. el cliente envía comandos (por ejemplo “docker build”) al demonio Docker, que los ejecuta. El comando docker usa la API Docker y además puede comunicarse con más de un deminio.
* El registro de Docker almacena imágenes de Docker. Docker Hub y Docker Cloud son registros públicos y Docker está configurado para buscar imágenes en Docker Hub de manera predeterminada.

Cuando utiliza los comandos de extracción, las imágenes necesarias se extraen de su registro configurado. Cuando creamos una imagen y utilizamos el comando docker push, nuestra imagen se envía a su registro configurado.

* Objetos Docker
* **Imagen,** es una plantilla de solo lectura, con las instrucciones para crear contenedores. Las imágenes se pueden actualizar fácilmente y se pueden compartir públicamente.
* DockerFile, es un archivo de configuración que se utiliza para crear imágenes. En este archivo se indica que va a contener la imagen y los distintos comandos para instalar las herramientas.
* **Contenedor** es una instancia en ejecución de una imagen. Son los que ejecutan la aplicación. A partir de una única imagen, se pueden ejecutar varios contenedores.

El contenedor alberga todo lo necesario para que una aplicación pueda ejecutarse de forma aislada.

En un entorno de máquinas virtuales, tenemos máquinas individuales donde se ejecuta un sistema operativo completo, con las librerías necesarias y el software que ejecuta las aplicaciones. Con Docker, el “sistema operativo invitado” se convierte en el “Hypervisor”, cada contenedor comparte recursos con el OS anfitrión, en particular, el núcleo de Linux. Este último punto es importante, ya que permite utilizar contenedores que usen distintos sistemas operativos, proporcionando gran flexibilidad. Así que podríamos tener un Ubuntu con SO anfitrión que contuviese contenedores donde corriese CentOS, Debian o Redhat Linux.

* Volúmenes, sirven para guardar los datos persistentes y compartir datos entre contenedores.
* Links, sirven para enlazar contenedores entre sí, que están dentro de una misma máquina, sin exponer a los contenedores cuáles son los datos de la máquina que los contiene.

## Otros componentes Docker

A continuación, una breve descripción de los componentes Docker usados en este proyecto. Los siguientes capítulos detallan como se instalan y se usan estos componentes.

* **Docker-Compose**, permite definir aplicaciones de varios contenedores en un archivo con las mismas propiedades que indicaríamos con el comando *“Docker run”* individualmente. Con un único comando, se inician todos los contenedores en el orden especificado.
* **Docker-Machine**, permite instalar el motor Docker en hosts virtuales y administrar los hosts con comandos de Docker Machine. Los hosts se pueden crear en un equipo local, en un centro de datos, etc.

Docker machine automatiza todas las tareas de aprovisionamiento e instalación de un único host Docker.

* **Docker Swarm**, consiste en varios hosts Docker que se ejecutan en modo enjambre y actúan como administradores (para gestionar la membresía y la delegación) y los trabajadores (que ejecutan servicios de enjambre). Un host Docker determinado puede ser un administrador, un trabajador o realizar ambos roles. Cuando crea un servicio, define su estado óptimo (cantidad de réplicas, recursos de red y almacenamiento disponibles, puertos que el servicio expone al mundo exterior y más). Docker trabaja para mantener ese estado deseado. Por ejemplo, si un nodo de trabajador deja de estar disponible, Docker programa las tareas de ese nodo en otros nodos. Una tarea es un contenedor en ejecución que es parte de un servicio de enjambre y administrado por un administrador de enjambre, en lugar de un contenedor independiente.

## Instalación de Docker

Docker está disponible en dos ediciones: Community Edition (CE) y Enterprise Edition (EE).

Este proyecto usa Docker Community Edition (CE), ejecutándose en Ubuntu 16.04 TLS.

Para la instalación de Docker he seguido las instrucciones de <https://www.digitalocean.com/community/tutorials/como-instalar-y-usar-docker-en-ubuntu-16-04-es>. Una vez instalado, comprobamos que todo ha ido bien ejecutando el comando ***systemctl status Docker,*** que genera una salida como la que se muestra en la siguiente figura.

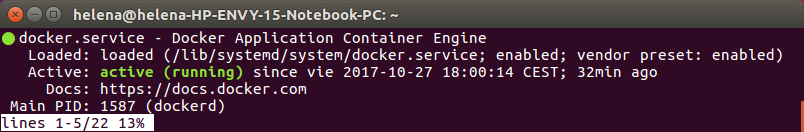


Figura ‑: Comprobación instalación Docker

Para ver la versión de Docker instalada se utiliza el comando ***Docker –version,*** la versión que estoy usando es la mostrada en la siguiente figura.

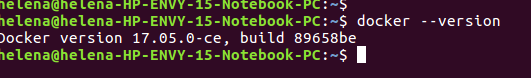


Figura ‑: Ver versión Docker

Una vez Docker está instalado ya puedo crear una de las imágenes usadas en este proyecto. Para ello uso el siguiente fichero Dockerfile.

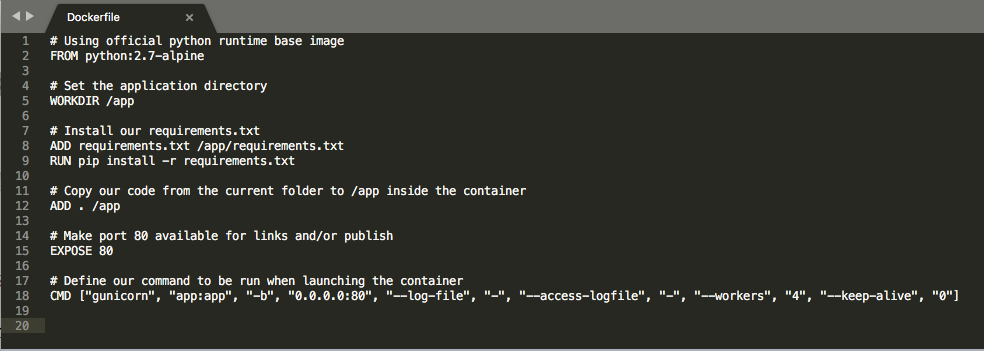


Figura ‑: Fichero Dockerfile para creación de imagen vote

El comando “***docker build –t vote-he:1.0*** .” crea la imagen ***vote-he*** *versión* ***1.0***como se muestra en la siguiente figura.

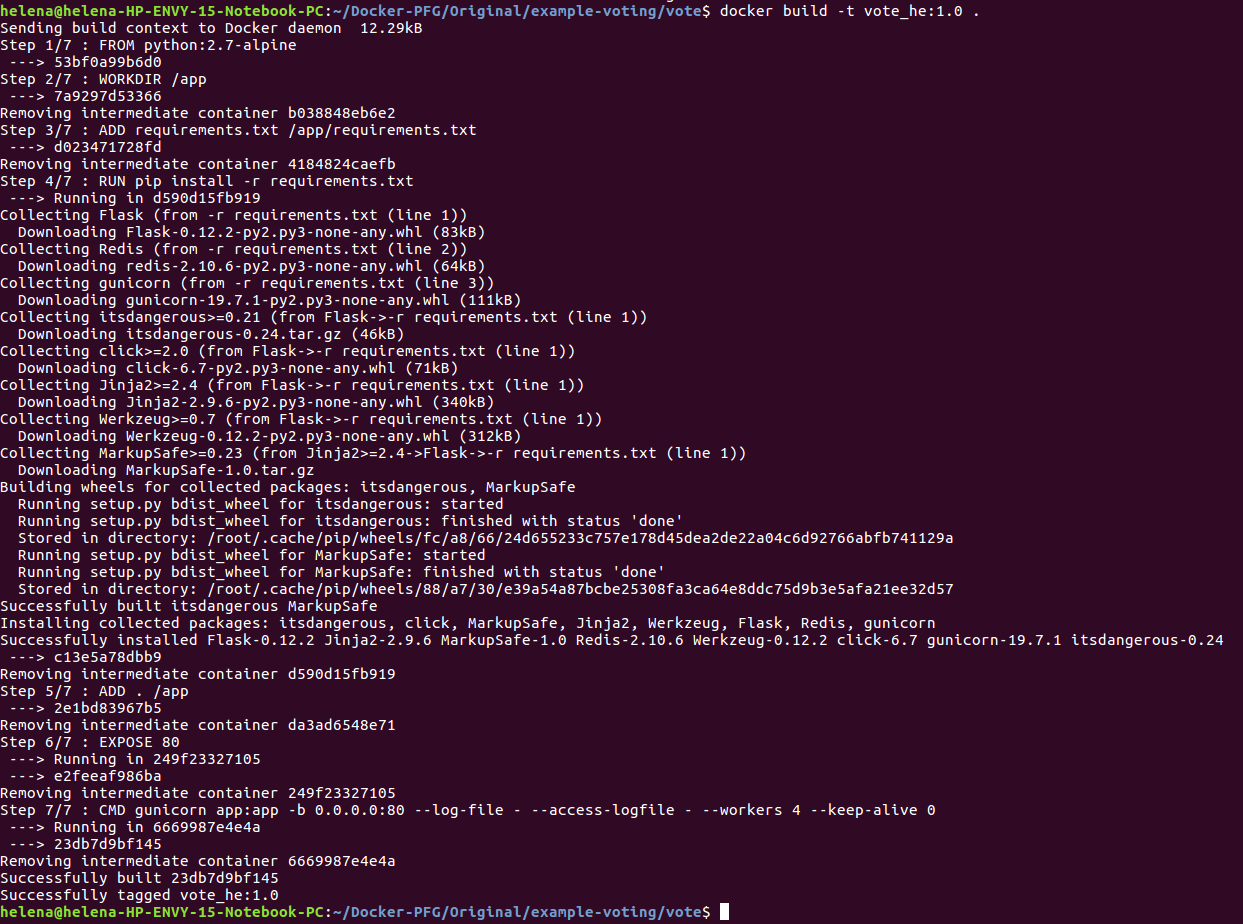


Figura 3‑6: Docker build vote-he:1.0

Con el comando ***docker run –name vote –d –p 5000:80 vote-he:1.0*** creo el contenedor y le doy el nombre *vote\_he,* como puede verse en la siguiente figura

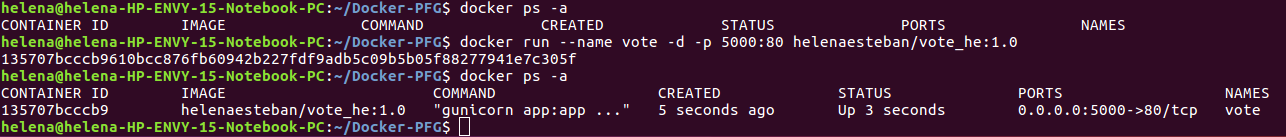


Figura ‑: Contenedores Docker en ejecución

## Instalación de Docker-Machine

Para la instalación de Docker he seguido las instrucciones de <https://docs.docker.com/machine/install-machine/>. Una vez instalado, se comprueba que todo ha ido bien ejecutando el comando ***docker-machine –version,*** que devuelve la versión de **docker-machine** instalada, como puede verse en la siguiente figura.

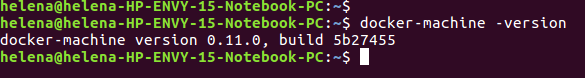


Figura ‑: Versión de Docker-machine instalada

Como ya he dicho en un capítulo anterior **docker-machine** permite instalar el motor Docker en hosts virtuales y administrar los hosts con comandos de **docker-machine**. Utiliza distintos drivers que permiten crear y configurar el motor de Docker en distintos entornos y proveedores, por ejemplo, virtualbox, AWS o VMWare. En este proyecto he usado virtualBox.

Es necesario tener instalado virtualBox para poder crear los hosts virtuales. Para la instalación de virtualBox en Ubuntu he descargado la última versión de virtualBox desde su página de descargas <https://www.virtualbox.org/wiki/Downloads> y he instalado el paquete descargado usando el comando “***sudo dpkg –i <nombre del paquete>”.***

Una vez que **docker-machine** y virtualBox están instalados se pueden crear los hosts virtuales necesarios.

Para la creación de un nuevo host se usa el comando “***docker-machine create --driver virtualbox <nombre host>”.***

Para comprobar el estado del host creado uso el comando “***docker-machine ls***” que muestra los hosts virtuales existentes en el PC y su estado (ver insertar referencia a figura 6.1 ).

## Instalación de Docker Compose

Para la instalación de docker-compose he seguido las instrucciones de <https://docs.docker.com/compose/install/#install-compose>. Una vez instalado, se comprueba que todo ha ido bien ejecutando el comando “***docker-compose –version”,*** que devuelve la versión instalada como puede verse en la siguiente figura.

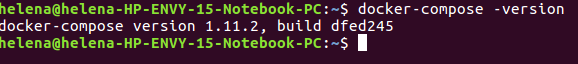


Figura ‑: Versión docker-compose

Docker-compose, usa un archivo YAML para configurar los servicios de la aplicación. Con un solo comando, crea e inicia todos los servicios configurados en el fichero. Para usar docker-compose es necesario seguir estos pasos:

* Definir el fichero Dockerfile de la aplicación.
* Definir los servicios que componen la aplicación en un fichero ***yml*** para que puedan ejecutarse juntos en un entorno aislado.
* Ejecución de docker-compose mediante el comando ***<nombre fichero>.yml up*** que iniciará y ejecutará toda la aplicación.

Por ejemplo, para ejecutar la aplicación vote del capítulo 4.1 Instalación de Docker, el fichero sería como el que muestra la siguiente figura.

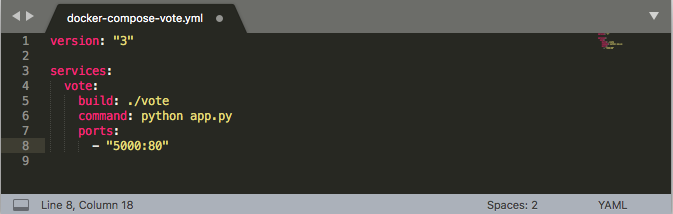


Figura ‑: Fichero doocker-compose-vote.yml

El comando que inicia y ejecuta la aplicación es “***docker-compose –f docker-compose-vote.yml” up***, como puede verse en la siguiente figura.

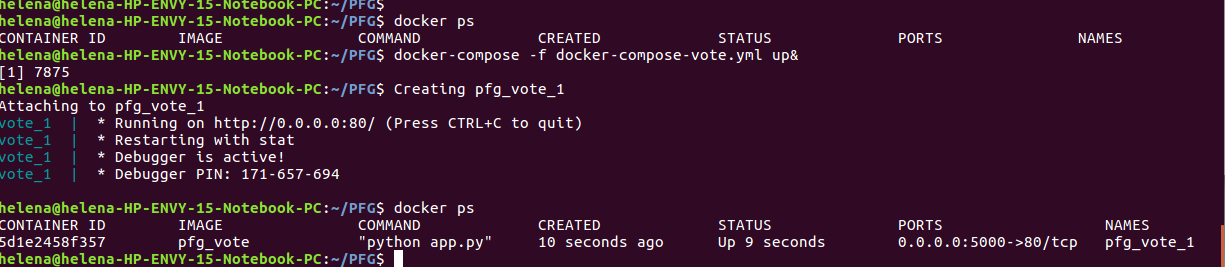


Figura ‑: Ejecución docker-compose

Una vez creado se puede comprobar el resultado invocando la url de la aplicación.

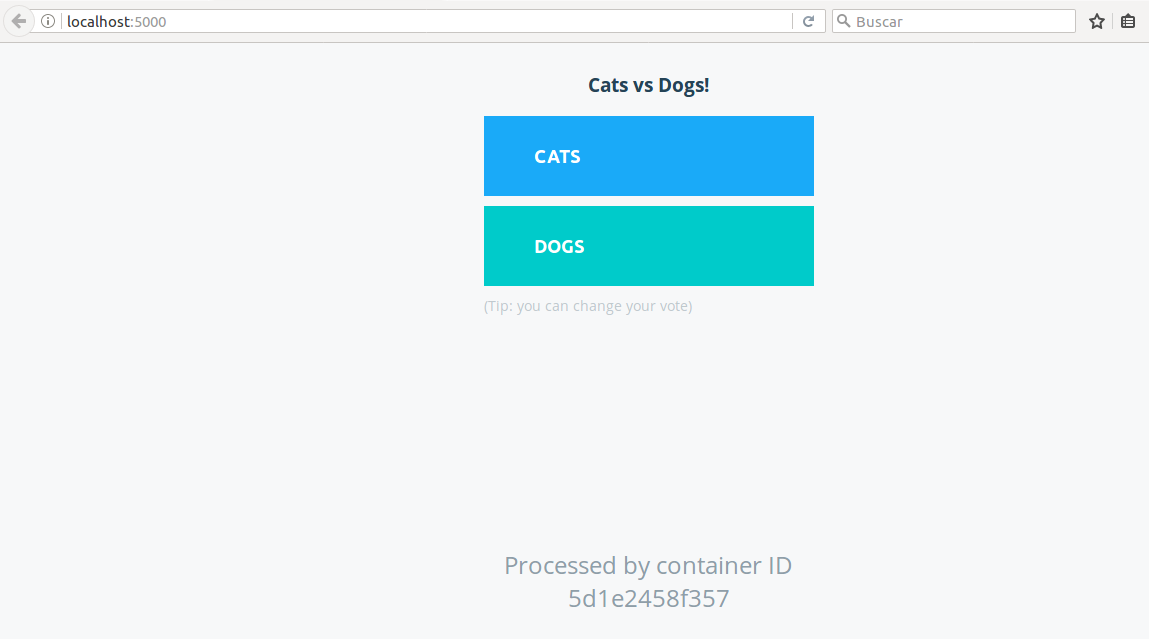


Figura ‑: Página web para votación

# Docker en el entorno de Desarrollo y Pruebas

Incluir explicación de la aplicación y que hace

La aplicación usada para demostrar el despliegue y control de aplicaciones con Docker la he obtenido github (<https://github.com/dockersamples/example-voting-app)>.

He elegido esta aplicación ya que se compone de varios micro-servicios cada uno de ellos usando diferentes tecnologías. Esto me permite demostrar la utilidad de la plataforma Docker para el despliegue y gestión de contenedores que incluyen servicios usando diferentes tecnologías que corren en un mismo host virtual.

Esta aplicación se compone de tres micro-servicios:

* Vote, aplicación web implementada en Python que permite votar entre opciones
* Worker, aplicación implementada en .NET y en java que recibe los votos y los almacena en una Base de datos.
* Result, aplicación web implementada sobre Node.js que obtiene los votos de la base de datos y muestra los resultados de los votos realizados en tiempo real.

Usa:

* Cola Redist, que recibe los votos.
* Base de datos Postgres, para almacenar los votos, esta base de datos está respaldada por un volumen Docker.

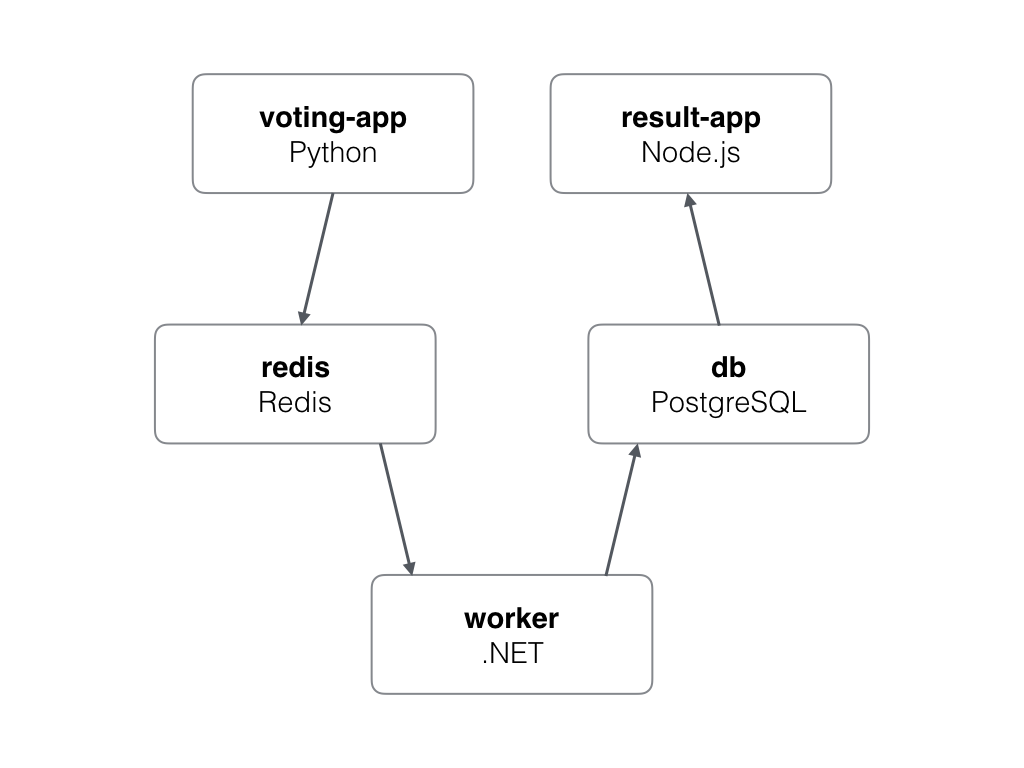


Figura ‑: arquitectura de la aplicación – Obtenida de Github

Para demostrar las posibilidades de Docker y sus componentes, voy a simular un entorno de desarrollo en el que el desarrollador en su host virtual crea imágenes y ejecuta contenedores donde puede probar el desarrollo realizado. Una vez depuradas las aplicaciones, con docker-compose se pueden desplegar los distintos contenedores para realizar pruebas de integración.

Se van a crear varias versiones de una misma imagen para demostrar cómo es de fácil volver a una versión anterior en caso de fallo, o ver cómo ha ido evolucionando el desarrollo de la aplicación.

La aplicación descargada de github solo acepta un voto por cliente. No registra los votos si ya se ha enviado un voto de un cliente. He realizado cambios mínimos, para que permita visualizar los resultados de distinta forma, así como cambios para poder realizar más votos usando un cliente ejecutados desde el mismo entorno.

## Imágenes

En este documento se usan las imágenes descargadas de docker hub (redis:alpine y postgres:9.4) así como las imágenes que se detallan en el capítulo Dockerfile

### Dockerfile

Docker puede construir imágenes automáticamente leyendo las instrucciones de un Dockerfile. Un Dockerfile es un documento de texto que contiene todos los comandos que se podrían llamar en la línea de comando para ensamblar una imagen. Al usar Docker se realiza una compilación automatizada que ejecuta varias instrucciones de línea de comandos en sucesión.

El fichero Dockerfile y el contexto (aplicación, ficheros auxiliares, etc..) deben de estar en el mismo directorio.

Docker ejecuta las instrucciones en un Dockerfile en orden. Debe comenzar con una instrucción `FROM`. FROM solo puede estar precedido por una o más instrucciones ARG, que declaran argumentos que se utilizan en líneas FROM en Dockerfile.

Docker trata las líneas que comienzan con # como comentario, a menos que la línea sea una directiva de analizador válida. Un marcador # en cualquier otro lugar de una línea se trata como un argumento

Como crear un Dockerfile se pueden ver en <https://docs.docker.com/engine/reference/builder/>, a continuación, describo los comandos usados en este proyecto.

* FROM, inicializa una compilación y establece la Imagen base para las instrucciones posteriores. La imagen puede ser cualquier imagen válida; es especialmente fácil comenzar tirando de una imagen de los repositorios públicos.

Puede aparecer más de una vez para crear varias imágenes o realizar una nueva compilación como dependencia de otra. Si se usan varios FROM en un mismo Dockerfile se debe tomar nota de la última ID de imagen emitida por la confirmación antes de cada nueva instrucción FROM. Cada instrucción FROM borra cualquier estado creado por instrucciones anteriores.

* WORKDIR, establece el directorio de trabajo para las instrucciones RUN, CMD, ENTRYPOINT, COPY y ADD que lo siguen en Dockerfile. Si el WORKDIR no existe, se creará incluso si no se utiliza en ninguna instrucción posterior de Dockerfile.

La instrucción WORKDIR se puede utilizar varias veces en un Dockerfile. Si se proporciona una ruta relativa, será relativa a la ruta de la instrucción WORKDIR previa.

* ADD, copia nuevos archivos, directorios o URL de archivos remotos de <src> y los agrega al sistema de archivos de la imagen en la ruta <dest>.
* RUN, ejecuta los comandos en una nueva capa en la parte superior de la imagen actual y confirmará los resultados. La imagen comprometida resultante se usará para el siguiente paso en el archivo Docker.
* EXPOSE, informa a Docker que el contenedor escucha en los puertos de red especificados en tiempo de ejecución. Se puede especificar si el puerto escucha en TCP o UDP, por defecto TCP.

En realidad, la instrucción EXPOSE no publica el puerto. Solo informa qué puertos están destinados a ser publicados. Para publicar realmente el puerto cuando se ejecuta el contenedor, se usa el indicador -p para publicar y asignar uno o más puertos, o -P para publicar todos los puertos expuestos y asignarlos a puertos de orden superior.

* ENV, establece la variable de entorno <clave> al valor <valor>. Este valor estará en el entorno de todos los comandos Dockerfile "descendientes" y también se puede reemplazar en línea en muchos
* CMD, proporciona valores predeterminados para un contenedor de ejecución. Estos valores predeterminados pueden incluir un ejecutable, o pueden omitir el ejecutable, en cuyo caso se debe especificar una instrucción ENTRYPOINT.

#### Aplicación vote\_he – Dockerfile

Ver Figura 4‑3

#### Aplicación worker\_he – Dockerfile

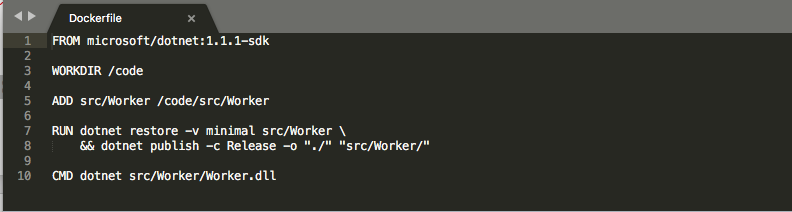


Figura ‑: Dockerfile para crear imagen worker\_he

#### Aplicación Result\_he – Dockerfile



Figura ‑: Dockerfile para crear imagen result\_he

### Docker Build

Las imágenes se pueden crear desde línea de comandos o por medio de un Dockerfile.

En este proyecto, las imágenes se crean utilizando el comando build usando un Dockerfile y un "contexto". El contexto de construcción es el conjunto de archivos ubicados en la RUTA o URL especificada. El proceso de compilación puede referirse a cualquiera de los archivos en el contexto.

Dado que he creado varias imágenes, y el Dockerfile no permite el uso de TAG, he creado un script “build.sh” para poder versionar las imágenes.

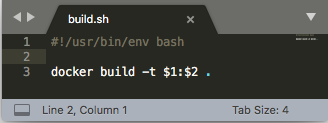


Figura ‑: script para creación de imágenes

#### Creación de imágenes vote\_he:x.x

La imagen identificada como vote\_he:1.0 que implementa el código original bajado de Github fue creada en el capítulo 4.1 Instalación de Docker como puede verse en la Figura 4‑4. La Figura 4‑10 muestra la pantalla web generada por la imagen vote\_he:1.0

Un desarrollador puede tener varias versiones de la aplicación generando nuevas imagines con versiones distintas. Por ejemplo, he modificado la aplicación para mostrar los textos en español. Vuelve a construir la imagen vote\_he, en este caso con “tag” 2.0

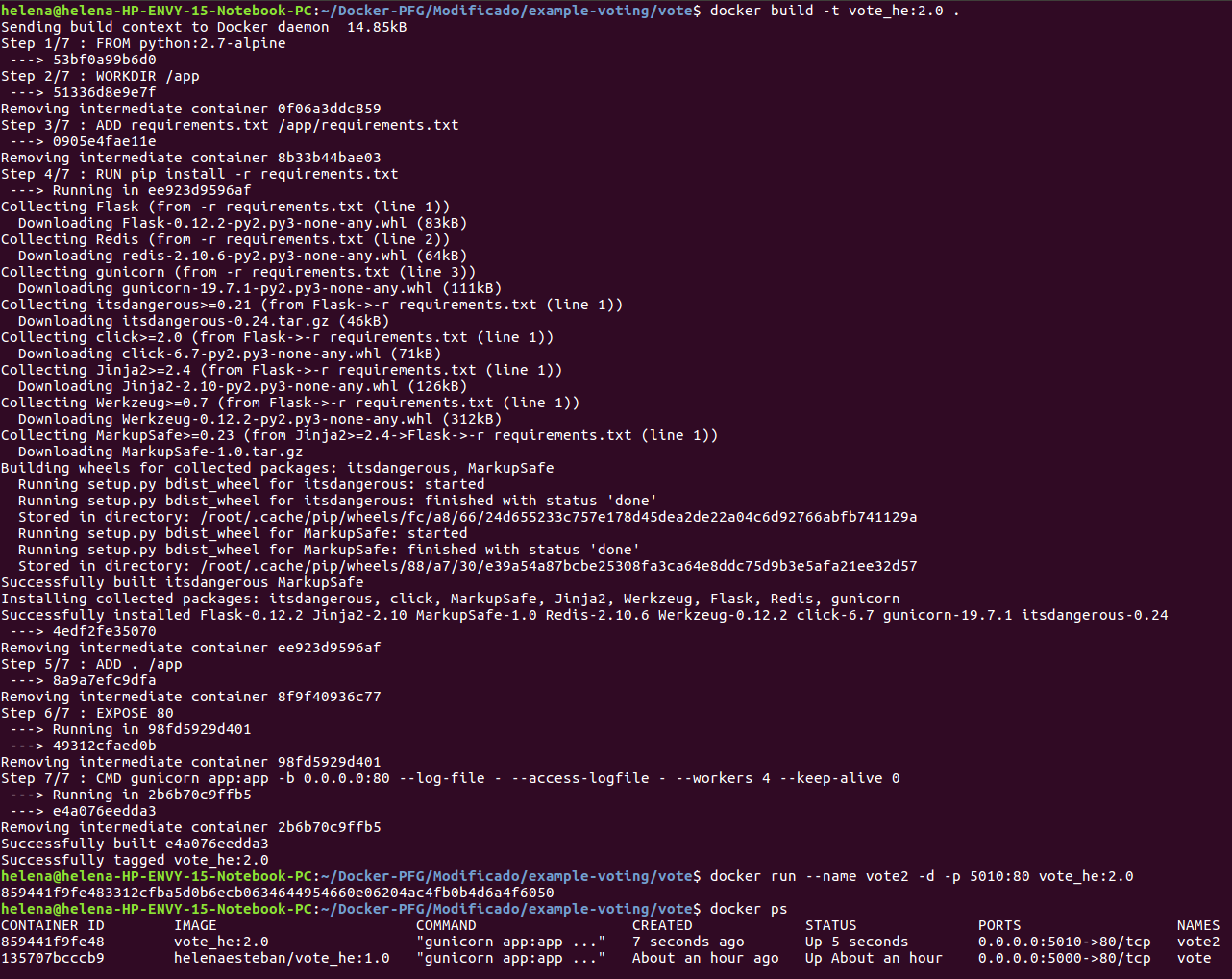


Figura ‑: Creación imagen vote\_he:2.0

He cambiado el puerto al 5010 para que se puedan utilizar los dos contenedores. Ahora al conectar allocalhost:5010, obtenemos una página como la de la siguiente figura.

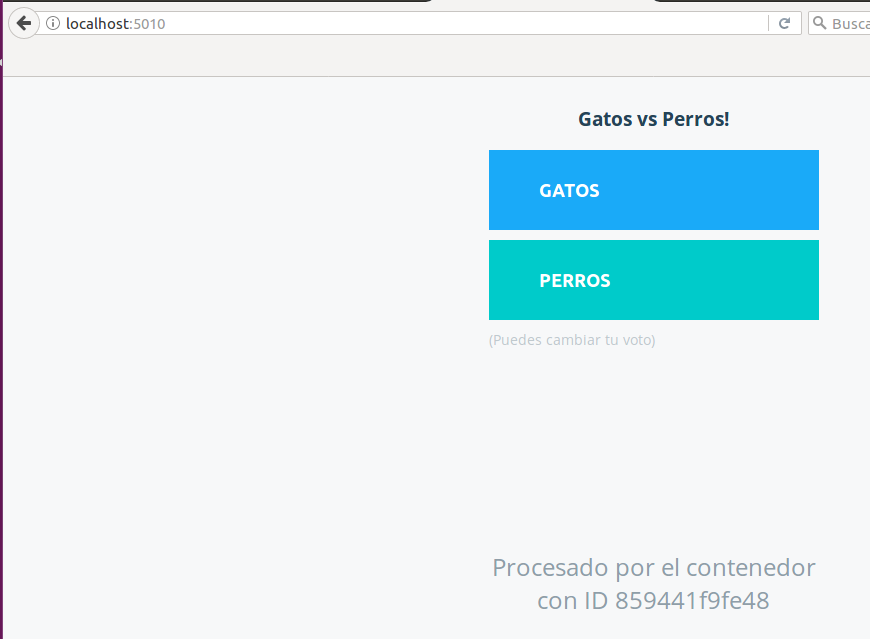


Figura ‑: Página web – aplicación vote\_he:2.0

#### Creación de imagen worker\_he:x.x

El código de la aplicación worker no permite realizar dos votos seguidos desde el mismo cliente. Se ha modificado el programa para evitar esta restricción y poder votar varias veces desde el mismo cliente opción.

Se han creado dos imágenes como en el caso anterior.

* worker:1.0 basada en el código original
* worker:2.0 basada en el código modificado.

Esta aplicación puede ser ejecutada con .NET o en Java, he usado el entorno Java. La creación de esta imagen es bastante larga, debido a todas las dependencias Maven, a continuación, se muestra la primera parte de la creación de la imagen 1.0 y después el resultado y la lista de imágenes creadas.

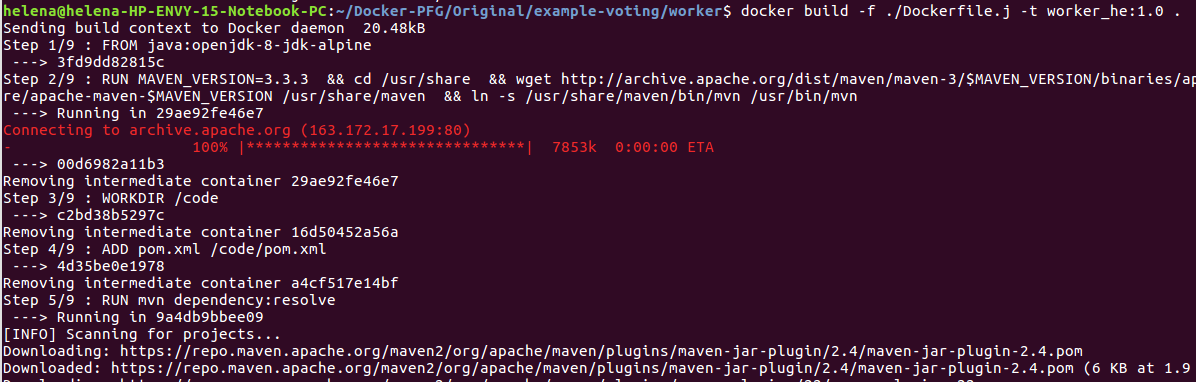


Figura ‑: Creación imagen worker\_he:1.0 – parte1

En la siguiente imagen podemos ver que la imagen worker\_he1.0 ha sido creada.

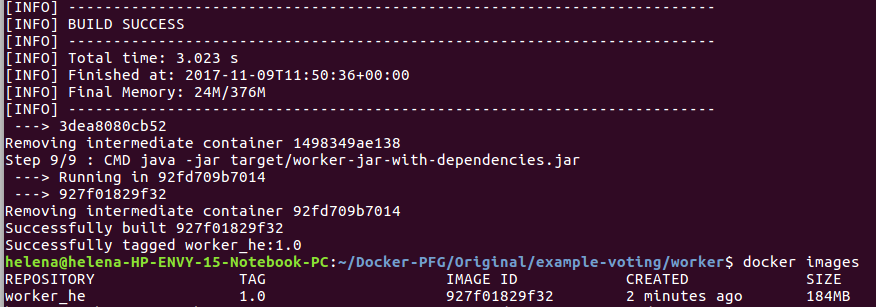


Figura ‑: Creación imagen worker:1.0 – parte2

#### Creación de imagen result\_he:x.x

El código de la aplicación **result** muestra el porcentaje de votos a cada opción. Se ha modificado el programa para mostrar el número total de votos de cada opción.

Se han creado dos imágenes como en el caso anterior.

* result:1.0 basada en el código original
* result:2.0 basada en el código modificado.

A continuación, la creación de la imagen result\_he:2.0.

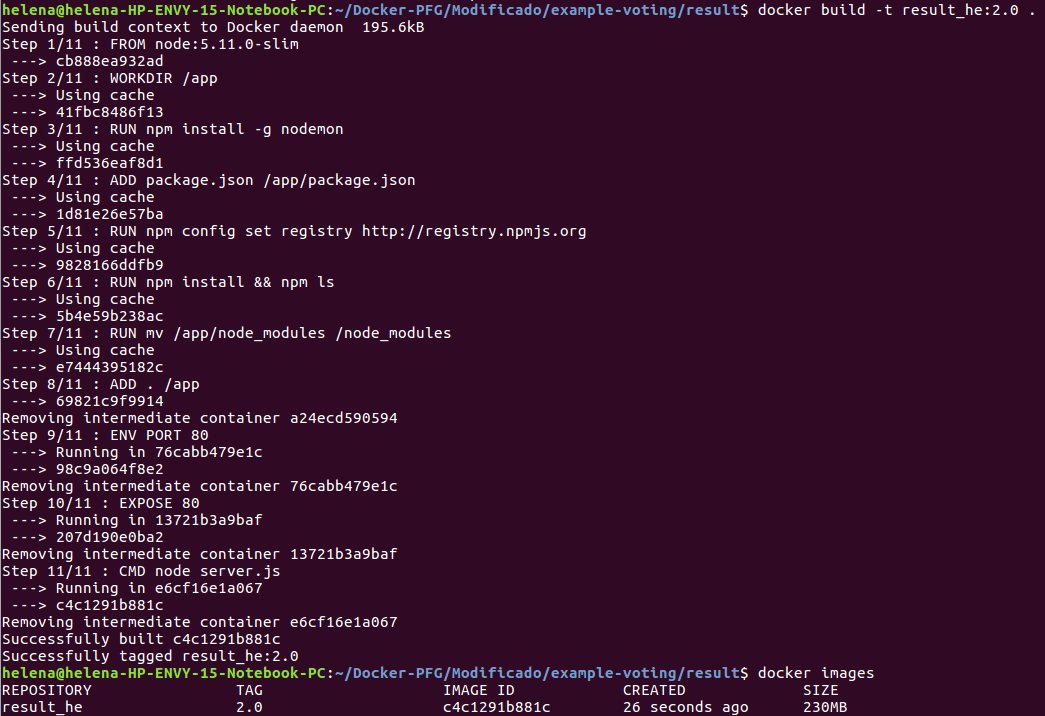


Figura ‑: Creación imagen result:2.0

A continuación, se muestra la página web para result:1.0 y result:2.0

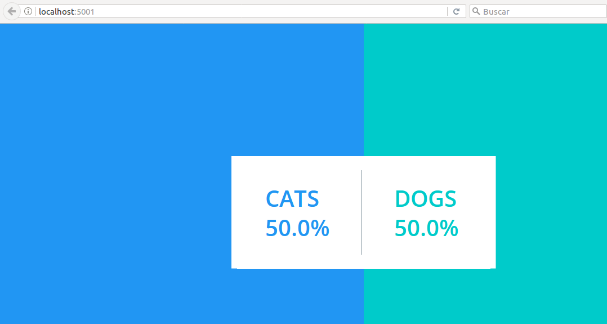
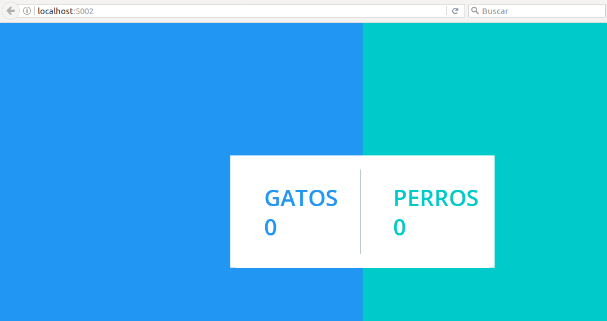
 

Figura ‑: Página web de result:1.0 y página web de result:2.0

### Docker Push

Una vez que las pruebas son satisfactorias, se pueden subir las imágenes creadas a “Docker Hub”, desde donde se pueden descargar por otros desarrolladores, y ser usadas para el entorno de producción como puede verse en el siguiente capítulo.

He creado una cuenta “helenaesteban” en las que he ido subiendo las distintas imágenes, empezando por las imágenes creadas usando el código origen y las distintas modificaciones que he ido realizando.

Antes de subir la imagen a Docker Hub hay que añadir a la imagen el nombre del usuario de Docker Hub en mi caso “helenaesteban” para esto se usa el comando “docker tag” y una vez realizado el “docker tag” se realiza el push de la imagen, como se puede ver en la siguiente figura para la imagen :1.0.

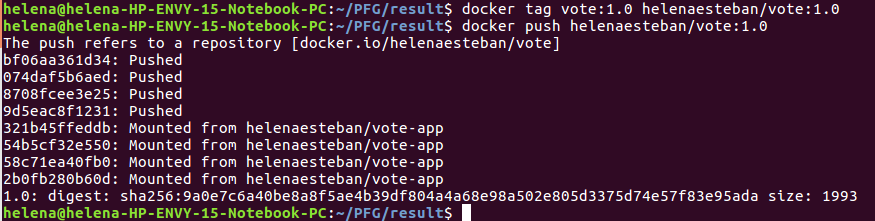


Figura ‑: Docker tag y docker push al repositorio “docker hub”

### Docker-Compose

Una vez finalizadas todas las aplicaciones, se pueden ejecutar todas ellas en un host virtual para pruebas end-to-end. Para ello uso el fichero docker-compose.

A continuación, se muestran los ficheros docker-compose necesarios para lanzar la aplicación original y la modificada.

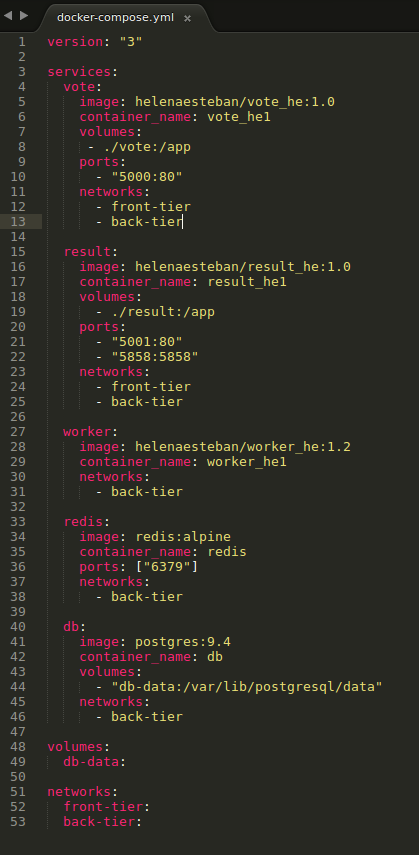
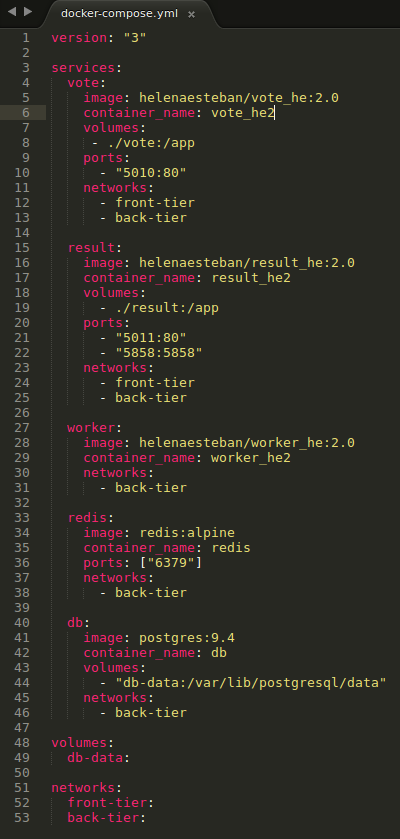
 

Figura ‑: Ficheros docker-compose

En estos ficheros pueden verse todas las aplicaciones que componen la solución obtenidas desde el repositorio de Docker, además de los volúmenes necesarios para que los distintos contenedores puedan compartir datos y para mantener los datos cuando los contenedores se paran y las redes definidas.

Una vez ejecutado el fichero docker-compose se puede realizar una prueba end-to-end de la aplicación.

Ver siguientes figuras

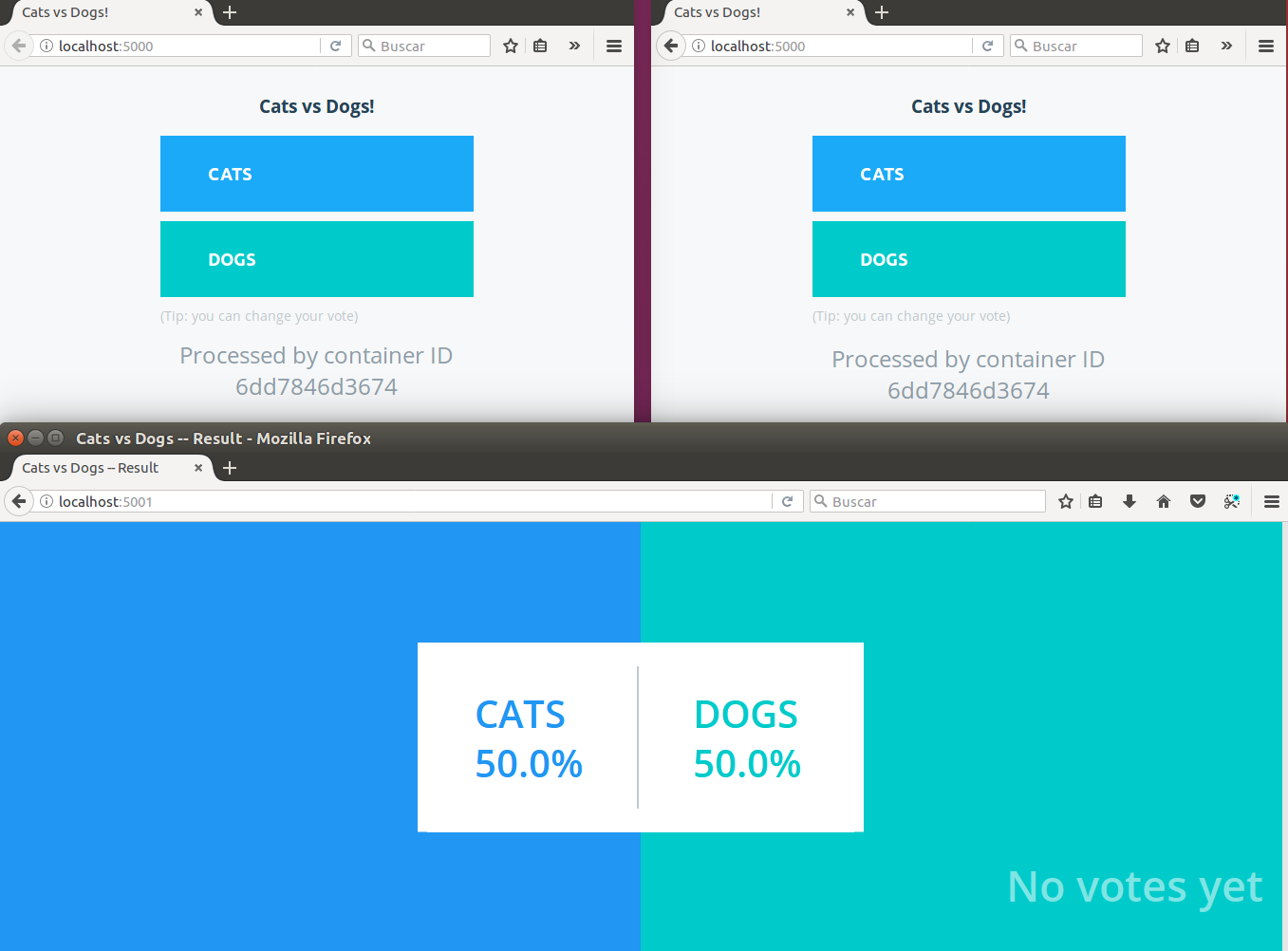


Figura ‑: Aplicación Original - No hay votos

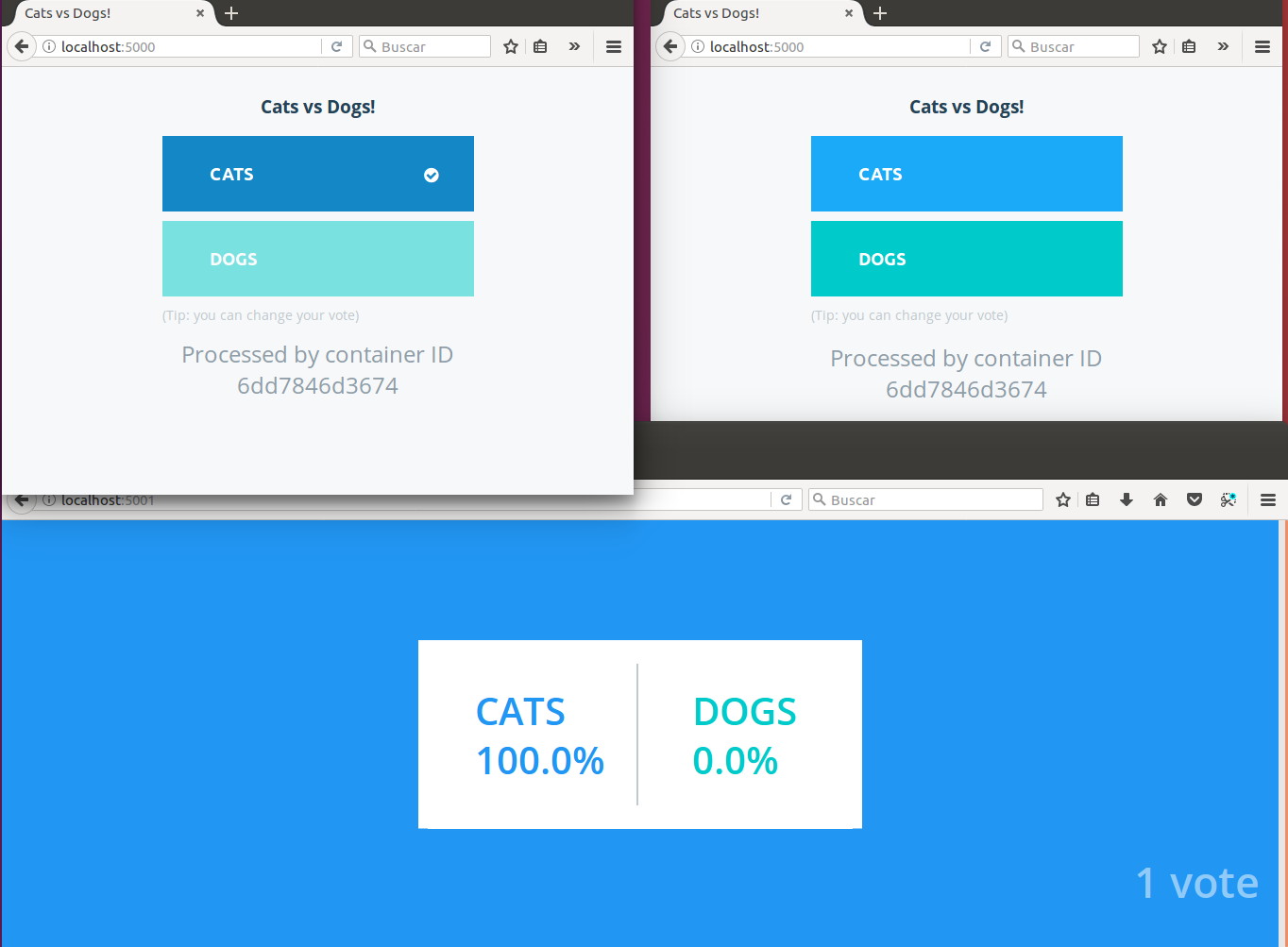


Figura ‑: Aplicación Original - 1 voto emitido

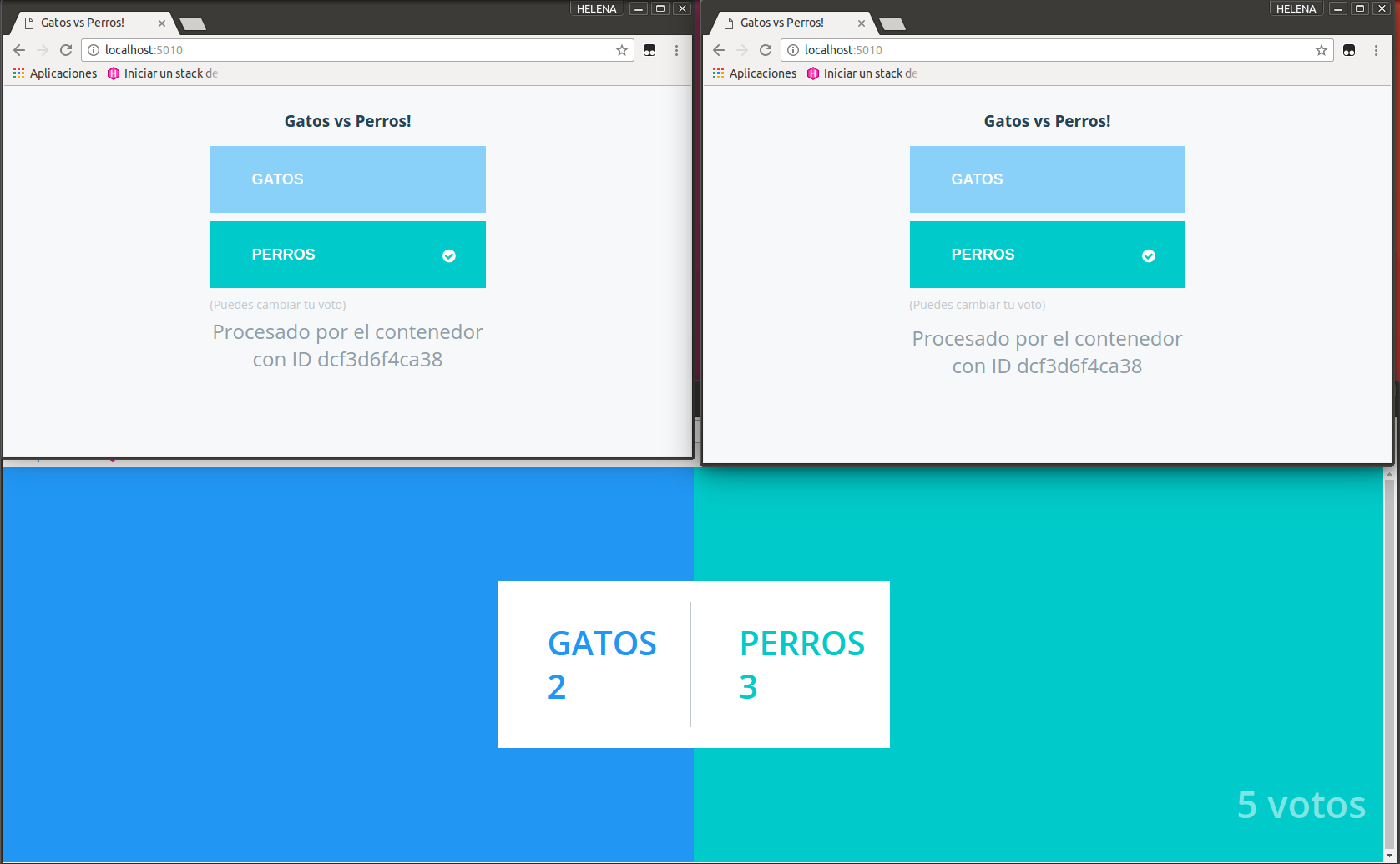


Figura ‑: Aplicación Modificada - 5 votos emitidos

# Docker en el Entorno de Producción

Introducción de la aplicación, ya comentada anteriormente, y por que Swarm en el contexto de la aplicación

En el punto anterior la aplicación se ha desplegado en un host, este tipo de despliegue es válido durante el proceso de desarrollo y pruebas de la aplicación.

En un entorno de producción se requiere que la aplicación este siempre disponible, esto requiere un despliegue en cluster, que permita tener réplicas de los servicios para que, en caso de fallo de una de ellas, exista otra disponible.

Para este tipo de entornos Docker proporciona Docker Swarm.

Para este proyecto he usado tres hosts en los que se desplegaran los servicios en la figura 4-1

En el fichero compose creado para desplegar los servicios, podemos dejar al administrador del Swarm que distribuya los servicios o podemos indicarle donde crearlos como se muestra en la figura-6-8

Antes de realizar el despliegue voy a describir lo que es Swarm que e Docker Swarm y otros conceptos relacionados

## Docker Swarm

Un Swarm consiste en varios hosts Docker que se ejecutan en modo Swarm y actúan como administradores (para gestionar la membresía y la delegación) y los trabajadores (que ejecutan servicios del swarm). Un host Docker determinado puede ser un administrador, un trabajador o realizar ambos roles. Cuando crea un servicio, define su estado óptimo (cantidad de réplicas, recursos de red y almacenamiento disponibles, puertos que el servicio expone al mundo exterior y más). Docker trabaja para mantener ese estado deseado. Por ejemplo, si un nodo de trabajador deja de estar disponible, Docker programa las tareas de ese nodo en otros nodos. Una tarea es un contenedor en ejecución que es parte de un servicio de Swarm y administrado por un administrador de Swarm, en lugar de un contenedor independiente.

Debido a que Docker Swarm utiliza el API estándar Docker, cualquier herramienta que se comunique con el demonio Docker puede utilizar Swarm para escalar de forma transparente a múltiples host. De entre las herramientas soportadas en este proyecto se usa Docker Compose.

El uso de Docker Machine es el mejor método para comenzar con Docker Swarm

Una de las ventajas clave de los servicios del Swarm sobre los contenedores independientes es que puede modificar la configuración de un servicio, incluidas las redes y volúmenes a los que está conectado, sin la necesidad de reiniciar manualmente el servicio. Docker actualizará la configuración, detendrá las tareas de servicio con la configuración desactualizada y creará nuevas que coincidan con la configuración deseada.

Cuando Docker se ejecuta en modo Swarm, aún puede ejecutar contenedores independientes en cualquiera de los hosts Docker que participan en el Swarm, así como en servicios del Swarm. Una diferencia clave entre los contenedores independientes y los servicios del Swarm es que solo los administradores del Swarm pueden administrar un Swarm, mientras que los contenedores independientes pueden iniciarse en cualquier demonio. Los demonios de Docker pueden participar en un Swarm como administradores, trabajadores o ambos.

De la misma manera que se puede usar **docker-compose** para definir y ejecutar contenedores, puede definir y ejecutar pilas de servicios del Swarm.

### Nodos

Un nodo es una instancia del motor de Docker que participa en el Swarm. Se pueden ejecutar uno o más nodos en un único ordenador o servicio cloud, pero las implementaciones de Swarm de producción suelen incluir nodos Docker distribuidos en varias máquinas físicas y en la nube.

Para desplegar una aplicación en un Swarm, se envía una definición a un nodo administrador y el nodo administrador distribuye unidades de trabajo llamadas tareas a nodos de trabajadores.

Los nodos administradores también realizan las funciones de orquestación y administración de clúster necesarias para mantener el estado deseado del Swarm. Los nodos administradores eligen a un solo líder para realizar las tareas de orquestación.

Los nodos trabajadores reciben y ejecutan tareas enviadas desde los nodos administradores. Por defecto, los nodos administradores también ejecutan servicios como nodos trabajadores, pero se pueden configurar para ejecutar tareas de administrador exclusivamente y ser nodos solo administradores. Un agente se ejecuta en cada nodo trabajador e informa sobre las tareas asignadas. El nodo trabajador notifica al nodo administrador el estado actual de sus tareas asignadas para que el administrador pueda mantener el estado deseado de cada trabajador.

Arquitectura de Swarm (incluir un dibujo con los contenedores, relaciones, etc…)



Figura ‑: Cluster de servicios

Como puede verse en la figura anterior, en el Swarm se desplegarán:

* Un volumen “db-data” para respaldo de los votos insertados en la Base de Datos.
* Dos redes, “frontend” y “backend”
* Tres nodos, dos de ellos con funciones de administrador/trabajador y otro con funciones de trabajador

### Servicios y Tareas

Un servicio es la definición de las tareas a ejecutar en los nodos administradores o trabajadores. Es la estructura central del sistema Swarm y la raíz primaria de la interacción del usuario con el Swarm.

Cuando se crea un servicio, se especifica qué imagen de contendor usar y qué comandos se deben ejecutar dentro de los contenedores en ejecución.

En el modelo de servicios replicados, el administrador del Swarm distribuye una cantidad específica de tareas de réplica entre los nodos según la escala que establezca en el estado deseado.

Para los servicios globales, el Swarm ejecuta una tarea para el servicio en cada nodo disponible en el clúster.

Una tarea lleva un contenedor Docker y los comandos para ejecutar dentro del contenedor. Es la unidad de programación atómica del Swarm. Los nodos administradores asignan tareas a los nodos trabajadores de acuerdo con la cantidad de réplicas configuradas en la escala de servicio. Una vez que una tarea se asigna a un nodo, no puede moverse a otro nodo. Solo puede ejecutarse en el nodo asignado o fallar.

## Creación del Docker Swarm

Para la creación del Swarm, primero hay que crear los hosts virtuales. Para este proyecto voy a crear 3 hosts.

* Host-1, se configura como administrador
* Host -2, se configura como trabajador
* Host-3, se configura como administrador (secundario)

La siguiente figura muestra la creación del host-1



Figura ‑: Creación de host-1

Como driver he usado virtualbox, con la configuración por defecto.

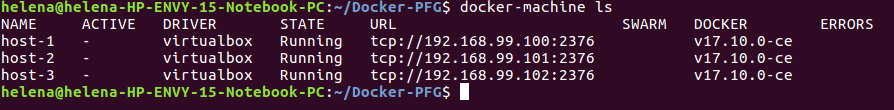


Figura ‑: Lista de hosts

Una vez creados los tres hosts, como puede verse en la figura anterior, me conecto al host-1 con el comando **docker-machine** ssh <nombre host> para inicializar el Swarm, como se muestra en la siguiente figura.

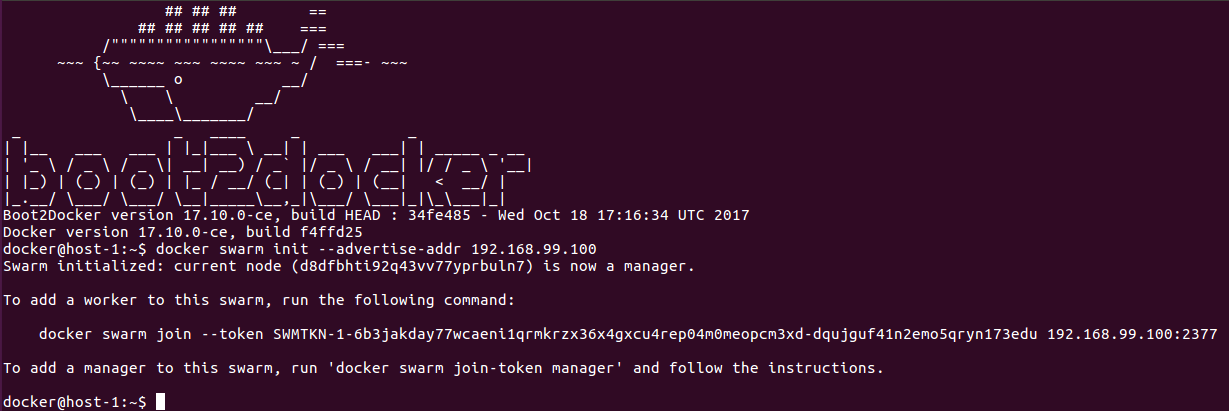


Figura ‑: docker swarm init

En otra ventana de Ubuntu me conecto al host-2 para añadir el worker al swarm con el comando indicado al iniciar el “swarm”, como se indica en la Figura 6‑3

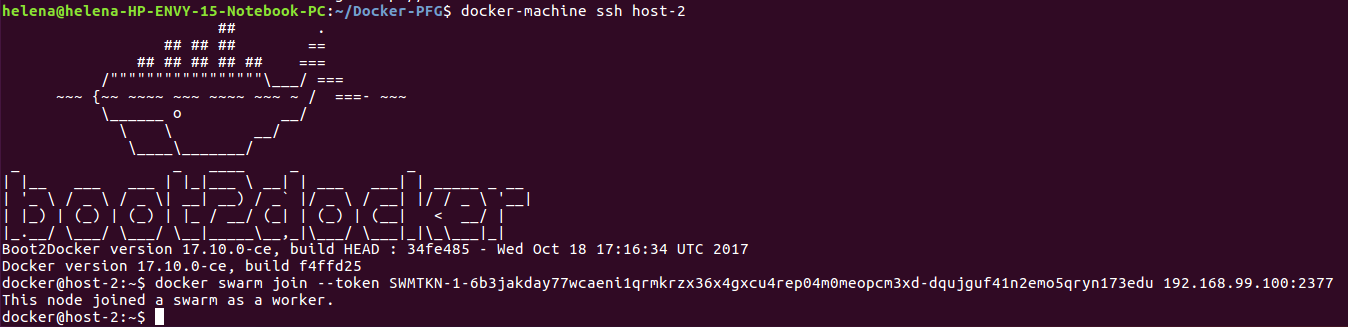


Figura 5‑4: añadir worker al docker swarm

Para añadir un segundo master al swarm, es necesario ejecutar el comando “docker swarm join-token manager”, que nos devuelve el token para poder añadir el nuevo master, el host configurado como master, en mi caso host-1

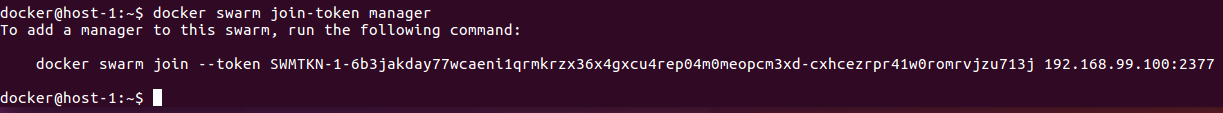


Figura ‑: token para añadir un segundo master

Y por último me conecto al host-3 y añado el master secundario al “swarm

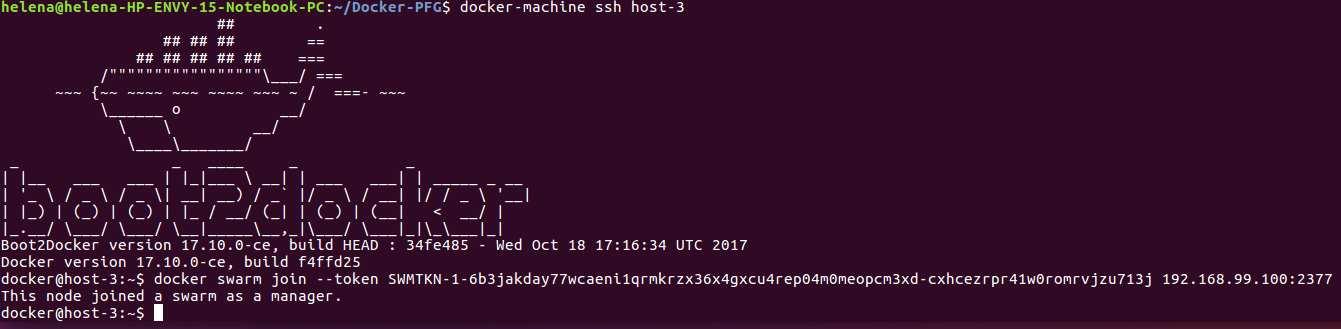


Figura ‑: añadir segundo master

La siguiente figura muestra los hosts que forman el docker swarm

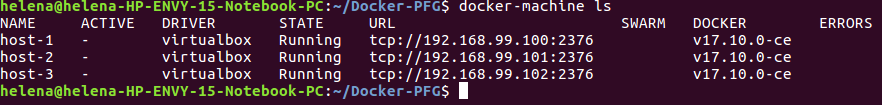


Figura ‑: docker swarm

Una vez creado el Swarm, despliego las aplicaciones usando el fichero docker-compose creado para esta tarea, esto se realiza en el nodo configurado como administrador.

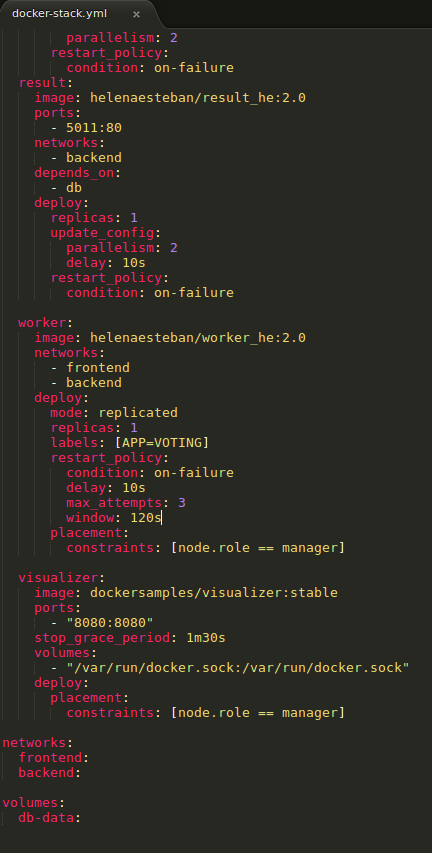


Figura 5‑8: fichero docker-compose para Swarm

El fichero se copia en el host configurado como master usando el comando



Figura ‑: Copiar fichero compose a host-1

El fichero docker-satck.yml, se configuran todos los servicios que componen la aplicación.

Por defecto docker Swarm utiliza al máximo cada host, es decir, despliega tantos servicios como puede contener el host, pero también da la opción de indicar que un servicio se despliegue en un host determinado.

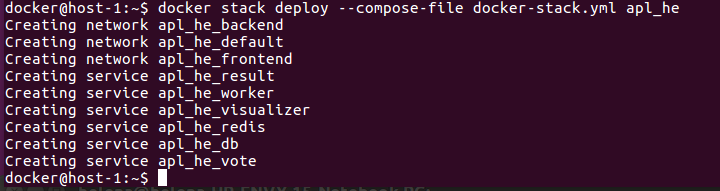


Figura ‑: docker stack deploy

Con los comandos: **docker stack services** <nombre stack> o **docker-machine services** ls muestran el estado del despliegue.

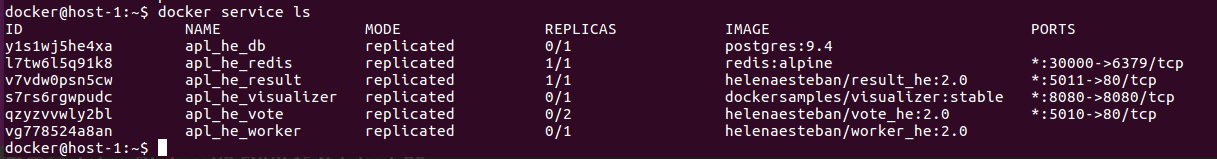


Figura ‑: estado del despliegue de servicios

Docker proporciona una herramienta gráfica, la imagen “visualizer”. En el fichero docker-satck.yml usado para realizar el despliegue de los servicios, está incluida “visualizer”

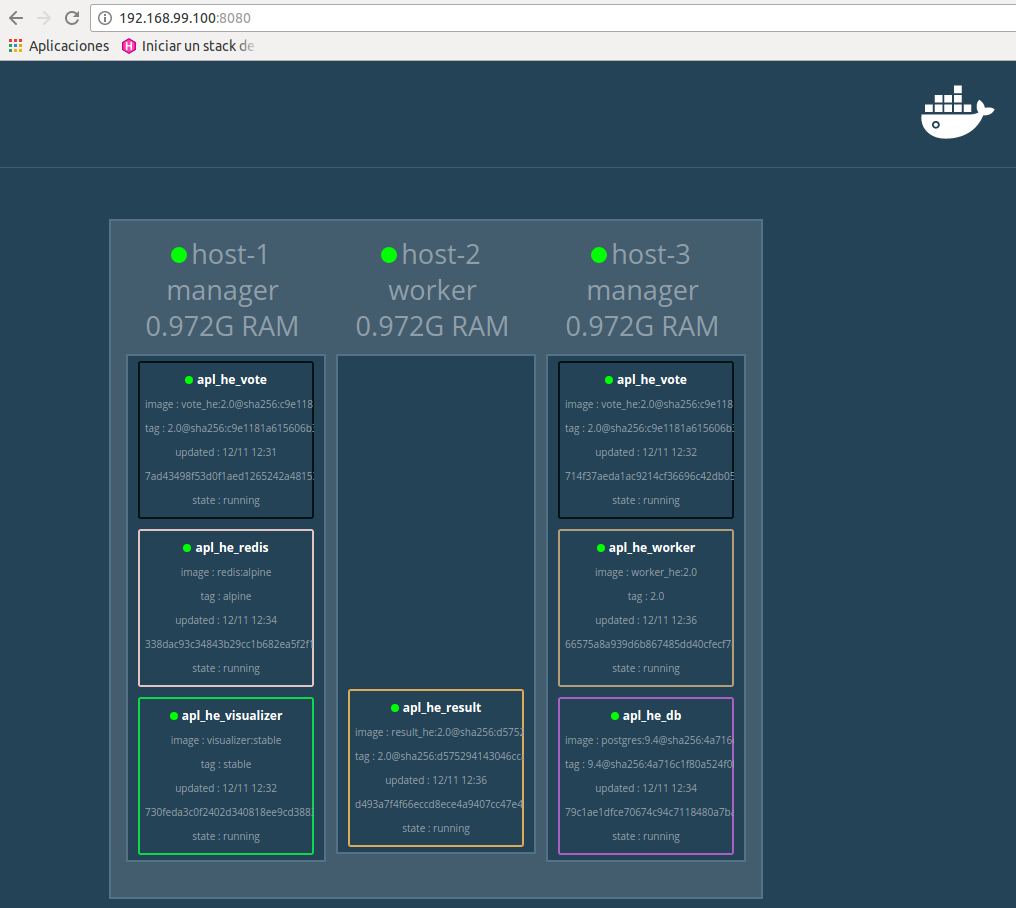


Figura ‑: visualizer

Una vez que todos los servicios están desplegados, pruebo la aplicación usando cualquiera de las IPs de los distintos nodos y el puerto designado para el servicio. El host administrador se ocupa de en-rutar la petición al host donde este corriendo la aplicación.

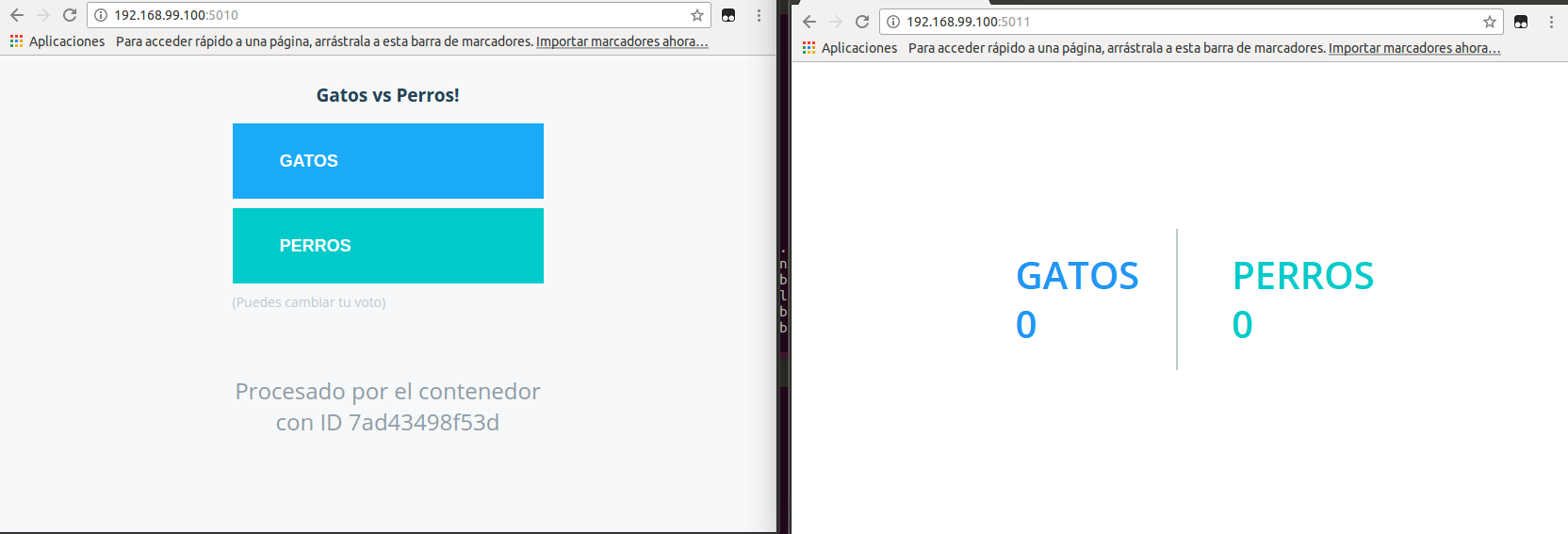


Figura ‑:

# Experimentos y Validación

|  |  |  |  |
| --- | --- | --- | --- |
| FASE I: PLANIFICACIÓN DEL PROYECTO | | | |
| *SUB-FASE* | *OBJETIVO* | *TAREA* | *INSTRUMENTO* |
| Sub-fase de planificación | Planificación de tiempo y fases de trabajo | Elaboración de la estructura | Experiencias de otros compañeros |
| Sub-fase de investigación | Roma de contacto con Docker y sus componentes | Lectura de documentación | Blogs, websites |
| Sub-fase de búsqueda de servicios | Búsqueda en la web de aplicaciones que se puedan desplegar en Docker | Análisis de aplicaciones | Blogs, Websites |

Empecé el proyecto buscando documentación en la Web sobre docker, donde me encontré con los distintos componentes que se pueden usar. A continuación, busqué de una aplicación que cumpliese con los requerimientos necesarios (varias tecnologías) para demostrar las capacidades de docker y docker Swarm.

|  |  |  |  |
| --- | --- | --- | --- |
| FASE II: PREPARACION ENTORNO DE TRABAJO | | | |
| *SUB-FASE* | *OBJETIVO* | *TAREA* | *INSTRUMENTO* |
| Sub-fase instalación de herramientas base | Preparar el entorno de trabajo | Descarga e instalación de Ubuntu y Docker | Paquete de Ubuntu y Docker |
| Sub-fase de instalación de componentes Docker | Añadir nuevos componentes al entorno de trabajo | Descarga e instalación de docker-compose, docker-machine y virtualbox | Software de los componentes instalados |

Después de analizar qué sistema operativo (Windows, Mac o Linux) sería mejor para realizar las pruebas me decidí por Linux en concreto Ubuntu, ya que Docker inicialmente se desarrolló en Linux, y existe mucha documentación en la Web.

Una vez instalado Ubuntu, instale Docker y los componentes necesarios para crear imágenes y ejecutarlas en forma aislada, a continuación, seguí con la instalación de docker-compose y finalmente el siguiente paso era desplegar la aplicación en un Swarm, por lo que instale virtualBox y docker-machine.

|  |  |  |  |
| --- | --- | --- | --- |
| FASE III: DESPLIEGUE APLICACIONES | | | |
| *SUB-FASE* | *OBJETIVO* | *TAREA* | *INSTRUMENTO* |
| Sub-fase de análisis de Docker | Familiarización con Docker (contenedores, imágenes, comandos) | Descarga de imágenes, crear contenedores, utilización de los comandos run, etc.. | Docker hub registry |
| Sub-fase de utilización de Dockerfile | Familiarización con Dockerfile para creación de imágenes | Creación de imágenes | Dockerfile |
| Sub-fase de utilización de docker-compose | Despliegue de todos los servicios que componen la aplicación | Prueba de las aplicaciones desplegadas | browser |
| Sub-fase de modificación y despliegue de aplicación descargada de internet | Comprobar ventajas de docker en creación de nuevos entornos | Modificación de código, de dockerfiles, de fichero docker compose | Ubuntu, docker |
| Sub-fase inicialización de Swarm | Desplegar la aplicación en docker Swarm | Inicialización del Swarm y despliegue de aplicaciones | Docker, Ubuntu, servicios |
| Sub-fase utilización del Swarm | Familiarización con docker swarm (comandos, usos, etc) | Pruebas de despliegue | Ubuntu, docker swarm, docker-compose, servicios |

Comencé con la creación, ejecución y gestión de contenedores basados en imágenes descargadas de Docker Hub. Esto me permitió conocer los comandos básicos de docker: Docker run, docker stop, docker rmi, etc.

A continuación, creación de las imágenes de la aplicación usando Dockerfile y creación de contenedor.

Una vez creadas las tres imágenes que componen la aplicación, despliegue de la aplicación usando docker-compose, esto permito realizar una prueba end-to-endde la aplicación

Modificación de la aplicación, para mostrar texto en español y permitir que desde un mismo cliente se pueda realizar más de un voto. Una vez comprobado que funcionaba correctamente.

Me cree una cuenta en Docker Hub para subir las imágenes creadas

Despliegue de la aplicación usando las imágenes subidas a Docker Hub, modificación del fichero docker-compose para que utilizase las imágenes en docker-hube lugar de crearlas.

Pruebas de modificación de servicios desplegados, comprobando que el servicio recoge los cambios realizados y la imagen no se modifica hasta que no se realiza un commit de la misma

Por último creación del Swarm y despliegue de la aplicación, probando con la creación de distinto número de réplicas, ubicación de servicios, modificación de servicios, añadir replicas, etc..

# Conclusiones

El eslogan de Docker: *“Build, Ship, and Run Any App, Anywhere”*, describe perfectamente las áreas principales de su uso, ya que según he podido comprobar durante el desarrollo de este proyecto es posible usar alguna de sus herramientas en cualquier punto, desde entornos de desarrollo hasta entornos de operación.

Docker nos da una solución fácil y efectiva para todos estos problemas: un container de Docker se puede ejecutar de la misma manera en cualquiera de estos entornos.

Al principio me costó entender los conceptos de Docker, contenedor, imagen, Dockerfile, etc.. y como relacionarlos. Una vez conseguí entender cómo manejarlos, entendí su potencial en la construcción de los distintos entornos necesarios para la creación, pruebas y puesta en producción de una aplicación.

Una de las cosas más fáciles que he encontrado es la búsqueda de imágenes docker para poder realizar mis pruebas iniciales. Dada la popularidad de Docker, se pueden encontrar en la red muchos lugares de los que descargar imágenes, aunque he usado **Docker Hub**(plataforma de compartición de Docker donde se puede encontrar casi cualquier sistema o servicio que necesitemos como una imagen **Ubuntu**, **SUSELinux**, un **Wordpress**preinstalado, un **MySQL**ya pre-configurado, etc…), ya que algunas de las imágenes que circulan por la red pueden contener código malicioso.

Bajo me punto de vista la mayor complejidad, a la hora de usar Docker es la creación del fichero Dockerfile, afortunadamente existen muchos ejemplos en la web que ayudan a la hora de crearlo, pero una vez creado este fichero, en muy poco tiempo podemos crear tantos entornos como sea necesario.

La creación y manipulación de los ficheros docker-compose, una vez entendido el Dockerfile, es sencilla.

Como ya he comentado en otro capítulo he usado el driver de virtualbox para la creación de hosts, esta parta también ha sido bastante sencilla. La inicialización y gestión del Swarm me costó un poco más hasta que me familiarizé con los comandos necesarios para su gestión.

Mi conclusión final es que me alegro de haber elegido este tema para mi proyecto fin de grado ya que aunque al principio creí que no lo iba a conseguir, al final me ha gustado, y creo que he adquirido los conocimientos base para poder encontrar un trabajo y seguir formándome en Docker ya que como he dicho anteriormente, creo que tiene mucho futuro y me gustaría seguir mi carrera profesional especializándome en esta tecnología.



## Trabajo futuro

# REFERENCIAS

1. Página Oficial de Docker
2. <https://docs.docker.com/get-started/part2/#define-a-container-with-a-dockerfile>