Docker crash course

Raul Estrada

Octubre, 2020

- Es posible que se haya sentido abrumado por la gran cantidad de desarrolladores que hablan sobre contenedores, máquinas virtuales, hipervisores y otra magia vudú relacionada con DevOps.
- Es hora de comprender qué es **C**ontainers **a**s a **S**ervice y por qué lo necesita.

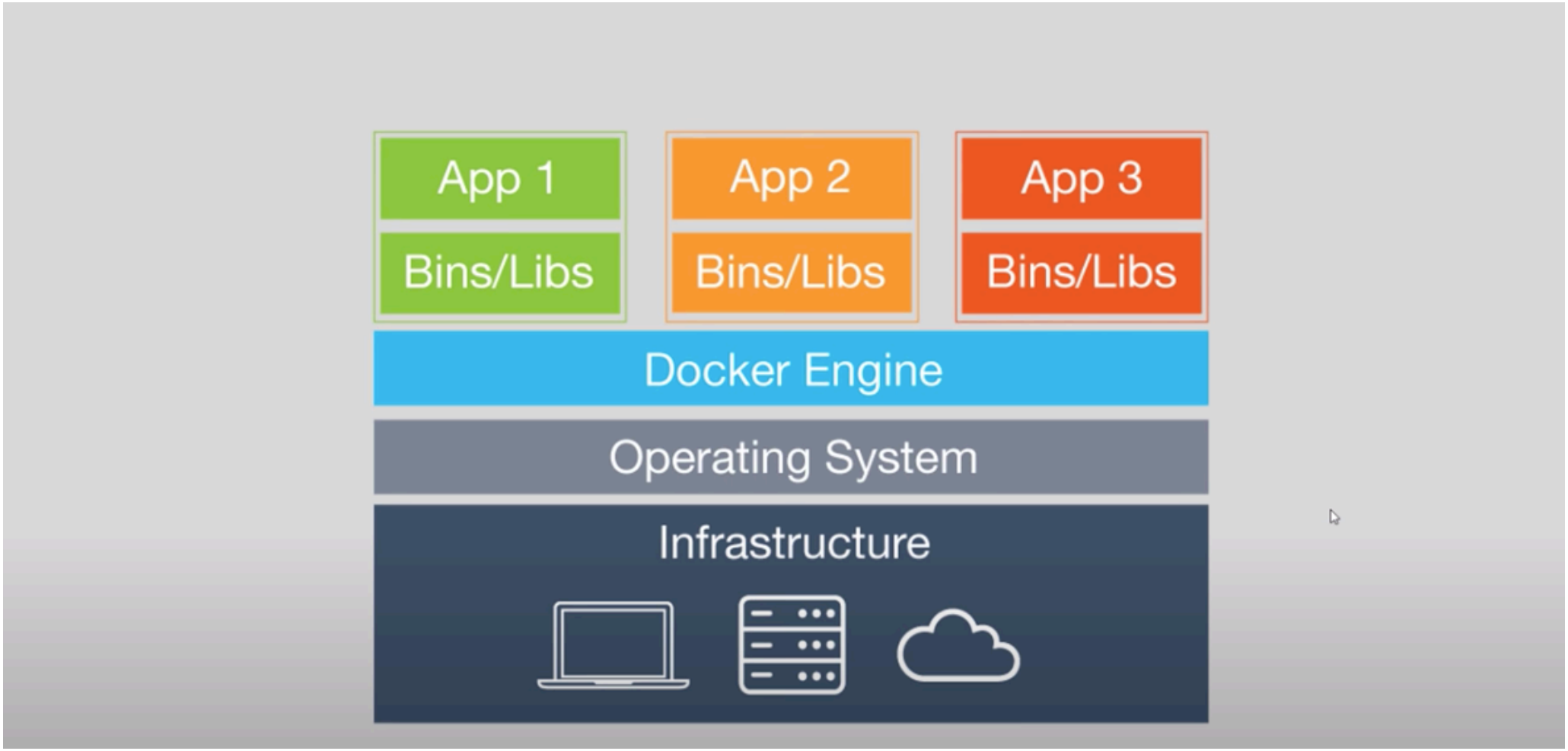
TL;DR

- ***"¿Por qué necesito esto"***:
  - Descripción de los términos clave
- - Por qué necesitamos CaaS y Docker
- **Quick Start**:
  - Instalación de Docker
  - Creación de un container
- **Escenario de la vida real**:
  - Creación de un contenedor nginx para alojar un sitio web estático
- - Aprender a usar herramientas de compilación para automatizar los comandos de Docker

¿Docker?

- Docker es un software para
  - Crear aplicaciones containerizadas.
  - El concepto de contenedor es ser un entorno pequeño y sin estado para ejecutar una pieza de software.

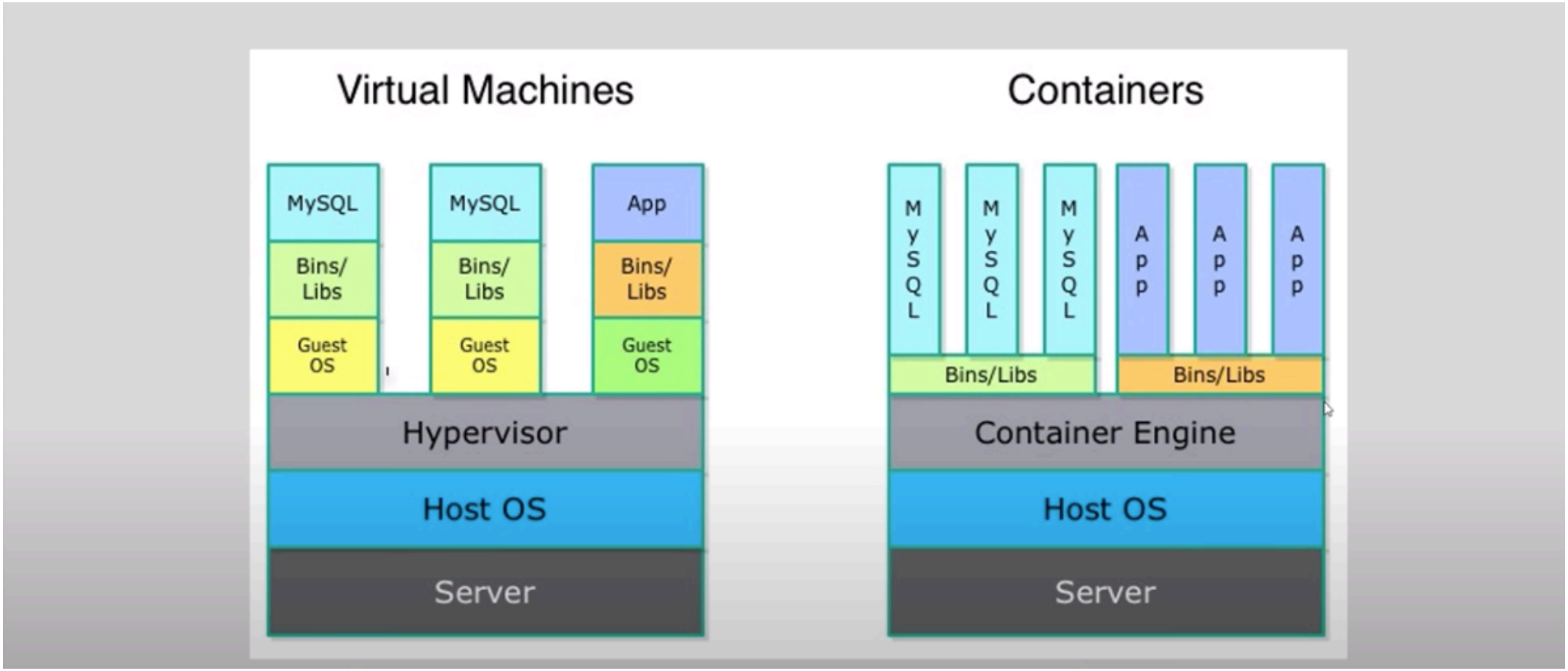
- Una imagen de contenedor es un paquete ligero, autónomo y ejecutable de un software que incluye todo lo necesario para ejecutarlo: código, entorno de ejecución, herramientas del sistema, bibliotecas del sistema, configuraciones.
- — Official Docker website


App 1
Bins/Libs
App 2
Bins/Libs
App 3
Bins/Libs
Docker Engine
Operating System
Infrastructure

Virtual machine?

Una máquina virtual (VM) es literalmente lo que su nombre indica

- Una versión virtual de una máquina real.
- Simula el hardware de una máquina dentro de una máquina más grande.
- Es decir, puede ejecutar muchas máquinas virtuales en un servidor más grande.
- Como en la película Inception.
- Lo que permite que las máquinas virtuales funcionen es un software genial llamado hipervisor


Virtual Machines
MySQL
Bins/ Libs
Guest OS
MySQL
Bins/ Libs
Guest OS
App
Bins/ Libs
Guest OS
Hypervisor
Host OS
Server
Containers
MySQL
MySQL
MySQL
App
App
App
Bins/Libs
Bins/Libs
Container Engine
Host OS
Server

¿Hypervisor?

- ¡Muchos términos, lo sé!
- Las máquinas virtuales solo funcionan gracias al hipervisor.
- Es un software especial que permite que una máquina física albergue varias máquinas virtuales diferentes.
- Todas estas máquinas virtuales pueden ejecutar sus propios programas y parecerá que utilizan el hardware del host.
- Sin embargo, en realidad es el hipervisor el que asigna recursos a la máquina virtual.

Respondiendo mis preguntas…

- ¿Por qué realmente necesitamos CaaS?
- Hemos estado usando máquinas virtuales durante tanto tiempo,
- ¿cómo es que los contenedores de repente son tan buenos?
- Bueno, nadie dijo que las máquinas virtuales sean malas, simplemente son difíciles de administrar.
- DevOps es generalmente difícil y necesita una persona dedicada para hacer el trabajo todo el tiempo.
- Las máquinas virtuales ocupan una gran cantidad de almacenamiento y RAM, y su configuración es especializada.
- Sin mencionar que necesita una buena cantidad de experiencia para administrarlos de la manera correcta.

Mantra: Si lo haces más de una vez, lo automatizas

- Con Docker, se pueden abstraer todas las configuraciones oportunas y las configuraciones del entorno, y concentrarse en la codificación.
- Con Docker Hub, puedes capturar imágenes prediseñadas y comenzar a funcionar en una fracción del tiempo que tomaría con una VM normal.
- La mayor ventaja es la creación de un entorno homogéneo. En lugar de tener que instalar una lista de diferentes dependencias para ejecutar su aplicación, ahora solo necesita instalar una cosa, Docker.
- Al ser multiplataforma, todos los desarrolladores de su equipo trabajarán exactamente en el mismo entorno.
- Lo mismo se aplica a sus servidores de desarrollo, preparación y producción. Ahora, esto es genial. No más "funciona en mi máquina”.

Quick Start

- Comencemos con la instalación.
- Es increíble que pueda tener solo una pieza de software instalada en su máquina de desarrollo y aún así estar seguro de que todo funcionará bien. Docker es, literalmente, todo lo que necesita.

Instalación de Docker

- Por suerte, el proceso de instalación es muy sencillo. Cómo lo haces en Ubuntu.
- $ sudo apt-get update
  $ sudo apt-get install -y docker.io

- Eso es todo lo que necesitas. Para asegurarse de que se esté ejecutando, ejecutar este otro comando..

    $ sudo systemctl status docker

Debe regresar una salida como esta

```
docker.service - Docker Application Container Engine
Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset: enabled)
Active: active (running) since Sun 2020-10-18 12:42:17 CET; 4h 46min ago
Docs: https://docs.docker.com
Main PID: 2156 (dockerd)
Tasks: 26
Memory: 63.0M
CPU: 1min 57.541s
CGroup: /system.slice/docker.service
 ├─2156 /usr/bin/dockerd -H fd://
 └─2204 docker-containerd --config /var/run/docker/containerd/containerd.toml
```

- Si el servicio del sistema está detenido, puede ejecutar una combinación de dos comandos para activarlo y asegurarse de que se inicie en el arranque.

```
$ sudo systemctl start docker && sudo systemctl enable docker
```

- Eso es todo, estás listo para comenzar.
- Con la instalación básica de Docker, deberá ejecutar el comando docker como sudo. Sin embargo, puede agregar su usuario al grupo de Docker y podrá ejecutar el comando sin sudo

```
$ sudo usermod -aG docker ${USER}
$ su - ${USER}
```

- Al ejecutar estos comandos, su usuario se agregará al grupo de Docker. Para verificar esto, ejecute

```
$ id -nG
```

si obtiene un resultado con su nombre de usuario en la lista, tenga la seguridad de que hizo todo bien.

Construyendo un contenedor

- Teniendo Docker instalado y en ejecución, podemos seguir adelante y jugar un poco.
- Los cuatro primeros comandos que necesita para comenzar a ejecutar Docker son:
  - **create**: crea un contenedor a partir de una imagen
  - **ps**: enumera los contenedores en ejecución, opcional -a marca para enumerar todos los contenedores
  - **start**: inicia un contenedor creado
  - **attach**: redirecciona la entrada y salida estándar del terminal a un contenedor en ejecución, lo que literalmente lo conecta al contenedor como lo haría con cualquier máquina virtual

- Empecemos con algo pequeño. Tomaremos una imagen de Ubuntu del Docker Hub y crearemos un contenedor a partir de él

  $ docker create **-it** ubuntu:16.04 **bash**

- Estamos agregando -it como una opción para darle al contenedor una terminal integrada, para que podamos conectarnos a él, mientras también le decimos que ejecute el comando bash, para que obtengamos una interfaz de terminal adecuada.

- Al especificar ubuntu: 16.04, extraemos la imagen de Ubuntu, con la etiqueta de versión 16.04, del Docker Hub.

- Una vez que haya ejecutado el comando de creación, continúe y verifique que se creó el contenedor.

  $ docker ps **–a**

- La lista debe verse como esto

CONTAINERID	IMAGE	COMMAND	CREATED	STATUS PORTS	NAMES
7643dba89904	ubuntu:16.04	"bash"	X min ago	Created	name

- Perfecto, el contenedor está creado y listo para ser iniciado.

- Ejecutar el contenedor es tan simple como darle al comando de inicio el **ID** del contenedor.
- $ docker start 7643dba89904
- Una vez más, comprobar si el contenedor se está ejecutando, pero ahora sin el indicador –a
- $ docker ps
- Si es así, adelante y dele attach

  $ docker attach 7643dba89904

- El cursor cambia.
- ¿Por qué? Porque acabas de entrar al contenedor.
- Cuan genial es esto que ahora puedes ejecutar cualquier comando bash al que estés acostumbrado en Ubuntu, como si fuera una instancia que se ejecuta en la nube.

- Adelante, pruébelo

  $ ls
- Funcionará bien y enumerará todos los directorios
- Incluso $ ll funciona
- Este pequeño contenedor Docker es todo lo que necesita
- Es su propio pequeño patio de recreo, donde puede desarrollar, probar o lo que quiera
- Si desea salir del contenedor, todo lo que necesita hacer es escribir literalmente exit
- El contenedor se detendrá y podrá listarlo nuevamente escribiendo

  $ docker ps -a.

- ***Nota**: Todos los contenedores Docker se ejecutan con sudo de forma predeterminada, lo que significa que el comando sudo no existe. Cada comando que ejecute se ejecutará automáticamente con privilegios de sudo*

Escenario de la vida real

- Es hora de entrar en cosas reales.
- Esto es lo que se usa en la vida real para sus propios proyectos y aplicaciones de producción

¿Los contenedores son stateless?

- Cada contenedor está aislado y sin estado, lo que significa que una vez que elimine un contenedor, el contenido se eliminará **para siempre**.

```
$ docker rm 7643dba89904
```

- Entonces, ¿Cómo persistimos nuestros datos?

  Se hace uso de los “volúmenes”

- Los volúmenes le permiten mapear directorios en su máquina host a directorios dentro del contenedor. Así es cómo.

  $ docker create -it **-v $(pwd):/var/www** ubuntu:latest bash

- Al crear un nuevo contenedor, agregamos la bandera -v para especificar qué volumen crear y conservar.
- Este comando vincula el directorio de trabajo actual en su máquina al directorio /var/www dentro del contenedor.

- Una vez que inicie el contenedor con el comando

  $ docker start <container_id>
- podrá editar el código en la máquina host y ver los cambios inmediatamente en el contenedor.
- Le brinda la capacidad de conservar datos para varios casos de uso, desde guardar imágenes hasta almacenar archivos de base de datos y, por supuesto, para fines de desarrollo donde necesita capacidades de recarga en vivo.
- También puede ejecutar los comandos de creación e inicio con el comando de ejecución.
  - $ docker ejecutar -it -d ubuntu: 16.04 bash
- La única adición es la bandera -d que le dice al contenedor que se ejecute separado, en segundo plano, lo que significa que puede continuar y adjuntarlo de inmediato.

- Cree un nuevo directorio, asígnele el nombre **myapp** para su comodidad.
- Todo lo que necesita es crear un archivo **index.html** simple en el directorio myapp y pegarlo.

```
<!-- index.html --><html>
        <head>
        <link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.5/css/bootstrap.min.css"
rel="stylesheet" integrity="sha256-MfvZlkHCEqatNoGiOXveE8FIwMzZg4W85qfrfIFBfYc= sha512-
dTfge/zgoMYpP7QbHy4gWMEGsbsdZeCXz7irItjcC3sPUFtf0kuFbDz/ixG7ArTxmDjLXDmezHubeNikyKG
VyQ==" crossorigin="anonymous">
        <title>Docker Quick Start</title>
        </head>
        <body>
                <div class="container">
                        <h1>Hello Docker</h1>
                        <p>This means the nginx server is working.</p>
                </div>
        </body>
</html>
```

- Tenemos una página web genérica, con algún texto de encabezado. Lo que queda es ejecutar un contenedor nginx.

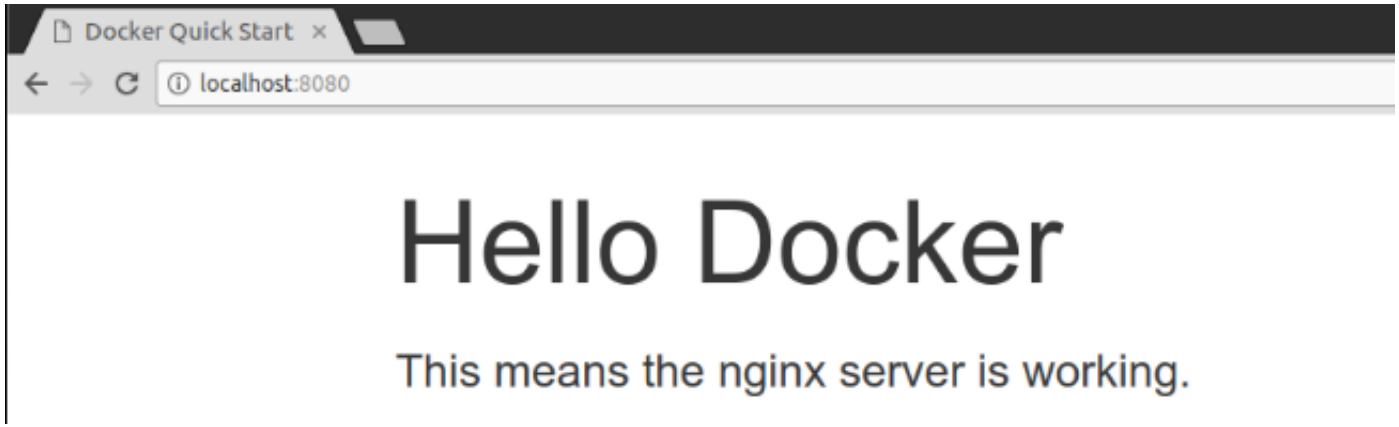
  ```
  $ docker run --name webserver -v $ (pwd):/usr/share/nginx/html -d -p 8080: 80 nginx
  ```

- Aquí puede ver que estamos tomando una imagen nginx de Docker Hub para que podamos obtener una configuración instantánea de nginx.
- La configuración del volumen es similar a lo que hicimos anteriormente, solo apuntamos al directorio predeterminado donde nginx aloja archivos HTML.
- Lo nuevo es la opción --name que configuramos en webserver y la opción -p 8080: 80. Esto asigna el puerto 80 del contenedor al puerto 8080 en la máquina host.
- Por supuesto, no olvide ejecutar el comando mientras se está en el directorio **myapp**.

- Compruebe si el contenedor está funcionando con

  $ docker ps

- Abra una ventana del navegador.
- Vaya a http://localhost:8080 y véalo en ejecución.



- Simple como eso. Tenemos un servidor web nginx en funcionamiento con solo un par de comandos.

- Siéntase libre y edite algo en el index.html. Vuelva a cargar la página y verá que el contenido ha cambiado.

- ***Nota:*** Puede detener un contenedor en ejecución con el comando stop. Asegúrese de detener el contenedor antes de continuar.

  $ docker stop <container_id>

¿cómo hacer mi vida más fácil?

- Si necesito hacer algo dos veces, prefiero automatizarlo.
- Junto con el archivo **index.html**, agregue un **Dockerfile**. Su nombre es literalmente **Dockerfile**, sin extensión de archivo.

```
# Dockerfile
FROM nginx:alpine
VOLUME /usr/share/nginx/html
EXPOSE 80
```

- El **Dockerfile** es literalmente la configuración de compilación para las imágenes de Docker.
- La clave está en las imágenes. Con esto estamos especificando que queremos tomar la imagen **nginx: alpine** como base para nuestra imagen, crear un volumen y exponer el puerto 80.

- Para construir una imagen tenemos el comando build

  $ docker build . -t webserver:v1

- El "." es para especificar donde está localizado nuestro **Dockerfile** y usarlo para construir la imagen, la bandera -t es el tag de la imagen.

- Esta imagen se conocerá como servidor web: v1.

- Con este comando, no extrajimos inmediatamente una imagen de Docker Hub, sino que creamos nuestra propia imagen. Para enumerar todas sus imágenes, use el comando *images*

  $ docker images

- Ahora, queremos correr la imagen que creamos

  $ docker run -v $(pwd):/usr/share/nginx/html -d -p 8080:80 **webserver:v1**

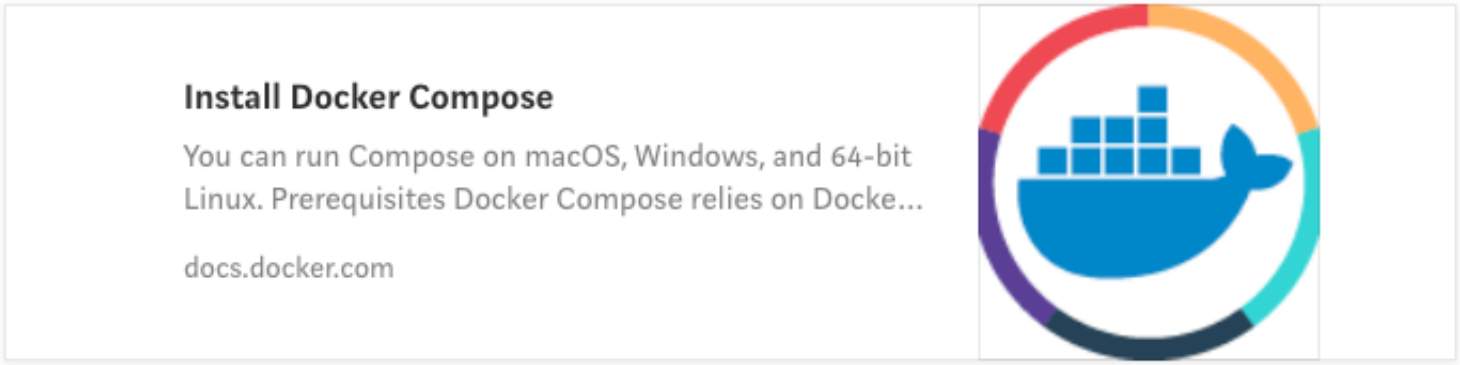
- El poder del **Dockerfile** es la customización que puedes darle al contenedor

- Pueden preconstruirse imagenes basado en la necesidades

- Pero, si en realidad no nos gustan las tareas repetitivas, siempre se puede instalar **docker-compose**.

Docker-compose?

- It lets you both build and run the container in one command.
- But, what's even more important is that you can build a whole cluster of containers and configure them by using docker-compose.
- Jump over to their install page and get it installed on your machine, for your respective operating system.



- Back in the terminal run
  - $ docker-compose --version
- it outputs something back to you. If it does, you're set.
- Let's get start with some compositions
- Alongside the **Dockerfile** add another file named **docker-compose.yml** and paste this snippet in.

```
# docker-compose.yml
version: '2'
services:
    webserver:
        build: .
        ports:
        - "8080:80"
        volumes:
        - .:/usr/share/nginx/html
```

- Be careful with the indentations, otherwise it won't work properly.

- That's it. What's left is to run docker-compose.
  $ docker-compose up (-d)
- ***Note****: The -d signals docker-compose to run detached, then you can use*

  $ docker-compose ps

to see what's currently running, or stop docker-compose with

  $ docker-compose stop

- Docker will:
  - build the image from the **Dockerfile** in the current directory (.)
  - map the ports as we did above
  - as well as share the volumes

- See what's happening? The exact same thing we did with the build and run commands, instead now only running one command,
  $ docker-compose up

- Jump back to the browser and you'll see everything works just as it did before.

- The only difference is that you've now escaped the tedious work with writing commands in the terminal, replacing them with two configuration files, the **Dockerfile** and the **docker-compose.yml** file.

- Both of these can be added to your Git repository, meaning every contributor to your project can have the development environment up and running in a fraction of the time it would take to install dependencies manually.

- Why is this important? Because it will always work in production as expected. The exact same network of containers will be spun up on the production server!

- To wrap this section up, go ahead and list all the containers once again.

  $ docker ps –a

- If you ever want to *delete* a container you can run the rm command I mentioned above

- Use the rmi command for deleting images.

  $ docker rmi <image_id>

- Try your best to not leave residual containers lying around and make sure to delete them if you don't need them.

A broader perspective?

- Docker is not the only container technology.
- Docker is merely the most widely used containerization option we have since 2014.

Container Orchestration?

- We've only talked about Docker and Docker-compose as a tool for creating networks of containers.
- Managing all that and ensuring maximum up time is where orchestration comes into play.
- As the number of containers grow we need a way of automating the various DevOps tasks we usually do.
- Orchestration is what helps us out with provisioning hosts, creating or removing containers when you need to scale out or down, re-creating failed containers, networking containers, and much more.
- All the big ones out there use Google's solution called Kubernetes or Docker's own Swarm Mode.

Finally

- A Docker container really is just a tiny VM where you can do anything you like, from development, staging, testing to hosting production applications.

- The homogeneous nature of Docker is like magic for production environments.

- It will ease the stresses of deploying applications and managing servers.

- Because now you'll know for sure whatever works locally will work in the cloud. That's what I call peace of mind.

- No more hearing the infamous sentence we have all heard one too many times.

  "Well it works on my machine…"