Comencemos, Parte 2: Contenedores

Tiempo estimado de lectura: 13 minutos

Requisitos previos

- Instale Docker versión 1.13 o superior (https://docs.docker.com/engine/installation/) .
- Lea la orientación en la Parte 1 (https://docs.docker.com/get-started/) .
- Dale a tu entorno una prueba rápida para asegurarte de que estás listo:

```
docker run hello-world
```

Introducción

Es hora de comenzar a construir una aplicación al estilo Docker. Comenzamos en la parte inferior de la jerarquía de dicha aplicación, que es un contenedor, que cubrimos en esta página. Por encima de este nivel hay un servicio que define cómo se comportan los contenedores en la producción, cubierto en la Parte 3 (https://docs.docker.com/get-started/part3/) . Finalmente, en el nivel superior se encuentra la pila, definiendo las interacciones de todos los servicios, cubiertos en la Parte 5 (https://docs.docker.com/get-started/part5/) .

- Apilar
- Servicios
- **Contenedor** (usted está aquí)

Su nuevo entorno de desarrollo

En el pasado, si comenzaba a escribir una aplicación de Python, su primera tarea era instalar un tiempo de ejecución de Python en su máquina. Pero eso crea una situación en la que el entorno de su máquina debe ser perfecto para que su aplicación se ejecute como se espera y también debe coincidir con su entorno de producción.

Con Docker, puede tomar un tiempo de ejecución de Python portátil como imagen, sin necesidad de instalación. Luego, su compilación puede incluir la imagen base de Python junto con el código de su aplicación, asegurando que su aplicación, sus dependencias y el tiempo de ejecución, viajen juntos.

Estas imágenes portátiles están definidas por algo llamado a `Dockerfile` .

Definir un contenedor con `Dockerfile`

`Dockerfile` define lo que sucede en el ambiente dentro de su contenedor. El acceso a recursos como interfaces de red y unidades de disco se virtualiza dentro de este entorno, que está aislado del resto de su sistema, por lo que necesita asignar puertos al mundo exterior y especificar qué archivos desea "copiar" en ese ambiente. Sin embargo, después de hacer eso, puede esperar que la compilación de su aplicación definida en esto se `Dockerfile` comporte exactamente igual donde sea que se ejecute.

`Dockerfile`

Crea un directorio vacío Cambie los directorios (`cd`) al nuevo directorio, cree un archivo llamado `Dockerfile` , copie y pegue el siguiente contenido en ese archivo y guárdelo. Tome nota de los comentarios que explican cada declaración en su nuevo archivo Docker.

```
# Use an official Python runtime as a parent image
FROM python:2.7-slim

# Set the working directory to /app
WORKDIR /app

# Copy the current directory contents into the container at /app
ADD . /app

# Install any needed packages specified in requirements.txt
RUN pip install --trusted-host pypi.python.org -r requirements.txt

# Make port 80 available to the world outside this container
EXPOSE 80

# Define environment variable
ENV NAME World

# Run app.py when the container launches
CMD ["python", "app.py"]
```

¿Estás detrás de un servidor proxy?

Los servidores proxy pueden bloquear las conexiones a su aplicación web una vez que esté en funcionamiento. Si está detrás de un servidor proxy, agregue las siguientes líneas a su archivo Docker, utilizando el `ENV` comando para especificar el host y el puerto para sus servidores proxy:

```
# Set proxy server, replace host:port with values for your servers
ENV http_proxy host:port
ENV https_proxy host:port
```

Agregue estas líneas antes de la llamada para `pip` que la instalación tenga éxito.

Esto se `Dockerfile` refiere a un par de archivos que aún no hemos creado, concretamente `app.py` y `requirements.txt` . Vamos a crear los siguientes.

La aplicación en sí

Cree dos archivos más `requirements.txt` y `app.py` , y colóquelos en la misma carpeta con `Dockerfile` . Esto completa nuestra aplicación, que como puede ver es bastante simple. Cuando lo anterior `Dockerfile` se basa en una imagen, `app.py` y `requirements.txt` está presente debido a que `Dockerfile` 's `ADD` de comandos, y la salida de `app.py` es accesible a través de HTTP gracias al `EXPOSE` mando.

`requirements.txt`

```
Flask
Redis
```

app.py

```
from flask import Flask
from redis import Redis, RedisError
import os
import socket

# Connect to Redis
redis = Redis(host="redis", db=0, socket_connect_timeout=2, socket_timeout=2)

app = Flask(__name__)

@app.route("/")
def hello():
    try:
        visits = redis.incr("counter")
    except RedisError:
        visits = "<i>cannot connect to Redis, counter disabled</i>"

    html = "<h3>Hello {name}!</h3>" \
           "<b>Hostname:</b> {hostname}<br/>" \
           "<b>Visits:</b> {visits}"
    return html.format(name=os.getenv("NAME", "world"), hostname=socket.gethos

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=80)
```

Ahora vemos que `pip install -r requirements.txt` instala las bibliotecas de Flask y Redis para Python, y la aplicación imprime la variable de entorno `NAME` , así como el resultado de una llamada a `socket.gethostname()` . Finalmente, dado que Redis no se está ejecutando (ya que solo hemos instalado la biblioteca de Python, y no Redis en sí misma), deberíamos esperar que el intento de usarlo aquí falle y genere el mensaje de error.

> **Nota** : Al acceder al nombre del host cuando está dentro de un contenedor, se recupera el ID del contenedor, que es como el ID del proceso de un ejecutable en ejecución.

¡Eso es! No necesita Python ni nada en `requirements.txt` su sistema, ni construir o ejecutar esta imagen los instala en su sistema. No parece que hayas configurado realmente un entorno con Python y Flask, pero lo has hecho.

Crea la aplicación

Estamos listos para construir la aplicación. Asegúrate de que todavía estás en el nivel superior de tu nuevo directorio. Esto es lo que `ls` debe mostrar:

```
$ ls
Dockerfile              app.py                  requirements.txt
```

Ahora ejecuta el comando de compilación. Esto crea una imagen Docker, que vamos a etiquetar usando `-t` para que tenga un nombre descriptivo.

```
docker build -t friendlyhello .
```

¿Dónde está tu imagen construida? Está en el registro de imagen de Docker local de tu máquina:

```
$ docker image ls

REPOSITORY            TAG                 IMAGE ID
friendlyhello         latest              326387cea398
```

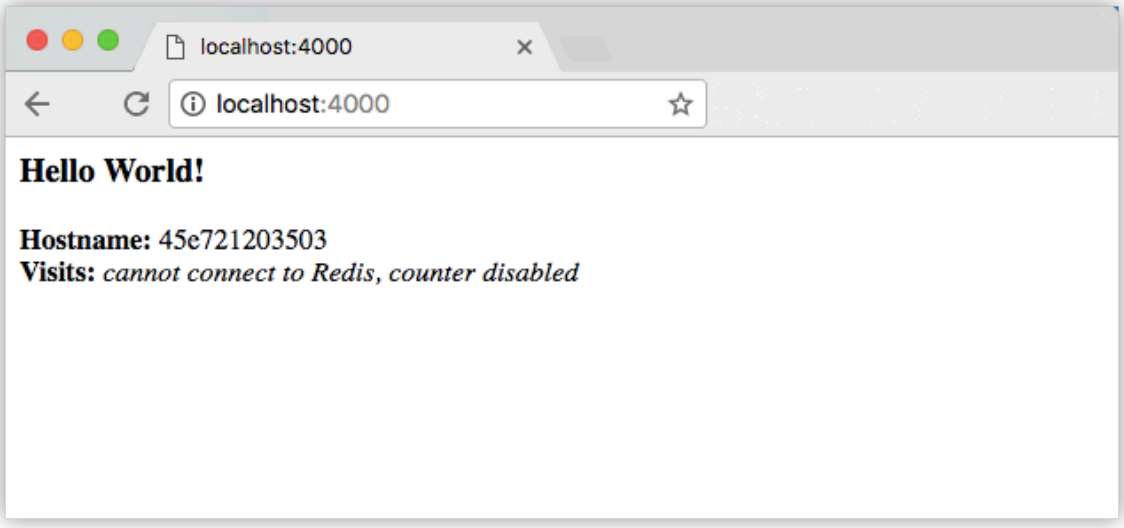
Ejecuta la aplicación

Ejecute la aplicación, asignando el puerto 4000 de su máquina al puerto 80 del contenedor utilizando `-p` :

```
docker run -p 4000:80 friendlyhello
```

Debería ver un mensaje en el que Python está publicando su aplicación `http://0.0.0.0:80` . Pero ese mensaje viene del interior del contenedor, que no sabe que usted asignó el puerto 80 de ese contenedor a 4000, creando la URL correcta `http://localhost:4000` .

Vaya a esa URL en un navegador web para ver el contenido mostrado en una página web.



> **Nota** : Si está utilizando Docker Toolbox en Windows 7, use la IP Docker Machine en lugar de `localhost` . Por ejemplo, http://192.168.99.100:4000/. Para encontrar la dirección IP, use el comando `docker-machine ip` .

También puede usar el `curl` comando en un shell para ver el mismo contenido.

```
$ curl http://localhost:4000

<h3>Hello World!</h3><b>Hostname:</b> 8fc990912a14<br/><b>Visits:</b> <i>canno
```

Esta reasignación de puerto `4000:80` es para demostrar la diferencia entre lo que está `EXPOSE` dentro `Dockerfile` y lo que `publish` usa `docker run -p` . En los pasos posteriores, simplemente asignamos el puerto 80 en el host al puerto 80 en el contenedor y lo usamos `http://localhost` .

Pulse `CTRL+C` en su terminal para salir.

> **En Windows, detenga explícitamente el contenedor**
>
> En sistemas Windows, `CTRL+C` no detiene el contenedor. Por lo tanto, primero escriba `CTRL+C` para recuperar el mensaje (o abra otro shell), luego escriba `docker container ls` para enumerar los contenedores en ejecución, y luego `docker container stop <Container NAME or ID>` para detener el contenedor. De lo contrario, recibirá una respuesta de error del daemon cuando intente volver a ejecutar el contenedor en el siguiente paso.

Ahora ejecutemos la aplicación en segundo plano, en modo separado:

```
docker run -d -p 4000:80 friendlyhello
```

Obtienes la identificación larga del contenedor para tu aplicación y luego la regresas a tu terminal. Su contenedor se ejecuta en segundo plano. También puede ver el identificador de contenedor abreviado con `docker container ls` (y ambos funcionan indistintamente al ejecutar comandos):

```
$ docker container ls
CONTAINER ID        IMAGE               COMMAND             CREATED
1fa4ab2cf395        friendlyhello       "python app.py"     28 seconds ago
```

Tenga en cuenta que `CONTAINER ID` coincide con lo que está encendido `http://localhost:4000` .

Ahora use `docker container stop` para finalizar el proceso, usando el `CONTAINER ID` , como ese:

```
docker container stop 1fa4ab2cf395
```

Comparte tu imagen

Para demostrar la portabilidad de lo que acabamos de crear, carguemos nuestra imagen construida y ejecútela en otro lugar. Después de todo, debe saber cómo enviar datos a los registros cuando desee implementar contenedores en producción.

Un registro es una colección de repositorios, y un repositorio es una colección de imágenes, algo así como un repositorio de GitHub, excepto que el código ya está construido. Una cuenta en un registro puede crear muchos repositorios. La `docker` CLI usa el registro público de Docker por defecto.

> **Nota** : utilizamos el registro público de Docker aquí simplemente porque es gratuito y preconfigurado, pero hay muchos públicos entre los que puede elegir, e incluso puede configurar su propio registro privado mediante el Registro de confianza Docker (https://docs.docker.com/datacenter/dtr/2.2/guides/) .

Inicia sesión con tu ID de Docker

Si no tiene una cuenta de Docker, inscríbase para obtener una en cloud.docker.com (https://cloud.docker.com/) . Tome nota de su nombre de usuario.

Inicie sesión en el registro público de Docker en su máquina local.

```
$ docker login
```

Etiquetar la imagen

La notación para asociar una imagen local con un repositorio en un registro es `username/repository:tag` . La etiqueta es opcional, pero se recomienda, ya que es el mecanismo que usan los registros para dar una versión a las imágenes de Docker. Proporcione el repositorio y etiquete nombres significativos para el contexto, como `get-started:part2` . Esto coloca la imagen en el `get-started` repositorio y la etiqueta como `part2` .

Ahora, ponlo todo junto para etiquetar la imagen. Ejecute `docker tag image` con su nombre de usuario, repositorio y nombres de etiquetas para que la imagen se cargue en su destino deseado. La sintaxis del comando es:

```
docker tag image username/repository:tag
```

Por ejemplo:

```
docker tag friendlyhello john/get-started:part2
```

Ejecute la imagen de la ventana acoplable (https://docs.docker.com/engine/reference/commandline/image_ls/) para ver su imagen recién etiquetada. (También puedes usar `docker image ls`)

```
$ docker image ls
REPOSITORY                TAG                 IMAGE ID            CREATED
friendlyhello             latest              d9e555c53008        3 minutes ago
john/get-started          part2               d9e555c53008        3 minutes ago
python                    2.7-slim            1c7128a655f6        5 days ago
...
```

Publica la imagen

Suba su imagen etiquetada al repositorio:

```
docker push username/repository:tag
```

Una vez completado, los resultados de esta carga están disponibles públicamente. Si inicia sesión en Docker Hub (https://hub.docker.com/) , verá la nueva imagen allí, con su comando de extracción.

Tire y ejecute la imagen desde el repositorio remoto

A partir de ahora, puede usar `docker run` y ejecutar su aplicación en cualquier máquina con este comando:

```
docker run -p 4000:80 username/repository:tag
```

Si la imagen no está disponible localmente en la máquina, Docker la extrae del repositorio.

```
$ docker run -p 4000:80 john/get-started:part2
Unable to find image 'john/get-started:part2' locally
part2: Pulling from john/get-started
10a267c67f42: Already exists
f68a39a6a5e4: Already exists
9beaffc0cf19: Already exists
3c1fe835fb6b: Already exists
4c9f1fa8fcb8: Already exists
ee7d8f576a14: Already exists
fbccdcced46e: Already exists
Digest: sha256:0601c866aab2adcc6498200efd0f754037e909e5fd42069adeff72d1e243906
Status: Downloaded newer image for john/get-started:part2
 * Running on http://0.0.0.0:80/ (Press CTRL+C to quit)
```

No importa dónde se `docker run` ejecute, extrae su imagen, junto con Python y todas las dependencias de `requirements.txt` , y ejecuta su código. Todo viaja en un paquete pequeño y ordenado, y no necesita instalar nada en el equipo host para que Docker lo ejecute.

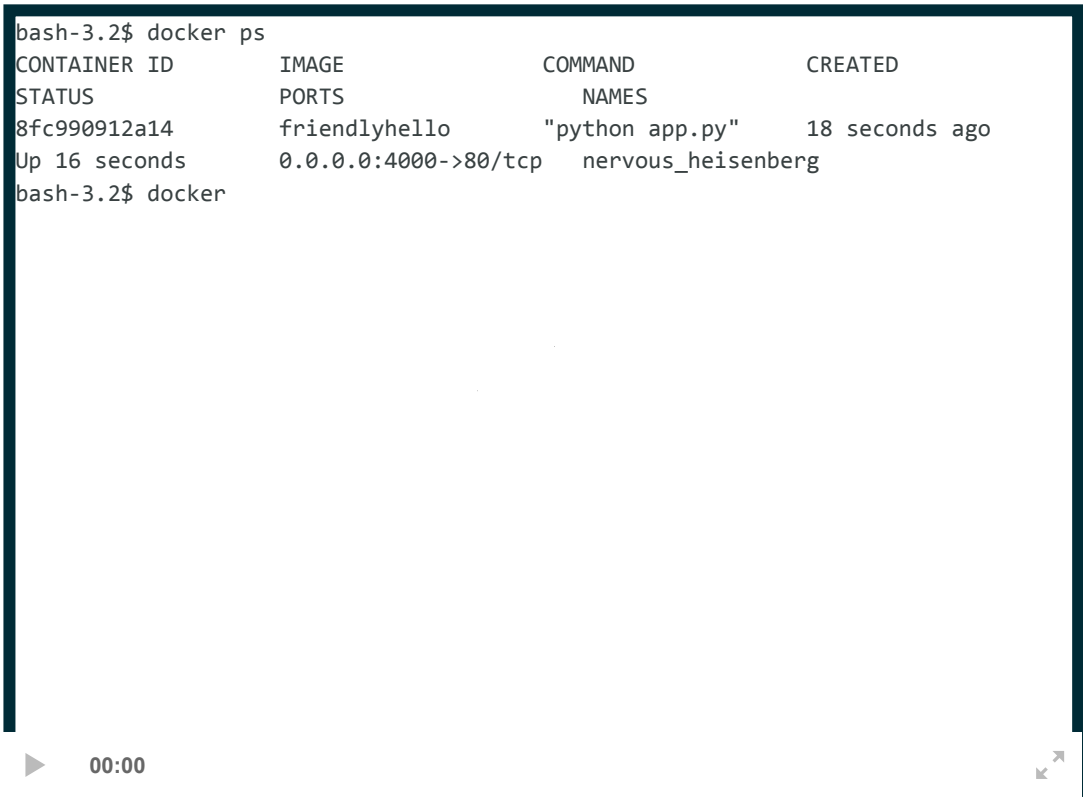
Conclusión de la segunda parte

Eso es todo por esta página. En la siguiente sección, aprendemos a escalar nuestra aplicación ejecutando este contenedor en un **servicio** .

Continúa a la Parte 3 » (https://docs.docker.com/get-started/part3/)

Resumen y hoja de trucos (opcional)

Aquí hay una grabación final de lo que se cubrió en esta página (https://asciinema.org/a/blkah0l4ds33tbe06y4vkme6g) :



Recorded with asciinema

Aquí hay una lista de los comandos Docker básicos de esta página, y algunos relacionados si desea explorar un poco antes de continuar.

```
docker build -t friendlyhello .  # Create image using this directory's Dockerf
docker run -p 4000:80 friendlyhello  # Run "friendlyname" mapping port 4000 to
docker run -d -p 4000:80 friendlyhello         # Same thing, but in detached m
docker container ls                                # List all running containe
docker container ls -a             # List all containers, even those not runni
docker container stop <hash>           # Gracefully stop the specified contain
docker container kill <hash>         # Force shutdown of the specified contain
docker container rm <hash>        # Remove specified container from this machi
docker container rm $(docker container ls -a -q)         # Remove all containe
docker image ls -a                             # List all images on this machi
docker image rm <image id>            # Remove specified image from this machi
docker image rm $(docker image ls -a -q)   # Remove all images from this machi
docker login             # Log in this CLI session using your Docker credentia
docker tag <image> username/repository:tag  # Tag <image> for upload to regist
docker push username/repository:tag            # Upload tagged image to regist
docker run username/repository:tag                   # Run image from a regist
```

contenedores (https://docs.docker.com/glossary/?term=containers), python (https://docs.docker.com/glossary/?term=python), código (https://docs.docker.com/glossary/?term=code), codificación (https://docs.docker.com/glossary/?term=coding), compilación (https://docs.docker.com/glossary/?term=build), empuje

(https://docs.docker.com/glossary/?term=push) , ejecución (https://docs.docker.com/glossary/?term=run)