2. Despliegue de software

Aplicación en Python

Usando docker-compose

Víctor Ponz

Institut Educació Secundària
El Caminàs

Curso de especialización en Ciberseguridad

Contenidos

Qué aprenderemos

- a usar `docker`
- a usar `docker-compose` para crear una aplicación basada en microservicios
- a crear `tags` para cada nueva característica en *git* y a hacer un mezclado con la rama principal

Hello World!

Para empezar, crea un nuevo repositorio en GitHub y clónalo en local

Para esta práctica hemos de crear la siguiente estructura:

```
[victorponz@localhost Using Docker]$ tree identidock
identidock
├── app
│   └── identidock.py
└── Dockerfile
```

Figura 1: Estructura ficheros

Crea esta estructura de directorio y súbela a git

Creamos la primera versión de una pequeña aplicación que, de momento, va a mostrar el famoso «**Hello World!**». Este es el contenido del `Dockerfile`:

```
FROM python:3.4
RUN pip install Flask==0.10.1
WORKDIR /app
COPY app /app
CMD ["python", "identidock.py"]
```

y ahora creamos el archivo `identidock.py`

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello World!\n'
```

```
if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0')
```

Este programa :

- (1) usa Flask para desarrollar la aplicación
- (4) Crea la ruta / asociada a una URL y el método `hello_word` que responde a dicha ruta
- (9) Ejecuta el servidor web de python

> Según la Wikipedia Flask es un framework minimalista escrito en Python que permite crear aplicaciones web rápidamente y con un mínimo número de líneas de código.

Vamos a generar la imagen y a correr el contenedor:

```
docker build -t identidock .
docker run -d -p 5000:5000 identidock
```

Una vez iniciado el contenedor, usaremos el comando `curl` para realizar una petición:

```
curl localhost:5000
```

que nos devuelve:

```
Hello World!
```

El problema con este tipo de flujo de trabajo es que cada vez que se hace una modificación en el código se debe parar y reiniciar el contenedor. Se puede mejorar haciendo un *bind mount* del directorio local app de tal forma que quedaría de esta forma:

```
docker run -d -p 5000:5000 -v "$(pwd)"/app:/app identidock
```

Este argumento `-v $(pwd)/app:/app` hace que la carpeta app del host pase al contenedor ya montada en el directorio `/app` del mismo.

Pero antes de continuar debemos haber parado el contenedor creado anteriormente. Una forma fácil es usar el contenedor que devuelve el comando `$(docker ps -lq)` que mediante el flag `l` lista sólo el último lanzado.

```
docker stop $(docker ps -lq)
docker rm $(docker ps -lq)
```

Si todo ha ido bien, actualiza tu repositorio remoto

Identicons

Vamos a convertir la aplicación anterior en una aplicación web que, dado un nombre de usuario, genere un identicon

El primer caso es modificar nuestra aplicación para que muestre un formulario en el que se pueda escribir.

Reemplaza el fichero `identidock.py` con el siguiente contenido:

```
from flask import Flask
app = Flask(__name__)
default_name = 'Víctor Ponz'
@app.route('/')
def get_identicon():
    name = default_name
    header = '<html><head><title>Identidock</title></head><body>'
    body = '''<form method="POST">
            Hola <input type="text" name="name" value="{}">
            <input type="submit" value="submit">
            </form>
            <p>Te pareces a:
            <img src="/monster/monster.png"/>
            '''.format(name)
    footer = '</body></html>'
    return header + body + footer
if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0')
```

- en las líneas (9 al 14) , crea un formulario para poder ser enviado por POST

Este es el resultado

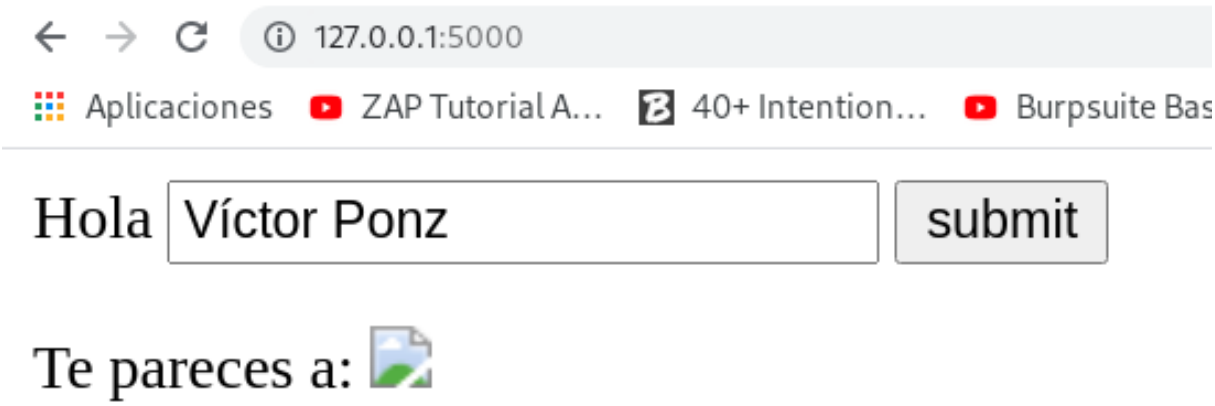

Figura 2: Formulario

De momento el enlace a la imagen aparece roto, pero es normal. En el siguiente apartado lo enlazaremos con el generador de identicon.

Sacando partido a un generador de imágenes

Vamos a iniciar un contenedor con una imagen de dnmonster que proporciona avatares únicos para una cadena dada. Para ello usaremos una imagen de este servicio más adelante que expone una API REST para generar las imágenes.

Primero modificamos el archivo `identidock.py` con el siguiente contenido:

```
from flask import Flask, Response
import requests
app = Flask(__name__)
salt = "UNIQUE_SALT"
default_name = 'Víctor Ponz'

@app.route('/')
def mainpage():
    name = default_name
    header = '<html><head><title>Identidock</title></head><body>'
    body = '''<form method="POST">
            Hola <input type="text" name="name" value="{0}">
            <input type="submit" value="submit">
```

```
            </form>
            <p>Te pareces a:
            <img src="/monster/monster.png"/>
            '''.format(name)
    footer = '</body></html>'

    return header + body + footer

@app.route('/monster/<name>')
def get_identicon(name):
    r = requests.get('http://dnmonster:8080/monster/' + name + '?size=80')
    image = r.content
    return Response(image, mimetype='image/png')

if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0')
```

- (23) La parte novedosa es el método `get_identicon` que se conecta a la URL http://dnmonster:8080 para obtener una imagen y devolverla en el objeto `Response` El puerto 8080 es donde escucha la imagen `dmonster`

Hemos de incluir el paquete `requests` en la instalación, por lo que modificamos el `Dockerfile`

```
FROM python:3.4
RUN pip install Flask==0.10.1 requests==2.5.1
WORKDIR /app
COPY app /app
CMD ["python", "identidock.py"]
```

Ya podemos instalar la imagen para lanzar el contenedor de **dmonster** y enlazarlo con nuestra aplicación.

Primero lo haremos mediante comandos y luego usaremos `docker-compose`.

```
docker run -d --name dnmonster amouat/dnmonster:1.0
```

Ahora iniciamos el contenedor de la aplicación casi de la misma manera que en anteriormente, excepto que agregamos el argumento `--link dnmonster: dnmonster` para conectar los contenedores. Esta es la magia que hace que la URL http://dnmonster: 8080 sea direccionable en el código Python

```
docker run -d -p 5000:5000 --link dnmonster:dnmonster identidock
```

Si ahora abres http://localhost:5000 deberías ver una página como la siguiente:

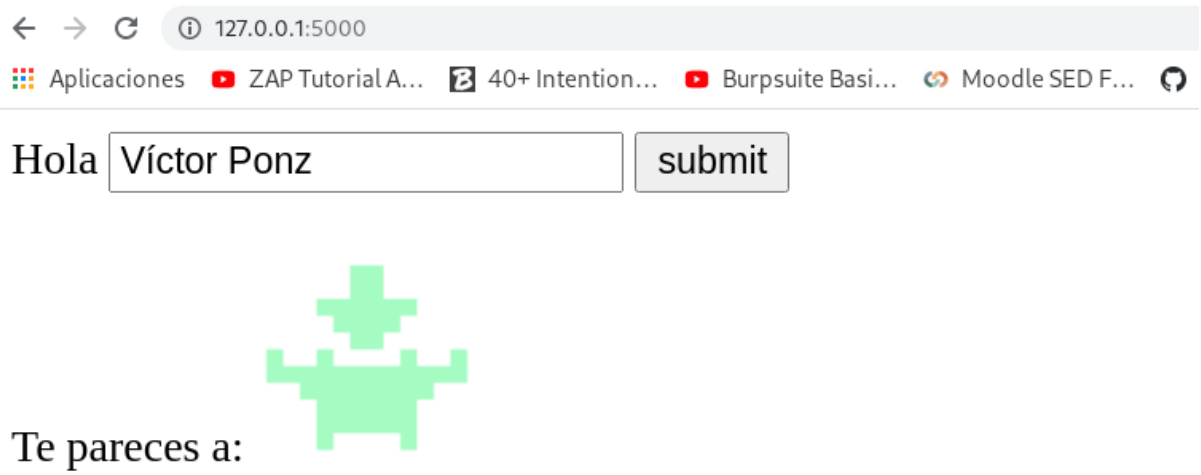

Figura 3: dnmonsters

No parece mucho, pero acabamos de generar nuestro primer icono de identificación.

El botón de enviar todavía está roto, por lo que en realidad no estamos usando ninguna entrada del usuario, pero lo arreglaremos en un minuto.

Primero, hagamos un archivo `Compose` (para que no tengamos que recordar todos esos comandos de ejecución).

Creamos `docker-compose.yml`:

```
identidock:
  build: .
  ports:
    - "5000:5000"
  volumes:
    - ./app:/app
  links:
    - dnmonster
dnmonster:
  ports:
    - "5080:8080"
  image: amouat/dnmonster:1.0
```

Mediante `Compose` definimos los distintos servicios que componen nuestra aplicación de tal forma que no hemos de estar parando los contenedores uno a uno.

Además define que el contenedor `identidock` depende de `dnmonster` por lo que primero lanza este contenedor.

Paramos el contenedor y hacemos `docker-compose up -d`

Ya sólo nos queda hacer que funcione el botón submit. Para ello hacemos que el controlador `/` también responda a peticiones POST (`@app.route('/', methods=['GET', 'POST'])`) y pasar el nombre como un parámetro para el generador de identicon.

```
from flask import Flask, Response, request
import requests
import hashlib
app = Flask(__name__)
salt = "UNIQUE_SALT"
default_name = 'Víctor Ponz'

@app.route('/', methods=['GET', 'POST'])
def mainpage():
    name = default_name
    if request.method == 'POST':
        name = request.form['name']
    salted_name = salt + name
    name_hash = hashlib.sha256(salted_name.encode()).hexdigest()
    header = '<html><head><title>Identidock</title></head><body>'
    body = '''<form method="POST">
            Hola <input type="text" name="name" value="{0}">
            <input type="submit" value="submit">
            </form>
            <p>Te pareces a:
            <img src="/monster/{1}"/>
            '''.format(name, name_hash)
    footer = '</body></html>'

    return header + body + footer
@app.route('/monster/<name>')
def get_identicon(name):
    r = requests.get('http://dnmonster:8080/monster/' + name + '?size=80')
    image = r.content
    return Response(image, mimetype='image/png')
```

```
if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0')
```

Añadir cacheo

Hasta aquí todo bien. Pero hay una cosa horrible sobre esta aplicación en este momento (aparte de los monstruos): cada vez que se solicita un monstruo, hacemos una llamada computacionalmente costosa al servicio `dnmonster`. No hay necesidad de esto; el objetivo de un icono de identificación es que la imagen sigue siendo la misma para una entrada determinada, por lo que deberíamos almacenar en **caché** el resultado.

Usaremos `Redis` para lograrlo. `Redis` es un almacén de datos de clave-valor en memoria. Es excelente para tareas como esta en las que no hay una gran cantidad de información y no nos preocupa la durabilidad (si una entrada se pierde o se elimina, podemos simplemente regenerar la imagen).

Podríamos agregar el servidor `Redis` a nuestro contenedor identidock, pero es más fácil y más idiomático crear un contenedor nuevo. De esta manera, podemos aprovechar la imagen oficial de Redis que ya está disponible en Docker Hub y evitar la molestia adicional de ejecutar varios procesos en un contenedor.

```
identidock:
  build: .
  ports:
    - "5000:5000"
  volumes:
    - ./app:/app
  links:
    - dnmonster
    - redis
dnmonster:
  ports:
    - "5080:8080"
  image: amouat/dnmonster:1.0
redis:
  image: redis:3.0
```

Y modificamos el código para usar la imagen `redis`

```
from flask import Flask, Response, request
import requests
import hashlib
import redis
app = Flask(__name__)
salt = "UNIQUE_SALT"
default_name = 'Víctor Ponz'

@app.route('/', methods=['GET', 'POST'])
def mainpage():
    name = default_name
    if request.method == 'POST':
        name = request.form['name']
    salted_name = salt + name
    name_hash = hashlib.sha256(salted_name.encode()).hexdigest()
    header = '<html><head><title>Identidock</title></head><body>'
    body = '''<form method="POST">
            Hola <input type="text" name="name" value="{0}">
            <input type="submit" value="submit">
            </form>
            <p>Te pareces a:
            <img src="/monster/{1}"/>
            '''.format(name, name_hash)
    footer = '</body></html>'

    return header + body + footer
@app.route('/monster/<name>')
def get_identicon(name):
    r = requests.get('http://dnmonster:8080/monster/' + name + '?size=80')
    image = r.contentimage = cache.get(name)
    if image is None:
        print ("Cache miss", flush=True)
        r = requests.get('http://dnmonster:8080/monster/' + name +
 ↪  '?size=80')
        image = r.content
        cache.set(name, image)

    return Response(image, mimetype='image/png')

if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0')
```

Ahora sólo nos queda parar el contenedor, construirlo y levantarlo

```
docker-compose stop
docker-compose build
docker-compose up -d
```

Referencias

- Using - Docker: O'Reilly, Adrian Mouat