edix
Digital Workers

Actividad de desarrollo: Servidores HTTP

DEVOPS - UF4

Vamos a introducir un tipo de aplicaciones de uso extremadamente extendido en la industria por su versatilidad y potencia: los servidores HTTP.

Dada la complejidad de este tipo de software, centraremos esta tarea en dejar una serie de enlaces para tener como referencia y en mostrar unos usos prácticos basados en Docker que nos pueden ser muy útiles durante el curso (y en la actividad de desarrollo de este módulo).

Definición

En su caso más básico, un servidor HTTP sirve contenido estático de una ubicación física: un directorio donde almacenamos una página web, por ejemplo. En este contexto, servir significa que se puede acceder a ese contenido, usando el protocolo HTTP, en algún puerto de la IP de la máquina host.

El servidor web por antonomasia en internet es, sin duda, Apache HTTP Server, un proyecto Open Source de la fundación Apache.

Otros ejemplos que debemos conocer, y que dan muchísimas más funcionalidades de Apache (a costa de una complejidad mucho mayor), son:

- Nginx.
- HAProxy.

Una alternativa más ligera que Apache es Lighttpd, aunque por su reducido tamaño no se debe pensar que no ofrece un elevado rendimiento, ni mucho menos.

Ejemplo

El caso más sencillo que se me ocurre es empleando el módulo HTTP incluido con Python. Por defecto, convierte el directorio de invocación en el *webroot* de una web, accesible en la dirección 0.0.0.0 y en el puerto que le digamos:

```
$ python3 -m http.server
Serving HTTP on :: port 8000 (http://[::]:8000/) ...
```

En este caso, la traza muestra la dirección IPv6 correspondiente a 0.0.0.0, aunque se puede comprobar que también está disponible la v4.

Reverse-Proxying

Algunos servidores HTTP pueden hacer funciones de Proxy inverso. Recordemos que un proxy es un servidor por el que pasan todas las comunicaciones salientes de una red. La función de un proxy inverso es, precisamente, la de **recibir** todas las comunicaciones entrantes a una red, y decidir a qué hosts de esa red **reencaminar el tráfico**, en base a unos criterios o buscando una funcionalidad que dependerá del caso.

Un ejemplo de proxy inverso sería un servicio de **balanceo de carga**, encargado de recibir todo el tráfico de un servicio (una página web, por ejemplo) y repartirlo entre varios hosts que tengan recursos para atender esa petición, y evita así una **denegación de servicio** (DoS, *Denial of Service*) por sobrecarga de peticiones. Esto es precisamente lo que hacen los **Elastic Load Balancers** de AWS, con el añadido de que pueden gestionar los hosts a los que enviar la carga de múltiples maneras.

Otro ejemplo de uso, muy común, es donde el proxy escucha en una IP que tiene asociados múltiples nombres de dominio en un servidor DNS, y es capaz de reencaminar el tráfico al host adecuado para la petición solicitada.

Imaginemos que nuestro proxy escucha en la dirección 14.45.35.18, y que hemos registrado todo el dominio **.pruebas.com* en un servidor DNS, con varios subdominios: *statics.pruebas.com, www.pruebas.com, dev.pruebas.com*… En la configuración de nuestro proxy, podríamos hacer que cada uno de esos subdominios vaya a un host distinto (incluso balanceando carga en alguno de ellos).

Terminación TLS

Sabiendo que se puede hacer todo esto, podríamos pensar: ¿qué pasa con las comunicaciones seguras? ¿Soporta SSL/TLS?

La respuesta es que sí (obviamente), y que además nos habilita un patrón en el que el servidor HTTP, como **punto de entrada a mi red** desde el exterior, esté configurado para aceptar peticiones HTTPS exclusivamente (garantizando así la seguridad de los datos que llegan a ella) y después, reparta tráfico HTTP, ya en claro, por los diferentes servicios que tengamos desplegados.

Esta manera de trabajar, comúnmente llamada 'terminar TLS', tiene la ventaja de que me permite exponer servicios sin preocuparme de la sobrecarga de la gestión de certificados y con la mayor eficiencia de usar HTTP 'a secas'. Directamente me olvido de TLS, porque sé que existe un servicio que asegura que el tráfico sensible no viajará en claro fuera de la red.

Ejemplo

Vamos a ver un caso básico de uso Nginx con un ejemplo práctico. Vamos a necesitar Docker, y haber al menos leído las prácticas de la UF2 en las que creamos dos simples servicios web, uno en Java y otro en Docker.

Para referencia, se dejan los ficheros empleados en el ejercicio en esta ruta. Si lo descomprimimos, encontraremos lo siguiente:

```
$  tree
.
├── docker-compose.yml
├── java
│   ├── Dockerfile
│   └── server.java
├── nginx
│   ├── generate_certs.sh
│   ├── load_balancer.conf
│   ├── load_balancer_ssl.conf
│   ├── router.conf
│   └── ssl
│       ├── certificates
│       │   └── test.domain.crt
│       └── private
│           └── test.domain.key
└── python
    ├── Dockerfile
    ├── app.py
    └── requirements.txt

6 directories, 12 files
```

El contenido de las carpetas java y python es una versión, más sencilla si cabe, de esos servidores sencillos que habíamos comentado. Adjunto a ellos, un Dockerfile que crea un servicio web en el puerto 8080.

El orquestador en este caso será Docker Compose. Una manera muy sencilla de lanzar ambos servidores a la vez que sería así:

```yaml
---
version: "3"
services:
  jserver:
    build: java/
    ports:
      - "8080:8080"
```

```
  pserver:
    build: python/
    ports:
      - "8081:8080"
```

Lanzando esto con docker compose, veremos que ambos están up y funcionando, cada uno en su puerto.

```
$ docker-compose up --build pserver jserver
Creating network "desarrollo_default" with the default driver
Creating desarrollo_jserver_1 ... done
Creating desarrollo_pserver_1 ... done
Attaching to desarrollo_jserver_1, desarrollo_pserver_1
pserver_1    |  * Serving Flask app 'app' (lazy loading)
pserver_1    |  * Environment: production
pserver_1    |    WARNING: This is a development server. Do not use it
in a production deployment.
pserver_1    |    Use a production WSGI server instead.
pserver_1    |  * Debug mode: off
pserver_1    |  * Running on all addresses.
pserver_1    |    WARNING: This is a development server. Do not use it
in a production deployment.
pserver_1    |  * Running on http://172.23.0.3:8080/ (Press CTRL+C to
quit)
jserver_1    | Java server started at 8080
pserver_1    | 172.23.0.1 - - [17/Dec/2021 08:39:37] "GET / HTTP/1.1"
200 -
pserver_1    | 172.23.0.1 - - [17/Dec/2021 08:39:45] "GET / HTTP/1.1"
200 -


(Otro terminal)
 $  curl  127.0.0.01:8081
Hello from Python!

$  curl  127.0.0.01:8080
Hello from Java!
```

Reverse-proxying

Vamos a usar nginx como reverse-proxy: lo vamos a poner delante de nuestros servicios, de modo que estos ya no estén expuestos al exterior, y lo vamos a configurar para que encamine las peticiones de manera equitativa entre los dos (esto se conoce como *round-robin*).

Para ello, emplearemos el fichero de configuración en 'nginx/load_balancer.conf':

```
events { worker_connections 1024; }
http {
    upstream app_servers {
        server jserver:8080;
        server pserver:8080;
    }

    server {
        listen 80;
        location / {
            proxy_pass         http://app_servers;
        }
    }
}
```

Este fichero se monta (usando Docker compose) en la ruta en la que el servicio espera encontrar la configuración del servicio. Además, debemos modificar la configuración de puertos, para que solo se expongan al host puertos de nginx (en este caso, el 8080 local se mapea al 80 del proxy), y por sencillez, crear una relación de dependencia entre el servicio del proxy y los servicios que maneja:

```
---
version: "3"
services:
  nginx:
    image: nginx:1.16.0-alpine
    volumes:
      - ./nginx/load_balancer.conf:/etc/nginx/nginx.conf:ro
    ports:
      - "8080:80"
    depends_on:
      - jserver
      - pserver
  jserver:
    build: java/
  pserver:
    build: python/
```

Levantemos este servicio y veamos la diferencia:

```
$ docker-compose up nginx
Starting desarrollo_nginx_1 ... done
```

```
Attaching to desarrollo_nginx_1
(Otro terminal)
$  curl  127.0.0.01:8080
Hello from Java!
 $  curl  127.0.0.01:8080
Hello from Python!
 $  curl  127.0.0.01:8080
Hello from Java!
 $  curl  127.0.0.01:8080
Hello from Python
 $  curl  127.0.0.01:8080
Hello from Java
 $  curl  127.0.0.01:8080
Hello from Python!
```

¡Efectivamente! El servidor está balanceando el tráfico, enviando el 50% a cada uno.

Terminación TLS

Usando nginx y unos certificados autofirmados, podemos simular un servidor TLS en local (totalmente extrapolable a un caso productivo, contando con unos certificados expedidos por una CA válida). Se ha dejado un ejemplo de configuración en 'nginx/load_balancer_ssl.conf'.

```
events { worker_connections 1024; }

http {

    upstream app_servers {
        server jserver:8080;
        server pserver:8080;
    }

    server {
        listen 443 ssl;
        ssl_certificate /etc/ssl/test.domain.crt;
        ssl_certificate_key /etc/ssl/test.domain.key;
        server_name test.domain;
        location / {
            proxy_pass         http://app_servers;
        }
    }
}
```

Se ha añadido un sencillo script que genera un par de claves para un dominio falso, 'test.domain'. Para su ejecución hace falta tener instalado 'openssl' y 'ssl-cert'. Si no lo tenéis disponible, podéis ejecutar el script montando el volumen en una imagen de Ubuntu e instalar el software necesario:

```
$  cd nginx/
$  ./generate_certs.sh
Generating a 2048 bit RSA private key
.............+++
......+++
writing new private key to
'/Users/xxxxxx/desarrollo/nginx/ssl/private/test.domain.key'
-----
```

Con estos certificados, podemos crear un nuevo servicio en nuestro docker-compose, que monta los certificados en su sitio:

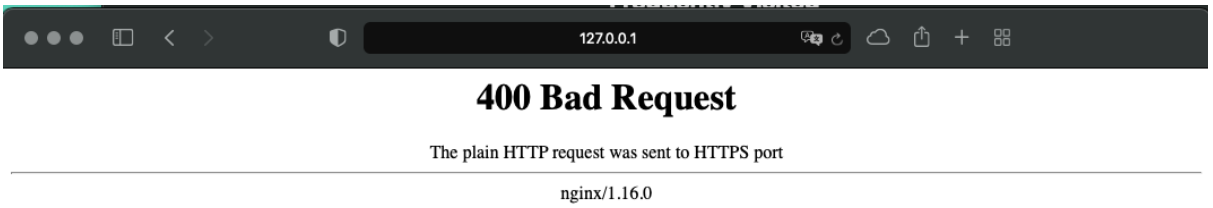
```
 nginx_ssl:
    image: nginx:1.16.0-alpine
    volumes:
      - ./nginx/load_balancer_ssl.conf:/etc/nginx/nginx.conf:ro
      - ./nginx/ssl/certificates/test.domain.crt:/etc/ssl/test.domain.crt:ro
      - ./nginx/ssl/private/test.domain.key:/etc/ssl/test.domain.key:ro
    ports:
      - "8080:443"
    depends_on:
      - jserver
      - pserver
```

Vamos a probarlo:

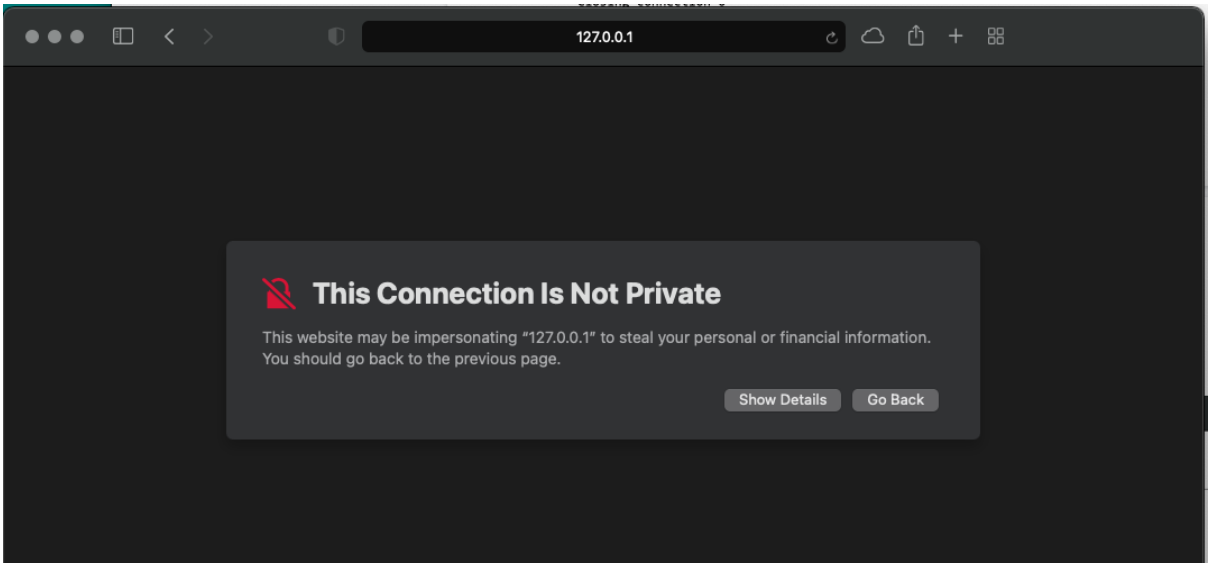
```
$  curl -v  127.0.0.01:8080
*   Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to 127.0.0.01 (127.0.0.1) port 8080 (#0)
> GET / HTTP/1.1
> Host: 127.0.0.01:8080
> User-Agent: curl/7.64.1
> Accept: */*
>
< HTTP/1.1 400 Bad Request
< Server: nginx/1.16.0
< Date: Fri, 17 Dec 2021 09:21:11 GMT
< Content-Type: text/html
< Content-Length: 255
< Connection: close
<
<html>
<head><title>400 The plain HTTP request was sent to HTTPS port</title></head>
<body>
```

```
<center><h1>400 Bad Request</h1></center>
<center>The plain HTTP request was sent to HTTPS port</center>
<hr><center>nginx/1.16.0</center>
</body>
</html>
* Closing connection 0
```

Vaya, vaya... Nginx nos devuelve un error, HTTP 400 Bad Request. Recordemos que esto significa que el cliente ha hecho algo mal. En este caso (como pone bien claro más abajo) se está enviando una petición HTTP a un puerto HTTPS:



Esto es porque el protocolo **por defecto** es HTTP, algo que afortunadamente parece sencillo de cambiar, visitando https://127.0.0.1:8080:



Esto es esperable, ya que usamos un certificado no confiable (¡pero válido!).

Si lo hacemos vía *curl*, debemos usar el flag *--insecure* para obtener una respuesta válida, sino nos pasará lo mismo.

```
$  curl -v  --insecure https://127.0.0.01:8080
*   Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to 127.0.0.01 (127.0.0.1) port 8080 (#0)
* ALPN, offering h2
```

```
* ALPN, offering http/1.1
* successfully set certificate verify locations:
*   CAfile: /etc/ssl/cert.pem
  CApath: none
* TLSv1.2 (OUT), TLS handshake, Client hello (1):
* TLSv1.2 (IN), TLS handshake, Server hello (2):
* TLSv1.2 (IN), TLS handshake, Certificate (11):
* TLSv1.2 (IN), TLS handshake, Server key exchange (12):
* TLSv1.2 (IN), TLS handshake, Server finished (14):
* TLSv1.2 (OUT), TLS handshake, Client key exchange (16):
* TLSv1.2 (OUT), TLS change cipher, Change cipher spec (1):
* TLSv1.2 (OUT), TLS handshake, Finished (20):
* TLSv1.2 (IN), TLS change cipher, Change cipher spec (1):
* TLSv1.2 (IN), TLS handshake, Finished (20):
* SSL connection using TLSv1.2 / ECDHE-RSA-AES256-GCM-SHA384
* ALPN, server accepted to use http/1.1
* Server certificate:
*  subject: CN=test.domain
*  start date: Dec 17 09:14:41 2021 GMT
*  expire date: Dec 15 09:14:41 2031 GMT
*  issuer: CN=test.domain
*  SSL certificate verify result: self signed certificate (18), continuing anyway.
> GET / HTTP/1.1
> Host: 127.0.0.01:8080
> User-Agent: curl/7.64.1
> Accept: */*
>
< HTTP/1.1 200 OK
< Server: nginx/1.16.0
< Date: Fri, 17 Dec 2021 09:26:48 GMT
< Content-Length: 16
< Connection: keep-alive
<
* Connection #0 to host 127.0.0.01 left intact
Hello from Java!* Closing connection 0

$  curl  --insecure https://127.0.0.01:8080
Hello from Python
```

Routing basado en nombre de dominio

Por último, vamos a usar Nginx para ***enrutar*** el tráfico entre diferentes servicios según la ruta a la que vengan, simulando el caso expuesto más arriba.

Crearemos una nueva entrada en nuestro docker-compose, similar a la primera, que apunta a un fichero de configuración de Nginx con este contenido:

cat nginx/router.conf

```
events { worker_connections 1024; }
http {

  server {
    server_name java.test.domain;
    location / {
      proxy_pass http://jserver:8080;
    }
  }
  server {
    server_name python.test.domain;
    location / {
      proxy_pass http://pserver:8080;
    }
  }
}
```

Probemos, a ver qué pasa:

```
$  curl   http://127.0.0.01:8080
Hello from Java!
 $  curl   http://127.0.0.01:8080
Hello from Java!
 $  curl   http://127.0.0.01:8080
Hello from Java!
 $  curl   http://127.0.0.01:8080
Hello from Java!
 $  curl   http://127.0.0.01:8080
Hello from Java!
```

Parece que no está balanceando… ¿qué pasa?

Sencillo: **no estamos balanceando**, sino enrutando. Y al no haber ningún *match*, utiliza la sección 'server' por defecto: la primera que encuentra. Vamos a usar el nombre dominio y ya está, ¿no?

```
$  curl   http://python.test.domain:8080
curl: (6) Could not resolve host: python.test.domain

$  curl   http://java.test.domain:8080
curl: (6) Could not resolve host: java.test.domain
```

… ¿Qué significa esto? ¿Por qué no puede resolver ese dominio?

```
$ dig +short java.test.domain @8.8.8.8
$
```

Pues porque no existe, obviamente, se trata de un nombre inventado para pruebas locales. Para estos casos, en los que necesitamos 'simular' nombres de dominio reales en nuestro entorno local, lo que se hace es modificar un fichero que DNS vaya a consultar. En la mayoría de los sistemas operativos, este fichero es /etc/hosts, y se añadiría una entrada como la siguiente:

```
# /etc/hosts
# Local development for test.domain
127.0.0.1 java.test.domain python.test.domain
```

Y ahora ya sí:

```
$  curl   http://python.test.domain:8080
Hello from Python!

$  curl   http://java.test.domain:8080
Hello from Java!
```