Práctica 1: Micro-servicios REST API en contenedor

Versión 1.0 11/09/2025

Contenido

1. Objetivos docentes

Tras la realización de la Práctica 1, el estudiante debe ser capaz de:

- Desarrollar micro-servicios en Python que implementen una REST API
- Desplegar un micro-servicio en contenedores usando docker-compose

2. Introducción

En esta práctica implementaremos un servicio de almacenamiento de documentos de texto compartidos. Este servicio se divide realmente en dos, el servicio de usuarios y el de biblioteca.

Cada usuario tendrá una biblioteca, y podrá añadir, listar, descargar, o borrar documentos de su biblioteca. En el caso de que se añada un documento y este ya exista, se reemplazará su contenido.

Además, cada documento de la biblioteca podrá marcarse como público o privado.

- Los documentos públicos podrán ser descargados por otros usuarios si conocen el ID del usuario (UID) del propietario y el nombre del documento.
- Los documentos privados no podrán ser descargados por terceros.
  El propietario siempre podrá listar y gestionar sus documentos (crear, actualizar, borrar y cambiar su visibilidad).

El resto de los usuarios podrán descargar el contenido de los documentos públicos de otros usuarios, siempre que conozcan el UID y el nombre del documento, pero no podrán listar, o modificar la biblioteca de otros usuarios. Se recomienda que el servicio de archivos mantenga la visibilidad de cada documento en una estructura persistente (p. ej., un índice JSON).

Para que el sistema sea seguro, ha de permitir crear usuarios, con un nombre y contraseña. Al crear un usuario se le asignará un UID único, y si todo es correcto se devolverá dicho UID junto con el token de acceso. Un usuario puede entrar al sistema de forma similar, indicando su nombre y contraseña, y en caso de éxito, el sistema le devolverá su token de acceso y el UID. Tanto el UID como el token de acceso serán necesarios para poder acceder a la biblioteca de un usuario. Además, el servicio de usuarios debe permitir cambiar la contraseña de un usuario autenticado.

El UID único se puede crear con la función uuid.uuid4(), que genera un ID aleatorio suficientemente grande como para que estemos casi seguros de que es único.

Para simplificar, el token de acceso será un hash (sha-1) derivado del UID, con el formato ‘UID.hash(Secret.UID)’, por lo que el servicio de biblioteca podrá verificar fácilmente si el token es válido.

Secret es conocido por ambos servicios, pero no por el cliente. En el caso de la librería UUID de python Secret es a su vez un UUID, y usaremos la función uuid5 para usar sha-1 como hash. Ejemplo:

```
Secret_uuid = uuid.UUID('00010203-0405-0607-0809-0a0b0c0d0e0f')
uuid.uuid5(secret_uuid, uid_del_usuario_como_string)
```

Aviso: Para mayor seguridad, en un caso real, se deben usar otros mecanismos mejores para crear los tokens, que incluyan un tiempo de expiración, y no requieran que todos los servicios conozcan la clave Secret, por ejemplo, usando tokens JWT con expiración y firma con un sistema de clave pública/privada.

Para probar que todo funciona, se creará un cliente en Python que interactúe con ambos servicios.

3.Microservicios

Los microservicios son una alternativa, opuesta, a las aplicaciones monolíticas.

Mientras que, en una aplicación monolítica, todo en uno, toda la funcionalidad se encuentra en un único servidor, en una arquitectura de microservicios cada funcionalidad se desarrolla y despliega de forma independiente, desacoplada del resto.

Esto permite:

- Facilitar y acelerar cambios en la funcionalidad
- Escalar sólo las piezas (funcionalidad) necesarias
- Compartir recursos (funcionalidad) entre distintas aplicaciones

4.REST API

REST es una manera, la más extendida, de implementar microservicios.

Se basa en definir recursos (por ejemplo, *user*, *file*, ...), cada uno con una URL asociada.

La gestión de los recursos se realiza mediante peticiones HTTP en que el verbo indica la acción a realizar:

- GET: recuperación
- PUT: creación o reemplazo (actualización completa)
- PATCH: actualización parcial
- DELETE: elimina el recurso
- POST: ejecución de tareas asociadas al recurso, por ejemplo, añadir un elemento al carrito

Nota: GET, PUT, PATCH y DELETE deben ser idempotentes; POST no tiene por qué serlo.

En esta práctica se implementará un prototipo de los recursos user y file, implementando los servicios en user.py y file.py respectivamente. No es necesario gestionar la biblioteca como un recurso distinto a los archivos, ya que solo tendremos una biblioteca por usuario, por lo que no tiene sentido crearlas o destruirlas, aunque si es posible listar su contenido.

Las rutas de los recursos de tipo archivo será: /file/UID/nombre_archivo.extensión

La biblioteca de un usuario estará accesible en la ruta /file/UID

Además, el propietario de un documento puede generar un enlace temporal para compartirlo con otros usuarios. Ese enlace permite descargar el documento durante cierto tiempo (por ejemplo, un minuto), aunque no se tenga un token de acceso normal. Una vez pasado ese tiempo, el enlace deja de ser válido.

Para crear este enlace se usará la ruta /file/UID/nombre_archivo/share. El servicio devolverá un *share_token*, que identifica el documento y su caducidad. Cualquier usuario que disponga de ese enlace podrá usarlo en la ruta /share/share_token para descargar el documento. Si el enlace es incorrecto o ya ha caducado, se devolverá un error.

El *share_token* tendrá el formato:

UID.nombre.exp.hash

donde UID es el identificador del usuario, nombre es el nombre del archivo, exp es el instante de caducidad, y hash es un sha-1 calculado con una clave secreta conocida por los servicios, de forma parecida a los tokens normales. De esta forma el servicio puede comprobar fácilmente si el enlace es válido y si todavía no ha caducado.

Nota: Cuando se crea un enlace temporal, no se tiene en cuenta si el documento es público o privado. La idea del enlace es que, durante cierto tiempo, cualquier persona que lo reciba pueda descargar el documento. Fuera de ese caso, siguen aplicándose las reglas normales de visibilidad (los documentos privados no pueden descargarse sin token y los públicos sí).

En el caso de los usuarios no hay que añadir nada a la ruta, y se usará simplemente /user. La información con el nombre, o la contraseña se añadirá al cuerpo de la petición, usando los campos 'name' y 'password' respectivamente. De forma similar, el servicio de usuarios devolverá el token y el UID del usuario en el cuerpo de la respuesta, usando los campos 'token' y 'uid'. En ambos casos el cuerpo de la petición/respuesta estará en formato JSON.

En el caso del token de seguridad para acceder al servicio de archivos, este se pasará en el campo Authorization, en la cabecera de la petición, indicando que es de tipo "Bearer". Por ejemplo:

Authorization: Bearer 0b43d6ba-b431-4fee-b649-1d14a871ec00.93096740305b3341c

Se aconseja desarrollar este apartado siguiendo los siguientes pasos:

1. Creación y pruebas del microservicio de archivos, sin comprobación de los tokens.

   Para las pruebas se creará un cliente en Python que llame al API del servicio. No es necesario que el cliente envíe o reciba archivos reales, ya que al ser documentos de texto se podrán enviar cadenas de texto, y mostrar dicho texto al obtenerlos del servidor. Esto además puede facilitará las pruebas.

   En cuanto a la persistencia del servicio, se recomienda usar una estructura sencilla de directorios y archivos de texto.

2. Creación del microservicio de usuarios, añadiendo la comprobación de los tokens al servicio de biblioteca y el cliente de pruebas.

   Cada vez que se cree un nuevo usuario, se creará un archivo con su nombre, que contenga en formato JSON su UID y un hash con la clave. Existen distintas bibliotecas en Python para trabajar con UIDs. Para esta práctica se recomienda usar la librería uuid para generar los UIDs de forma aleatoria. Como hash se puede utilizar la función nativa de Python.

Nota: Cada microservicio será un programa distinto, que escuchará en un puerto diferente, mientras que para las pruebas se puede usar un mismo programa cliente que llame a ambos servicios.

4.1.Entorno de Python

Para disponer de un entorno de python propio, independiente de los paquetes instalados en el sistema, vamos a usar un entorno generado con el módulo venv (virtualenv).

Ejecuta en un terminal:

```
$> mkdir -p venv/si1p1
$> python3 -m venv venv/si1p1
$> source ./venv/si1p1/bin/activate
$> pip install quart requests
```

Con esto tendremos los paquetes necesarios para esta práctica.

También podemos automatizar la instalación de paquetes usando un fichero con las dependencias.

Usando “pip3 freeze > requirements.txt” se crea el archivo con las dependencias actuales, y luego se pueden instalar usando “pip install -r requirements.txt”.

Cuidado: No se debe copiar el directorio venv/si1p1, si no que se debe recrear usando pip install. Tampoco se ha de entregar el contenido de dicha carpeta al entregar la práctica o usar un sistema de control de versiones, ya que es suficiente con indicar las dependencias en un archivo (requirements.txt)

4.2.Micro-servicios en Quart

Tanto Flask como Quart son entornos en python para el desarrollo de aplicaciones web y micro-servicios. Quart es una evolución de Flask que optimiza su ejecución.

Los servicios web en Quart hacen uso de funciones asíncronas (async def), las cuales permiten seguir ejecutando otras tareas mientras se espera (en await) a que termine la ejecución de otra función, típicamente debido a funciones de entrada/salida. Esto se hace en un solo hilo o proceso, por lo que se evitan gran parte de los problemas de concurrencia típicos de otros modelos de computación paralela. Las funciones asíncronas por lo tanto son funciones capaces de detenerse a esperar, cediendo el hilo de ejecución a otra parte del código que pueda continuar la ejecución, sin necesidad de intervención por parte del sistema operativo.

Para ejecutar los servicios, Quart usa las anotaciones @app.route, @app.get, @app.put, etc., para identificar que función será necesario llamar dependiendo del método http de la petición.

Consulta la documentación de Quart para más información:

https://quart.palletsprojects.com/en/latest/

En las primeras pruebas, también puedes usar el navegador para probar las rutas del API que usen GET, o el comando curl para poder enviar datos en formato JSON, o usar distintos métodos HTTP.

Por ejemplo:

```
curl -X PUT http://127.0.0.1:5050/user/antonio -H 'Content-Type: application/json' -d '
{"password": "my_password"}'
```

El comando anterior realiza una petición PUT al puerto 5050, en la ruta /user/antonio, y envía un objeto JSON con un campo ‘password’ con el valor ‘my_password’. En la petición http, podemos añadir cabeceras con la opción -H. En este caso se ha especificado que el tipo del contenido es application/json.

Para automatizar las pruebas, se pueden utilizar frameworks que faciliten esta tarea. Para esta práctica sin embargo se creará un cliente sencillo, no interactivo, en Python utilizando la biblioteca Requests, que simplemente indique el nombre de la prueba y si el resultado es el esperado o no. Se deberán probar todas las rutas implementadas, intentando que con la ejecución del cliente se ejecute todo el código de los servicios, incluidos los casos de error implementados.

Puedes obtener más información sobre el uso de Requests en: https://pypi.org/project/requests/

5.Contenedores

La tecnología de contenedores permite aislar el entorno de ejecución de una aplicación del resto del sistema de forma que, por ejemplo, se puede desarrollar una aplicación que requiera de un entorno (paquetes) de Debian y ejecutarla en un sistema que ejecute Ubuntu.

También permite disponer de unos paquetes de software propios en ese entorno de ejecución; por ejemplo, se pueden tener versiones anteriores a las instaladas en el sistema o paquetes que no existen en dicho sistema.

Es parecido a lo que hace con el entorno de python venv usado anteriormente, pero a una escala mucho mayor, sin llegar a la independencia que da una máquina virtual.

5.1.Modo rootless

El gestor (demonio) de contenedores se puede ejecutar de muchas formas. En modo rootful (con permisos de superusuario) o rootless (como un usuario). También es posible ejecutar un gestor de contenedores anidado (docker in docker) o virtualizado (Docker Desktop).

En modo rootful, los contenedores se arrancan con prácticamente todos los privilegios de root, si bien en un entorno aislado. Esto presenta vulnerabilidades que exigen un control exhaustivo de los contenedores desplegados.

En modo rootless el gestor se arranca con los privilegios del usuario que lo arranca, y no presenta estos peligros, aunque también presenta una serie de limitaciones, si bien se está haciendo un esfuerzo importante (incluso a nivel del kernel de linux) para resolverlas.

Docker Desktop es una forma de ejecutar el gestor docker como si fuera rootful, pero dentro de una máquina virtual, por lo que se pierde algo de eficiencia, aunque ganamos en seguridad y compatibilidad. Este sistema no sería apropiado en producción, pero facilita el desarrollo, ya que además posee una interfaz gráfica.

5.2.Ejecución en los laboratorios

El modo rootful de docker, por cuestiones de seguridad, está desactivado, y no lo usaremos en los laboratorios.

Para iniciar Docker en lo loboratorios ejecuta en un terminal ($> es el prompt):

```
$> sudo /usr/local/bin/si2setuid.sh 3000000:100000 $USER
# Este comando establece la configuración de usuario para tener suficientes subuids
# editando como sudo los archivos /etc/subuid y /etc/subgid
```

```
# Si lo quieres hacer en un ordenador personal tendrás que editar los archivos manualmente
$> dockerd-rootless-setuptool.sh install
# usar docker propio en lugar del docker del sistema
$> DOCKER_HOST=unix:///run/user/$(id -u)/docker.sock
# comprobación de estado del demonio
$> systemctl --user status docker
```

5.3.Creación de las imágenes

Para ejecutar una aplicación en un contenedor se ha de partir de una imagen base del contenedor.

La imagen consta de los archivos y configuración (red, volúmenes externos, puertos publicados, valores por defecto de las variables de entorno usadas, programa inicial a ejecutarse, …) que la aplicación requiere.

En este apartado vamos a crear la imagen base de nuestros microservicios. Para ello crearemos un directorio para guardar los archivos de cada servicio de forma independiente, y en cada directorio crearemos un archivo Dockerfile, siguiendo los ejemplos que puedes encontrar en la imagen oficial de Python: https://hub.docker.com/_/python

La sintaxis completa de los ficheros Dockerfile la puedes encontrar en la documentación oficial: https://docs.docker.com/reference/dockerfile/

Puedes probar a construir la imagen de cada servicio y añadirles un nombre, ejecutando desde cada directorio el comando:

$> docker build –tag nombre_imagen:version

Una vez construida, se puede ejecutar con el comando:

$> docker run -p puerto_host:puerto_contenedor nombre_imagen:version

5.4.Docker compose

Podemos automatizar los pasos del punto anterior usando un solo archivo docker-compose.yml

Puesto que ya tenemos los servicios en directorios separados, crearemos el fichero docker-compose.yml en el directorio padre de los servicios, indicando el nombre de cada subcarpeta en la opción “build” de cada servicio.

Un ejemplo sencillo de una aplicación similar usando docker compose, con flask en lugar de Quart, lo puedes encontrar en: https://docs.docker.com/compose/gettingstarted/

Cada servicio será independiente, y podrá escuchar en un puerto distinto. Además, se puede configurar donde se guardarán realmente los datos, configurando un volumen distinto para cada servicio.

6. Planificación

Esta práctica está planteada para seguir la planificación:

Semana 1	Codificación de la API
Semana 2	Codificación de la API
Semana 3	Dockers y cliente de prueba
Semana 4	Dudas

7.Entrega

La fecha y normas de entrega de las prácticas se encuentran en Moodle. Como mínimo se han de entregar los siguientes archivos:

- file.py
- user.py
- docker-compose.yml
- cliente.py
- Dockerfiles
- Memoria