Sistemas Informáticos I

Práctica 1

Francisco Javier Zurita Vílchez

Javier San Andrés de Pedro

Índice

1. Introducción

En esta práctica, hemos desarrollado un entorno de microservicios para gestionar un sistema de biblioteca de usuarios. Además, hemos desarrollado un entorno de ejecución independiente para cada servicio, denotando aún más esta independencia al hacer que cada uno acceda a su propia base de datos. Si bien la elección de este tipo de arquitectura para el sistema en cuestión parece demasiado sobrecargada (al haber únicamente dos servicios), la escalabilidad de este tipo es abrumadora frente a la vertiente monolítica.

Además, en cuanto a la contenerización, ella permite virtualizar la ejecución de ambos servicios otorgando un entorno de desarrollo independiente para cada uno, de tal forma que se pueda ir alterando uno de ellos a la par que se trabaja con la última versión publicada del otro, lo cual confiere una ventaja importante. Es más, es tal la independencia, que uno de ellos podría ejecutarse en una máquina Debían y el otro en un Windows sin que hubiera ninguna incompatibilidad. Otra de las ventajas que tiene es la posibilidad de gestionar ambos servicios en un sistema de redes diseñado a nuestro antojo y que puede resultar muy útil el día de mañana en aplicaciones más profesionales.

2. Archivos

En esta sección, primero se comentarán los ficheros relativos al código y, más tardes, aquellos relativos a Docker. La estructura de archivos que se ha utilizado es la siguiente:

```
/
│
├── library_server/
│   ├── Dockerfile  (for file.py)
│   ├── file.py
│
├── users_server/
│   ├── Dockerfile  (for user.py)
│   ├── user.py
│
├── tests/
│   ├── client.py
│
├── utils.py           # <-- Common file imported by both file.py and user.py
├── .env               # <-- Shared .env file for environment variables
├── requirements.txt   # <-- Shared dependencies
├── docker-compose.yml
```

2.1. User.py

En este script de Pyhton vienen todos los métodos relativos a la API del servidor de usuarios.

2.1.1. **register_user()** (Ruta: /user)

Para registrar un nuevo usuario, se necesita una petición utilizando el verbo PUT (ya que se trata de la creación de nuevos recursos), un nombre de usuario único en el sistema y una contraseña de cualquier índole para proteger el acceso a la cuenta. Una vez le llega esta petición, este método se comunica con el servidor de biblioteca con el patrón SAGA enviándole una solicitud de creación de biblioteca para ese nuevo usuario. Si no se recibiera respuesta por parte del otro servidor o la respuesta no fuera positiva, entonces el usuario no quedaría como registrado en el sistema.

Después, se encarga de generar un archivo en el directorio user con nombre el del propio usuario y como información la contraseña con un hash sencillo y el id único del susodicho. Este archivo nuevo creado se almacena en formato json. Una vez todos los recursos necesarios han sido creados, se encarga de devolver el uid y un token de acceso para que el usuario pueda interactuar con su biblioteca.

2.1.2. **login_user()** (Ruta: /user)

Su principal función es facilitar el inicio de sesión de un usuario ya registrado en el sistema. Para ello, necesitamos que el cliente mande una petición a la ruta descrita con el verbo GET. En el cuerpo de la petición, se ha de incluir el nombre de usuario y la contraseña. Este método se encarga de comprobar que esas credenciales sean válidas y, si lo son, devuelve el uid y el token de acceso para que el usuario pueda interactuar con su biblioteca.

Si bien podría haberse optado por otra opción de diseño en la que la ruta fuera /user/username, hemos preferido pasar el nombre en el cuerpo de la petición porque, hoy en día, lo más común es acceder a una ruta de loggeo en el que tanto el nombre de usuario como la contraseña se introducen dentro de la misma vista.

2.1.3. **delete_user()** (Ruta: /user/<username>)

Este método permite a un usuario eliminar su cuenta del sistema. Evidentemente, el verbo asociado a esta ruta es DELETE y la ruta incluye el nombre de usuario porque normalmente la eliminación de una cuenta se suele realizar desde el menú de ajustes de la propia cuenta. Como es lógico, no se permite a una persona externa eliminar la cuenta de otro, sino que tiene que ser el propio usuario quien autorice dicha operación.

La principal delicadeza de esta función es que ha de eliminar todos los recursos asociados al usuario, tanto su biblioteca como el fichero que contiene sus credenciales. La parte de borrar el fichero con sus credenciales es sencilla ya que depende del servidor de usuarios. No obstante, el borrado de la biblioteca se gestiona enteramente en el servidor de biblioteca. Por ello, en esta función se comunican ambos servidores mediante el patrón SAGA para eliminar la biblioteca asociada. Los detalles de esta comunicación serán explicados en una sección aparte mismo epígrafe.

2.2. File.py

En este archivo, vienen todas las funciones de la API del servidor de biblioteca

2.2.1. create_user_library() (Ruta: /file/<uid>)

Este método se encarga de crear los recursos de biblioteca para un usuario con cierto uid. El verbo asociado es PUT (pues se trata de la creación de un recurso) y la ruta incluye el uid porque resulta más cómodo que tener que especificarlo en el cuerpo de la petición. Aunque se pierda coherencia con la ruta de creación de usuario, hay que matizar que esta API es interna a los servidores y oculta a los usuarios; por lo que el cambio de sintaxis puede prevenir intentos fraudulentos de creación de bibliotecas de manera directa por clientes ilícitos (lo lógico es que alguien externo al sistema intente atacar la ruta /file para la que no hay un método PUT definido).

Además, cabe destacar que solo serán admitidas las peticiones realizadas por parte del servidor de usuarios, de la manera que se especificará en su sección correspondiente y que cualquier intento de creación directa de bibliotecas será revocado.

2.2.2. delete_user_library () (Ruta: /file/<uid>)

Este método se encarga de destruir la biblioteca de un usuario. Este tipo de eventualidad solo puede producirse cuando un usuario borra su cuenta y, al igual que el método anterior, es el servidor de usuarios el único que puede ejecutar esta orden. El verbo HTTP asociado es DELETE. Cualquier intento de eliminación directa por otro cliente que no sea el servidor vecino, será revocado y contestado con el código 401.

2.2.3. list_documents() (Ruta: /file/<uid>)

Esta función se encarga de elaborar un listado de los documentos que se hallan en la biblioteca de un usuario determinado. El verbo para ello es GET pues supone la adquisición de un recurso del servidor. Por otro lado, cabe mencionar que es el propio usuario el único capaz de listar su propia biblioteca y que, para garantizar esto, se utilizan los tókenes de acceso provistos por el servidor de usuarios en el registro o inicio de sesión.

2.2.4. add_file() (Ruta: /file/<uid>/<filename>)

Esta rutina permite a un usuario añadir archivos a su biblioteca. El verbo permitido es PUT ya que se crea un nuevo recurso o se reemplaza completamente si ya existe. De nuevo, es el usuario el único que puede añadir archivos a su biblioteca y se comprobaran los tókenes de acceso para garantizar esto.

2.2.5. send_file() (Ruta: /file/<uid>/<filename>)

Este método permite a los usuarios de la biblioteca descargar un archivo de la biblioteca de otro usuario (o de su propia biblioteca). Se necesita conocer, por ende, tanto el uid del usuario como el nombre del fichero. Además, el sistema comprueba que el token de acceso referido pertenece a un usuario del sistema (sea cual sea, pero un usuario del sistema). Por tanto, la única forma de acceder al contenido de la aplicación es estando registrado en la propia aplicación; dando mayor robustez y control al sistema en general (toda descarga puede ser monitorizada si se quisiera).

2.2.6. delete_file() (Ruta: /file/<uid>/<filename>)

Permite a un usuario eliminar algún archivo de su biblioteca si este existe en ella. El verbo asociado es DELETE y, al igual que los métodos de listado y adición de archivos, el único capaz de realizar estas transacciones es el propio usuario y se utilizan de nuevo los tókenes para asegurar que así sea.

2.3. Utils.py

En este fichero vienen incluidas funciones comunes que utilizan tanto file.py como user.py. Las más destacadas son una función para obtener las rutas absolutas a los ficheros (independientemente de dónde se ejecuten los scripts), distintas funciones para construir un string de código html que refleje las respuestas de ambos servidores, una función para sacar el token de la cabecera de autorización de una petición y otra función para comprobar si ese token es válido en el sistema. Como nota, se asume que todos los tipos de autorización son de tipo Bearer ya que se trata de autorización a través de tókenes.

2.4. Comunicación entre los servidores

En una arquitectura de microservicios en que cada servicio gestiona su propia base de datos (en nuestro ejemplo, los directorios file y user) uno de los patrones más utilizados es el patrón SAGA. Aunque la verdadera eficacia de este patrón se ve reflejado con transacciones más complejas que involucran múltiples servicios, con solo dos servicios también hemos implementado este patrón para dos funcionalidades: creación de usuarios y eliminación de los mismos.

La primera idea que se nos ocurrió para implementar esto es el uso de un broker como RabbitMQTT para que los servidores se comunicaran. No obstante, aunque de hecho esa solución es la más habitual a escala profesional, nos pareció que también estaba fuera de las competencias de esta práctica; por lo que hemos optado por emular la comunicación a través de colas mediante una API específica.

Así, para comprobar que es el otro servidor el que se comunica, se utiliza como token de acceso la clave SECRET (conocida por ambos servidores). Como el propósito es que ambos se ejecuten y comuniquen en un contenedor mediante Docker, ambos formarán parte de la misma red privada, impidiendo el acceso a estos datagramas desde fuera y, por consiguiente, evitando que esta clave sea conocida por clientes externos o atacantes.

Por otro lado, en cuanto a la implementación del patrón SAGA, decir que, antes de crear un usuario, se envía la petición de creación de biblioteca al otro servidor. Si la respuesta es positiva, entonces se creará correctamente el usuario; si no, entonces se desharán todas las acciones anteriormente realizadas y se devolverá un código de error. De manera equivalente, antes de eliminar un usuario, se solicita al servidor de biblioteca que elimine su biblioteca asociada.

2.5. .env y requirements.txt

El primero contiene las variables de entorno del sistema, donde se localizan la clave SECRET de encriptación y los puertos de ambos servidores. En requirements.txt se hayan todas las dependencias necesarias para el correcto funcionamiento de los servidores.

2.6. Archivos de Docker

Para permitir la contenerización, tenemos tres archivos: dos Dockerfiles relativas a cada servicio y un docker compose para ambos. Los Dockerfile contienen instrucciones sencillas para copiar los ficheros correspondientes, descargar las dependencias y definir un directorio de trabajo para cada aplicación. Como ambos servicios utilizan tanto los ficheros requirements.txt y utils.py que se hallan en la carpeta padre, por asuntos particulares de Dockerfile (que solo permite inspeccionar la carpeta en que se encuentra el fichero y subcarpetas), no sería posible disponer de esos ficheros si lanzamos cada aplicación con el comando docker build <Dockerfile_path>.

Para solventar esta deficiencia, el fichero docker-compose.yml sirve de herramienta fundamental. En este fichero, podemos otorgar a los anteriores Dockerfiles el directorio padre como contexto, permitiendo así que ambos puedan acceder a los dos archivos previamente mencionados. Además, en docker-compose.yml podemos especificar las variables de entorno, el mapeo de puertos y los volúmenes que se usarán para mantener la persistencia de los datos.

Para los volúmenes, hemos optado por volúmenes con nombre. El motivo que sostiene esta implementación, es que necesitamos que se mantenga la persistencia incluso si el contenedor es borrado, incluso si hay cambios en el software y es necesario actualizar ficheros. Si hubiéramos utilizado volúmenes anónimos, que suelen estar destinados de hecho a almacenamiento temporal, no contaríamos con esta ventaja y, si necesitáramos por ejemplo mover la aplicación a otro puerto, perderíamos todos los usuarios y bibliotecas hasta ese punto almacenados o necesitaríamos iniciar un tedioso proceso de migración de datos de un volumen a otro.

3. Instalación y ejecución

Un requisito indispensable para la ejecución es tener instalado Docker y tener activo el servidor. Para correr y despertar ambos servidores, basta ubicarse en la carpeta padre y ejecutar el comando **docker compose up –build**. El cual se encargará de levantar ambos servidores ejecutando el archivo docker-compose.yml.

Para parar la ejecución de los mismos se puede ejecutar **Ctrl+C** y esperar a que paren. Si se quisiera eliminar los contenedores, ejecutar el comando **docker compose down**. Como los volúmenes son con nombre, al eliminar los contenedores no se eliminan los mismos. Para hacerlo, basta con utilizar los siguientes comandos **docker volumes rm <volumen>**. En este caso, los dos nombres son **users_data** (para el volumen que almacena

información de los usuarios) y **library_data** (para el volumen que almacena información de la biblioteca.

Es importante notar que **la ejecución por separado de los Dockerfiles fallará** pues ambos necesitan copiar archivos de una carpeta padre y esto solo es posible mediante una contextualización de un archivo docker-compose en la carpeta padre (esto es ajeno a nosotros y depende enteramente del propio Docker).

Además, **es importante no modificar nada del archivo docker-compose.yml**, pues los nombres de los servicios indicados en este fichero, le dan nombre a la interfaz IP que utilizan para comunicarse entre ellos. Por tanto, modificar esto conllevará que no se puedan comunicar entre sí y no sea posible crear nuevos usuarios ni eliminar existentes. Además, los ficheros de prueba tampoco funcionarán. De igual forma, tampoco se ha de cambiar los puertos asignados para ambos contenedores y, si se hace, modificar las variables de entorno correspondientes.

4. Pruebas

En cuanto a las pruebas ejecutadas para observar el correcto funcionamiento de la práctica, hemos diseñado un script localizado en la carpeta tests llamado client.py. En este fichero, se ejecutan pruebas de caja blanca que revisan el comportamiento del programa.

Se ejecuta como un script de Python usual y el único requisito para **funcionar correctamente es tener activos ambos servidores y haber limpiado los datos de los volúmenes si el script ha sido ejecutado previamente.** Hemos preferido que esta prueba deje “restos” en los volúmenes de datos para poder analizar los resultados del programa sin tener que debuggear y parar la ejecución del mismo. Por ende, hay que vaciar los datos de los volúmenes manualmente si se quiere ejecutar la prueba una segunda vez.

A continuación, se muestran capturas de la ejecución de este archivo. Primero, hay que tener ambos servidores corriendo:

```
PS C:\Users\usuario\PycharmProjects\Sistemas-Informaticos> docker compose up --build
[+] Building 12.1s (10/11)
 => [file_app auth] library/python:pull token for registry-1.docker.io
 => [file_app internal] load .dockerignore
 => => transferring context: 2B
 => [file_app 1/6] FROM docker.io/library/python:latest@sha256:45803c375b95ea33f482e53a461
 => [file_app internal] load build context
 => => transferring context: 6.71kB
 => CACHED [file_app 2/6] WORKDIR /library_service
 => [file_app 3/6] COPY library_server/file.py ./
 => [file_app 4/6] COPY utils.py ./
```

```
[+] Running 5/5
 ✓ Network sistemas-informaticos_default       Created
 ✓ Volume "library_data"                       Created
 ✓ Volume "users_data"                         Created
 ✓ Container sistemas-informaticos-file_app-1  Created
 ✓ Container sistemas-informaticos-user_app-1  Created
Attaching to file_app-1, user_app-1
file_app-1  | [2024-10-14 07:10:49 +0000] [1] [INFO] Running on http://172.18.0.2:5010
user_app-1  | [2024-10-14 07:10:50 +0000] [1] [INFO] Running on http://172.18.0.3:5005
```

A continuación, una captura de la funcionalidad básica del servicio de usuarios. El test client.py realiza más acciones y se recomienda ejecutarlo para observar el correcto funcionamiento.

```
CREATING USER: (New user)
{"access_token":"d55b7c41-eea1-5608-a6c3-07af8892dc8e","uid":"1c739253-6177-47e3-af43-49c94be69f95"}

CREATING USER: (New user)
{"access_token":"49777b48-c617-557f-aeae-277b095d99a7","uid":"7c0fa185-ee96-4ab3-8cd1-4344752ef49b"}

CREATING USER: (Already existing user)
<!DOCTYPE html>
<html lang="en">
<head>
 <meta charset="UTF-8">
 <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>400 Bad Request</title>
</head>
<body>
    <h1>400 Bad Request</h1>
    <p>   There is something wrong with the request and could not be processed.   </p>
   <p>  The user 'hola1' already exists.   </p>
</body>
</html>

LOGIN USER: (Valid credentials)
{"access_token":"d55b7c41-eea1-5608-a6c3-07af8892dc8e","uid":"1c739253-6177-47e3-af43-49c94be69f95"}
```

En cuanto a lo que debería realizar los tests. Deberían crear dos ususarios, eliminar uno y volver a crearlo. Después, añadir dos ficheros a la biblioteca de uno de ellos, actualizar uno de ellos y borrar otro. Así, en el volumen de user deberíamos tener, al final, el registro de dos usuarios y en el volumen de biblioteca, un directorio vacío y otro conteniendo un archivo. Veamos que así es: