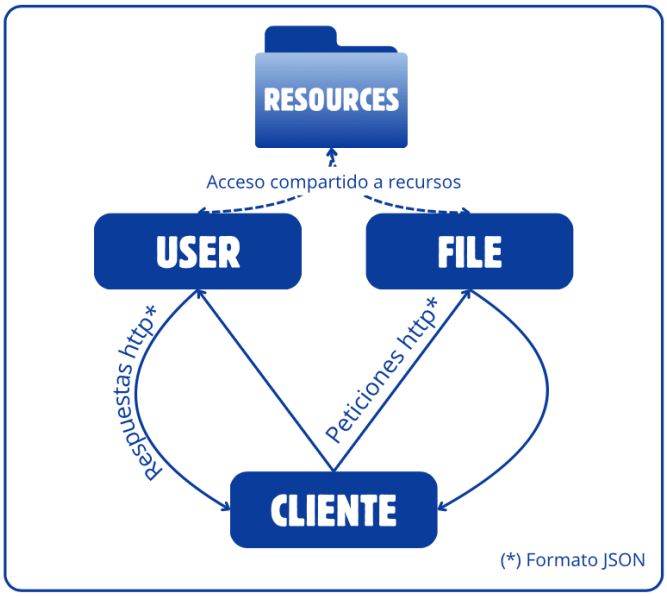
PRACTICA 1
Micro-servicios
REST API en contenedor

Ana Pardo Jiménez
ana.pardoj@estudiante.uam.es

Juan Larrondo Fernández de Córdoba
juan.larrondo@estudiante.uam.es

INTRODUCCIÓN

En esta práctica hemos realizado el código de un sistema aislado mediante docker, en el que implementamos un sistema gestor de archivos. Para este hemos incluido dos APIs, una de usuario y otra de file de tal forma que el sistema sigue la siguiente estructura:



La carpeta resources es donde se guardan todos los archivos de usuario y las bibliotecas. Sigue la siguiente estructura:

```
resources/
├── users.txt
└── files/
    ├── uid1.txt
    ├── uid2.txt
    ├── uid3.txt
    └── uid4.txt
```

Estos archivos los gestionamos mediante la librería pandas de python y en ellos guardamos la información de los usuarios (Nombre, contraseña y uid) y de los archivos de sus bibliotecas (dentro de uid1.txt guardamos nombre de archivo, visibilidad y contenido).

API USER

Los usuarios contienen toda su información de usuario (contraseña, nombre, uid...) en un archivo dentro de resources llamado users.txt. Este archivo no existe en el momento de ejecutar la api por primera vez, sino que se crea con la primera creación de un usuario.

La API user incluye todos los handlers necesarios para gestionar los usuarios. Cada handler tiene una url (ruta en el que se especifica, en nuestro caso, el nombre del usuario que se quiere crear/gestionar) y, en caso de necesitarlo, un cuerpo JSON con el que se pasan el resto de argumentos necesarios, como la contraseña. Todos los handlers devuelven un objeto JSON con el estado de la ejecución (OK/ERROR) y otra información necesaria en caso de éxito o un mensaje de error en caso de fallo. Además devuelven un código de estado HTTP (200, 404, 500...) dependiendo del resultado de la ejecución.

Los handlers son los siguientes:

- http_create_user:
  - **Url**: http://0.0.0.0:5050/create_user/<nombre_usuario>
  - **JSON**:
    {
     “password”: <contraseña>
    }
  - **Retorno JSON en éxito**:
    {
     'status': 'OK',
     'username': <nombre_usuario>,
     'UID': <uid>,
     'Token': <token>
    }

- http_login:
  - **Url**: http://0.0.0.0:5050/login/<nombre_usuario>
  - **JSON**:
    {
     “password”: <contraseña>
    }
  - **Retorno JSON en éxito**:
    {
     'status': 'OK',
     'UID': <uid>,
     'Token': <token>
    }

- http_get_user_id:
  - **Url**: http://0.0.0.0:5050/get_user_id/<nombre_usuario>
  - **Retorno JSON en éxito**:
    {

'status': 'OK',
'UID': <uid>,
'username': <nombre_usuario>
}

- http_change_pass:
  - **Url**: http://0.0.0.0:5050/change_pass/<nombre_usuario>
  - **JSON**:
    {
    "password": <contraseña>,
    "new_password": <nueva_contraseña>
    }
  - **Retorno JSON en éxito**:
    {
    'status': 'OK'
    }

- http_change_username:
  - **Url**: http://0.0.0.0:5050/change_username/<nombre_usuario>
  - **JSON**:
    {
    "password": <contraseña>,
    "new_username": <nuevo_nombre>
    }
  - **Retorno JSON en éxito**:
    {
    'status': 'OK'
    }

- http_delete_user:
  - **Url**: http://0.0.0.0:5050/delete_user/<nombre_usuario>
  - **JSON**:
    {
    “password”: <contraseña>
    }
  - **Retorno JSON en éxito**:
    {
    'status': 'OK'
    }

Además de las funciones que implementan la funcionalidad de cada uno de los handlers (de mismo nombre, quitando “http_”), hay otras funciones auxiliares:

- open_or_create_txt() y open_users_txt(): Como dice el nombre, sirven para abrir o crear en caso de no existir (o solo abrir) el archivo que contiene la información de los usuarios.
- handle_file_error(error_code): Para gestionar los posibles errores provenientes de abrir el archivo users.txt.

API FILE

Cada usuario hace uso de una biblioteca que se genera en el momento de creación del usuario. Estos pueden:

- Añadir ficheros
- Listar los ficheros de la biblioteca (la suya propia)
- Descargar/leer ficheros de su biblioteca y archivos de otras en caso de:
    - Ser públicas y tener la uid del usuario propietario
    - Tener un share token en un tiempo válido
- Borrar archivos (solo los suyos)

Para ello hemos implementado las siguientes funciones privadas y sus correspondientes funciones http:

- http_create_file:
    - **Url:** http://0.0.0.0:5051/create_file
    - **JSON:**
      {
       "uid": "user123",
       "filename": "mi_archivo.txt",
       "content": "Contenido del archivo",
       "visibility": "private"   // opcional: "public" o "private"
      }
    - **Retorno JSON en éxito:**
      {
       "ok": true,
       "uid": "user123",
       "filename": "mi_archivo.txt",
       "visibility": "private"
      }

- http_modify_file:
    - **Url:** http://0.0.0.0:5051/modify_file
    - **JSON:**
      {
       "uid": "user123",
       "filename": "mi_archivo.txt",
       "new_content": "Nuevo contenido del archivo",
       "visibility": "public"   // opcional
      }
    - **Retorno JSON en éxito:**
      {
       "ok": true,
       "uid": "user123",
       "filename": "mi_archivo.txt",
       "visibility": "public"

}

- http_remove_file:
  - **Url:** http://0.0.0.0:5051/remove_file
  - **JSON:**
    {
     "uid": "user123",
     "filename": "mi_archivo.txt"
    }
  - **Retorno JSON en éxito:**
    {
     "ok": true,
     "uid": "user123",
     "filename": "mi_archivo.txt"
    }
- http_read_file:
  - **Url:** http://0.0.0.0:5051/read_file
  - **JSON:**
    {
     "uid": "user123",
     "filename": "mi_archivo.txt"
    }
  - **Retorno JSON en éxito:**
    {
     "ok": true,
     "uid": "user123",
     "filename": "mi_archivo.txt",
     "content": "Contenido del archivo"
    }
- http_list_files:
  - **Url:** http://0.0.0.0:5051/list_files
  - **JSON:**
    {
     "uid": "user123"
    }
  - **Retorno JSON en éxito:**
    {
     "ok": true,
     "uid": "user123",
     "files": [
      "mi_archivo.txt",
      "otro_archivo.txt"
     ]
    }

Todas contienen además un encabezado obligatorio (excepto en read_file que es opcional) con el token del cliente correspondiente:

Authorization: Bearer <token>

Al igual que en la API de user atendemos las peticiones JSON al igual que damos respuesta a las mismas con dicho formato, pero internamente lo gestionamos con las funciones privadas.

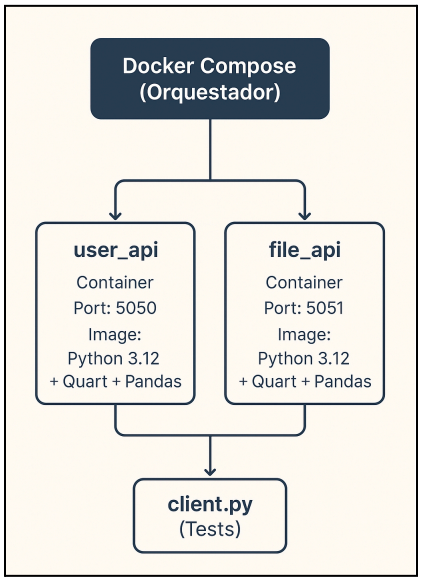
DOCKER

Para el despliegue del sistema de la práctica con docker hemos usado cuatro archivos: Dockerfile, docker-compose.yml, .dockerignore y requirements.txt.

Los archivos de las diferentes APIs están todos en el mismo directorio, de modo que no tenemos que preocuparnos de la comunicación entre APIs que no "se conocen" entre sí. Como consecuencia de esto, aquellas instrucciones para la construcción del contenedor que estén en el Dockerfile se aplicarán por igual a ambas APIs. Esto no es un problema en esta práctica por la simplicidad del sistema.

El Dockerfile usa como imagen base python:3.12. Copia únicamente el archivo de dependencias (requirements.txt) dentro del directorio base del contenedor (/app) instala las dependencias y expone ambos puertos, 5050 y 5051. Esto lo hace para ambas APIs, aunque cada una solo use uno de los puertos, porque se usa el mismo Dockerfile para ambas.

El archivo docker-compose-yml se encarga de orquestar los dos contenedores de los servicios user y file. Define los dos servicios (user_api y file_api), y para cada uno crea la imagen, mapea los puertos correctos (previamente expuestos por el Dockerfile), crea un volumen del directorio de la práctica en el directorio base del contenedor y define el comando a ejecutar para ejecutar cada servidor.



Por otra parte, el archivo .dockerignore incluye aquellos archivos y directorios que no queremos copiar ni incluir en el volumen de los contenedores. Y por último, requirements.txt, en nuestro caso, contiene las únicas librerías que se necesita instalar en nuestro caso: pandas y quart.

Requisitos del Sistema

- Software Requerido:

Componente	**Versión Mínima**
Docker	20.10+
Docker Compose	1.29+
Python	3.12

- Dependencias del Cliente:

Librería	**Versión**
requests	2.25+
pytest	6.0+

Probar el funcionamiento del sistema:

1. Comprobar limpieza de contenedores, imágenes, etc. (Copiar en terminal):

   ```
   # Contenedores
   sudo docker ps -a
   # Imágenes
   sudo docker images
   # Networks
   sudo docker network ls
   # Volúmenes
   sudo docker volume ls
   # Directorios
   ls -la resources/
   ls -la resources/files/
   ```

2. Construir imágenes, contenedores y lanzarlos:

   ```
   sudo docker-compose up --build
   ```

3. Ejecutar client.py en otra terminal (Flag -v o -vv para más información):

   ```
   sudo pytest client.py
   ```

Estructura del Proyecto:

```
./
├── client.py                 # Cliente de pruebas
├── user.py                   # API de gestión de usuarios
├── file.py                   # API de gestión de archivos
├── docker-compose.yml        # Configuración de contenedores
├── Dockerfile                # Imagen base de contenedores
├── requirements.txt          # Dependencias de Python
├── .dockerignore             # Archivos ignorados por Docker
└── resources/                # Directorio de datos (auto-creado)
    ├── users.txt             # Base de datos de usuarios
    └── files/                # Archivos de usuarios
        └── <uid>.txt         # Archivos por usuario
```

PRUEBAS

El fichero client.py implementa un cliente de pruebas completo para el sistema de gestión de usuarios y archivos desarrollado en la práctica.

Su función principal es verificar automáticamente que todos los endpoints de la API REST de los servidores user.py (puerto 5050) y file.py (puerto 5051) funcionen correctamente.

Estas pruebas se ejecutan utilizando el framework pytest, lo que permite automatizar el proceso de comprobación, validar las respuestas HTTP y garantizar que el sistema cumpla con las especificaciones esperadas.

Funcionalidades principales

El cliente ejecuta una batería de pruebas que cubre **todo el ciclo de vida del sistema**:

Gestión de usuarios
- Creación de usuarios nuevos y detección de duplicados.
- Inicio de sesión (login) con credenciales correctas e incorrectas.
- Modificación de contraseña y nombre de usuario.
- Eliminación de usuarios y comprobación de errores por validaciones o inexistencia.

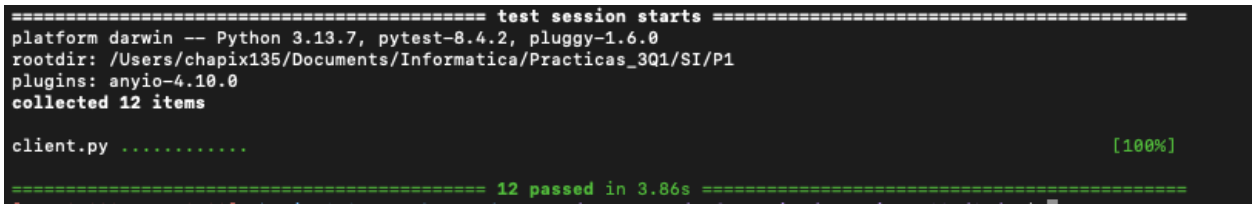
Gestión de archivos
- Creación de archivos privados y públicos.
- Lectura, modificación, eliminación y listado de archivos.
- Verificación de permisos: acceso autorizado y no autorizado.

Tokens de compartición
- Generación de **tokens temporales de compartición** para acceder a archivos privados.
- Comprobación del acceso válido y del comportamiento ante tokens expirados.

Control de errores y validaciones
- Casos de prueba con cuerpos JSON ausentes o incompletos.
- Validación de respuestas con códigos de estado HTTP correctos (200, 201, 400, 401, 403, 404).
- Limpieza automática de los archivos de prueba antes y después de ejecutar los tests.

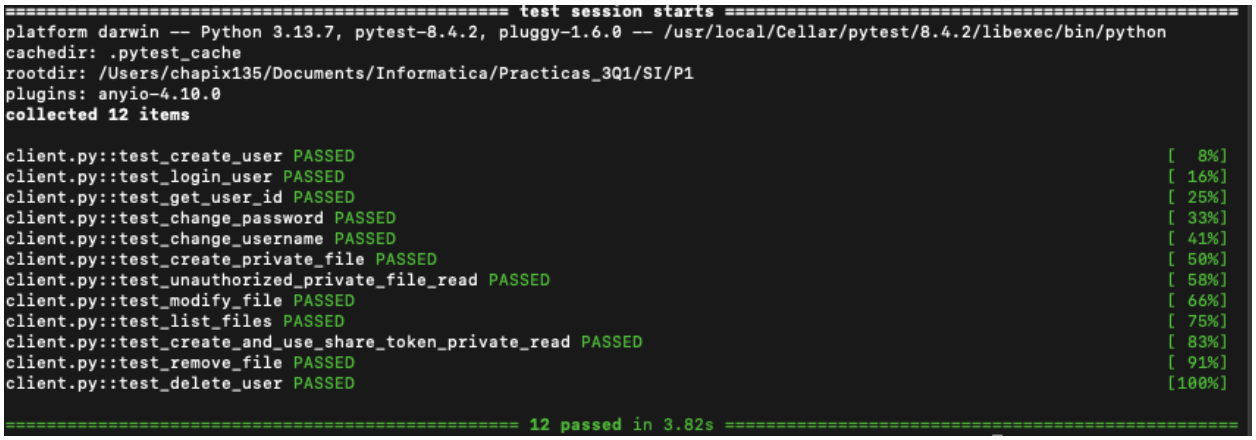
Para utilizar el cliente automatizado primero se deberá poner en marcha el docker y las API tal y como se especifica en el apartado anterior. Para comprobar el cliente puede ejecutar directamente el comando:

pytest client.py



Para una versión de los test menos completa. Aunque se recomienda ejecutar con:

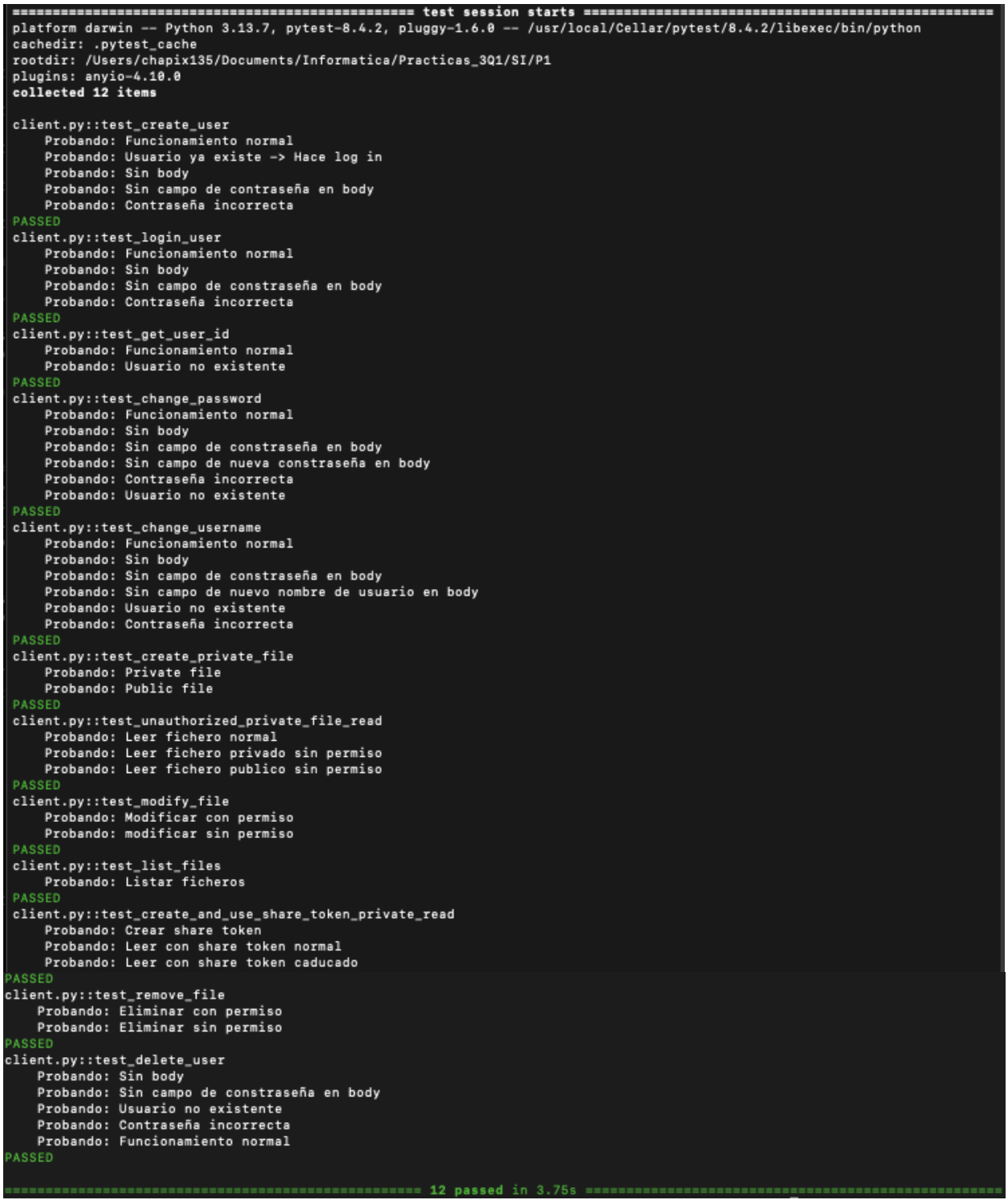
pytest client.py -vv



Para una mejor visualización del resultado de los test tal y como se muestra a continuación.

Asimismo, se puede ejecutar con el siguiente comando:

pytest client.py -vv -s



Para la visualización más completa de los test, que muestra prints con las diferentes partes del test y las comprobaciones que se están realizando.