Práctica 1: Micro-servicios

Parte 1: REST API en contenedor

Versión 1.0 11/09/2023

1. Objetivos docentes

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

- Desarrollar un micro-servicio en Python que implemente una REST API
- Gestionar contenedores:
  - descargar de imágenes
  - ejecución de contenedores
  - revisión de imágenes y contendores
- Desplegar un micro-servicio en un contenedor:
  - creación de un Dockerfile
  - generación de una imagen
- Gestionar un registro de contenedores local
- Usar servicios cloud:
  - Usar localStack para probar los servicios en local
  - Usar los servicios de almacenamiento de archivos y datos S3 y DynamoDB
  - Implementar y desplegar funciones Lambda
  - Crear un API de un microservicio e integrarlo con funciones Lambda

2. Introducción

Se pretende explorar la implementación de una alternativa al proyecto Solid (ver bibliografía): Un repositorio de archivos con información relativa a un usuario (historial médico, compras, ...) que pueda ser accedida por terceros para su creación o modificación, siempre según los permisos otorgados por ambas partes.

Por ejemplo, una entidad sanitaria estará autorizada para crear un nuevo documento (por ejemplo, el resultado de unos análisis), pero no estará autorizada a borrarlo, y el usuario no estará autorizado a modificarlo.

En esta práctica se desarrollarán unos prototipos básicos de los micro-servicios para gestionar los usuarios y los archivos de un usuario.

El desarrollo de la práctica se ha dividido en dos partes, en la primera se trabajarán los aspectos más básicos, y en la segunda se utilizarán servicios cloud, usando localstack, y la arquitectura serverless, para el despliegue de nuestros microservicios.

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, normalmente fuertemente imbricada (tight coupling), en una arquitectura de microservicios cada funcionalidad se desarrolla y despliega de forma independiente, desacoplada (loose coupling) 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.

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).

Tarea: entorno python

Ejecuta en un terminal:

```
$> mkdir -p venv/si1p1
$> python3 -m venv venv/si1p1
$> source ./venv/si1p1/bin/activate
$> pip install hypercorn quart localstack awscli awscli-local
```

Con esto tendremos los paquetes necesarios para esta práctica.

Recuerda usar este entorno python cada vez que necesites ejecutar hypercorn, quart, localstack, aws, awslocal, o ejecutar código python que necesite estas bibliotecas.

Cuidado: Si se desea, se puede copiar el directorio venv/si1p1 a una memoria USB, pero hay que tener presente que cuando se copie en otro PC debe encontrase en la misma ruta absoluta, ya que hay archivos que la contienen (v. g. venv/si1p1/bin/activate)

4.2. Micro-servicio 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.

Tarea: revisión de user_rest.py

Revisar el archivo src/user_rest.py suministrado y la bibliografía de Quart y Flask.

Pregunta: user_rest.py

1. ¿Qué crees que hacen las anotaciones @app.route, @app.get, @app.put?
2. ¿Qué ventajas o inconvenientes crees que pueden tener?

Tarea: ejecución de user_rest.py

Ejecuta en un terminal:

$> source venv/si1p1/bin/activate \

$> [[-d build/users]] || mkdir -p build/users \

$> QUART_APP=src.user_rest:app quart run –p 5050

Pregunta: ejecución de user_rest.py

¿Qué crees que ha sucedido?

Tarea: petición a user_rest.py

Ejecuta en otro terminal

```
$> curl -X PUT \
  http://localhost:5050/user/myusername \
  -H 'Content-Type: application/json' \
  -d '{"firstName": "myFirstName"}'
```

Pregunta: petición a user_rest.py

Revisa la ayuda de curl (man curl) y contesta:

1. ¿Qué crees que ha sucedido al ejecutar el mandato curl?
2. ¿Cuál es la respuesta del microservicio?
3. ¿Se ha generado algún archivo? ¿Cuál es su ruta y contenido?

Tarea: petición HTTP

Revisa la ayuda de nc (man nc) y ejecuta en otro terminal este programa en modo escucha en el puerto 8888.

Ejecuta de nuevo curl, pero haciendo la petición al puerto 8888

Pregunta: petición HTTP

1. ¿Cuál es la petición HTTP que llega al programa nc?
2. ¿En qué campo de la cabecera HTTP se indica el tipo de dato del cuerpo (body) de la petición y cuál es ese tipo?
3. ¿Qué separa la cabecera del cuerpo?

Tarea: prototipo user

Completa el micro-servicio user:

1. implementa el método DELETE para eliminar un usuario
2. implementa el método PATCH para actualizar algún campo del usuario

4.3. Despliegue con Hypercorn

Hypercorn es una evolución de Gunicorn que aprovecha las características de Quart frente a Flask.

Es un programa que permite dotar de encriptación TLS a las peticiones realizadas a los microservicios desarrollados en Quart así como escalar horizontalmente ejecutando instancias del microservicio en paralelo.

Tarea: ejecución hypercorn

Ejecuta en un terminal:

```
$> hypercorn --bind 0.0.0.0:8080 --workers 2 src.user_rest:app
```

Pregunta: ejecución hypercorn

1. ¿Qué crees que ha sucedido?
2. ¿Qué petición curl debes hacer ahora para hacer un GET de un usuario? ¿Qué ha cambiado respecto de la ejecución sin hypercorn?

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

En los laboratorios están instalados tanto los paquetes de docker como de podman, una alternativa a docker mucho más orientada al modo rootless.

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

En principio, podman es más potente que docker rootless y tiene mucha más actividad en cuanto a nueva funcionalidad. Sin embargo, de momento presenta alguna incompatibilidad con localstack (ver siguiente parte de la práctica), así es que usaremos docker rootless o docker desktop. Docker desktop no está instalado en los laboratorios, pero puedes usarlo para ejecutar docker tanto en linux o mac como en Windows (junto a WSL2).

Tarea: arranque de docker rootless

Ejecuta en un terminal:

```
# doc de configuración de usuario necesaria
$> man subuid
# EPS labs: establece configuración de usuario
$> sudo /usr/local/bin/si2setuid.sh 3000000:100000 $USER
# HOME: establece configuración de usuario
# - edita como sudo los archivos /etc/subuid y /etc/subgid e introduce entradas
#   semejantes a las de los labs, con el nombre de tu usuario en tu PC
# sudo vim /etc/subuid
# sudo vim /etc/subgid
# arranque del demonio
$> 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
```

Pregunta: modo rootless

Revisa el contenido de /etc/subuid y la documentación de docker rootless: ¿para qué crees que sirve ese archivo?

5.3. Contenedores docker

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.

Al ejecutar un contenedor basado en una imagen, se hace una réplica de la imagen (si bien puede que no haya una copia física inicial de todos los archivos) y se ejecuta el programa indicado. De esta forma, se pueden tener tantas réplicas del contenedor (aplicación) como se desee, siempre que haya recursos en el servidor.

Revisa la documentación del cliente de docker con el mandato man o en la bibliografía

Los comandos docker que se indican a continuación sirven tanto para docker rootles como para docker desktop.

Tarea: docker info

Ejecuta en un terminal:

```
$> docker info
```

Pregunta: docker info

1. ¿Dónde se guarda la configuración del demonio arrancado?
2. ¿Dónde se almacenan los contenedores?
3. ¿Qué tecnología de almacenamiento se usa para los contenedores? ¿Qué es copy-on-write? ¿Lo usa docker?

Tarea: run whoami

Ejecuta en un terminal:

```
$> docker run alpine whoami
```

Pregunta: run whoami

1. ¿Qué crees que ha sucedido?
2. ¿Crees que el usuario root del contenedor es el mismo que el del host?

Tarea: list images

Ejecuta en un terminal:

```
$> docker images
```

y revisa su resultado.

Pregunta: list images

¿Qué crees que tiene que ver con el mandato docker pull?

Tarea: docker ps

Ejecuta en un terminal:

```
$> docker ps -a
```

Pregunta: docker ps

¿Qué crees que muestra la salida?

Tarea: docker pull y run -ti

Ejecuta en un terminal:

```
$> docker pull ubuntu:22.04
$> docker run -ti --name u22.04 ubuntu
```

Pregunta: docker pull y run -ti

1. ¿Qué hacen las opiones -ti?
2. ¿Qué ha sucedido?

Sal del contenedor con Ctrl-D ó exit.

Tarea: docker inspect

Revisa la ayuda del mandato jq (man jq).
Ejecuta en un terminal:

```
$> docker inspect u22.04 | jq '.[].Config.Cmd[]'
```

Pregunta: docker inspect

1. ¿Qué hace el mandato jq?
2. ¿Qué tiene que ver el resultado del mandato con la tarea anterior?

Tarea: docker start

Ejecuta en un terminal:

```
$> docker start u22.04
```

Pregunta: docker start

1. ¿Qué ha sucedido?
2. ¿Cómo se borra el contenedor u22.04?

Tarea: docker run -d

Ejecuta en un terminal:

```
$> docker run -ti -d --name u22.04d ubuntu
```

Pregunta: docker run -d

¿Qué hace la opción -d?

Tarea: docker exec

Ejecuta en un terminal:

```
$> docker exec -ti u22.04d /bin/bash
```

y dentro del contenedor:

```
$> mkdir -p /si1/users
$> touch /si1/users/test
```

Pregunta: docker exec

¿Qué ha sucedido?

Sal del contenedor

Tarea: docker volume

Ejecuta en un terminal:

```
$> [[ -d si1/users/testv ]] || mkdir -p si1/users/testv
$> docker run -ti -v ${PWD}/si1/users/testv:/si1/users/testv alpine \
  touch /si1/users/testv/afile
$> docker run -ti -v ${PWD}/si1/users/testv:/si1/users/testv alpine \
  ls -al /si1/users/testv/afile
$> ls -al si1/users/testv/afile
$> echo "a test" >>si1/users/testv/afile
$> docker run -ti -v ${PWD}/si1/users/testv:/si1/users/testv alpine \
  cat /si1/users/testv/afile
```

Pregunta: docker volume

1. ¿Qué hace la opción -v?
2. ¿Qué ha sucedido al ejecutar los mandatos anteriores?

Tarea: limpieza
Revisa la documentación y elimina todos los contenedores e imágenes empleados.

Pregunta: limpieza
1. ¿Qué mandato has usado para eliminar los contenedores?
2. ¿Qué mandato has usado para eliminar las imágenes?

5.4. Creación de una imagen de contenedor con un Dockerfile

En este apartado vamos a crear la imagen base de nuestro microservicio de usuarios

Tarea: Dockerfile
Revisa la documentación sobre el Dockerfile en la bibliografía y el archivo src/Dockerfile suministrado.

Pregunta: Dockerfile
1. ¿Qué hace la sentencia FROM?
2. ¿Qué hace la sentencia ENV?
3. ¿Qué diferencia presenta ENV respecto de ARG?
4. ¿Qué hace la sentencia RUN?
5. ¿Qué hace la sentencia COPY?
6. ¿Qué otra sentencia del Dockerfile tienen un cometido similar a COPY?
7. ¿Qué hace la sentencia EXPOSE?
8. ¿Qué hace la sentencia CMD?
9. ¿Qué otras sentencias del Dockerfile tienen un cometido similar a CMD?

Tarea: docker build
Ejecuta en un terminal

```
$> cd src
$> docker build --tag si1p1:latest .
$> docker build --tag si1p1:1.0 .
$> docker images
$> cd -
```

Cuidado con el '.' al final del mandato, indica la ubicación (ruta) del archivo Dockerfile.

Pregunta: docker build

1. ¿Por qué es mucho más rápido la creación de la segunda imagen?
2. Las imágenes constan de distintas capas: ¿Qué quiere decir esto?

Tarea: docker build with user

Descomenta las líneas del Dockerfile que se refieren al usuario si1 y vuelve a construir la imagen, esta vez con el tag 1.0u.

Pregunta: docker build with user

¿Por qué en los últimos pasos no utiliza la caché?

Tarea: docker run myimage

Ejecuta en un terminal (borra el contenedor si1p1 si ya existía):

```
$> docker run --name si1p1 -e NUMWORKERS=5 -d -p 8080:8000 si1p1
$> docker exec -ti si1p1 /bin/bash
$> # … dentro del contenedor
$> ps --forest -aef
$> exit
```

Para borrar el contenedor, ejecuta:

```
$> docker rm -f si1p1
```

Pregunta: docker run myimage

1. ¿Cuántos subprocesos de hypercorn aparecen? ¿Por qué?
2. ¿Qué hace la opción '-p' del comando docker run?

3. Si deseáramos ejecutar otra réplica (contenedor) del microservicio, ¿qué deberíamos cambiar en la sentencia docker run?

Tarea: docker run myimage on ./si1

Usando un volumen (opción -v), crea en local el directorio ./si1 y arranca otra réplica de nuestra aplicación montando el directorio creado en el directorio /si1 del contenedor.

Pregunta: docker run myimage on ./si1

1. ¿Qué mandato has ejecutado?
2. De cara a la realización de backups, ¿cuál crees que es la utilidad de montar volúmenes externos respecto de usar los internos de docker?

5.5. Docker registry

Las imágenes de los contenedores se almacenan habitualmente en repositorios conocidos como registries; por ejemplo docker hub de la compañía Docker o quay.io de Redhat (ver bibliografía).

En este apartado vamos a arrancar nuestro propio registry, para poder trabajar sin acceso a internet, lo cuál suele ser lo habitual en servidores de empresa.

Tarea: docker registry

Localiza en docher hub la imagen registry, revisa sus tags y su documentación y descarga la última versión con el mandato de docker adecuado.

Arranca un contenedor de dicha imagen con el nombre si1_registry y escuchando en el puerto 5050.

Pregunta: docker registry

1. ¿Qué mandato has usado para descargar la imagen del registry?
2. ¿Cuál es el número de versión de dicha imagen?
3. ¿Qué mandato has usado para ejecutar el contenedor del registry?
4. ¿En qué puerto escucha por defecto el contenedor del registry?

Tarea: docker tag push

Descarga la imagen de ubuntu, si no lo has hecho ya.

Ejecuta los siguientes mandatos (asumiendo que el registro está escuchando en el puerto 5000):

```
$> docker tag ubuntu:latest localhost:5000/ubuntu:latest
$> docker push localhost:5000/ubuntu:latest
$> docker images
```

Pregunta: docker tag push

¿Qué ha sucedido?

Tarea: app from local

Comenta la sentencia FROM del Dockerfile e introduce una nueva que use nuestra copia local de Ubuntu.

Genera la imagen de nuestra aplicación y súbela a nuestro registro.

Ejecuta un contenedor con nuestra aplicación usando la imagen de nuestro registro.

Pregunta: docker app from local

¿Qué mandatos has usado?

6. Documentación a aportar

En esta parte de la práctica deberéis entregar:

1. Memoria:
    1. Respuesta a las preguntas planteadas.
    2. Listado de archivos empleados junto con una breve descripción.
    3. Detalles de implementación relevantes.
2. Archivos, al menos:
    1. src/Dockerfile
    2. src/user_rest.py

7. Bibliografía

Bibliografía de los distintos apartados.

7.1. Proyecto Solid

1. Tim Berners-Lee Solid Pods: https://solidproject.org/

7.2. Python

1. Aprendizaje interactivo de Python: http://www.diveintopython3.net
2. Uso de JSON en Python: https://docs.python.org/2/library/json.html
3. Tipos de datos: https://docs.python.org/2/tutorial/datastructures.html

7.3. REST API

1. HTTP codes: https://en.wikipedia.org/wiki/List_of_HTTP_status_codes
2. manifesto: https://medium.com/@sathyap/the-api-manifesto-3bc587cfb007
3. constraints: https://restfulapi.net/rest-architectural-constraints/
4. openapi doc: https://swagger.io/docs/specification/about/
5. api design swagger: https://swagger.io/blog/api-design/api-design-best-practices/
6. api design google: https://cloud.google.com/blog/products/application-development/api-design-why-you-should-use-links-not-keys-to-represent-relationships-in-apis
7. naming: https://restfulapi.net/resource-naming/
8. ejemplo: https://www.marqeta.com/docs/core-api/card-products
9. petición asíncrona: https://docs.microsoft.com/es-es/azure/architecture/patterns/async-request-replyv

7.4. Flask y Quart:

1. quart quickstart: https://pgjones.gitlab.io/quart/tutorials/quickstart.html

2. quart API reference: https://pgjones.gitlab.io/quart/reference/api.html
3. quart cheatsheet: https://pgjones.gitlab.io/quart/reference/cheatsheet.html
4. flask quickstart: https://flask.palletsprojects.com/en/2.2.x/quickstart/
5. flask tutorial: https://flask.palletsprojects.com/en/2.2.x/tutorial/

7.5. Gunicorn y Hypercorn

1. Hypercorn: https://hypercorn.readthedocs.io/en/latest/
2. Gunicorn: https://docs.gunicorn.org/en/stable/

7.6. Docker y Podman

1. docs: https://docs.docker.com/
2. command line: https://docs.docker.com/engine/reference/commandline/cli/
3. dockerfile: https://docs.docker.com/engine/reference/builder/
4. docker build: https://docs.docker.com/engine/reference/commandline/build/
5. docker hub: https://hub.docker.com/
6. rootless: https://docs.docker.com/engine/security/rootless/
7. docker daemon configuration: https://docs.docker.com/engine/reference/commandline/dockerd/#daemon-configuration-file
8. docker registry API: https://docs.docker.com/registry/spec/api/
9. Podman doc: https://docs.podman.io/en/latest/index.html