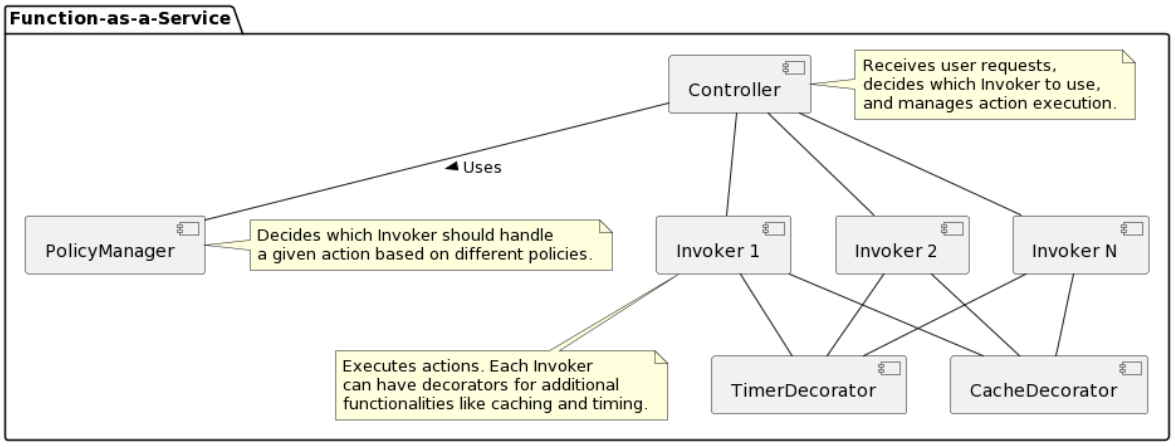
TAP - Diseño e implementación de un sistema de Funciones (Function as a Service) - Versión 2

Function-as-a-Service (FaaS) es un paradigma Cloud que permite la ejecución de código sencillo, llamadas *funciones* o *acciones*, en la nube en servidores virtualizados. En un sistema FaaS, los usuarios pueden registrar, listar, invocar y eliminar acciones. Para registrar una nueva acción, el usuario debe definir su código, e indicar un identificador y la cantidad de memoria RAM (en megabytes) que necesita la acción. Tras crear una acción, ésta se puede invocar mediante el identificador pudiendo indicar además unos parámetros de entrada.

Queremos modelar con clases un sistema FaaS completo que permita la ejecución de funciones de manera adaptativa a los recursos disponibles. Nos basaremos en la arquitectura de OpenWhisk, un sistema de funciones de código abierto creado por IBM, usado internamente como servicio en IBM Cloud.

En OpenWhisk, la arquitectura está principalmente formada por dos componentes: Un Controller y múltiples Invokers.

- El **Controller** es responsable de recibir las peticiones de invocación de los usuarios y seleccionar los Invokers que ejecutarán las acciones. El controller es único, y tiene una vista global de los Invokers disponibles y los recursos libres de cada uno.
- El **Invoker** recibe las órdenes de ejecución de acciones del Controller, y es el responsable de reservar los recursos (memoria) necesarios y ejecutar el código de las acciones. Para simular la reserva de memoria, indicaremos al Invoker la cantidad total de memoria RAM en megabytes en un atributo, que iremos modificando a medida que se le asignan acciones. **El número de invokers es constante durante toda la ejecución del programa y se indica al inicializar el programa.**



Un ejemplo de la interfaz para crear e invocar una acción podría ser el siguiente:

```
public class Main {
    public static void main(String[] args) {
        Controller controller = new Controller(4, 1024);
        Function<Map<String, Integer>, Integer> f = x -> x.get("x") + x.get("y");
        controller.registerAction("addAction", f, 256);
        int res = (int) controller.invoke("addAction", Map.of("x", 6, "y", 2));
        System.out.println(res);
    }
}
```

En este ejemplo, creamos el componente Controller, indicando que queremos usar 4 Invokers con capacidad de 1024MB de memoria. Definimos una acción *en línea* usando Java Functions que recibe un *Map* por parámetro y que retorna la suma de los dos valores con las claves “x” e “y”. Registramos la acción indicando el identificador “addAction” y que ocupa 256MB de memoria. Finalmente invocamos la función mediante el método *invoke* del Controller, que ejecutará la lógica de delegar la ejecución de la función a algún Invoker, y finalmente retorna el resultado.

Invocaciones Grupales

Al Invoker le llegarán peticiones de ejecutar una única función o un grupo de funciones (el mismo código ejecutado N veces) si se indica una lista de parámetros. Por ejemplo, podríamos ejecutar:

```
List<Map<String, Integer>> input = Arrays.asList(new Map[]{
    Map.of("x", 2, "y", 3),
    Map.of("x", 9, "y", 1),
    Map.of("x", 8, "y", 8),
});
List<Integer> result = controller.invoke("addAction", input);
System.out.println(result.toString());
```

El Controller se encargaría de distribuir las funciones de esta ejecución grupal sobre uno o más Invokers si tienen disponibilidad de recursos. Las llamadas en grupo solo se deben ejecutar si tenemos recursos disponibles para ejecutar todo el grupo a la vez. Si no se pueden denegar o retrasar hasta haber liberado recursos.

Multithreading e invocaciones asíncronas

Queremos evitar bloquear el hilo de ejecución del programa principal esperando el resultado de acciones de larga duración. Para ello, implementaremos un método *invoke_async* que invoque acciones de manera asíncrona. Con ello, podremos invocar múltiples acciones en *background* de manera concurrente.
En las invocaciones asíncronas, en lugar de devolver el resultado de la acción, devolveremos un *Future*. Un Future es un *placeholder* a un resultado de una función que todavía no está disponible. Podemos guardar el Future en una variable, y esperar al resultado más adelante en el código. Por ejemplo:

```
Function<Integer, String> sleep = s -> {
    try {
        Thread.sleep(Duration.ofSeconds(s));
        return "Done!";
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
};
controller.registerAction("sleepAction", sleep);
ResultFuture fut1 = controller.invoke_async("sleepAction", 5);
ResultFuture fut2 = controller.invoke_async("sleepAction", 5);
ResultFuture fut3 = controller.invoke_async("sleepAction", 5);
fut1.get();
fut2.get();
fut3.get();
```

El tiempo total de ejecución del programa debería ser 5 segundos y no 15 (la suma de los tres *Thread.sleep* en secuencial).

Modificaremos el Invoker para que pueda utilizar múltiples hilos para ejecutar más de una acción de manera concurrente. Utilizad ***ExecutorService*** de Java y otros mecanismos de sincronización (*Queues, Locks…*) para gestionar los hilos y Futures.

Nota: La gestión de la memoria es una sección crítica. Usad algún método de sincronización para acceder al valor de manera atómica.

Policy Manager

Queremos diseñar un sistema extensible de políticas de gestión de recursos (Policy Manager) que decida cómo se asignan las funciones a los diferentes Invokers.

Por ejemplo, queremos tener una política ***RoundRobin***, que simplemente distribuya uniformemente las funciones entre los Invokers que tengan recursos disponibles. También queremos tener una política ***GreedyGroup*** que intente siempre que pueda rellenar un Invoker todo lo posible antes de mover carga a otro. Si nos hacen una invocación grupal de 12 funciones, la política sería ejecutar el máximo de ellas en cada Invoker, por ejemplo: 8 en un Invoker y 4 en otro. Queremos incluir también una política ***UniformGroup*** que defina un tamaño de grupo (6 por ejemplo) y que coloque de manera uniforme entre todos los Invokers. Para un grupo de 18 funciones, distribuimos la carga en 6 funciones a cada invoker. Por último queremos una política ***BigGroup*** que defina un tamaño de grupo (6 por ejemplo) pero que intente colocar los grupos empaquetados si es posible. Por ejemplo, si ejecuto un grupo de 18 podría ejecutar 2 grupos de 6 en un Ivoker y otro de 6 en otro. O los tres en el mismo si cupieran.

Comparad con diferentes cargas de trabajo para procesar ficheros de texto la eficiencia de los diferentes gestores de recursos. Calculad si se está haciendo un uso eficiente de los recursos disponibles en una política respecto a otra.

Usad el patrón **strategy** para implementar el *policy manager*.

Observer

Queremos agrupar en el Controller diferentes métricas del sistema. Por ejemplo, de cada invocación guardaremos el tiempo de ejecución, el Invoker asignado, la memoria utilizada, etc. Para ello, los diferentes Invokers deberán notificar al Controller de las diferentes métricas que se realicen. El Controller guardará las métricas recolectadas.

Usad el patrón **observer** para implementar la notificación de las métricas.

Posteriormente, queremos consultar estas métricas para calcular diferentes datos. Por ejemplo, queremos saber el tiempo máximo, mínimo, media de invocación por cada acción, el tiempo agregado de ejecución de cada acción y la utilización de memoria de cada Invoker.

Usad Java Collections y Streams para implementar las diferentes consultas sobre las métricas.

Decorator

Queremos añadir lógica extra a las acciones definidas por el usuario de manera transparente. Primero, queremos implementar un “cronómetro” que monitoree el tiempo de ejecución de la función de usuario e imprima por pantalla el tiempo transcurrido. Segundo, queremos aplicar *memoization* para guardar en una caché el resultado de una acción. El sistema accede a una caché guardada en el Invoker, y guardará el resultado de una acción, junto con su identificador y parámetros de entrada. En invocaciones futuras donde coincidan los parámetros de entrada, devolveremos el resultado de la acción directamente recuperándose de la caché, sin ejecutar la acción.

Usad el patrón **decorator** para implementar la lógica del cronómetro y memoization. Comprobad el funcionamiento usando los decorator por separado y encadenados. Usad los decoradores con una acción que calcule el factorial de un número pasado por parámetro.

Reflection

Hasta ahora el usuario tiene que interactuar con el Controller para invocar una acción mediante su identificador. Queremos poder crear e invocar acciones siguiendo un patrón orientado a objetos. Para ello, implementaremos un *ActionProxy* que nos permita invocar acciones llamando a métodos directamente.

Usad **reflection** de Java y **DynamicProxy** para implementar el *ActionProxy*.

El *DynamicProxy* permite crear dinámicamente implementaciones de interfaz en tiempo de ejecución. En este caso, tal como se especifica anteriormente, nos permite interactuar con el Controller sin especificar el identificador correspondiente. El *ActionProxy* por tanto, será el responsable de corresponder cada invocación al Controller.

Concretando en el concepto anterior, podemos decir que en tiempo de ejecución, creamos una implementación proxy de esta interfaz. En cada invocación de un método de dicha

interfaz, la llamada se redirige a un *InvocationHandler* que hemos definido con anterioridad. Cabe remarcar, que la lógica de redirección de llamadas al *Controller* se encuentra en este mismo *InvocationHandler*.

Composite (opcional)

El Controller puede sobrecargarse si debe gestionar muchos Invokers. Para simplificar el proceso, cada Invoker podrá gestionar internamente diferentes niveles Invokers. Un Invoker podrá pasar una petición de invocación a los Invokers de niveles más bajos. Implementad el sistema para soportar varias jerarquías (3 niveles) de Invokers que operen con normalidad y de manera transparente al Controller.

Usad el patrón **composite** para implementar la jerarquía de Invokers.

Java RMI (opcional)

Queremos implementar el sistema distribuido, para que cada componente (*Controller*, múltiples Invokers) pueda ser ejecutado en diferentes servidores. Para ello, usaremos J*ava RMI* para comunicar los diferentes componentes. Para comprobar que funcione, podemos desplegar los servidores en una misma máquina en puertos diferentes.

Concretamente, el objetivo final es crear un sistema distribuido con la capacidad de que, por ejemplo, el *Controller* pueda invocar una acción en un Invoker que se ubica en otro servidor remoto, todo ello de manera transparente y sin necesidad de intervención manual.

Entrando en detalle, *Java RMI* logra dichos objetivos serializando los datos para poder enviarlos mediante la red, recibirlos desde el servidor, y deserializarlos para ser procesados correctamente. Una vez finalizado el procedimiento, el resultado se vuelve a serializar para ser enviados al cliente.

Javadocs y Junit

Utilizad Javadocs para generar documentación sobre el sistema. Utilizad Junit para unit testing de los componentes del sistema.

Aplicación Map Reduce

Para verificar el funcionamiento del sistema, queremos implementar las aplicaciones WordCount y CountWords siguiendo el modelo de programación MapReduce. WordCount cuenta el número de ocurrencias de cada palabra en un texto, mientras que CountWords cuenta el número de palabras totales en un texto. Ambas aplicaciones aceptan un string con un texto y devuelven un diccionario donde la clave será cada palabra y el valor un entero con el número de apariciones de esa palabra en el texto en el caso de WordCount, o el total de palabras en CountWords.

Usaremos las invocaciones grupales para procesar en paralelo múltiples ficheros (10) con una invocación grupal sobre las funciones WordCount o CountWords. En nuestro caso, la

invocación grupal será la fase Map, y ejecutaremos después una única función Reduce que reciba los resultados de todas las funciones y agregue el resultado.

Para los ficheros de texto, podéis usar los libros disponibles en Gutenberg Project.

Evaluación

- La evaluación es individual o en grupos de dos máximo. Se realizará una entrevista grupal para evaluar la práctica. La nota es individual.
- Los detalles de implementación de la práctica són libres siempre que estén justificados.
- Es **obligatorio** implementar todos los apartados para optar al aprobado, excepto Composite y Java RMI, que sirven para subir la nota o optar a un excelente.
- Para probar el funcionamiento de los diferentes apartados, utilizad múltiples clases con métodos main para poder elegir cual ejecutar sin comentar/descomentar código.