Documentación sistema FaaS

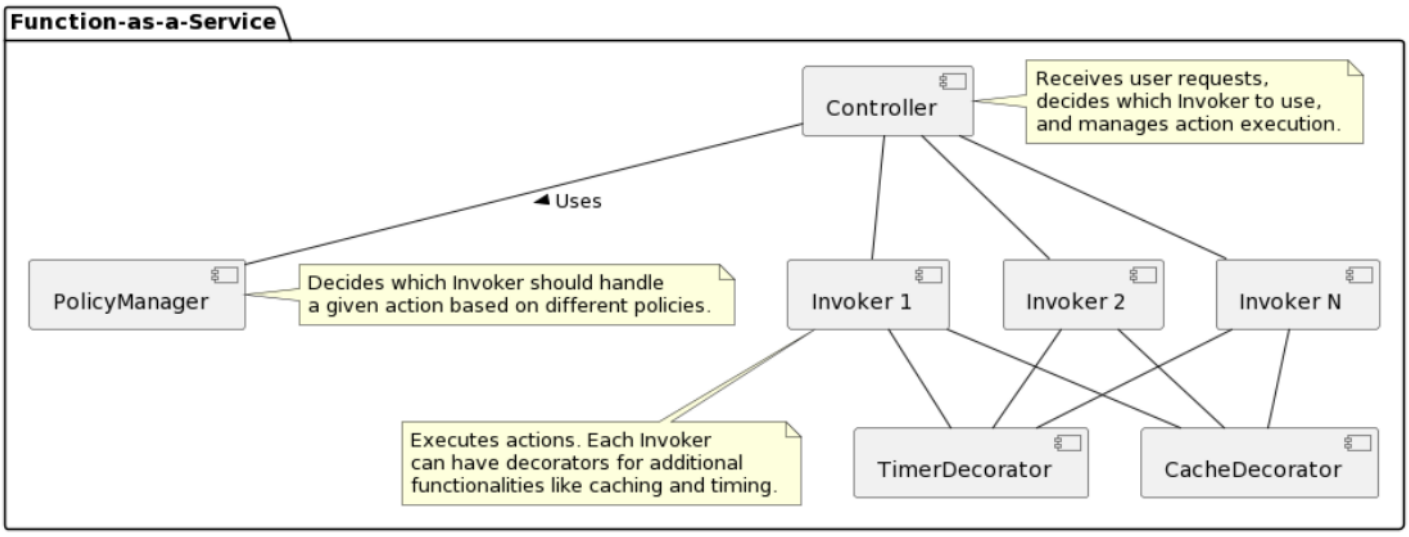
Function-as-a-Service (FaaS) es un paradigma Cloud que permite la ejecución de código sencillo, llamadas funciones o acciones, en la nube en servidores virtualizados. En un sistema FaaS, los usuarios pueden registrar, listar, invocar y eliminar acciones. Para registrar una nueva acción, el usuario debe 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.

Descripción del proyecto

Modelar con clases un sistema FaaS completo que permita la ejecución de funciones de manera adaptativa a los recursos disponibles. El proyecto se basa 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.



Indice de contenido

API Contratos

Class Request

Se ha diseñado un sistema de comunicación más orientado a graphql que a un sistema api rest. El motivo es que siempre van a utilizar un mismo punto de entrada que recibe una clase request.

La clase Request contiene:

El tipo de la acción que se pretende ejecutar. Equivalente a una query de graphql o equivalente a un endpoint de un servicio rest. Adicionalmente se le indica si esta debe ser ejecutada de manera asincrónica (isAsync) y la opción "isGroupalInvoke" que indica si la tarea es única o es un grupo de tareas a ejecutar.

Para las respuestas, disponemos de el atributo "response" tipado en función de la acción a ejecutar y "isDone" que indica si la tera se ejecutó correctamente. como indico este "contrato" es único ya que he diseñado el sistema más enfocado a Graphql

Ejemplo de una petición asíncrona, grupal, aplicar la función Word Count, sobre un texto:

```
Request request = new Request(wordCountMapAction, textos, true, true);
```

Diseño de sistema

Diseño de sistema

Class Controller

La clase `Controller` es la interfaz pública que permite a los usuarios interactuar con el sistema "Function as a Service". La clase `Controller` utiliza las clases `Invoker`, `InvokerAction`, `PolicyManager`, `Metrics` y `ActionProxyInvocationHandler` para la gestión de los invokers, las acciones, las políticas de ejecución y otras funcionalidades del sistema.

Métodos	Descripción
Controller	El constructor de la clase `Controller` recibe como parámetros el número de invokers a crear, la memoria asignada a cada invoker y un objeto `PolicyManager` que se encarga de gestionar las políticas de ejecución. En el constructor, se inicializan las listas de invokers y acciones, se registra una acción por defecto y se configuran los invokers.
registerAction	Este método permite registrar una nueva acción en el sistema. Recibe como parámetros el nombre de la acción, un objeto `Action` que representa la implementación de la acción y la cantidad de memoria necesaria para ejecutarla.
invokeAction	Este método se encarga de ejecutar una acción en el sistema. Recibe como parámetros el nombre de la acción y el input necesario para la ejecución. Dependiendo de la política de ejecución configurada en el `PolicyManager`, se selecciona uno o varios invokers para ejecutar la acción. El resultado de la ejecución se devuelve como resultado del método.
invokeAction_async	Este método es similar a invokeAction, pero permite ejecutar la acción de forma asíncrona, es decir, en un hilo separado. El resultado de la

Métodos	Descripción
	ejecución se devuelve como un objeto Future, que permite obtener el resultado cuando esté disponible.
invokeAction (sobrecarga)	Esta versión del método invokeAction permite ejecutar una lista de acciones en paralelo. Recibe como parámetros el nombre de la acción y una lista de inputs necesarios para la ejecución. Se seleccionan los invokers correspondientes para cada acción y se ejecutan en paralelo. El resultado de cada ejecución se devuelve como una lista de resultados.
invokeAction_async (sobrecarga)	Esta versión del método invokeAction_async permite ejecutar una lista de acciones en paralelo de forma asíncrona. Recibe como parámetros el nombre de la acción y una lista de inputs necesarios para la ejecución. Se seleccionan los invokers correspondientes para cada acción y se ejecutan en paralelo en hilos separados. El resultado de cada ejecución se devuelve como una lista de objetos Future, que permiten obtener los resultados cuando estén disponibles.
createActionProxy	Este método permite crear un proxy dinámico para una acción. Recibe como parámetros el nombre de la acción, indicadores de si la ejecución es asíncrona y si se trata de una invocación grupal. El proxy se encarga de ejecutar la acción y gestionar su ejecución.
update	Este método implementa la interfaz `Observer` y se ejecuta cuando un invoker notifica sobre la finalización de una tarea. Actualiza las métricas del sistema con la información proporcionada.
showMetrics	Este método muestra las métricas del sistema.
add	Este método implementa la interfaz Calculator y realiza una suma de dos números enteros.
Getters y Setters	Getters y setters para cada atributo.

Class Controller

La clase `Controller` es la interfaz pública que permite a los usuarios interactuar con el sistema "Function as a Service". La clase `Controller` utiliza las clases `Invoker`, `InvokerAction`, `PolicyManager`, `Metrics` y `ActionProxyInvocationHandler` para la gestión de los invokers, las acciones, las políticas de ejecución y otras funcionalidades del sistema.

Métodos	Descripción
Controller	El constructor de la clase `Controller` recibe como parámetros el número de invokers a crear, la memoria asignada a cada invoker y un

Métodos	Descripción
	objeto `PolicyManager` que se encarga de gestionar las políticas de ejecución. En el constructor, se inicializan las listas de invokers y acciones, se registra una acción por defecto y se configuran los invokers.
registerAction	Este método permite registrar una nueva acción en el sistema. Recibe como parámetros el nombre de la acción, un objeto `Action` que representa la implementación de la acción y la cantidad de memoria necesaria para ejecutarla.
invokeAction	Este método se encarga de ejecutar una acción en el sistema. Recibe como parámetros el nombre de la acción y el input necesario para la ejecución. Dependiendo de la política de ejecución configurada en el `PolicyManager`, se selecciona uno o varios invokers para ejecutar la acción. El resultado de la ejecución se devuelve como resultado del método.
invokeAction_async	Este método es similar a invokeAction, pero permite ejecutar la acción de forma asíncrona, es decir, en un hilo separado. El resultado de la ejecución se devuelve como un objeto Future, que permite obtener el resultado cuando esté disponible.
invokeAction (sobrecarga)	Esta versión del método invokeAction permite ejecutar una lista de acciones en paralelo. Recibe como parámetros el nombre de la acción y una lista de inputs necesarios para la ejecución. Se seleccionan los invokers correspondientes para cada acción y se ejecutan en paralelo. El resultado de cada ejecución se devuelve como una lista de resultados.
invokeAction_async (sobrecarga)	Esta versión del método invokeAction_async permite ejecutar una lista de acciones en paralelo de forma asíncrona. Recibe como parámetros el nombre de la acción y una lista de inputs necesarios para la ejecución. Se seleccionan los invokers correspondientes para cada acción y se ejecutan en paralelo en hilos separados. El resultado de cada ejecución se devuelve como una lista de objetos Future, que permiten obtener los resultados cuando estén disponibles.
createActionProxy	Este método permite crear un proxy dinámico para una acción. Recibe como parámetros el nombre de la acción, indicadores de si la ejecución es asíncrona y si se trata de una invocación grupal. El proxy se encarga de ejecutar la acción y gestionar su ejecución.
update	Este método implementa la interfaz `Observer` y se ejecuta cuando un invoker notifica sobre la finalización de una tarea. Actualiza las métricas del sistema con la información proporcionada.
showMetrics	Este método muestra las métricas del sistema.

Métodos	Descripción
add	Este método implementa la interfaz Calculator y realiza una suma de dos números enteros.
Getters y Setters	Getters y setters para cada atributo.

Interface Action<T,R>

La interfaz `Action` es una representación de una acción que consta de dos tipos: `T` (tipo del parámetro de entrada) y `R` (tipo del valor de retorno).

Métodos	Descripción
run	Este método representa la ejecución de la tarea en la acción o función seleccionada.
getName	Obtención del nombre función/acción

Interface PolicyManager

La interfaz "PolicyManager" se utiliza para gestionar políticas de ejecución de acciones.

Métodos	Descripción
invokeAction	Este método se utiliza para ejecutar una acción con la política seleccionada utilizando hilos. Toma como parámetros una lista de objetos `Invoker` que representan los invocadores de la acción, un objeto "T" que representa los parámetros necesarios según el tipo de acción a ejecutar, y un entero "memoryNeeded" que representa la memoria necesaria para la acción a ejecutar. Devuelve una lista de enteros, que representa a los invokers a invocar para la ejecució de la/s tarea/s.
invokeAction (sobrecarga)	Este método se utiliza para ejecutar múltiples acciones con la política seleccionada utilizando hilos. Toma como parámetros una lista de objetos `Invoker` que representan los invocadores de la acción, una lista de objetos "T" que representan los parámetros necesarios según el tipo de acción a ejecutar, y un entero "memoryNeeded" que representa la memoria necesaria para la acción a ejecutar. Devuelve una lista de enteros, que representa a los invokers a invocar para la ejecució de la/s tarea/s.

Interface Observer

La interfaz `Observer` se utiliza para representar objetos que desean ser notificados de cambios en un objeto observado.

Métodos	Descripción
taskName	Representa el nombre de la tarea, un long "executionTime" que representa el tiempo de ejecución y un entero "memoryNeeded" que representa la memoria necesaria.
update	Este método se utiliza para notificar cuando el estado del objeto observado cambia. Toma como parámetros un objeto `Observable` que representa el objeto observado

Interface Observable

La interfaz `Observable` se utiliza para representar objetos que pueden estar suscritos a otros objetos y recibir notificaciones de cambios.

Métodos	Descripción
subscribeObserver	Este método se utiliza para suscribir un objeto a otro objeto. Toma como parámetro un objeto `Observer` que representa el objeto a suscribir.
unsubscribeObserver	Este método se utiliza para desuscribir un objeto de otro objeto. Toma como parámetro un objeto `Observer` que representa el objeto a desuscribir.
notifyObserver	Este método se utiliza para notificar cambios a los objetos suscritos. Toma como parámetros un String "taskName" que representa el nombre de la tarea, un long "executionTime" que representa el tiempo de ejecución y un entero "memoryNeeded" que representa la memoria necesaria.
getId	Este método se utiliza para obtener el identificador del invoker.

Puntos de extensión y ciclo de vida

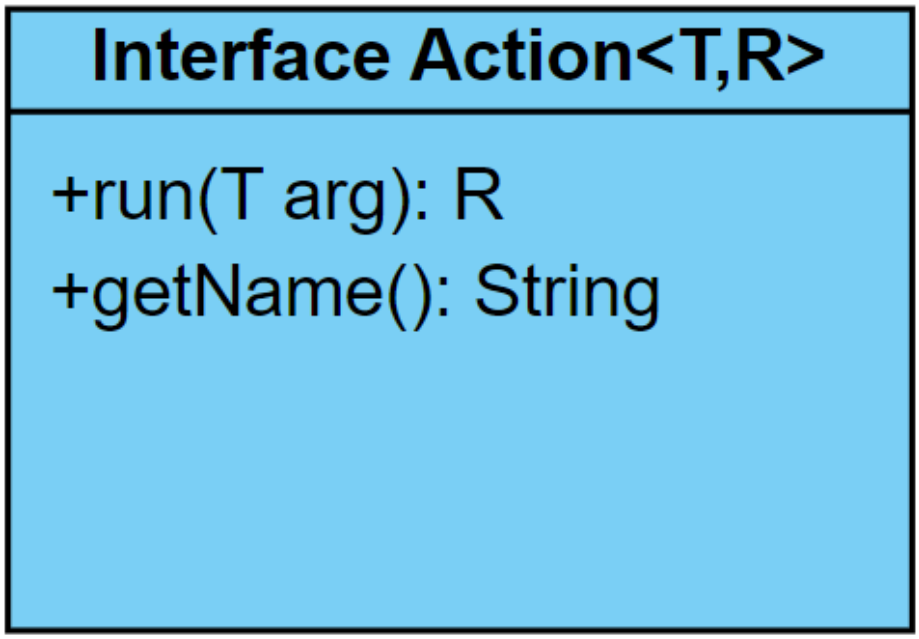
Implementación de nuevas funciones

Para implementar una nueva función en el sistema se debe implementar cumplir con la interfaz `Action` con sus contratos.

Interface Action<T,R>

La interfaz `Action` es una representación de una acción que consta de dos tipos: `T` (tipo del parámetro de entrada) y `R` (tipo del valor de retorno).

Métodos	Descripción
run	Este método representa la ejecución de la tarea en la acción o función seleccionada.
getName	Obtención del nombre función/acción



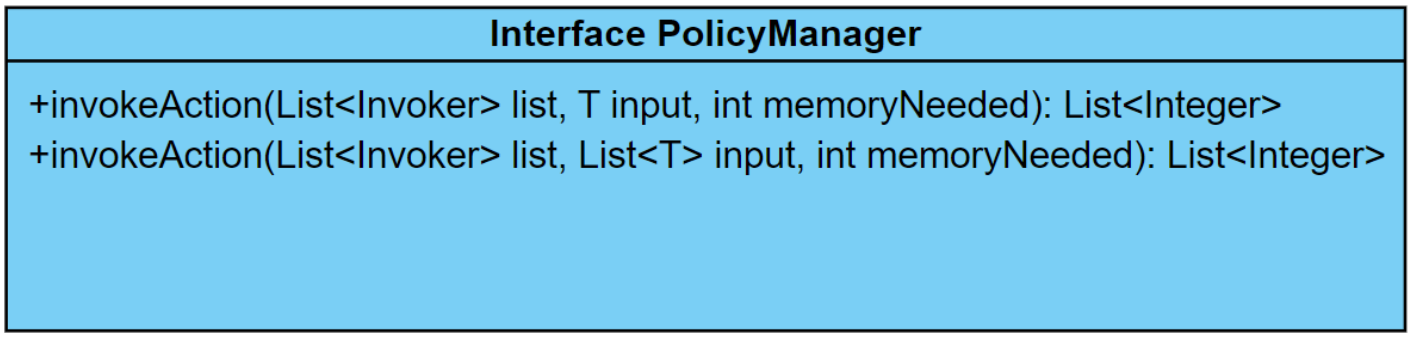
Implementación de nuevas políticas de gestión de recursos

Para implementar una nueva función en el sistema se debe cumplir con la interfaz `PolicyManager` con sus contratos.

Interface PolicyManager

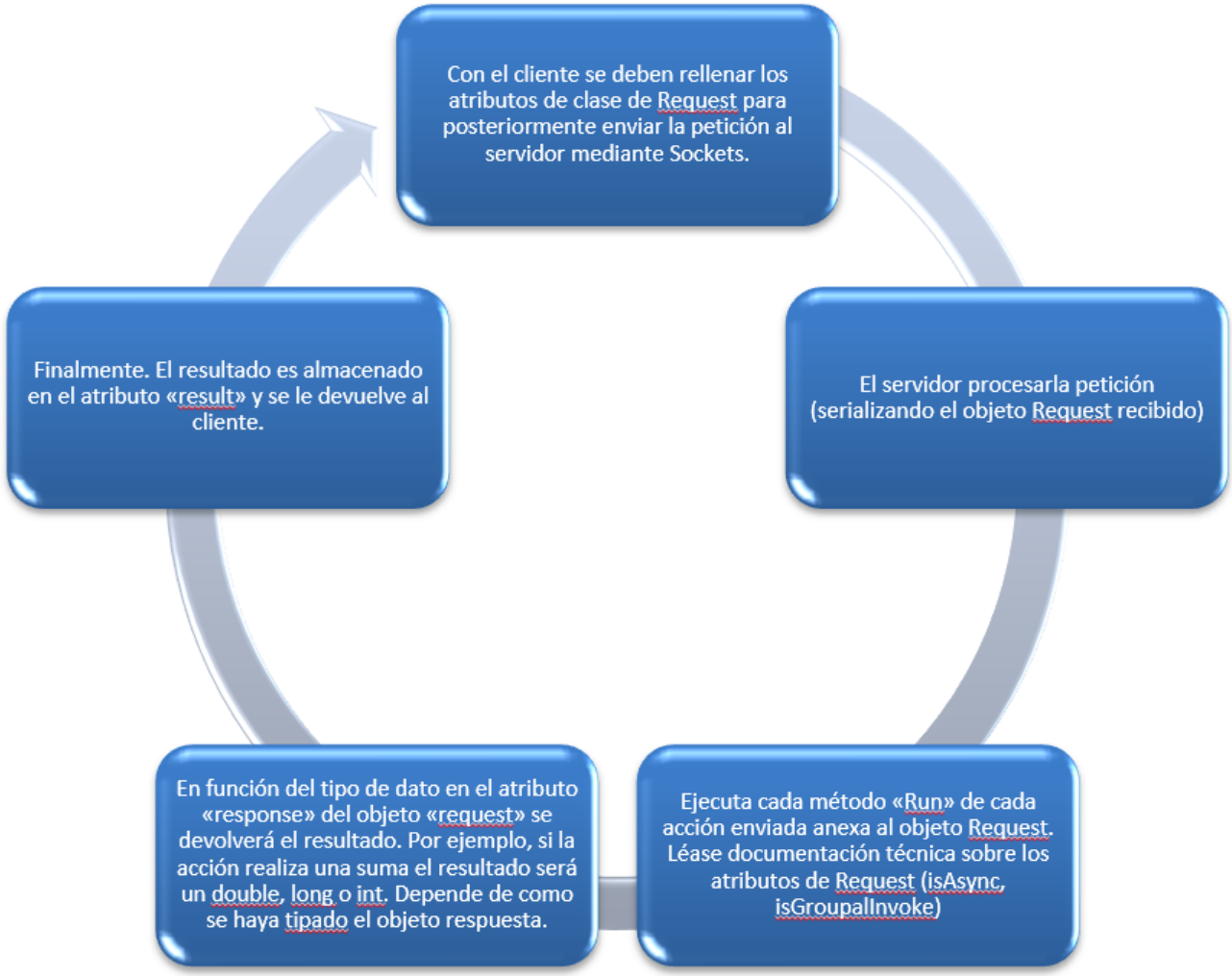
La interfaz "PolicyManager" se utiliza para gestionar políticas de ejecución de acciones.

Métodos	Descripción
invokeAction	Este método se utiliza para ejecutar una acción con la política seleccionada utilizando hilos. Toma como parámetros una lista de objetos `Invoker` que representan los invocadores de la acción, un objeto "T" que representa los parámetros necesarios según el tipo de acción a ejecutar, y un entero "memoryNeeded" que representa la memoria necesaria para la acción a ejecutar. Devuelve una lista de enteros, que representa a los invokers a invocar para la ejecució de la/s tarea/s.
invokeAction (sobrecarga)	Este método se utiliza para ejecutar múltiples acciones con la política seleccionada utilizando hilos. Toma como parámetros una lista de objetos `Invoker` que representan los invocadores de la acción, una lista de objetos "T" que representan los parámetros necesarios según el tipo de acción a ejecutar, y un entero "memoryNeeded" que representa la memoria necesaria para la acción a ejecutar. Devuelve una lista de enteros, que representa a los invokers a invocar para la ejecució de la/s tarea/s.



Ciclo de vida

El sistema està pensado para enviar y recibir peticiones de un servidor, cumpliendo los contratos establecidos en el `Request`, las peticiones son interceptadas en tiempo de ejecución, por el proxy dinamico, el cuál procesa los contratos e interviene entre la petición y la respuesta, que envía de vuelta el Controller ejecutando la acción, antes aplicando la política de asignación de recursos/tareas a los diferentes invocadores del sistema pre-establecido.



Validacion y testeo

Explicación de los tests unitarios que se han implementado y que prueban cada uno.

@Test testReflection

Probar la funcionalidad de reflexión del sistema Function as a Service (FaaS), verificando si la suma de 5 y 28 utilizando un objeto Calculator creado con ciertos parámetros y un algoritmo de balanceo de carga RoundRobin da como resultado 33.

```
@Test
void testReflection() throws Exception {
    Calculator calculator = new Controller(3,1024, new RoundRobin());
    int result = calculator.add(5, 28);
```

```
        assertEquals(33, result);
    }
```

@Test testAdd

Testeo de flujo de ejecución de una acción, mediante Proxy. Registramos una acción de suma en el controlador, se crea un proxy para la acción, se ejecuta la acción a través del proxy con ciertos valores de entrada, y se verifica si el resultado es el esperado.

```
    @Test
    void testAdd() throws Exception {
        Controller controller = new Controller(3,1024, new RoundRobin());
        Action addAction = new AddAction();
        controller.registerAction("addAction", addAction, 100);
        Action<Map<String,Integer>, Integer> proxy =
        controller.createActionProxy("addAction", false, false);
        int result = proxy.run(Map.of("x",5,"y",28));
        assertEquals(33, result);
    }
```

@Test testAddGroup

Testeo de flujo de ejecución de un grupo de acción, mediante Proxy. Registramos acciones grupales de acción de suma en el controlador, se crea un proxy para la acción grupal, se ejecuta las acciones a través del proxy con ciertos valores de entrada, y se verifica si el resultado es el esperado para cada acción del grupo.

```
    @Test
    void testAddGroup() throws Exception {
        Controller controller = new Controller(3,1024, new RoundRobin());
        Action addAction = new AddAction();
        controller.registerAction("addAction", addAction, 100);

        List<Map<String,Integer>> list = new ArrayList<>();
        list.add(Map.of("x",12,"y",25));
        list.add(Map.of("x",2,"y",5));
        list.add(Map.of("x",4,"y",265));
        list.add(Map.of("x",9,"y",218));


        Action<List<Map<String,Integer>>, List<Integer>> proxy = controller.createActionProxy(
        List<Integer> result = proxy.run(list);
        assertEquals(37, result.get(0));
        assertEquals(7, result.get(1));
        assertEquals(269, result.get(2));
```

```
        assertEquals(227, result.get(3));
    }
```

@Test testAddGroup_async

Testeo de flujo de ejecución de un grupo de acción, mediante Proxy. Registramos acciones grupales de acción de suma en el controlador, se crea un proxy para la acción grupal, se ejecuta las acciones a través del proxy con ciertos valores de entrada, y se verifica si el resultado es el esperado para cada acción del grupo, en multi-hilo.

```
    @Test
    void testAddGroup_async() throws Exception {
        Controller controller = new Controller(3,1024, new RoundRobin());
        Action addAction = new AddAction();
        controller.registerAction("addAction", addAction, 100);

        List<Map<String,Integer>> list = new ArrayList<>();
        list.add(Map.of("x",12,"y",25));
        list.add(Map.of("x",2,"y",5));
        list.add(Map.of("x",4,"y",265));
        list.add(Map.of("x",9,"y",218));


        Action<List<Map<String,Integer>>, List<Integer>> proxy = controller.createActionProxy(
        List<Integer> result = proxy.run(list);
        assertEquals(37, result.get(0));
        assertEquals(7, result.get(1));
        assertEquals(269, result.get(2));
        assertEquals(227, result.get(3));
    }
```

@Test testSleepGroup_async

Concurrencia y robustez, del sistema. Si la aplicacion no fuera asincrona, es decir no funcionase correctamente, el tiempo de ejecucion de este test seria de 38 segundos minimo, porque son los sleep que hay. Pues, el test se ejecuta en 20 segundos, por lo que concluimos en que se realiza correctamente, ya que el tiempo iterativo acumulado serian minimo 38. Si acaba en 20 es porque las tareas se hacen concurrentemente.

```
    @Test
    void testSleepGroup_async() throws Exception {
        Controller controller = new Controller(3,1024, new RoundRobin());
        Action sleepAction = new SleepAction();
        controller.registerAction("sleepAction", sleepAction, 100);
```

```
        List<Integer> list = new ArrayList<>();
        list.add(1000);
        list.add(2000);
        list.add(5000);
        list.add(10000);
        list.add(20000);

        Action<List<Integer>, List<String>> proxy = controller.createActionProxy("sleepAction"
        List<String> result = proxy.run(list);
        assertEquals("Done!", result.get(0));
        assertEquals("Done!", result.get(1));
        assertEquals("Done!", result.get(2));
        assertEquals("Done!", result.get(3));
        assertEquals("Done!", result.get(3));
    }
```

Instalacion y configuracion

****Maven:****

Utilizar el siguiente comando en caso de utilizar Maven.

```
mvn clean package
```

****MV Java:****

```
java -main tumain.class -cp <ruta_a_jar_de_faas>
```

****Importar .jar:****

> Si está compilando el código fuente. En caso de que se esté importando el .jar añadir al classpath de tu aplicación.
> Verá un .jar del sistema que puede descargarse en el repositorio, clic aquí para acceder.

Configuración

> El servidor utiliza el puerto `12345` , por favor si encuentras que no funciona, en caso de tener un ****SO Windows****, añadir una regla de entrada y salida para el puerto que recibe y envía

> peticiones el servidor `12345` , ver aquí como configurar reglas de entrada y salidas en Windows. En caso de utilizar un ****SO Linux****, añadir reglas de entrada y salida para el puerto del servidor en IPTABLES (consulta la documentación de tu distribución para los detalles)

Ejemplos

Ejemplos practicos de la funcionalidad del framework.

WordCount in MapReduce

TextToASCII en Scala

Add en Java