TOLERANCIA A FALLAS CON MICROPROFILE, QUARKUS Y DOCKER

Computación Tolerante a Fallas

Murillo Cortes, Jeanette.

Computación Tolerante a Fallas
Dr. Michel Emanuel López Franco

Sección D06
Calendario 2022A

Lunes 21 de Marzo, 2022
Guadalajara, Jalisco.

Tolerancia a Fallas con MicroProfile, Quarkus y Docker

¿Qué es MicroProfile?

MicroProfile es un proyecto que permite usar microservicios portables mediante de APIs, de manera que pueda brindar portabilidad a una aplicación en distintos entornos de ejecución.
MicroProfile Fault Tolerance permite construir soluciones que sean estratégicas para la Tolerancia a Fallas, de manera estandarizada, donde se reciben llamadas realizadas por un cliente, utilizando Retry, Fallback, Circuit Breaker y entre otros patrones de resiliencia.

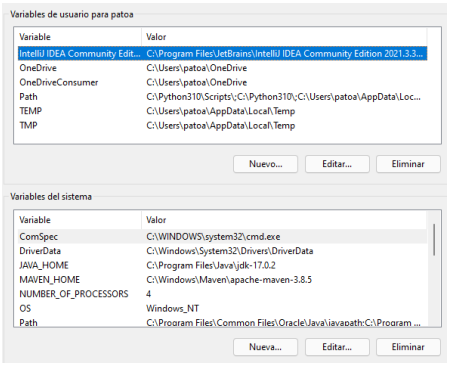
¿Qué es Quarkus?

Es un framework que permite compilar el código en un ejecutable Nativo o una Máquina Virtual de Java (JVM) para Linux o MAC, reduciendo el tiempo de arranque y consumo de memoria.
Utiliza las bibliotecas y los estándares principales de Java, como Eclipse MicroProfile y Spring.

¿Qué es Docker?

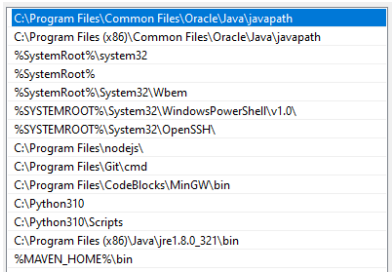
Es un proyecto de código abierto que permite que las aplicaciones se puedan desarrollar, ejecutar y desplegar mediante contenedores de software.
Para crear uno, se deben compilar una serie de comandos que crea una imagen local o remotamente, creando un contenedor.
Se puede utilizar Docker instalándolo en Windows, Mac, o creando un servidor de Linux.

PREINSTALACIONES



Para el desarrollo de esta práctica, se hizo uso de:

- IDLE IntelliJ que proporciona el uso de Maven y JDK. Obtenido del url: https://www.jetbrains.com/es-es/idea/download/#section=windows
- Maven: https://maven.apache.org/download.cgi
- JAVA: https://www.oracle.com/java/technologies/downloads/#jdk17-windows
- Docker: https://www.docker.com/products/docker-desktop/



Ambos agregados en las Variables del entorno del Sistema:

DESARROLLO DEL TUTORIAL

Para comenzar un nuevo proyecto en Quarkus, desde su página principal damos click en 'Start Coding' https://code.quarkus.io/
Configurando por el momento solo los detalles que se muestran en la imagen.

También, se seleccionan las extensiones que se van a incluir en el proyecto (estas se preparan para usar Gradal BM Native y se pueda compilar a código dependiente del sistema operativo). De esta manera, se seleccionó:

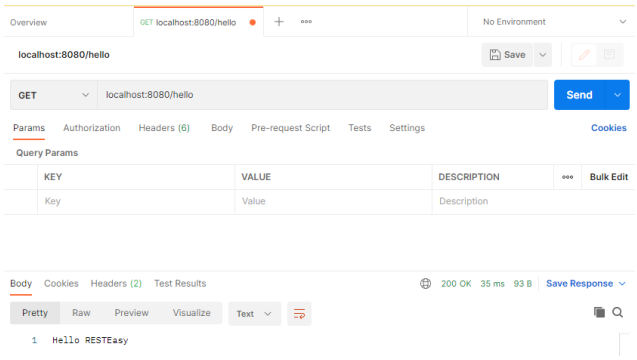
- RESTEasy JAX-RS
- RESTEasy JSON-B: Para la serialización entre Java y JSON
- SmallRye Fault Tolerance: bibliotecas como Fault Tolerance, que implementan los patrones de Eclipse de MicroProfile

Al terminar, se genera la aplicación dando *click* en el botón azul y descargando el proyecto como .zip. Se descomprime en la ubicación que queramos, en mi caso una carpeta utilizada para las tareas de la materia, y desde el cmd entramos a la carpeta descomprimida, abriendo IntelliJ con el comando 'idea'.
Una vez dentro del proyecto vamos a la ruta donde se encuentra el archivo GreetingResource:

`'demo-fault-tolerance>src>main>java>com>vorozco>GreetingResource.java'`

En el archivo entonces vamos a tener el recurso GreetingResource que está en REST, utilizando JAX-RX (de manera que se pueden hacer API's con Fault Tolerance):

1. Como Quarkus permite ejecutar compilación en modo de developer mode, ejecutamos '`mvn clean package`' y eso va a producir un fat-jar y va a tener el código de un API REST.
2. Para ejecutar el development mode se ejecuta con quarkus:dev; entonces en la consola se escribe '`mvn compile quarkus:dev`'.
3. Desde Postman se observa el resultado.



Para realizar el ejemplo de la API REST

1. En com.vorozco se agregan dos nuevos 'Package' llamados 'model' y 'controller' para organizar el código.
2. Metemos el archivo 'GreetingResource' en el Package 'controller' y en model agregamos una nueva clase de Java llamada 'Person'.
3. En person se agregan los atributos del Id, nombre y correo de la persona. Se genera el código de rutina con *click derecho* 'Generate>Getter and Setter' seleccionando todos los Getter's y Setter's con *Ctrl*.
4. Generamos un constructor normal con 'Generate>Constructor' y copeamos el constructor para generar un constructor vacio.

```
public Person() {
}
```

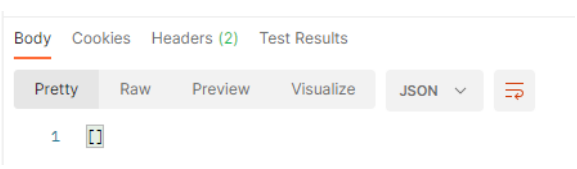
5. En el Paquete 'controller' hacemos una nueva clase de Java llamada 'PersonController'. En esta clase, importaremos java con @Path y dentro de la clase, generaremos un almacén de personas por medio de una lista, importando ArrayList y List, y terminando con el código de la siguiente manera:

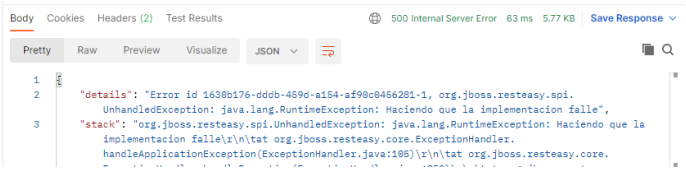
```
package com.vorozco.controller;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import java.util.ArrayList;
import java.util.List;

@Path("/persons")
@Produces(MediaType.APPLICATION_JSON)
public class PersonController {

    List<Person> personList = new ArrayList<>();

    @GET
    public List<Person> getPersonList(){
        return  this.personList;
    }
}
```

6. Continuamos, generando un Fallo aleatorio, por medio del segmento de código que utiliza las librerías Random y Logger, una nueva función llamada doFail() y que se ejecuta cuando se obtiene la lista de personas:

```
package com.vorozco.controller;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import java.util.ArrayList;
import java.util.List;
import java.util.Random; // NEW
import java.util.logging.Logger; // NEW

@Path("/persons")
@Produces(MediaType.APPLICATION_JSON)
public class PersonController {

    List<Person> personList = new ArrayList<>();
    Logger LOGGER = Logger.getLogger("Demologger");

    @GET
    public List<Person> getPersonList(){
        LOGGER.info("Ejecutando person list"); /* Cada vez que entre va a
decir el msg*/ // NEW
        doFail(); /*Cada que se produzca, va a informar una falla*/ // NEW
        return  this.personList;
    }

    public void doFail(){ // NEW ALL FUNCTION
        var random = new Random();
        if(random.nextBoolean()){
            LOGGER.warning("Se produce una falla");
            throw new RuntimeException("Haciendo que la implementacion
falle");
        }
    }
}
```

Dando como resultado la lista de personas o una falla:





Mostrando que se ejecuta la lista de personas y en caso del error, el warning:

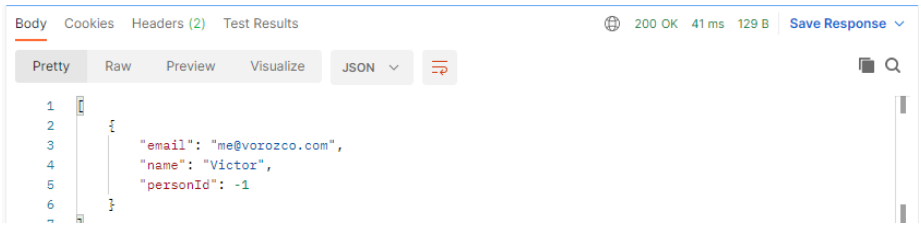
```
--
Tests paused
2022-03-21 21:32:17,514 INFO  [Demologger] (executor-thread-0) Ejecutando person list

--
```

```
--
Tests paused
2022-03-21 21:32:17,515 WARNING [Demologger] (executor-thread-0) Se produce una falla

--
```

FALLBACK Y TIMEOUT

7. Ahora, para ofrecer el camino alternativo de Fallback, se agrega MicroProfile Fault Tolerance, para que, si exista una falla, aun así se muestre una respuesta alternativa, o también, se pueda realizar una Ejecución con Fallo por tiempo con Timeout:

```
package com.vorozco.controller;

import com.vorozco.model.Person;
import org.eclipse.microprofile.faulttolerance.Fallback;
import org.eclipse.microprofile.faulttolerance.Timeout;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import java.util.logging.Logger;

@Path("/persons")
@Produces(MediaType.APPLICATION_JSON)
public class PersonController {

    List<Person> personList = new ArrayList<>();
    Logger LOGGER = Logger.getLogger("Demologger");

    @GET
    @Timeout(value = 5000L) /* Por tardar mucho tiempo en dar una
respuesta, en este caso, 5 segundos*/
    @Fallback(fallbackMethod = "getPersonFallbackList")
    public List<Person> getPersonList(){
        LOGGER.info("Ejecutando person list"); /* Cada vez que entre va a
decir el msg*/
        //doFail(); /* Cada que se produzca, va a informar una falla*/
        doWait();
        return  this.personList;
    }

    public List<Person> getPersonFallbackList(){
        var person = new Person(-1L, "Victor", "me@vorozco.com");
        return List.of(person);
    }

    public void doWait(){
        var random = new Random();
        try {
            LOGGER.warning("Haciendo un sleep");
            Thread.sleep((random.nextInt(10) + 4) * 1000L);
        }catch (Exception ex){

        }
    }

    public void doFail(){
        var random = new Random();
        if(random.nextBoolean()){
            LOGGER.warning("Se produce una falla");
            throw new RuntimeException("Haciendo que la implementacion
falle");
        }
    }
}
```



RETRY

8. También se puede hacer uso de Retry, para realizar la detección de fallas ejecutar determinado número de llamadas hasta que corresponda el Retry con los 4 intentos:

```
import org.eclipse.microprofile.faulttolerance.Retry;
@GET
//@Timeout(value = 5000L)
@Retry(maxRetries = 4) /* Puede intentar 4 veces antes de dar fallo
completo*/
@Fallback(fallbackMethod = "getPersonFallbackList")
public List<Person> getPersonList(){
    LOGGER.info("Ejecutando person list");
    doFail();
    //doWait();
    return  this.personList;
}
```

CIRCUIT BREAKER

9. Para la implementación del Circuit Breaker para la limitación de cantidad de fallas que están ocurriendo en el sistema, combinando reglas más completas:
   9.1. Para esto primero determinamos que 1 de cada 10 intentos fallen con 'failureRatio'
   9.2. Determinamos el delay para determinar cuando se va a tardar en cerrar el circuito

```
import org.eclipse.microprofile.faulttolerance.CircuitBreaker;
@GET
//@Timeout(value = 5000L)
//@Retry(maxRetries = 4)
@CircuitBreaker(failureRatio = 0.1, delay = 15000) /*Con delay de 15s para
cerrar cuando falle y continuar*/
@Fallback(fallbackMethod = "getPersonFallbackList")
public List<Person> getPersonList(){
    LOGGER.info("Ejecutando person list");
    doFail();
    //doWait();
    return  this.personList;
}
```

BULKHEAD

10. En Bulkhead(), se determinan cuántos clientes simultáneos pueden estar a la vez; en este caso, para que se provocara el fallo se determinó que con 0 clientes:

```
import org.eclipse.microprofile.faulttolerance.*;
@GET
//@Timeout(value = 5000L)
//@Retry(maxRetries = 4)
//@CircuitBreaker(failureRatio = 0.1, delay = 15000)
@Bulkhead(value = 0) /*Determinando cuántos clientes simultáneos pueden
estar a la vez*/
@Fallback(fallbackMethod = "getPersonFallbackList")
public List<Person> getPersonList(){
    LOGGER.info("Ejecutando person list");
    doFail();
    //doWait();
```

```
    return  this.personList;
}
```

Entonces, para comprobar la Tolerancia a Fallas usando Quarkus, se pudieron implementar las siguientes patrones o reglas en modo developer:

FALLBACK: Envía al usuario a un camino alternativo.

TIMEOUT: Si se cumple el tiempo fuera, se manda a un camino alternativo

RETRY: El sistema automáticamente reintenta la petición en un número de veces, si no va al camino alternativo

CIRCUIT BREAKER: Abre el circuito si cae en una condición de fallo de acuerdo con el failure Ratio, para cerrarse solo después de un delay

BULKHEAD: Cuántos clientes simultáneos va a permitir hasta rechazar las peticiones hacia un camino alternativo

11. Para empacar en Docker, primeramente, se pueden algunos visualizar archivos de Docker en la ruta 'src>main>docker', como lo son:

- Dockerfile.jvm: estándar con el cuál se descarga un Linux, se descargan paquetes y se instalan para dar soporte al fat-jar
- Dockerfile.fast-jar: copea como dependencias los libs, bibliotecas y algunas opciones para Quarkus
- Dockerfile.native: invoca la construcción del container con JVM

12. Del archivo Dockerfile.jvm, en la línea 10 se puede seleccionar el comando para construir el Docker

```
docker build -f src/main/docker/Dockerfile.jvm -t quarkus/demo-fault-tolerance-jvm .
```

13. Por último, corremos el Docker:

'`docker run -p 9090:8080  quarkus/demo-fault-tolerance .`'