LAB 18: QUARKUS TOLERANCE POLICIES

Autor: José Díaz

Github Repo: https://github.com/joedayz/quarkus-bcp-2025.git

Abre el proyecto **15-tolerance-policies-start**.

Este ejercicio requiere que tu agregues resiliencia a la aplicación monitor. Esta aplicación es un microservicio que brinda información de instancias cloud, como información del sistema o utilización CPU. La aplicación monitor provee datos invocando a otros microservicios, los cuales, son simulados por simplicidad.

1. Abre el proyecto con tu editor favorito.
2. Revisa los endpoints en **src/main/java/com/bcp/training/MonitorResource.java**. Estos endpoints llaman a otros servicios.

   /info
   Invoca InforService para obtener información del sistema, acerca de, la instancia cloud.

   /status
   Invoca StatusService para obtener el status de la instancia cloud.

   /cpu/stats
   Invoca CputStatsService para obtener datos de CPU de la instancia cloud.

   /cpu/predict
   Invoca CpuPredictionService para predecir el futuro de carga CPU de la instancia cloud.

3. Instala la extensión smallrye-fault-tolerance e inicia la aplicación.

   mvn quarkus:add-extension -Dextension=smallrye-fault-tolerance

```
[student@workstation tolerance-policies]$ mvn \
 quarkus:add-extension -Dextension=smallrye-fault-tolerance
...output omitted...
[INFO] [SUCCESS] ... Extension io.quarkus:quarkus-smallrye-fault-tolerance has
 been installed
...output omitted...
```

E inicia la aplicación en modo desarrollo:

mvn quarkus:dev

```
[student@workstation tolerance-policies]$ mvn quarkus:dev
...output omitted...
[io.quarkus] (...) started in 1.312s. Listening on: http://localhost:8080
...output omitted...
```

4. Usa la política de **reintentos** para hacer que la aplicación sea resiliente a fallas en **InfoService**.

   a. Abre una nueva terminal y haz un request al endpoint **/info**. El endpoint falla.

curl localhost:8080/info

```
[student@workstation ~]$ curl localhost:8080/info; echo
{"details":"Error id ...output omitted... }
```

   b. Inspecciona el archivo **/src/main/java/com/bcp/training/sysinfo/InfoService.java**. Solo una de las cinco invocaciones al método **getInfo** son exitosas.

   c. Agrega la anotación **@Retry** al método **getInfo**. Establece **maxRetries** a **5**.

```
@Retry( maxRetries = 5 )
public Info getInfo() {…}
```

   d. Re-ejecuta el request y verifica que este trabaja.

curl localhost:8080/info

```
[student@workstation ~]$ curl localhost:8080/info; echo
{"NAME":"Linux","ARCH":"amd64","VERSION":"4.18.0-372.32.1.el8_6.x86_64"}
```

   e. Inspecciona los logs de la aplicación y verifica que Quarkus reintento los requests varias veces.

```
ERROR [com.red.tra.ser.InfoService] (...) Request #1 has failed
ERROR [com.red.tra.ser.InfoService] (...) Request #2 has failed
ERROR [com.red.tra.ser.InfoService] (...) Request #3 has failed
ERROR [com.red.tra.ser.InfoService] (...) Request #4 has failed
INFO  [com.red.tra.ser.InfoService] (...) Request #5 has succeeded
```

5. Usa la política de **Timeout** para hacer que la aplicación sea resiliente a delays en **StatusService**.

   a. Hacer un request al endpoint /status. El requests toma cerca de 5 segundos para completar.

   curl localhost:8080/status

```
[student@workstation ~]$ curl localhost:8080/status; echo
Running
```

   b. Inspecciona los logs de la aplicación y verifica que los requets están tomando cerca de 5 segundos en completarse.

```
WARN  [com.red.tra.sta.StatusService] (...) Request #1 is taking too long...
INFO  [com.red.tra.sta.StatusService] (...) Request #1 completed in 5001
 milliseconds
```

   c. Inspecciona el archivo **src/main/java/com/bcp/training/status/StatusService**. Observemos dos aspectos:
      - El método getStatus experimenta delays en 4 de 5 invocaciones.
      - El método getStatus reintenta invocaciones que fallan debido al timeout.

   d. Agrega la anotación **@Timeout** a el método **getStatus**. Arroja un error timeout despues de 200 ms.

```
@Timeout(200)
@Retry( maxRetries = 5, retryOn =
TimeoutException.class )
public String getStatus() {
    …
}
```

   e. Reejecuta el request y verifica que la respuesta es rápida.

   curl http://localhost:8080/status

```
[student@workstation ~]$ curl localhost:8080/status; echo
Running
```

f. Revisa los logs de la aplicación y verifica que Quarkus ha interrumpido las invocaciones lentas y reintenta de nuevo ellas.

```
WARN  [com.red.tra.sta.StatusService] (...) Request #1 is taking too long...
WARN  [com.red.tra.sta.StatusService] (...) Request #1 has been interrupted after
 200 milliseconds
WARN  [com.red.tra.sta.StatusService] (...) Request #2 is taking too long...
WARN  [com.red.tra.sta.StatusService] (...) Request #2 has been interrupted after
 200 milliseconds
WARN  [com.red.tra.sta.StatusService] (...) Request #3 is taking too long...
WARN  [com.red.tra.sta.StatusService] (...) Request #3 has been interrupted after
 200 milliseconds
WARN  [com.red.tra.sta.StatusService] (...) Request #4 is taking too long...
WARN  [com.red.tra.sta.StatusService] (...) Request #4 has been interrupted after
 200 milliseconds
INFO  [com.red.tra.sta.StatusService] (...) Request #5 completed in 0 milliseconds
```

6. Usa una política Fallback para hacer la aplicación resiliente cuando hay data faltante en **CputStatsService**.

   a. Haz un request al endpoint **/cput/stats**. La respuesta contiene uso de CPU en formato time series. La respuesta también contiene la media y desviación estándar, calculados a partir de la data time series.

   curl -s localhost:8080/cpu/stats | jq

```
[student@workstation ~]$ curl -s localhost:8080/cpu/stats | jq
{
  "usageTimeSeries": [
    0.987903386804749,
    0.34275471439780536,
    0.020709840667124446,
    0.35121416539390315,
    0.9863511894860464,
    0.31318722701672885,
    0.7856415790758161,
    0.42147674186562323,
    0.284121753040781
  ],
  "mean": 0.4992622886387308,
  "standardDeviation": 0.319795785721884
}
```

   b. Repetir el request hasta que un error ocurra.

   curl -s localhost:8080/cpu/stats | jq

```
[student@workstation ~]$ curl -s localhost:8080/cpu/stats | jq
{
  "details": "Error id ..., org.jboss.resteasy.spi.UnhandledException:
 java.lang.NullPointerException",
  "stack": ...output omitted...
}
```

c. Inspecciona los logs de la aplicación. El error ocurre cuando el método **getCpuStats** llama a **calculateMean**.

```
WARN  [com.red.tra.cpu.CpuStatsService] (...) Cpu usage data in request #3
 contains null values
ERROR [io.qua.ver.htt.run.QuarkusErrorHandler] (...) HTTP Request to /cpu/
stats failed, error id: ...: org.jboss.resteasy.spi.UnhandledException:
 java.lang.NullPointerException
        ...output omitted...
        at
 com.redhat.training.cpu.CpuStatsService.calculateMean(CpuStatsService.java:55)
        at
 com.redhat.training.cpu.CpuStatsService.getCpuStats(CpuStatsService.java:21)
```

d. Inspecciona el archivo **src/main/java/com/bcp/training/cpu/CputStatsService.java.** Una de las tres invocaciones del método **getCpuStats** falla porque la data contiene valores **null**. Los valores null resultan debido a un error cuando el servicio calcula la media y la desviación estándar.

e. Agrega la anotación **@Fallback** al método **getCpuStats**. Establece el método fallback **getCpuStatsWithMissingValues**.

```
@Fallback( fallbackMethod =
"getCpuStatsWithMissingValues" )
public CpuStats getCpuStats() {
…
}
```

f. Implementa el método fallback. Establece la media y desviación estándar a 0.0.

```
public CpuStats getCpuStatsWithMissingValues() {
    return new CpuStats( series, 0.0, 0.0 );
}
```

g. Repite el request al endpoint **/cpu/stats** hasta que recibas una respuesta con valores null. El request usa el método fallback y establece las propiedades agregadas a 0.0.

curl -s localhost:8080/cpu/stats | jq

```
[student@workstation {gls_lab_script}]$ curl -s localhost:8080/cpu/stats | jq
{
  "usageTimeSeries": [
    0.0965490271728523,
    null,
    0.22910115311828105,
    null,
    0.5527746943344609,
    null,
    0.006881053771782275,
    0.22669714994239298,
    0.3583567119779293
  ],
  "mean": 0.0,
  "standardDeviation": 0.0
}
```

7. Usa el patrón Circuit Breaker para detener el trafico que se envía a **CpuPredictionService** cuando este servicio no esta disponible.

   a. Haz un request al endpoint /cpu/predict. La respuesta es la carga prevista de CPU.

   curl localhost:8080/cpu/predict

```
[student@workstation ~]$ curl localhost:8080/cpu/predict; echo
0.9822281195867076
```

   b. Ejecuta el script **predict_many.sh** (linux o macosx) o **predict_many.ps1** (windows). Este script invoca el endpoint **/cpu/predict** cada segundo. El request comienza fallando.

   ./predict_many.sh o **./predict_many.ps1**

```
[student@workstation ~]$ ~/D0378/tolerance-policies/predict_many.sh
0.4997873140920043
{"details":"Error id 63a4f993-db90-49fe-8c24-24f33...}
{"details":"Error id 63a4f993-db90-49fe-8c24-24f33...}
{"details":"Error id 63a4f993-db90-49fe-8c24-24f33...}
{"details":"Error id 63a4f993-db90-49fe-8c24-24f33...}
```

   c. Presiona **Ctrl+C** para detener el script.
   d. Revisa el archivo **src/main/java/com/bcp/training/cpu/CpuPredictionService.java**. El servicio puede solo manejar un request cada dos segundos. Caso contrario, el servicio arrojará un error.
   e. Agrega la anotación **@CircuitBreaker** al método **predictSystemLoad**. Establece la propiedad **requestVolumeThreshold** a 6, de esta manera, el mecanismo abre el circuito si 3 de los 6 requets fallan. Establece la propiedad **delay** a 3000, así que el circuito permanece abierto por 3 segundos.

```
@CircuitBreaker( requestVolumeThreshold = 6, delay
= 3000 )
public Double predictSystemLoad() {
    callCount++;
```

```
    crashPossibly();

    return Math.random();
}
```

f. Ejecuta el script **predict_many.sh** o **predict_many.ps1**. Después de 6 requests, mucho de ellos fallan, el circuit breaker abre el circuito. En este punto la aplilcación retorna la respuesta: **Prediction service is not available at the moment**. El circuito permanecerá abierto por 3 segundos, y luego el prediction service retornará una respuesta valida nuevamente.

./predict_many.sh o **./predict_many.ps1**

```
[student@workstation ~]$ ~/DO378/tolerance-policies/predict_many.sh
0.9050798759755502
{"details":"Error id 63a4f993-db90-49fe-8c24-24f33...}
{"details":"Error id 63a4f993-db90-49fe-8c24-24f33...}
{"details":"Error id 63a4f993-db90-49fe-8c24-24f33...}
{"details":"Error id 63a4f993-db90-49fe-8c24-24f33...}
{"details":"Error id 63a4f993-db90-49fe-8c24-24f33...}
Prediction service is not available at the moment
Prediction service is not available at the moment
0.008288100728291115
{"details":"Error id 63a4f993-db90-49fe-8c24-24f33...}
{"details":"Error id 63a4f993-db90-49fe-8c24-24f33...}
{"details":"Error id 63a4f993-db90-49fe-8c24-24f33...}
{"details":"Error id 63a4f993-db90-49fe-8c24-24f33...}
{"details":"Error id 63a4f993-db90-49fe-8c24-24f33...}
Prediction service is not available at the moment
...output omitted...
```

g. Presiona Ctrl+C para detener el script.

8. Retorna a la terminal donde la aplicación es corriendo en modo desarrollo y presiona **q** para detener la aplicación.

Esto concluye el laboratorio.

Enjoy!

José