LAB 18: QUARKUS TOLERANCE POLICIES

Autor: José Díaz

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

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

Instructions

This exercise requires you to add resiliency to the `monitor` application. This application is a microservice that gathers monitoring information from cloud instances, such as system information or CPU utilization. The `monitor` application provides data by invoking other microservices, which are simulated, for the sake of simplicity.

1. Open the application project and review the endpoints.

    1.1. Navigate to the `~/D0378/tolerance-policies` directory.

```
[student@workstation ~]$ cd ~/D0378/tolerance-policies
```

    1.2. Open the project with your editor of choice, such as VSCodium or vim.

    1.3. Inspect the endpoints in the `src/main/java/com/redhat/training/MonitorResource.java` file. These endpoints call other services.

`/info`
: Invokes `InfoService` to get system information about the cloud instance.

`/status`
: Invokes `StatusService` to get the status of the cloud instance.

`/cpu/stats`
: Invokes `CpuStatsService` to get CPU data from the cloud instance.

`/cpu/predict`
: Invokes `CpuPredictionService` to predict the future CPU load of the cloud instance.

2. Install the `smallrye-fault-tolerance` extension and start the application.

2.1. Install the `smallrye-fault-tolerance` extension.

```
[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...
```

2.2. Start the application in development mode.

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

3. Use the Retry policy to make the application resilient to failures in `InfoService`.

3.1. Open a new terminal window and make a request to the `/info` endpoint. The request fails.

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

3.2. Inspect the `src/main/java/com/redhat/training/sysinfo/InfoService.java` file. Only one out of five invocations to the `getInfo` method succeed.

3.3. Add the `@Retry` annotation to the `getInfo` method. Set `maxRetries` to `5`.

```
@Retry( maxRetries = 5 )
public Info getInfo() {
    ...implementation omitted...
}
```

3.4. Rerun the request and verify that it works.

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

3.5. Inspect the application logs and verify that Quarkus retried the request several times.

```
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
```

4. Use the Timeout policy to make the application resilient to delays in `StatusService`.

4.1. Make a request to the `/status` endpoint. The request takes about five seconds to complete.

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

4.2. Inspect the application logs and verify that the request is taking about five seconds to complete.

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

4.3. Inspect the `src/main/java/com/redhat/training/status/StatusService.java` file. Note two aspects:

- The `getStatus` method experiences delays in four out of five invocations.
- The `getStatus` method retries invocations that fail due to a timeout.

4.4. Add the `@Timeout` annotation to the `getStatus` method. Throw a timeout error after 200 milliseconds.

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

4.5. Rerun the request and verify that the response is faster.

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

4.6. Review the application logs and verify that Quarkus has interrupted the slow invocations and retried them.

```
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
```

5. Use the Fallback policy to make the application resilient to missing data in `CpuStatsService`.

5.1. Make a request to the `/cpu/stats` endpoint. The response contains CPU usage time series data. The response also contains the mean and the standard deviation, calculated from the time series data.

```
[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
}
```

5.2. Repeat the request until an error occurs.

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

5.3. Inspect the application logs. The error occurs when the `getCpuStats` method calls `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)
```

5.4. Inspect the `src/main/java/com/redhat/training/cpu/CpuStatsService.java` file. One out of three invocations of the `getCpuStats` method fail because the data contains null values. Null values result in an error when the service calculates the mean and the standard deviation.

5.5. Add the `@Fallback` annotation to the `getCpuStats` method. Set `getCpuStatsWithMissingValues` as the fallback method.

```
@Fallback( fallbackMethod = "getCpuStatsWithMissingValues" )
public CpuStats getCpuStats() {
    ...implementation omitted...
}
```

5.6. Implement the fallback method. Set the mean and standard deviation values to `0.0`.

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

5.7. Repeat the request to the `/cpu/stats` endpoint until you receive a response with null values. The request uses the fallback method and sets the aggregate properties to 0.0.

```
[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
}
```

6. Use the Circuit Breaker pattern to stop sending traffic to the `CpuPredictionService` when this service becomes unstable.

6.1. Make a request to the `/cpu/predict` endpoint. The response is the predicted CPU load.

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

6.2. Run the `predict_many.sh` script. This script invokes the `/cpu/predict` endpoint every second. The requests start failing.

```
[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...}
```

6.3. Press `Ctrl+C` to stop the script.

6.4. Inspect the `src/main/java/com/redhat/training/cpu/CpuPredictionService.java` file. The service can only handle one request every two seconds. Otherwise, the service throws an error.

6.5. Add the `@CircuitBreaker` annotation to the `predictSystemLoad` method. Set the `requestVolumeThreshold` property to `6`, so that the mechanism opens the circuit if three out of six requests fail. Set the `delay` property to `3000`, so that the circuit remains open for three seconds.

```
@CircuitBreaker( requestVolumeThreshold = 6, delay = 3000 )
public Double predictSystemLoad() {
  ...implementation omitted...
}
```

6.6. Rerun the `predict_many.sh` script. After six requests, most of them failing, the circuit breaker opens the circuit. At this point, the application returns the response `Prediction service is not available at the moment`. The circuit remains open for three seconds, and then the prediction service returns a valid response again.

```
[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...
```

6.7. Press `Ctrl+C` to stop the script.

7. Return to the terminal where the application is running in development mode and press `q` to stop the application.

Finish

On the `workstation` machine, use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish tolerance-policies
```

This concludes the section.

enjoy!

Jose