Introduction to microservices observability with Eclipse MicroProfile

Microservices provide a modern approach to development, which is compliant with the cloud environment and gives us the ability to create cloud-native applications. With microservices, we promote resilience, fault tolerance, and scale; however, a microservice approach also presents different challenges than monolithic applications because of its distributed nature.

One of these challenges involves monitoring and logging, which naturally brings us to the concept of observability. In this article, we'll look at how Eclipse MicroProfile can help you implement observability in microservices.

What is observability?

The concept of observability comes from the control theory, that is, a math theory.

> Formally, a system is said to be **observable** if, for any possible sequence of state and control vectors (the latter being variables whose values one can choose), the current state (the values of the underlying dynamically evolving variables) can be determined in finite time using only the outputs. — Wikipedia

Some developers define observability in a microservice architecture as the set of metrics, logging, and tracing tools, but I think observability is a more general concept. Metrics, logging, and tracing are simply ways to provide observability.

To me, observability is the capacity of a system to expose precise information about its states in a quick and easy way. Unlike monitoring, observability is about the system. When we talk about monitoring, the focus is on the tools used to monitor the system, which can be easy or hard to monitor. When we talk about observability, the focus is on the system itself and the need to provide this information in an easier and faster way.

MicroProfile OpenTracing

The MicroProfile OpenTracing specification permits us to use distributed tracing using the OpenTracing API to trace the flow of a request across a service. This specification is compatible with Zipkin and Jaeger, and it permits us to use those tools to show information about the distributed tracing. Below is an example of how to use MicroProfile OpenTracing.

```
@Path("subjects")
@Traced
public  SubjectEndpoint {
```

MicroProfile Metrics

MicroProfile Metrics is a specification that permits us to expose metrics information about our applications. With this, we can expose precise metrics to be consumed more quickly and easily. Below is an example of how to use MicroProfile Metrics.

```
@Counted
public CounterBean() {
}
```

MicroProfile HealthCheck

The MicroProfile HealthCheck spec permits us to expose if the application is up or down in our environment. It works as a Boolean response (yes or no) to the question, “Is my application still running ok?“. Below is an example of how to use MicroProfile HealthCheck.

```
@Health
@ApplicationScoped
public class ApplicationHealthCheck implements HealthCheck {


  @Override
  public HealthCheckResponse call() {
    return HealthCheckResponse
        .named("application-check").up()
        .withData("CPUAvailable", Runtime.getRuntime().availableProcessors())
        .withData( "MemoryFree", Runtime.getRuntime().freeMemory())
        .withData("TotalMemory", Runtime.getRuntime().totalMemory())
        .build();
  }
}
```

Microservices Fault Tolerance Mechanism

Following are a few design patterns under the microservice friendly Microprofile spec, which are used for fault tolerance:

- **Bulkhead – isolate failures in part of the system.**
- **Circuit breaker – offer a way to fail fast.**
- **Retry – define criteria on when to retry.**
- **Fallback – provide an alternative solution for a failed execution.**

Timeout

If you annotate a method with this @Timeout, the execution will be aborted after the defined threshold. For this example that threshold is 500ms.

```
import org.eclipse.microprofile.faulttolerance.Timeout;


@Timeout(500)
public String callSomeService() {...}
```

Retry

This annotation allows you to define a strategy in case of failure, as an example, an IOException.

```
import org.eclipse.microprofile.faulttolerance.Retry;


@Retry(retryOn = IOException.class)
public String callSomeService() {...}
```

You can also stack multiple annotations, in here, the method will be retried once if the execution times out after 500ms.

```
import org.eclipse.microprofile.faulttolerance.Timeout;
import org.eclipse.microprofile.faulttolerance.Retry;


@Timeout(500)
@Retry(maxRetries = 1)
public String callSomeService() {...}
```

You can define very fine-grained behaviors. On the next example, if an IOException occurs, there will be 2 retries with a delay of 200ms minus or plus a random value between -100ms and +100ms, because of the jitter attribute. This helps to reduce peak loads.

```
@Retry(delay = 200, maxRetries = 2, jitter = 100, retryOn = IOException.class)
public String callSomeService() {...}
```

Fallback

The @Fallback annotation provides you an alternative execution path in case of failure, thus increasing the success rate of the requests.

Here is how you can ask for a different method in the same class to be executed in case of any Exception. The fallback method can be non-public:

```
import org.eclipse.microprofile.faulttolerance.Fallback;


@Fallback(fallbackMethod = "fallbackMethodInSameClass")
public String callSomeService() {...}
...
public String fallbackMethodInSameClass() {...}
```

In the next example, we combine different annotation. After 500ms of execution, the callSomeService() method will be re-called once, and if it times out again, the fallback method will be executed instead.

The CallAppologyService class implements the FallbackHandler interface

```
import org.eclipse.microprofile.faulttolerance.Timeout;
import org.eclipse.microprofile.faulttolerance.Retry;
import org.eclipse.microprofile.faulttolerance.Fallback;

@Timeout(500)
@Retry(maxRetries = 1)
@Fallback(CallAppologyService.class)
public String callSomeService() {...}
```

Circuit Breaker

The failure of a service results in higher latency for clients due to timeouts. This can trigger a cascading effect and propagate the failure to other services. If we know a service has problems, a circuit breaker can be used to force calls to fail immediately and stop subsequent invocations of that service.

In this example, the circuit breaker is closed or working normally, and we use a rolling window of the last 4 requests. If 75% (3 requests out of 4) fail, then the circuit will stay open, rejecting all subsequent requests for 1000ms.

After the 1000ms delay, the circuit is placed to half-open. At this point, trial calls will probe the destination and after 10 consecutive successes, the circuit will be placed back to closed (normal operation).

```
import org.eclipse.microprofile.faulttolerance.CircuitBreaker;

@CircuitBreaker(requestVolumeThreshold = 4, failureRatio=0.75, delay = 1000, successThreshold = 10)
public String callSomeService() {...}
```

You can combine the circuit breaker with other patterns, like the retry or the timeout. This way you can control the failures that lead to an open circuit.

```
import org.eclipse.microprofile.faulttolerance.CircuitBreaker;
import org.eclipse.microprofile.faulttolerance.Retry;
import org.eclipse.microprofile.faulttolerance.Timeout;

@CircuitBreaker(requestVolumeThreshold = 4, failureRatio = 0.75, delay = 1000, successThreshold = 10, )
@Retry(retryOn = {RuntimeException.class, TimeoutException.class}, maxRetries = 7)
@Timeout(500)
public String callSomeService() {...}
```

A `@Fallback` can be specified and it will be invoked if the `CircuitBreakerOpenException` is thrown.

Bulkhead

The @Bulkhead annotation also prevents the failure of a service from triggering a cascading effect to other services. In this case, it’s only effective if you are calling the method from multiple contexts.
The bulkhead works by limiting the number of concurrent requests to the method. This can be achieved in 2 ways:

Semaphore style

There a hard limit to the number of concurrent requests. In this example, after 5 parallel requests, the extra calls will receive a BulkheadException.

```
import org.eclipse.microprofile.faulttolerance.Asynchronous;
import org.eclipse.microprofile.faulttolerance.Bulkhead;


@Bulkhead(5)
public String callSomeService() {...}
```

Thread pool style

This will use a thread pool to hold the extra requests, up to a limit. In the example, 5 concurrent requests are allowed and 8 can be placed on a waiting queue. If the waiting queue is exhausted, the extra calls will receive a BulkheadException.

```
import org.eclipse.microprofile.faulttolerance.Bulkhead;


@Asynchronous
@Bulkhead(value = 5, waitingTaskQueue = 8)
public Future callSomeService() {...}
```

Asynchronous

The @Asynchronous annotation means that the method execution will happen in a separate thread. Any methods marked with this annotation must return one of:
java.util.concurrent.Future
java.util.concurrent.CompletionStage
There is an ongoing discussion around the @Asynchronous annotation and its implications when mixed with the other Fault Tolerance annotations. This behavior will be clarified in the upcoming 1.2 version.