Quarkus and Observability

Observability is the ability to watch the state of a system/application based on some external outputs.

These are the four important concepts to have observability correctly implemented:

- Monitoring/Metrics (Prometheus)
- Visualization (Graphana)
- Distributed tracing (Jaeger)
- Log aggregation (Sentry, GELF)

This cheat sheet covers the integrations you can find in the form of extensions for Quarkus to implement observability.

CREATING THE PROJECT

```
mvn "io.quarkus:quarkus-maven-plugin:1.4.2.Final:create" \
  -DprojectGroupId="org.acme" \
  -DprojectArtifactId="greeting" \
  -DprojectVersion="1.0-SNAPSHOT" \
  -DclassName="org.acme.GreetingResource" \
  -Dpath="/hello"
```

Tip You can generate the project in https://code.quarkus.io/

CREATING THE PROJECT

Quarkus can utilize the MicroProfile Metrics spec to provide metrics support.

```
./mvnw quarkus:add-extension -
Dextensions="io.quarkus:quarkus-smallrye-metrics"
```

The metrics can be provided in JSON or OpenMetrics format.

When the extension is present in the classpath, an endpoint is added automatically to /metrics providing default metrics. To add custom metrics, MicroProfile Metrics spec provides the next annotations:

`@Timed` / Tracks the duration of a call.

`@SimplyTimed` / Tracks the duration of a call without mean and distribution calculations.

`@Metered` / Tracks the frequency of invocations.

`@Counted` / Counts number of invocations.

`@Gauge` / Samples the value of the annotated object.

`@ConcurrentGauge` / Gauge to count parallel invocations.

`@Metric` / Used to inject a metric. Valid types are **Meter, Timer, Counter, Histogram. Gauge** type can only be produced in methods or fields.

```
@GET
//...
@Timed(name = "checksTimer",
unit = MetricUnits.MILLISECONDS)
public String hello() {}

@Counted(name = "countWelcome")
public String hello() {}
```

`@Gauge` annotation returning a measure as a gauge.

```
@Gauge(name = "hottestSauce", unit = MetricUnits.NONE)
public Long hottestSauce() {}
```

Injecting a histogram using `@Metric`.

```
@Metric(name = "histogram")
Historgram historgram;
```

Metrics can be configured in `application.properties` with the next properties:

`quarkus.smallrye-metrics.path`
The path to the metrics handler, defaults to /metrics.

`quarkus.smallrye-metrics.extensions.enabled`
If metrics are enabled or not, defaults to true.

`quarkus.smallrye-metrics.micrometer.compatibility`
Applies Micrometer compatibility mode, defaults to false.

You can apply metric annotations via CDI stereotypes:

```
@Stereotype
@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.TYPE, ElementType.METHOD,
ElementType.FIELD })
@Timed(name = "checksTimer", unit =
MetricUnits.MILLISECONDS)
public @interface TimedMilliseconds {
}
```

Metrics in Quarkus also integrates with other extensions:

`quarkus.smallrye-metrics.path`
If enabled Hibernate metrics are exposed under vendor scope.

`quarkus.mongodb.metrics.enabled`
If enabled MongoDB metrics are exposed under vendor scope.

DISTRIBUTED TRACING

Quarkus can utilize the MicroProfile OpenTracing spec to provide tracing support.

```
./mvnw quarkus:add-extension -
Dextensions="io.quarkus:quarkus-smallrye-opentracing"
```

All the requests sent to any endpoint are traced automatically.

This extension includes OpenTracing and Jaeger tracer support.

Jaeger tracer has multiple configuration properties, but a typical example is:

application.properties

```
quarkus.jaeger.service-name=myservice
quarkus.jaeger.sampler-type=const
quarkus.jaeger.sampler-param=1
quarkus.jaeger.endpoint=http://localhost:14268/api/traces
```

`@Traced(false)` annotation can be used to disable tracing at class or method level.

`io.opentracing.Tracer` interface can be injected into a class to manipulate the information that is traced.

```
@Inject
Tracer tracer;

tracer.activeSpan().setBaggageItem("key", "value");
```

You can disable the Jaeger extension by using quarkus.jaeger.enabled property.

You can log the traceId, spanId and sampled tracing information in the Quarkus logging system by configuring the log format:

```
quarkus.log.console.format=%d{HH:mm:ss} %-5p
traceId=%X{traceId}, spanId=%X{spanId},
sampled=%X{sampled} [%c{2.}] (%t) %s%e%n
```

Tracing in Quarkus also integrates with other extensions:

JDBC Tracer

Adds a span for each JDBC queries.

```
<dependency>
    <groupId>io.opentracing.contrib</groupId>
    <artifactId>opentracing-jdbc</artifactId>
</dependency>
```

Configure JDBC driver apart from tracing properties seen before:

```
# add ':tracing' to your database URL
quarkus.datasource.url
=jdbc:tracing:postgresql://localhost:5432/mydatabase
quarkus.datasource.driver
=io.opentracing.contrib.jdbc.TracingDriver
quarkus.hibernate-orm.dialect
=org.hibernate.dialect.PostgreSQLDialect
```

AWS XRay

If you are building native images, and want to use AWS X-Ray Tracing with your lambda you will need to include quarkus-amazon-lambda-xray as a dependency in your pom.

LOG AGGREGATION

Sentry

Quarkus integrates with Sentry for logging errors into an error monitoring system.

```
./mvnw quarkus:add-extension  -Dextensions="quarkus-
logging-sentry"
```

As an example if you want to send all errors occuring in the package org.example to Sentry with DSN https://abcd@sentry.io/1234, you should configure it in the follwoing way:

application.properties

```
quarkus.log.sentry=true
quarkus.log.sentry.dsn=https://abcd@sentry.io/1234
quarkus.log.sentry.level=ERROR
quarkus.log.sentry.in-app-packages=org.example
```

Full list of configuration properties for Sentry are:

`quarkus.log.sentry.enable` / Enables the Sentry logging extension, defaults to false.

`quarkus.log.sentry.dsn` / The DSN where events are sent.

`quarkus.log.sentry.level` / The log level, defaults to WARN.

`quarkus.log.sentry.in-app-packages` / Configures application package prefixes.

`quarkus.log.sentry.environment` / Sets the environment value.

`quarkus.log.sentry.release` / Sets the release value.

GELF format

You can configure the output logging to be in GELF format instead of plain text.

```
./mvnw quarkus:add-extension -Dextensions="quarkus-
logging-gelf"
```

`quarkus.log.handler.gelf.enabled` / Enables GELF logging handler, defaults to false.

`quarkus.log.handler.gelf.host` / The Hostname/IP of Logstash/Graylof. Prepend tcp: for using TCP protocol, defaults to udp:localhost.

`quarkus.log.handler.gelf.port` / The port, defaults to 12201.

`quarkus.log.handler.gelf.version` / The GELF version, defaults to 1.1.

`quarkus.log.handler.gelf.extract-stack-trace` / Posts Stack-Trace to StackTrace field, defaults to true.

`quarkus.log.handler.gelf.stack-trace-throwable-reference` / Gets the cause level to stack trace. 0 is fulls tack trace, defaults to 0.

`quarkus.log.handler.gelf.filter-stack-trace` / Sets the stack-Trace filtering, defaults to false.

`quarkus.log.handler.gelf.timestamp-pattern` / Sets the tiemstamp format in Java Date pattern, defaults to yyyy-MM-dd HH:mm:ss,SSS.

`quarkus.log.handler.gelf.level` / Sets the log level using java.util.logging.Level class, defaults to ALL.

`quarkus.log.handler.gelf.facility` / The name of the facility, defaults to jboss-logmanager.

`quarkus.log.handler.gelf.extract-stack-trace` / Posts Stack-Trace to StackTrace field, defaults to true.

`quarkus.log.handler.gelf.additional-field.<field>.<subfield>` / Posts additional fields (ie quarkus.log.handler.gelf.additional-field.field1.type=String)

Author Alex Soto
Java Champion, Working at Red Hat