Quarkus - MicroProfile Health

This guide demonstrates how your Quarkus application can utilize the MicroProfile Health specification through the SmallRye Health extension.

MicroProfile Health allows applications to provide information about their state to external viewers which is typically useful in cloud environments where automated processes must be able to determine whether the application should be discarded or restarted.

Prerequisites

To complete this guide, you need:

- less than 15 minutes
- an IDE
- JDK 1.8+ installed with JAVA_HOME configured appropriately
- Apache Maven 3.6.3

Architecture

In this guide, we build a simple REST application that exposes MicroProfile Health functionalities at the /health/live and /health/ready endpoints according to the specification.

Solution

We recommend that you follow the instructions in the next sections and create the application step by step. However, you can go right to the completed example.

Clone the Git repository: git clone https://github.com/quarkusio/quarkus-quickstarts.git, or download an archive.

The solution is located in the microprofile-health-quickstart directory.

Creating the Maven Project

First, we need a new project. Create a new project with the following command:

```
mvn io.quarkus:quarkus-maven-plugin:1.7.0.CR1:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=microprofile-health-quickstart \
    -Dextensions="health"
cd microprofile-health-quickstart
```

This command generates a Maven project, importing the `smallrye-health` extension which is an implementation of the MicroProfile Health specification used in Quarkus.

If you already have your Quarkus project configured, you can add the `smallrye-health` extension to your project by running the following command in your project base directory:

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

This will add the following to your `pom.xml`:

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-smallrye-health</artifactId>
</dependency>
```

Running the health check

Importing the `smallrye-health` extension directly exposes three REST endpoints:

- `/health/live` - The application is up and running.
- `/health/ready` - The application is ready to serve requests.
- `/health` - Accumulating all health check procedures in the application.

To check that the `smallrye-health` extension is working as expected:

- start your Quarkus application with `./mvnw compile quarkus:dev`
- access the `http://localhost:8080/health/live` endpoint using your browser or `curl http://localhost:8080/health/live`

All of the health REST endpoints return a simple JSON object with two fields:

- `status` — the overall result of all the health check procedures
- `checks` — an array of individual checks

The general `status` of the health check is computed as a logical AND of all the declared health check procedures. The `checks` array is empty as we have not specified any health check procedure yet so let's define some.

Creating your first health check

In this section, we create our first simple health check procedure.

Create the `org.acme.microprofile.health.SimpleHealthCheck` class:

```
package org.acme.microprofile.health;

import org.eclipse.microprofile.health.HealthCheck;
import org.eclipse.microprofile.health.HealthCheckResponse;
import org.eclipse.microprofile.health.Liveness;

import javax.enterprise.context.ApplicationScoped;

@Liveness
@ApplicationScoped ① ②
public class SimpleHealthCheck implements HealthCheck {

    @Override
    public HealthCheckResponse call() {
        return HealthCheckResponse.up("Simple health check");
    }
}
```

① It's recommended to annotate the health check class with `@ApplicationScoped` or the `@Singleton` scope so that a single bean instance is used for all health check requests.

② If a bean class annotated with one of the health check annotations declares no scope then the `@Singleton` scope is used automatically.

As you can see, the health check procedures are defined as CDI beans that implement the `HealthCheck` interface and are annotated with one of the health check qualifiers, such as:

- `@Liveness` - the liveness check accessible at `/health/live`

- `@Readiness` - the readiness check accessible at `/health/ready`

`HealthCheck` is a functional interface whose single method `call` returns a `HealthCheckResponse` object which can be easily constructed by the fluent builder API shown in the example.

As we have started our Quarkus application in dev mode simply repeat the request to `http://localhost:8080/health/live` by refreshing your browser window or by using `curl http://localhost:8080/health/live`. Because we defined our health check to be a liveness procedure (with `@Liveness` qualifier) the new health check procedure is now present in the `checks` array.

Congratulations! You've created your first Quarkus health check procedure. Let's continue by exploring what else can be done with the MicroProfile Health specification.

Adding a readiness health check procedure

In the previous section, we created a simple liveness health check procedure which states whether our application is running or not. In this section, we will create a readiness health check which will be able to state whether our application is able to process requests.

We will create another health check procedure that simulates a connection to an external service provider such as a database. For starters, we will always return the response indicating the application is ready.

Create `org.acme.microprofile.health.DatabaseConnectionHealthCheck` class:

```java
package org.acme.microprofile.health;

import org.eclipse.microprofile.health.HealthCheck;
import org.eclipse.microprofile.health.HealthCheckResponse;
import org.eclipse.microprofile.health.Readiness;

import javax.enterprise.context.ApplicationScoped;

@Readiness
@ApplicationScoped
public class DatabaseConnectionHealthCheck implements HealthCheck {

    @Override
    public HealthCheckResponse call() {
        return HealthCheckResponse.up("Database connection health check");
    }
}
```

If you now rerun the health check at `http://localhost:8080/health/live` the `checks` array will contain only the previously defined `SimpleHealthCheck` as it is the only check defined with the `@Liveness` qualifier. However, if you access `http://localhost:8080/health/ready` (in the browser or with `curl http://localhost:8080/health/ready`) you will see only the `Database connection health check` as it is the only health check defined with the `@Readiness` qualifier as the readiness health check procedure.

> ℹ️ If you access `http://localhost:8080/health` you will get back both checks.

More information about which health check procedures should be used in which situation is detailed in the MicroProfile Health specification. Generally, the liveness procedures determine whether the application should be restarted while readiness procedures determine whether it makes sense to contact the application with requests.

Negative health check procedures

In this section, we extend our `Database connection health check` with the option of stating that our application is not ready to process requests as the underlying database connection cannot be established. For simplicity reasons, we only determine whether the database is accessible or not by a configuration property.

Update the `org.acme.microprofile.health.DatabaseConnectionHealthCheck` class:

```java
package org.acme.microprofile.health;

import org.eclipse.microprofile.config.inject.ConfigProperty;
import org.eclipse.microprofile.health.HealthCheck;
import org.eclipse.microprofile.health.HealthCheckResponse;
import org.eclipse.microprofile.health.HealthCheckResponseBuilder;
import org.eclipse.microprofile.health.Readiness;

import javax.enterprise.context.ApplicationScoped;

@Readiness
@ApplicationScoped
public class DatabaseConnectionHealthCheck implements HealthCheck {

    @ConfigProperty(name = "database.up", defaultValue = "false")
    private boolean databaseUp;

    @Override
    public HealthCheckResponse call() {

        HealthCheckResponseBuilder responseBuilder =
HealthCheckResponse.named("Database connection health check");

        try {
            simulateDatabaseConnectionVerification();
            responseBuilder.up();
        } catch (IllegalStateException e) {
            // cannot access the database
            responseBuilder.down();
        }

        return responseBuilder.build();
    }

    private void simulateDatabaseConnectionVerification() {
        if (!databaseUp) {
            throw new IllegalStateException("Cannot contact
database");
        }
    }
}
```

Until now we used a simplified method of building a `HealthCheckResponse` through the `HealthCheckResponse#up(String)` (there is also `HealthCheckResponse#down(String)`) which will directly build the response object. From now on, we utilize the full builder capabilities provided by the `HealthCheckResponseBuilder` class.

If you now rerun the readiness health check (at `http://localhost:8080/health/ready`) the overall `status` should be DOWN. You can also check the liveness check at `http://localhost:8080/health/live` which will return the overall `status` UP because it isn't influenced by the readiness checks.

As we shouldn't leave this application with a readiness check in a DOWN state and because we are running Quarkus in dev mode you can add `database.up=true` in `src/main/resources/application.properties` and rerun the readiness health check again — it should be up again.

Adding user-specific data to the health check response

In previous sections, we saw how to create simple health checks with only the minimal attributes, namely, the health check name and its status (UP or DOWN). However, the MicroProfile specification also provides a way for the applications to supply arbitrary data in the form of key-value pairs sent to the consuming end. This can be done by using the `withData(key, value)` method of the health check response builder API.

Let's create a new health check procedure `org.acme.microprofile.health.DataHealthCheck`:

```
package org.acme.microprofile.health;

import org.eclipse.microprofile.health.Liveness;
import org.eclipse.microprofile.health.HealthCheck;
import org.eclipse.microprofile.health.HealthCheckResponse;

import javax.enterprise.context.ApplicationScoped;

@Liveness
@ApplicationScoped
public class DataHealthCheck implements HealthCheck {

    @Override
    public HealthCheckResponse call() {
        return HealthCheckResponse.named("Health check with data")
                .up()
                .withData("foo", "fooValue")
                .withData("bar", "barValue")
                .build();
    }
}
```

If you rerun the liveness health check procedure by accessing the `/health/live` endpoint you can see that the new health check `Health check with data` is present in the `checks` array. This check contains a new attribute called `data` which is a JSON object consisting of the properties we

have defined in our health check procedure.

This functionality is specifically useful in failure scenarios where you can pass the error along with the health check response.

```
        try {
            simulateDatabaseConnectionVerification();
            responseBuilder.up();
        } catch (IllegalStateException e) {
            // cannot access the database
            responseBuilder.down()
                    .withData("error", e.getMessage()); // pass the
exception message
        }
```

Extension health checks

Some extension may provide default health checks, including the extension will automatically register its health checks.

For example, `quarkus-agroal` that is used to manage Quarkus datasource(s) automatically register a readiness health check that will validate each datasources: Datasource Health Check.

You can disable extension health check via the property `quarkus.health.extensions.enabled` so none will be automatically registered.

Health UI

ℹ️   Experimental - not included in the MicroProfile specification

`health-ui` allows you to see your Health Checks in a Web GUI.

The Quarkus `smallrye-health` extension ships with `health-ui` and enables it by default in dev and test modes, but it can also be explicitly configured for production mode as well.

`health-ui` can be accessed from http://localhost:8080/health-ui/ .

Conclusion

MicroProfile Health provides a way for your application to distribute information about its healthiness state to state whether or not it is able to function properly. Liveness checks are utilized to tell whether the application should be restarted and readiness checks are used to tell whether the application is able to process requests.

All that is needed to enable the MicroProfile Health features in Quarkus is:

- adding the `smallrye-health` Quarkus extension to your project using the `quarkus-maven-plugin`:

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

- or simply adding the following Maven dependency:

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-smallrye-health</artifactId>
</dependency>
```

Configuration Reference

🔒 Configuration property fixed at build time - All other configuration properties are overridable at runtime

Configuration property	Type	Default
🔒 quarkus.health.extensions.enabled Whether or not extensions published health check should be enabled.	boolean	`true`
🔒 quarkus.smallrye-health.root-path Root path for health-checking endpoints.	string	`/health`
🔒 quarkus.smallrye-health.liveness-path The relative path of the liveness health-checking endpoint.	string	`/live`
🔒 quarkus.smallrye-health.readiness-path The relative path of the readiness health-checking endpoint.	string	`/ready`
🔒 quarkus.smallrye-health.group-path The relative path of the health group endpoint.	string	`/group`
UI configuration	**Type**	**Default**
🔒 quarkus.smallrye-health.ui.root-path The path where Health UI is available. The value `/` is not allowed as it blocks the application from serving anything else.	string	`/health-ui`
🔒 quarkus.smallrye-health.ui.always-include Always include the UI. By default this will only be included in dev and test. Setting this to true will also include the UI in Prod	boolean	`false`

🔒 quarkus.smallrye-health.ui.enable	boolean	true
If Health UI should be enabled. By default, Health UI is enabled.		

🔒 quarkus.smallrye-health.ui.enable

If Health UI should be enabled. By default, Health UI is enabled.