MicroProfile Health

The Microprofile community and it's contributors

2.2-RC1, January 22, 2020

Table of Contents

Specification: MicroProfile Health

Version: 2.2-RC1

Status: Draft

Release: January 22, 2020

MicroProfile Health

Rationale

The Eclipse MicroProfile Health Check specification defines a single container runtime mechanism for validating the availability and status of a MicroProfile implementation. This is primarily intended as a machine to machine (M2M) mechanism for use in containerized environments like cloud providers. Example of existing specifications from those environments include Cloud Foundry Health Checks and Kubernetes Liveness and Readiness Probes.

In this scenario health checks are used to determine if a computing node needs to be discarded (terminated, shutdown) and eventually replaced by another (healthy) instance.

The MicroProfile Health Check architecture consists of two `/health/ready` and `/health/live` endpoints in a MicroProfile runtime that respectively represent the readiness and the liveness of the entire runtime. These endpoints are linked to Health Check procedures defined with specifications API and annotated respectively with `@Liveness` and `@Readiness` annotations.

For backward compatibility, a 3rd endpoint `/health` may also be used to provide a combination of previous endpoints and Health Check procedures annotated with the deprecated `@Health` annotation.

These endpoints are expected to be associated with a configurable context, such as a web application deployment, that can be configured with settings such as port, virtual-host, security, etc. Further, the MicroProfile Health Check defines the notion of a procedure that represents the health of a particular subcomponent of an application.

In an application, there can be zero or more procedures bound to a given health endpoint. The overall application health for a given endpoint is the logical AND of all of the procedures bound to it.

The current version of the MicroProfile Health Check specification does not define how the defined endpoints may be partitioned in the event that the MicroProfile runtime supports deployment of multiple applications. If an implementation wishes to support multiple applications within a MicroProfile runtime, the semantics of individual endpoints are expected to be the logical AND of all the application in the runtime. The exact details of this are deferred to a future version of the MicroProfile Health Check specification.

Proposed solution

The proposed solution breaks down into two parts:

- A Java API to implement health check procedures
- A health checks protocol and wireformat

Contributors

- John Ament
- Heiko Braun
- Clément Escoffier
- Emily Jiang
- Werner Keil
- Jeff Mesnil
- Andrew Pielage
- Heiko Rupp
- Antoine Sabot-Durand
- Scott Stark
- Martin Stefanko
- Kevin Sutter

Java API Usage

This specification provides the following API to define health check procedures.

Common API check

The main API to provide health check procedures (readiness or liveness) on the application level is the `HealthCheck` interface:

```
@FunctionalInterface
public interface HealthCheck {

    HealthCheckResponse call();
}
```

Applications provide health check procedures (implementation of a `HealthCheck`), which will be used by the runtime hosting the application to verify the healthiness of the computing node.

Different kinds of Health Checks

This specification provides different kinds of health check procedures. Difference between them is only semantic. The nature of the procedure is defined by annotating the `HealthCheck` procedure with a specific annotation.

- Readiness checks defined with `@Readiness` annotation
- Liveness checks defined with `@Liveness` annotation
- Custom checks defined with `@HealthGroup` annotation
- Backward compatible checks defined with `@Health` annotation

A `HealthCheck` procedure with none of the above annotations is not an active procedure and should be ignored.

Readiness check

A Health Check for readiness allows third party services to know if the application is ready to process requests or not.

The `@Readiness` annotation must be applied on a `HealthCheck` implementation to define a readiness check procedure, otherwise, this annotation is ignored.

Liveness check

A Health Check for liveness allows third party services to determine if the application is running. This means that if this procedure fails the application can be discarded (terminated, shutdown).

The `@Liveness` annotation must be applied on a `HealthCheck` implementation to define a Liveness check procedure, otherwise, this annotation is ignored.

Custom check

A new kind of Health Check can be defined with the `@HealthGroup` annotation.

One or more Health Check procedure(s) can be annotated with `@HealthGroup` to apply a custom group name to the procedure.

All the Health Check procedures belonging to the same group define a new kind of Health Check.

The `@HealthGroup` annotation must be applied on a `HealthCheck` implementation to define a custom check procedure with the given name, otherwise, this annotation is ignored.

A `HealthCheck` implementation can have multiple `@HealthGroup` annotations to make it a member of multiple custom Health Check's groups.

Backward compatible check

To provide backward compatibility with previous specification version, a `HealthCheck` implementation with `@Health` annotation is still supported.

`@Health` annotation is deprecated, new procedures shouldn't use it.

Multiple HealthChecks procedures for a given kind

There can be one or several `HealthCheck` exposed for a given kind, they will all be invoked when an inbound protocol request is received (i.e. HTTP).

If more than one `HealthCheck` are invoked, they will be called in an unpredictable order.

The runtime will `call()` each `HealthCheck` which in turn creates a `HealthCheckResponse` that signals the health status to a consuming end:

```
public class HealthCheckResponse {

    public enum State { UP, DOWN }

    public abstract String getName();

    public abstract State getState();

    public abstract Optional<Map<String, Object>> getData();

    [...]
}
```

The status of all `HealthCheck` 's determines the overall status for the given Health check kind.

Combining multiple kinds of checks

A `HealthCheck` implementation may be annotated with multiple kinds of checks. The procedure will be used to resolve every kind of health check for which it is annotated.

For instance this procedure will be used to resolve liveness and readiness health check.

```
@Liveness
@Readiness
public class MyCheck implements HealthCheck {

  public HealthCheckResponse call() {
    ...
  }
}
```

Constructing `HealthCheckResponse` 's

Application level code is expected to use one of static methods on `HealthCheckResponse` to retrieve a `HealthCheckResponseBuilder` used to construct a response, i.e. :

```
public class SuccessfulCheck implements HealthCheck {
    @Override
    public HealthCheckResponse call() {
        return HealthCheckResponse.up("successful-check");
    }
}
```

The `name` is used to tell the different checks apart when a human operator looks at the responses. It may be that one check of several fails and it's useful to know which one. It's required that a response defines a name.

`HealthCheckResponse` 's also support a free-form information holder, that can be used to supply arbitrary data to the consuming end:

```
public class CheckDiskspace implements HealthCheck {

    @Override
    public HealthCheckResponse call() {
        return HealthCheckResponse.named("diskspace")
                .withData("free", "780mb")
                .up()
                .build();
    }
}
```

`HealthCheckResponse` also provides a constructor to allow instantiation on the consuming end.

Integration with CDI

Any enabled bean with a bean of type `org.eclipse.microprofile.health.HealthCheck` and `@Liveness`, `@Readiness`, `@HealthGroup` or `@Health` qualifier can be used as health check procedure.

Contextual references of health check procedures are invoked by runtime when the outermost protocol entry point (i.e. `http://HOST:PORT/health`) receives an inbound request

```
@ApplicationScoped
public class MyCheck implements HealthCheck {

    public HealthCheckResponse call() {
        [...]
    }
}
```

Health check procedures are CDI beans, therefore, they can also be defined with CDI producers:

```
@ApplicationScoped
class MyChecks {

  @Produces
  @Liveness
  HealthCheck check1() {
    return () -> HealthCheckResponse.named("heap-memory").state(getMemUsage() <
0.9).build();
  }

  @Produces
  @Readiness
  HealthCheck check2() {
    return () -> HealthCheckResponse.named("cpu-usage").state(getCpuUsage() <
0.9).build();
  }
}
```

Protocol and Wireformat

Abstract

This document defines the protocol to be used by components that need to ensure a compatible wireformat, agreed upon semantics and possible forms of interactions between system components that need to determine the “liveliness” or "readiness" of computing nodes in a bigger system.

Guidelines

Note that the force of these words is modified by the requirement level of the document in which they are used.

1. MUST This word, or the terms "REQUIRED" or "SHALL", mean that the definition is an absolute requirement of the specification.
2. MUST NOT This phrase, or the phrase "SHALL NOT", mean that the definition is an absolute prohibition of the specification.
3. SHOULD This word, or the adjective "RECOMMENDED", mean that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.
4. SHOULD NOT This phrase, or the phrase "NOT RECOMMENDED" mean that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label.
5. MAY – This word, or the adjective “OPTIONAL,” mean that an item is truly discretionary.

Goals

- MUST be compatibility with well known cloud platforms (i.e. http://kubernetes.io/docs/user-guide/liveness/)
- MUST be appropriate for machine-to-machine communication
- SHOULD give enough information for a human administrator

Terms used

Term	Description
Producer	The service/application that is checked
Consumer	The probing end, usually a machine, that needs to verify the liveness or readiness of a Producer
Health Check Procedure	The code executed to determine the liveliness of a Producer
Producer status	The overall status, determined by considering all health check procedure results
Health check procedure result	The result of single check

Protocol Overview

1. Consumer invokes the health check of a Producer through any of the supported protocols
2. Producer enforces security constraints on the invocation (i.e authentication)
3. Producer executes a set of Health check procedures (could be a set with one element)
4. Producer determines the overall status
5. The status is mapped to outermost protocol (i.e. HTTP status codes)
6. The payload is written to the response stream
7. The consumer reads the response
8. The consumer determines the overall status

Protocol Specifics

This section describes the specifics of the HTTP protocol usage.

Interacting with producers

How are the health checks accessed and invoked ? We don't make any assumptions about this, except for the wire format and protocol.

Protocol Mappings

Health checks (innermost) can and should be mapped to the actual invocation protocol (outermost). This section described some of guidelines and rules for these mappings.

- Producers MAY support a variety of protocols but the information items in the response payload MUST remain the same.
- Producers SHOULD define a well known default context to perform checks
- Each response SHOULD integrate with the outermost protocol whenever it makes sense (i.e. using HTTP status codes to signal the overall status)
- Inner protocol information items MUST NOT be replaced by outer protocol information items, rather kept redundantly.
- The inner protocol response MUST be self-contained, that is carrying all information needed to reason about the producer status

Mandatory and optional protocol types

REST/HTTP interaction

- Producer MUST provide a HTTP endpoint that follows the REST interface specifications described in Appendix A.

Protocol Adaptor

Each provider MUST provide the REST/HTTP interaction, but MAY provide other protocols such as TCP or JMX. When possible, the output MUST be the JSON output returned by the equivalent HTTP calls (Appendix B). The request is protocol specific.

Healthcheck Response information

- The primary information MUST be boolean, it needs to be consumed by other machines. Anything between available/unavailable doesn't make sense or would increase the complexity on the side of the consumer processing that information.
- The response information MAY contain an additional information holder
- Consumers MAY process the additional information holder or simply decide to ignore it
- The response information MUST contain the boolean `status` of each check
- The response information MUST contain the name of each check

Wireformats

- Producer MUST support JSON encoded payload with simple UP/DOWN states
- Producers MAY support an additional information holder with key/value pairs to provide further context (i.e. disk.free.space=120mb).
- The JSON response payload MUST be compatible with the one described in Appendix B
- The JSON response MUST contain the `name` entry specifying the name of the check, to support protocols that support external identifier (i.e. URI)
- The JSON response MUST contain the `status` entry specifying the state as String: "UP" or "DOWN"
- The JSON MAY support an additional information holder to carry key value pairs that provide additional context

Health Check Procedures

- A producer MUST support custom, application level health check procedures
- A producer SHOULD support reasonable out-of-the-box procedures
- A producer with no health check procedures expected or installed MUST return positive overall status (i.e. HTTP 200)
- A producer with health check procedures expected but not yet installed MUST return negative overall status (i.e. HTTP 503)

Policies to determine the overall status

When multiple procedures are installed all procedures MUST be executed and the overall status needs to be determined.

- Consumers MUST support a logical conjunction policy to determine the status
- Consumers MUST use the logical conjunction policy by default to determine the status
- Consumers MAY support custom policies to determine the status

Executing procedures

When executing health check procedures a producer MUST handle any unchecked exceptions and synthesize a substitute respone.

- The synthesized response MUST contain a `status` entry with a value of "DOWN".
- The synthesized response MUST contain a `name` entry with a value set to the runtime class name of the failing check.
- The synthesized response MAY contain additional information about the failure (i.e. exception message or stack trace)

Disabling default vendor procedures

An implementation is allowed to supply a reasonable default (out-of-the-box) procedures as defined in the Health Check Procedures section. To disable all default vendor procedures users can specify a MicroProfile Config configuration property `mp.health.disable-default-procedures` to `true`. This allows the application to process and display only the user-defined health check procedures.

Security

Aspects regarding the secure access of health check information.

- A producer MAY support security on all health check invocations (i.e. authentication)
- A producer MUST NOT enforce security by default, it SHOULD be an opt-in feature (i.e. configuration change)

Appendix A: REST interfaces specifications

Context	Verb	Status Code	Kind of procedure called	Response
/health/live	GET	200, 500, 503	Liveness	See Appendix B
/health/ready	GET	200, 500, 503	Readiness	See Appendix B
/health/group/{name}	GET	200, 500, 503	Custom with given name	See Appendix B
/health/group	GET	200, 500, 503	All custom groups	See Appendix B
/health	GET	200, 500, 503	Backward compatible + Liveness + Readiness	See Appendix B

Status Codes:

- 200 for a health check with a positive status (`UP`)
- 503 in case the overall status is negative (`DOWN`)
- 500 in case the producer wasn't able to process the health check request (i.e. error in procedure)

Appendix B: JSON payload specification

Response Codes and status mappings

The following table gives valid health check responses for all kinds of health checks:

Request	HTTP Status	JSON Payload	State	Comment
/health/live /health/ready /health/group/{name} /health/group /health	200	Yes	UP	Check with payload. See With procedures installed into the runtime.
/health/live /health/ready /health/group/{name} /health/group /health	200	Yes	UP	Check with no procedures expected or installed. See With no procedures expected or installed into the runtime
/health/live /health/ready /health/group/{name} /health/group /health	503	Yes	Down	Check failed
/health/live /health/ready /health/group/{name} /health/group /health	503	Yes	Down	Check with procedures expected but not yet installed. See With procedures expected but not yet installed into the runtime
/health/live /health/ready /health/group/{name} /health/group /health	500	No	Undetermined	Request processing failed (i.e. error in procedure)

JSON Schema:

```
{
 "$schema": "http://json-schema.org/draft-04/schema#",
 "type": "object",
 "properties": {
   "status": {
     "type": "string"
   },
   "checks": {
     "type": "array",
     "items": {
       "type": "object",
       "properties": {
         "name": {
           "type": "string"
         },
         "status": {
           "type": "string"
         },
         "data": {
           "type": "object",
           "patternProperties": {
             "[a-zA-Z_]*": {
               "type": [
                 "string",
                 "boolean",
                 "number"
               ]
             }
           },
           "additionalProperties": false
         }
       },
       "required": [
         "name",
         "status"
       ]
     }
   }
 },
 "required": [
   "status",
   "checks"
 ],
 "additionalProperties": false
}
```

(See http://jsonschema.net/#/)

Example response payloads

With procedures installed into the runtime

Status `200` and the following payload:

```
{
  "status": "UP",
  "checks": [
    {
      "name": "myCheck",
      "status": "UP",
      "data": {
        "key": "value",
        "foo": "bar"
      }
    }
  ]
}
```

Status `503` and the following payload:

```
{
  "status": "DOWN",
  "checks": [
    {
      "name": "firstCheck",
      "status": "DOWN",
      "data": {
        "key": "value",
        "foo": "bar"
      }
    },
    {
      "name": "secondCheck",
      "status": "UP"
    }
  ]
}
```

Status 500

```
{
  "status": "DOWN",
  "checks": [
    {
      "name": "example.health.FirstCheck",
      "status": "DOWN",
      "data": {
        "rootCause": "timed out waiting for available connection"
      }
    },
    {
      "name": "secondCheck",
      "status": "UP"
    }
  ]
}
```

With no procedures expected or installed into the runtime

Status `200` and the following payload:

```
{
  "status": "UP",
  "checks": []
}
```

With procedures expected but not yet installed into the runtime

Status `503` and the following payload:

```
{
  "status": "DOWN",
  "checks": []
}
```

Architecture

SPI Usage

Implementors of the API are expected to supply implementations of `HealthCheckResponse` and `HealthCheckResponseBuilder` by providing a `HealthCheckResponseProvider` to their implementation. The `HealthCheckResponseProvider` is discovered using the default JDK service loader.

A `HealthCheckResponseProvider` is used internally to create a `HealthCheckResponseBuilder` which is used to construct a `HealthCheckResponse`. This pattern allows implementors to extend a `HealthCheckResponse` and adapt it to their implementation needs. Common implementation details that fall into this category are invocation and security contexts or anything else required to map a `HealthCheckResponse` to the outermost invocation protocol (i.e. HTTP/JSON).

Release Notes

Release Notes for MicroProfile Health Check 2.1

The following changes occurred in the 2.1 release, compared to 2.0

A full list of changes may be found on the MicroProfile Health Check 2.1

API/SPI Changes

- Add new method to create responses
- Add config property to disable implementation Health Check procedures
- Improve javadoc

TCK enhancement

- Testing JSON format
- Add delayed test
- Add test name before each tests

Miscellaneous

- Remove duplicate Arquillian import
- Remove EL API transitive dependency

Release Notes for MicroProfile Health Check 2.0

The following changes occurred in the 2.0 release, compared to 1.0

A full list of changes may be found on the MicroProfile Health Check 2.0

API/SPI Changes

- Deprecation of `@Health` qualifier
- Introduction of `@Liveness` and `@Readiness` qualifiers

Protocol and wireformat changes

- In response JSON format replaced `outcome` and `state` by `status`. **This change breaks backward compatibility with version 1.0**
- Introduction of `/health/live` endpoint that must call all the liveness procedures
- Introduction of `/health/ready` endpoint that must call all the readiness procedures
- For backward compatibility, `/health` endpoint should now call all procedures having `@Health`, `@Liveness` or `@Readiness` qualifiers
- Correction and enhancement of response JSON format.

TCK enhancement

- Adding tests for new types of health check procedures
- Cleaning existing tests