Service Health Checks

The Microprofile community and it's contributors

1.0.0-RC2, 2017-08-30

Table of Contents

Service Health checks . . . 2
- Rationale . . . 3
- Proposed solution . . . 4
- Contributors . . . 5

Java API Usage . . . 6
- Constructing `HealthCheckResponse`s . . . 7

Integration with CDI . . . 8

Protocol and Wireformat . . . 9
- Abstract . . . 10
  - Guidelines . . . 10
- Goals . . . 11
- Terms used . . . 12
- Protocol Overview . . . 13
  - Protocol Specifics . . . 13
    - Interacting with producers . . . 13
    - Protocol Mappings . . . 13
  - Mandatory and optional protocol types . . . 13
    - REST/HTTP interaction . . . 13
    - Protocol Adaptor . . . 14
- Healthcheck Response information . . . 15
  - Wireformats . . . 15
- Health Check Procedures . . . 16
  - Policies to determine the overall outcome . . . 16
- Security . . . 17
- Appendix A: REST interface specifications . . . 18
  - Status Codes: . . . 18
- Appendix B: JSON payload specification . . . 19
  - Response Codes and status mappings . . . 19
  - JSON Schema: . . . 19
- Example response payloads . . . 21
  - With procedures installed into the runtime . . . 21
  - Without procedures installed into the runtime . . . 21

Architecture . . . 23
- SPI Usage . . . 24

Specification: Service Health Checks

Version: 1.0.0-RC2

Status: Release Candidate

Release: 2017-08-30

Service Health checks

Rationale

The Eclipse MicroProfile Health Check (MP-HC) 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 and containerized environments. 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 MP-HC architecture consists of a single `/health` endpoint in a MicroProfile runtime that represents the status of the entire runtime. This `/health` endpoint is 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 MP-HC defines the notion of a procedure that represents the health of a particular subcomponent of an application. There can be zero or more procedures in an application, and the overall health of the application as reflected via the `/health` endpoint is the logical AND of all of the procedure health status.

The 1.0 version of the MP-HC specification does not define how the `/health` endpoint 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 the /health endpoint 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 MP-HC 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

- Andrew Pielage
- Emily Jiang
- Heiko Braun
- Jeff Mesnil
- John Ament
- John D. Ament
- Kevin Sutter
- Scott Stark
- Werner Keil

Java API Usage

The main API to provide health check procedures 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 the runtime hosting the application to verify the healthiness of the computing node.

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

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

```
public abstract 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 outcome.

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.named("successful-check").up();
    }
}
```

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.

`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();
    }
}
```

Integration with CDI

Within CDI contexts, beans that implement `HealthCheck` and annotated with `@Health` are discovered automatically and are invoked by runtime when the outermost protocol entry point (i.e. `http://HOST:PORT/health`) receives an inbound request.

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

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

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” 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 of a Producer
Health Check Procedure	The code executed to determine the liveliness of a Producer
Producer Outcome	The overall outcome, 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 outcome (Producer outcome)
5. The outcome 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 outcome

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 state)
- 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 the producer outcome

Mandatory and optional protocol types

REST/HTTP interaction

- Producer MUST provide a HTTP endpoint that follow 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 state 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 `state` 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 without health check procedures installed MUST returns positive overall outcome (i.e. HTTP 200)

Policies to determine the overall outcome

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

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

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 interface specifications

Context	Verb	Status Code	Response
/health	GET	200, 500, 503	See Appendix B

Status Codes:

- 200 for a health check with a positive outcome
- 503 in case the overall outcome is negative
- 500 in case the consumer 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 give valid health check responses:

Request	HTTP Status	JSON Payload	State	Comment
/health	200	Yes	UP	Check with payload. See With procedures installed into the runtime.
/health	200	Yes	UP	Check without procedures installed. See Without procedures installed into the runtime
/health	503	Yes	Down	Check failed
/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": {
   "outcome": {
     "type": "string"
   },
   "checks": {
     "type": "array",
     "items": {
       "type": "object",
       "properties": {
         "name": {
           "type": "string"
         },
         "state": {
           "type": "string"
         },
         "data": {
           "type": "object",
           "properties": {
             "key": {
               "type": "string"
             },
             "value": {
               "type": "string|boolean|int"
             }
           }
         }
       },
       "required": [
         "name",
         "state"
         ]
     }
   }
 },
 "required": [
   "outcome",
   "checks"
 ]
}
```

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

Example response payloads

With procedures installed into the runtime

Status 200

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

Status 503

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

Without procedures installed into the runtime

Status `200` and the following payload:

```
{
  "outcome": "UP",
  "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).