Network Working Group
Internet-Draft
Intended status: Informational
Expires: 22 May 2021

I. Nadareishvili

18 November 2020

Health Check Response Format for HTTP APIs
draft-inadarei-api-health-check-05

Abstract

This document proposes a service health check response format for HTTP APIs.

Note to Readers

RFC EDITOR: please remove this section before publication

The issues list for this draft can be found at https://github.com/inadarei/rfc-healthcheck/issues (https://github.com/inadarei/rfc-healthcheck/issues).

The most recent draft is at https://inadarei.github.io/rfc-healthcheck/ (https://inadarei.github.io/rfc-healthcheck/).

Recent changes are listed at https://github.com/inadarei/rfc-healthcheck/commits/master (https://github.com/inadarei/rfc-healthcheck/commits/master).

See also the draft's current status in the IETF datatracker, at https://datatracker.ietf.org/doc/draft-inadarei-api-health-check/ (https://datatracker.ietf.org/doc/draft-inadarei-api-health-check/).

Status of This Memo

This Internet-Draft will expire on 22 May 2021.

Copyright Notice

Table of Contents

1. Introduction

The vast majority of modern APIs driving data to web and mobile applications use HTTP [RFC7230] as their protocol. The health and uptime of these APIs determine availability of the applications themselves. In distributed systems built with a number of APIs, understanding the health status of the APIs and making corresponding decisions, for caching, failover or circuit-breaking, are essential to the ability of providing highly-available solutions.

There exists a wide variety of operational software that relies on the ability to read health check response of APIs. However, there is currently no standard for the health check output response, so most applications either rely on the basic level of information included in HTTP status codes [RFC7231] or use task-specific formats.

Usage of task-specific or application-specific formats creates significant challenges, disallowing any meaningful interoperability across different implementations and between different tooling.

Standardizing a format for health checks can provide any of a number of benefits, including:

* Flexible deployment - since operational tooling and API clients can rely on rich, uniform format, they can be safely combined and substituted as needed.

* Evolvability - new APIs, conforming to the standard, can safely be introduced in any environment and ecosystem that also conforms to the same standard, without costly coordination and testing requirements.

This document defines a "health check" format using the JSON format [RFC8259] for APIs to use as a standard point for the health information they offer. Having a well-defined format for this purpose promotes good practice and tooling.

2. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

3. API Health Response

Health Check Response Format for HTTP APIs uses the JSON format described in [RFC8259] and has the media type "application/health+json".

Its content consists of a single mandatory root field ("status") and several optional fields:

3.1. status

status: (required) indicates whether the service status is acceptable or not. API publishers SHOULD use following values for the field:

* "pass": healthy (acceptable aliases: "ok" to support Node's Terminus and "up" for Java's SpringBoot),

* "fail": unhealthy (acceptable aliases: "error" to support Node's Terminus and "down" for Java's SpringBoot), and

* "warn": healthy, with some concerns.

The value of the status field is case-insensitive and is tightly related with the HTTP response code returned by the health endpoint. For "pass" status, HTTP response code in the 2xx-3xx range MUST be used. For "fail" status, HTTP response code in the 4xx-5xx range MUST be used. In case of the "warn" status, endpoints MUST return HTTP status in the 2xx-3xx range, and additional information SHOULD be provided, utilizing optional fields of the response.

A health endpoint is only meaningful in the context of the component it indicates the health of. It has no other meaning or purpose. As such, its health is a conduit to the health of the component. Clients SHOULD assume that the HTTP response code returned by the health endpoint is applicable to the entire component (e.g. a larger API or a microservice). This is compatible with the behavior that current infrastructural tooling expects: load-balancers, service discoveries and others, utilizing health-checks.

3.2. version

version: (optional) public version of the service.

3.3. releaseId

releaseId: (optional) in well-designed APIs, backwards-compatible changes in the service should not update a version number. APIs usually change their version number as infrequently as possible, to preserve stable interface. However, implementation of an API may change much more frequently, which leads to the importance of having separate "release number" or "releaseId" that is different from the public version of the API.

3.4. notes

notes: (optional) array of notes relevant to current state of health

3.5. output

output: (optional) raw error output, in case of "fail" or "warn" states. This field SHOULD be omitted for "pass" state.

3.6. checks

checks (optional) is an object that provides detailed health statuses of additional downstream systems and endpoints which can affect the overall health of the main API. Please refer to the "The Checks Object" section for more information.

3.7. links

links (optional) is an object containing link relations and URIs [RFC3986] for external links that MAY contain more information about the health of the endpoint. All values of this object SHALL be URIs. Keys MAY also be URIs. Per web-linking standards [RFC8288] a link relationship SHOULD either be a common/registered one or be indicated as a URI, to avoid name clashes. If a "self" link is provided, it MAY be used by clients to check health via HTTP response code, as mentioned above.

3.8. serviceId

serviceId (optional) is a unique identifier of the service, in the application scope.

3.9. description

description (optional) is a human-friendly description of the service.

4. The Checks Object

The "checks" object MAY have a number of unique keys, one for each logical downstream dependency or sub-component. Since each sub-component may be backed by several nodes with varying health statuses, these keys point to arrays of objects. In case of a single-node sub-component (or if presence of nodes is not relevant), a single-element array SHOULD be used as the value, for consistency.

The key identifying an element in the object SHOULD be a unique string within the details section. It MAY have two parts: "{componentName}:{measurementName}", in which case the meaning of the parts SHOULD be as follows:

* componentName: (optional) human-readable name for the component. MUST not contain a colon, in the name, since colon is used as a separator.

* measurementName: (optional) name of the measurement type (a data point type) that the status is reported for. MUST not contain a colon, in the name, since colon is used as a separator. The observation's name can be one of:

  - A pre-defined value from this spec. Pre-defined values include:

    o utilization

    o responseTime

    o connections

    o uptime

  - A common and standard term from a well-known source such as schema.org, IANA or microformats.

  - A URI that indicates extra semantics and processing rules that MAY be provided by a resource at the other end of the URI. URIs do not have to be dereferenceable, however. They are just a namespace, and the meaning of a namespace CAN be provided by any convenient means (e.g. publishing an RFC, Open API Spec document or a nicely printed book).

On the value side of the equation, each "component details" object in the array SHOULD have at least one key, and MAY have any or none of the following object keys:

4.1. componentId

componentId: (optional) is a unique identifier of an instance of a specific sub-component/dependency of a service. Multiple objects with the same componentID MAY appear in the details, if they are from different nodes.

4.2. componentType

componentType: (optional) SHOULD be present if componentName is present. It's a type of the component and could be one of:

* Pre-defined value from this spec. Pre-defined values include:

  - component

  - datastore

  - system

* A common and standard term from a well-known source such as schema.org, IANA or microformats.

* A URI that indicates extra semantics and processing rules that MAY be provided by a resource at the other end of the URI. URIs do not have to be dereferenceable, however. They are just a namespace, and the meaning of a namespace CAN be provided by any convenient means (e.g. publishing an RFC, Swagger document or a nicely printed book).

4.3. observedValue

observedValue: (optional) could be any valid JSON value, such as: string, number, object, array or literal.

4.4. observedUnit

observedUnit (optional) SHOULD be present if observedValue is present. Clarifies the unit of measurement in which observedUnit is reported, e.g. for a time-based value it is important to know whether the time is reported in seconds, minutes, hours or something else. To make sure unit is denoted by a well-understood name or an abbreviation, it SHOULD be one of:

* A common and standard term from a well-known source such as schema.org, IANA, microformats, or a standards document such as [RFC3339].

* A URI that indicates extra semantics and processing rules that MAY be provided by a resource at the other end of the URI. URIs do not have to be dereferenceable, however. They are just a namespace, and the meaning of a namespace CAN be provided by any convenient means (e.g. publishing an RFC, Swagger document or a nicely printed book).

4.5. status

status (optional) has the exact same meaning as the top-level "output" element, but for the sub-component/downstream dependency represented by the details object.

4.6. affectedEndpoints

affectedEndpoints (optional) is a JSON array containing URI Templates as defined by [RFC6570]. This field SHOULD be omitted if the "status" field is present and has value equal to "pass". A typical API has many URI endpoints. Most of the time we are interested in the overall health of the API, without diving into details. That said, sometimes operational and resilience middleware needs to know more details about the health of the API (which is why "checks" property provides details). In such cases, we often need to indicate which particular endpoints are affected by a particular check's troubles vs. other endpoints that may be fine.

4.7. time

time (optional) is the date-time, in ISO8601 format, at which the reading of the observedValue was recorded. This assumes that the value can be cached and the reading typically doesn't happen in real time, for performance and scalability purposes.

4.8. output

output (optional) has the exact same meaning as the top-level "output" element, but for the sub-component/downstream dependency represented by the details object. As is the case for the top-level element, this field SHOULD be omitted for "pass" state of a downstream dependency.

4.9. links

links (optional) has the exact same meaning as the top-level "output" element, but for the sub-component/downstream dependency represented by the details object.

4.10. Additional Keys

In addition to the above keys, additional user-defined keys MAY be included in the 'component details' object. Implementations MAY ignore any keys that are not part of the list of standard keys above.

5. Example Output

```
  GET /health HTTP/1.1
  Host: example.org
  Accept: application/health+json

  HTTP/1.1 200 OK
  Content-Type: application/health+json
  Cache-Control: max-age=3600
  Connection: close

{
  "status": "pass",
  "version": "1",
  "releaseId": "1.2.2",
  "notes": [""],
  "output": "",
  "serviceId": "f03e522f-1f44-4062-9b55-9587f91c9c41",
  "description": "health of authz service",
  "checks": {
    "cassandra:responseTime": [
      {
        "componentId": "dfd6cf2b-1b6e-4412-a0b8-f6f7797a60d2",
        "componentType": "datastore",
        "observedValue": 250,
        "observedUnit": "ms",
        "status": "pass",
        "affectedEndpoints" : [
          "/users/{userId}",
          "/customers/{customerId}/status",
          "/shopping/{anything}"
        ],
        "time": "2018-01-17T03:36:48Z",
        "output": ""
      }
    ],
    "cassandra:connections": [
      {
        "componentId": "dfd6cf2b-1b6e-4412-a0b8-f6f7797a60d2",
        "componentType": "datastore",
        "observedValue": 75,
        "status": "warn",
```

```
           "time": "2018-01-17T03:36:48Z",
           "output": "",
           "links": {
             "self": "http://api.example.com/dbnode/dfd6cf2b/health"
           }
         }
       ],
       "uptime": [
         {
           "componentType": "system",
           "observedValue": 1209600.245,
           "observedUnit": "s",
           "status": "pass",
           "time": "2018-01-17T03:36:48Z"
         }
       ],
       "cpu:utilization": [
         {
           "componentId": "6fd416e0-8920-410f-9c7b-c479000f7227",
           "node": 1,
           "componentType": "system",
           "observedValue": 85,
           "observedUnit": "percent",
           "status": "warn",
           "time": "2018-01-17T03:36:48Z",
           "output": ""
         },
         {
           "componentId": "6fd416e0-8920-410f-9c7b-c479000f7227",
           "node": 2,
           "componentType": "system",
           "observedValue": 85,
           "observedUnit": "percent",
           "status": "warn",
           "time": "2018-01-17T03:36:48Z",
           "output": ""
         }
       ],
       "memory:utilization": [
         {
           "componentId": "6fd416e0-8920-410f-9c7b-c479000f7227",
           "node": 1,
           "componentType": "system",
           "observedValue": 8.5,
           "observedUnit": "GiB",
           "status": "warn",
           "time": "2018-01-17T03:36:48Z",
           "output": ""
```

```
      },
      {
        "componentId": "6fd416e0-8920-410f-9c7b-c479000f7227",
        "node": 2,
        "componentType": "system",
        "observedValue": 5500,
        "observedUnit": "MiB",
        "status": "pass",
        "time": "2018-01-17T03:36:48Z",
        "output": ""
      }
    ]
  },
  "links": {
    "about": "http://api.example.com/about/authz",
    "http://api.x.io/rel/thresholds":
      "http://api.x.io/about/authz/thresholds"
  }
}
```

6. Security Considerations

Clients need to exercise care when reporting health information. Malicious actors could use this information for orchestrating attacks. In some cases, the health check endpoints may need to be authenticated and institute role-based access control.

7. IANA Considerations

The media type for health check response is application/health+json.

- Media type name: application

- Media subtype name: health+json

- Required parameters: n/a

- Optional parameters: n/a

- Encoding considerations: binary

- Security considerations: Health+JSON shares security issues common to all JSON content types. See RFC 8259 Section #12 for additional information.

  Health+JSON allows utilization of Uniform Resource Identifiers (URIs) and as such shares security issues common to URI usage. See RFC 3986 Section #7 for additional information.

Since health+json can carry wide variety of data, some data may require privacy or integrity services. This specification does not prescribe any specific solution and assumes that concrete implementations will utilize common, trusted approaches such as TLS/HTTPS, OAuth2 etc.

- Interoperability considerations: None

- Published specification: this RFC draft

- Applications which use this media: Various

- Fragment identifier considerations: Health+JSON follows RFC6901 for implementing URI Fragment Identification standard to JSON content types.

- Restrictions on usage: None

- Additional information:

  1. Deprecated alias names for this type: n/a

  2. Magic number(s): n/a

  3. File extension(s): .json

  4. Macintosh file type code: TEXT

  5. Object Identifiers: n/a

- General Comments:

- Person to contact for further information:

  1. Name: Irakli Nadareishvili

  2. Email: irakli@gmail.com

- Intended usage: Common

- Author/Change controller: Irakli Nadareishvili

8. Acknowledgements

Thanks to Mike Amundsen, Erik Wilde, Justin Bachorik and Randall Randall for their suggestions and feedback. And to Mark Nottingham for blueprint for authoring RFCs easily.

9. Creating and Serving Health Responses

When making an health check endpoint available, there are a few things to keep in mind:

- A health response endpoint is best located at a memorable and commonly-used URI, such as "health" because it will help self-discoverability by clients.

- Health check responses can be personalized. For example, you could advertise different URIs, and/or different kinds of link relations, to afford different clients access to additional health check information.

- Health check responses SHOULD be assigned a freshness lifetime (e.g., "Cache-Control: max-age=3600") so that clients can determine how long they could cache them, to avoid overly frequent fetching and unintended DDOS-ing of the service. Any method of cache lifetime negotiation provided by HTTP spec is acceptable (e.g. ETags are just fine).

- Custom link relation types, as well as the URIs for variables, SHOULD lead to documentation for those constructs.

10. Consuming Health Check Responses

Clients might use health check responses in a variety of ways.

Note that the health check response is a "living" document; links from the health check response MUST NOT be assumed to be valid beyond the freshness lifetime of the health check response, as per HTTP's caching model [RFC7234].

As a result, clients ought to cache the health check response (as per [RFC7234]), to avoid fetching it before every interaction (which would otherwise be required).

Likewise, a client encountering a 404 (Not Found) on a link is encouraged to obtain a fresh copy of the health check response, to assure that it is up-to-date.

11. References

11.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

[RFC6570] Gregorio, J., Fielding, R., Hadley, M., Nottingham, M., and D. Orchard, "URI Template", RFC 6570, DOI 10.17487/RFC6570, March 2012, <https://www.rfc-editor.org/info/rfc6570>.

[RFC7234] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Caching", RFC 7234, DOI 10.17487/RFC7234, June 2014, <https://www.rfc-editor.org/info/rfc7234>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

[RFC8288] Nottingham, M., "Web Linking", RFC 8288, DOI 10.17487/RFC8288, October 2017, <https://www.rfc-editor.org/info/rfc8288>.

11.2. Informative References

[RFC3339] Klyne, G. and C. Newman, "Date and Time on the Internet: Timestamps", RFC 3339, DOI 10.17487/RFC3339, July 2002, <https://www.rfc-editor.org/info/rfc3339>.

[RFC6838] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, DOI 10.17487/RFC6838, January 2013, <https://www.rfc-editor.org/info/rfc6838>.

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <https://www.rfc-editor.org/info/rfc7230>.

[RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <https://www.rfc-editor.org/info/rfc7231>.

Author's Address

Irakli Nadareishvili
114 5th Avenue
New York,
United States of America

Email: irakli@gmail.com
URI: http://www.freshblurbs.com