



How to Use the Payara Server Implementation of the MicroProfile Health API

**The Payara® Platform - Production-Ready,
Cloud Native and Aggressively Compatible.**

Contents

Introduction	1
What Do I Need to Use This Guide?	1
What is MicroProfile?	2
Getting Started with MicroProfile	3
Health vs Metrics API	3
Health API in Action	4
@Liveness check	5
@Readiness check	9
Compose health checks	11
@Health	13
Health check aggregation	14
Health Check Resource Paths	18
Health Check and Security	18
Health Check and CDI	18
Custom Health Endpoint	21
Conclusion	22

Introduction

The goal of this guide is to help you as a Java developer make the most of the [MicroProfile](#) Metrics API using the [Payara Server Platform](#). The guide starts by looking at what MicroProfile is, the individual APIs that make up MicroProfile and finally takes a deeper look at the Health API. By the end of this guide, you will be able to integrate MicroProfile into your application and learn how to build reliable and well functioning applications on Payara Server.

What Do I Need to Use This Guide?

You will need a copy of the latest [Payara Server full stream distribution](#) to follow along this guide. The sample code is built using Apache Maven and as such, any IDE that supports Maven is good enough. You should also have a minimum of Java SE 8 installed on your computer.

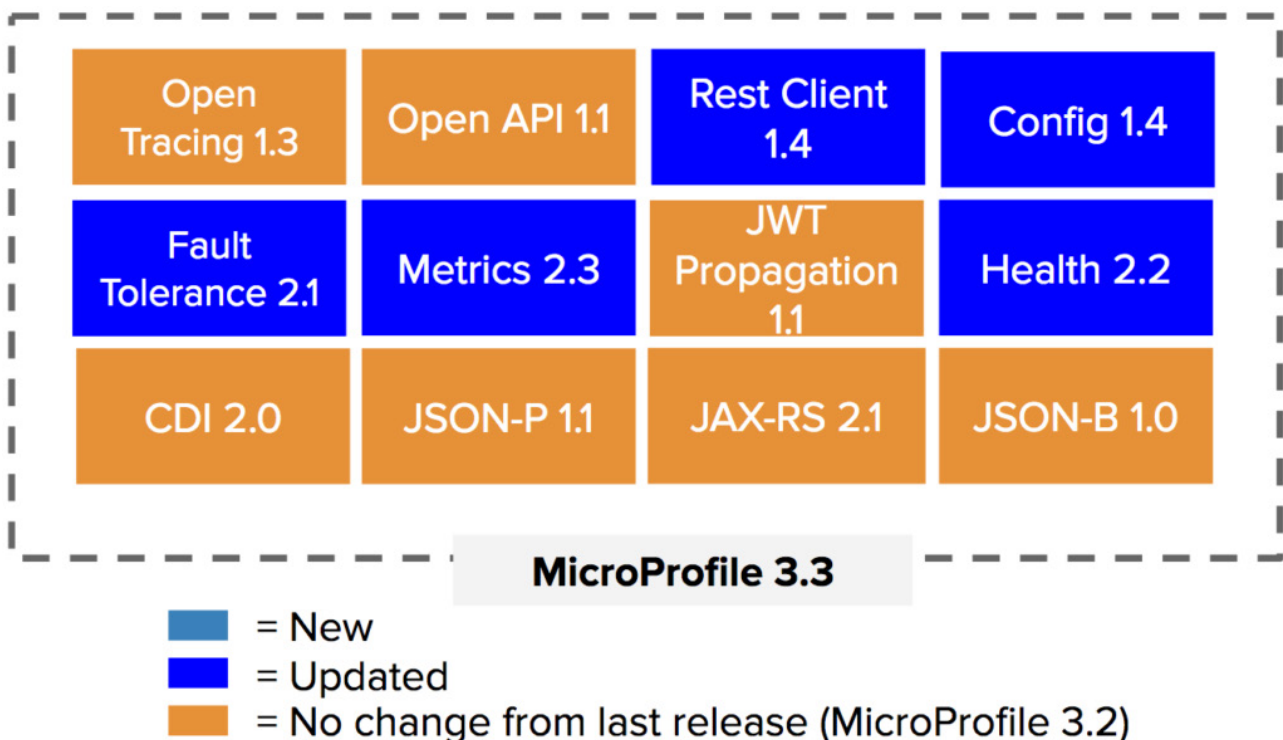
You can also follow along with the information in this guide with our video demo:



What is MicroProfile?

MicroProfile is a [community driven](#) initiative that is a collection of abstract specs that form a complete solution to developing cloud native, Java enterprise microservices. The goal is to create a set of APIs that abstracts you from their implementations so that you can create highly portable microservices across vendors.

The current release is version 3.3 which is an incremental release that includes an update to MicroProfile Config 1.4, MicroProfile Fault Tolerance 2.1, MicroProfile Health 2.2, MicroProfile Metrics 2.3, and MicroProfile Rest Client 1.4. Like its previous version, MicroProfile 3.3 continues to align itself with Java EE 8 as the foundational programming model for the development of Java microservices and consists of 12 different specifications as shown below.



As abstract specifications, the various implementations are free to implement the base specs and add custom features on top. Payara Server is one of the popular implementations of the MicroProfile spec and adds quite a number of custom features on top of the base specs. This guide will walk you through the Payara Server implementation of the MicroProfile Health API. By the end of this guide, you would have learned how to integrate the Health API into your applications to aid in running in automated cluster management cloud environments.

Getting Started with MicroProfile

To get started with the MicroProfile API, you need to include it as a dependency in your project as shown below.

```
1      <dependency>
2          <groupId>org.eclipse.microprofile</groupId>
3          <artifactId>microprofile</artifactId>
4          <version>3.3</version>
5          <type>pom</type>
6          <scope>provided</scope>
7      </dependency>
```

With the MicroProfile API dependency in place, you have access to all the APIs of the project. In our case, the Payara Server will provide the implementation for us.

MicroProfile Health - What is it?

The MicroProfile Health API is an API that lets your applications answer two questions related to their health and readiness to serve requests -

1. Are you live?
2. Are you ready to serve requests?

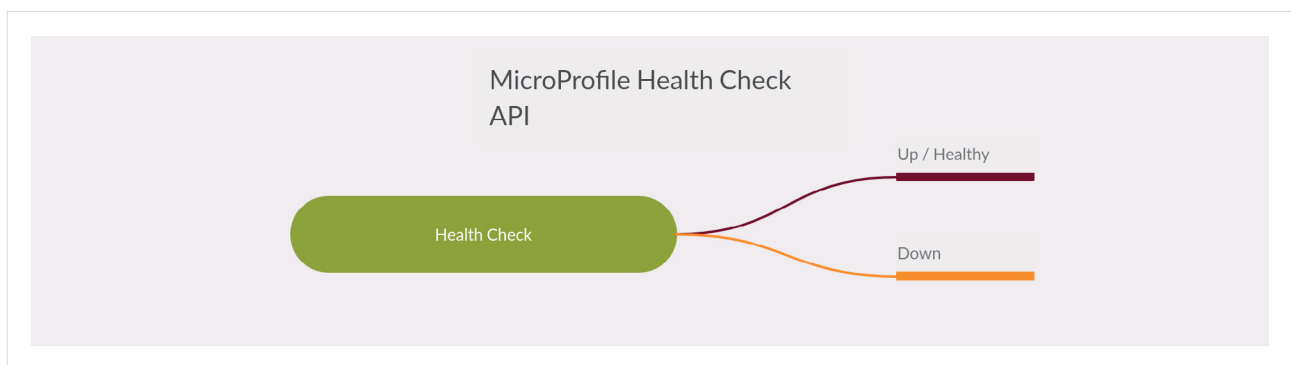
The objective of the Health API is to automate application response to third party queries with regards to the above questions. A typical example of such use case is when your application runs in a [Kubernetes](#) managed cluster. The kubelet will send liveness and readiness probes to your application in a container to check if it's live and/or ready to accept requests.

Health vs Metrics API

If you have heard of the [MicroProfile Metrics API](#), you might be wondering what the difference is between it and the Health API. The Metrics API helps you determine to what extent a given application is working correctly, or wrongly. For example, you could have a method that returns the list of

countries on a given continent. The Metrics API helps you determine - in detail - how that method is performing with metrics like how long it takes to execute, what the median runtime is among other examples.

The Health API on the other hand, is designed primarily for machine to machine communication. It helps your application answer automated queries from cluster management systems that need to know whether your application is alive and ready to service requests. It lets your application give binary answers of yes or no to automated probes in the form of UP or Down as shown in the image below.



MicroProfile Health API in Action

The MicroProfile Health API for you as a developer consists of three annotations (one of which is deprecated), a `HealthCheck` interface, and a `HealthCheckResponseBuilder` for building `HealthCheckResponse` objects. The annotations are

- `@Readiness`
- `@Liveness`
- `@Health`

The `@Readiness` annotation is used to test for the readiness of your application to service requests. The `@Liveness` is used to check if your application is up or down. The `@Health` annotation is deprecated and is only kept for backwards compatibility. You should not use it in new MicroProfile applications. It used to serve the purposes of the `@Liveness` and `@Readiness` annotations. All the annotations are CDI qualifiers.

You might be wondering what the technical differences are between the `@Liveness` and `@Readiness` annotations. According to the Health Spec, the difference between them is only semantic. This means the same health procedure can be annotated with both annotations.

To implement health checking in your application you

- Implement the HealthCheck interface
- Override the call() method to return a HealthCheckResponse object
- Annotate the class with either @Readiness or @Liveness or both.

@Liveness check

The image below shows our first implementation of a liveness health check.

```
1 @Liveness
2 @ApplicationScoped
3 public class LivenessHealthChecker implements HealthCheck {
4     private String time;
5
6     @PostConstruct
7     private void init() {
8         time = LocalDateTime.now().toString();
9     }
10    @Override
11    public HealthCheckResponse call() {
12        return HealthCheckResponse.builder().name("Liveness check").up()
13            .withData("status", "ALIVE")
14            .withData("from", time)
15            .build();
16    }
17 }
```

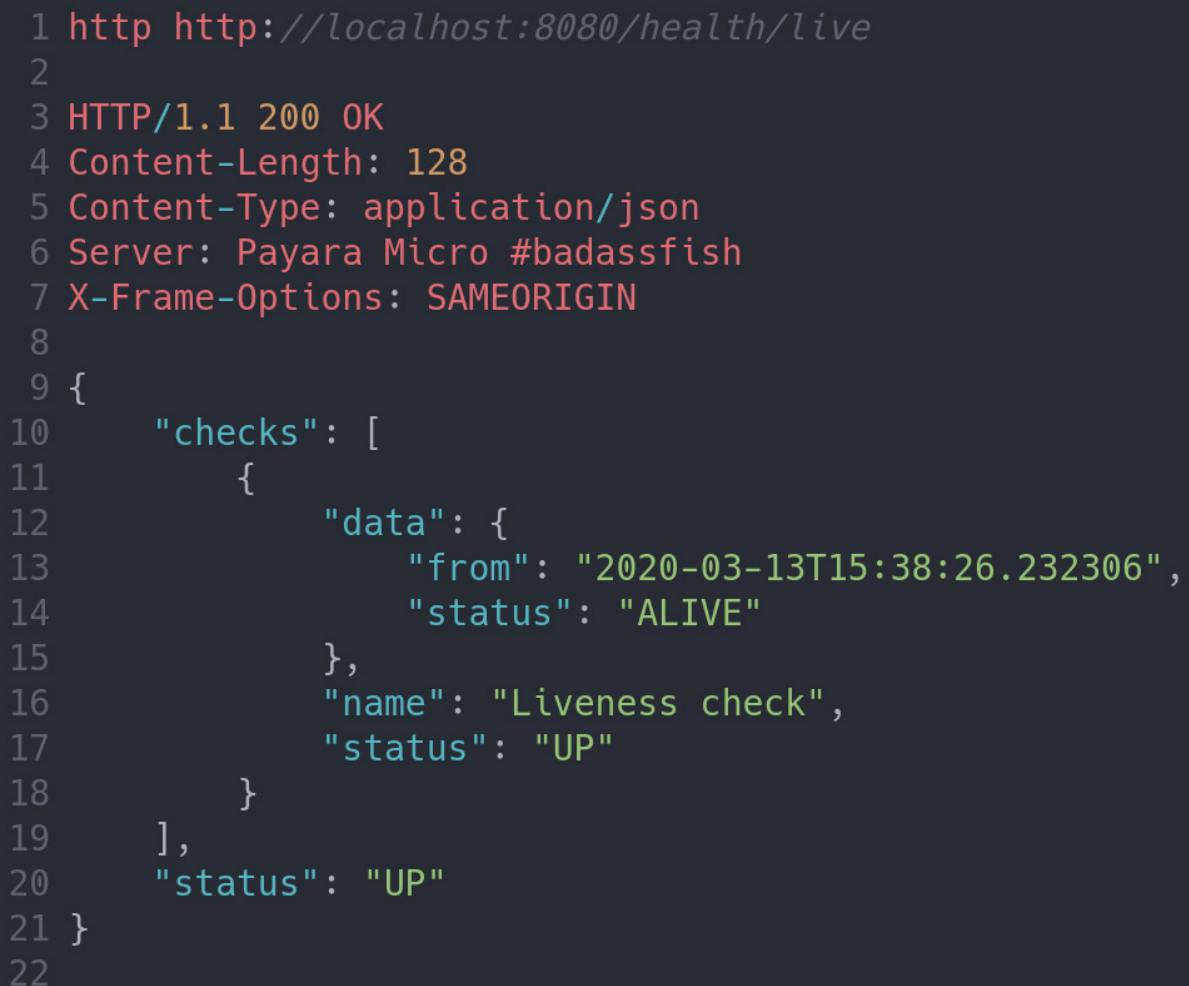
The code snippet above shows a LivenessHealthChecker class that implements HealthCheck on line 3. This class is annotated @Liveness on line 1 and @ApplicationScoped on line 2. MicroProfile Health is integrated with the CDI API so we can use the functionalities of CDI alongside Health. We declare this class as an application scoped singleton. The CDI runtime will create a single instance of this class the first time it is requested and only destroyed when the application shuts down.

On line 4, we declare a variable time of type String. Line 7 declares a private void method init() in which the value of variable time is set to the string representation of the LocalDateTime at the time this method is invoked. The init() is annotated @PostConstruct to tell the CDI runtime to automatically invoke this method whenever this class is constructed and all fields have been initialized. The goal of this method is to store the time this class is created to act as the time the application went

live. Line 11 implements the `call()` method of the `HealthCheck` interface. In this method we use the `HealthCheckResponseBuilder` to build a `HealthCheckResponse` object.

The two most important parts of the `HealthCheck` response object are the name and the `up()` method. The name sets the name of this particular health procedure, and the `up()` method sets the state of the `HealthCheck` response to UP meaning the application is live.

With our first health check in place, let us make an invocation to check on it.



```
1 http http://localhost:8080/health/live
2
3 HTTP/1.1 200 OK
4 Content-Length: 128
5 Content-Type: application/json
6 Server: Payara Micro #badassfish
7 X-Frame-Options: SAMEORIGIN
8
9 {
10     "checks": [
11         {
12             "data": {
13                 "from": "2020-03-13T15:38:26.232306",
14                 "status": "ALIVE"
15             },
16             "name": "Liveness check",
17             "status": "UP"
18         }
19     ],
20     "status": "UP"
21 }
22
```

Line 1 of the above image makes an HTTP GET invocation to the endpoint `http://localhost:8080/health/live`. This endpoint is the default resource for checking the liveness of a `MicroProfile`

application. Line 3 shows the start of the response returned from the server. It shows an HTTP status code 200, implying a “YES” to the question “Are you live?”

Line 5 shows the content return of the response as JSON. You should note that in the invocation to the resource endpoint on line 1, we did not explicitly specify the preferred response type. We got a JSON in return because the Health spec requires implementations to return data in JSON. Line 9-21 shows a JSON object that contains an array of `HealthCheckResponse` objects keyed to “checks.” Line 12-17 shows the `HealthCheckResponse` we constructed using the `HealthCheckResponseBuilder`. Line 16 for instance, shows the name we assigned to the `HealthCheckResponse` object we created - “Liveness check.”

Line 17 shows the status of the `HealthCheck` procedure as UP. This was set by our us when we invoked the `up()` on when building the `HealthCheckResponse` object. Setting the `HealthCheckResponse` state to up is also what set the HTTP status to 200 OK. Line 20 shows the overall status of the health checker as UP.

In the sample code shown so far, we just returned a `HealthCheckResponse` object whose state we manually set to UP. This will always cause an HTTP status 200 to be returned to any probing consumer. In your own application however, you would probably run some logic to verify your application is indeed up. An example could be making some invocation to a database and verifying the result to ascertain that your application is truly live.

The next code snippet shows a situation where we return a `HealthCheckResponse` object with status DOWN.

```
1  @Override
2  public HealthCheckResponse call() {
3      return HealthCheckResponse.builder().name("Liveness check").down()
4          .withData("status", "DOWN")
5          .withData("from", LocalDateTime.now().toString())
6          .build();
7  }
```

The above code snippet shows a `call()` method implementation that returns a `HealthCheckResponse` object whose state is set to DOWN with the invocation of the DOWN method on the response object. A request to the resource method `http://localhost:8080/health/live` shows the following results

```
1 http http://localhost:8080/health/live
2
3 HTTP/1.1 503 Service Unavailable
4 Connection: close
5 Content-Length: 131
6 Content-Type: application/json
7 Server: Payara Micro #badassfish
8 X-Frame-Options: SAMEORIGIN
9
10 {
11   "checks": [
12     {
13       "data": {
14         "from": "2020-03-13T17:28:31.891087",
15         "status": "DOWN"
16       },
17       "name": "Liveness check",
18       "status": "DOWN"
19     }
20   ],
21   "status": "DOWN"
22 }
23
```

Line 1 of the above image shows a HTTP GET request to the health /health/live resource. This time around, the HTTP response code returned is 503, Service Unavailable. Line 21 also shows the status as DOWN. 503 is returned because we set the HealthCheckResponse object returned to a state of DOWN. The Health runtime uses the state of the returned health check object to set the HTTP code which will be used by automated cluster management systems like Kubernetes to decide whether to discard a given node in a cluster or not.

@Readiness check

The process of creating a readiness health procedure is identical to that of creating a liveness procedure as shown below.

```
1 @Readiness
2 @ApplicationScoped
3 public class ReadinessHealthChecker implements HealthCheck {
4
5     @Override
6     public HealthCheckResponse call() {
7         return HealthCheckResponse.named("Readiness check").up()
8             .withData("status", "READY")
9             .withData("from", LocalDateTime.now().toString())
10            .build();
11    }
12 }
13
```

Line 3 of the above code snippet creates a `ReadinessHealthChecker` that implements the `HealthCheck` interface. This class is identical to the `LivenessHealthChecker` in that they are both application scoped singletons as declared on line 2 in above code snippet. Line 1 however, annotates this class with `@Readiness` annotation from the Health API.

Line 6 implements the `call()` method of the `HealthCheck` interface, returning a `HealthCheckResponse` object constructed in its body. Except the name and custom payload of the response object, this health procedure is identical to our first `@Liveness` implementation. This is because the only distinction between `@Liveness` and `@Readiness` is just semantic. However, in practice they are both the same.

An invocation to `http://localhost:8080/health/ready` returns the following data

```
1 http http://localhost:8080/health/ready
2
3 HTTP/1.1 200 OK
4 Content-Length: 129
5 Content-Type: application/json
6 Server: Payara Micro #badassfish
7 X-Frame-Options: SAMEORIGIN
8
9 {
10     "checks": [
11         {
12             "data": {
13                 "from": "2020-03-13T18:01:39.219146",
14                 "status": "READY"
15             },
16             "name": "Readiness check",
17             "status": "UP"
18         }
19     ],
20     "status": "UP"
21 }
22
```

Line 1 makes invocation to the `/health/ready` endpoint and the server returns an HTTP 200 code on line 3. As you can see, the only difference between this response and that of the `@Liveness` health check is in the endpoint that is invoked. `@Readiness` health check is hosted at the `/health/ready` endpoint and `@Liveness` at the `/health/live` endpoint. The structure of the responses is identical.

Just as we were able to have the Health runtime return a 503 error by setting the state of the `HealthCheckResponse` object to `DOWN`, we can do the same for the readiness check. Again, in your application, you will implement some kind of logic to determine the kind of response to return in your `HealthCheck` implementations.

Compose health checks

So far we have looked at the `@Liveness` and `@Readiness` health check annotations. However, you can use both annotations on one `HealthCheck` implementation. In such a case, the same health check will be used to service both a `/health/ready` and `/health/live` requests. The code below shows a `HealthCheck` implementation for both live and readiness checks in one class.

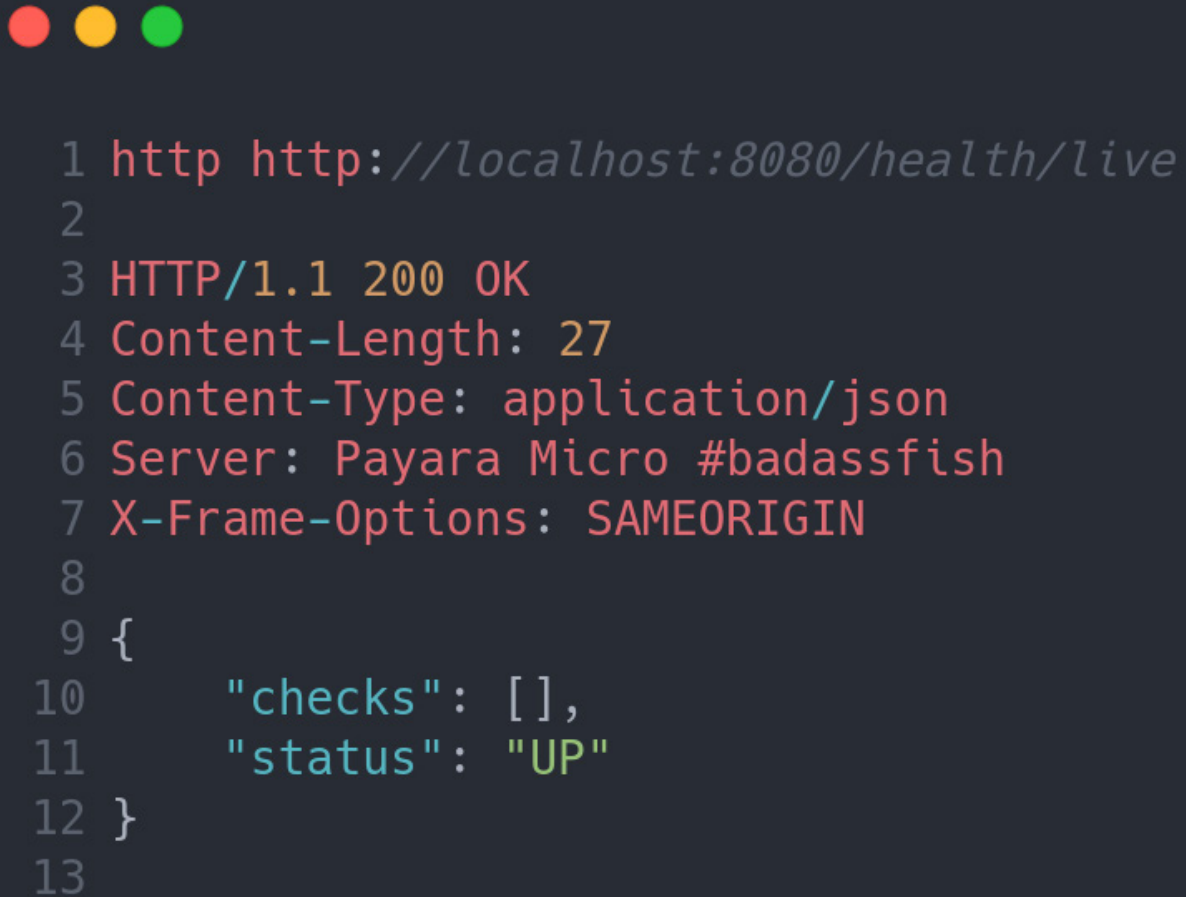
```
1 @Liveness
2 @Readiness
3 @ApplicationScoped
4 public class CompositeHealthCheck implements HealthCheck {
5     @Override
6     public HealthCheckResponse call() {
7         return HealthCheckResponse.named("Composite Health Check").up()
8             .withData("readinessStatus", "READY")
9             .withData("livenessStatus", "LIVE")
10            .withData("from", LocalDateTime.now().toString())
11            .build();
12     }
13 }
14
```

Lines 1 and 2 annotate class `CompositeHealthCheck` with `@Liveness` and `@Readiness`, respectively. This class implements the `HealthCheck` interface and overrides the `call()`, returning a `HealthCheckResponse` object. Note that this `HealthCheck` implementation is identical to all that we have seen so far.

We can make HTTP GET requests to `/health/ready` and `/health/live` resources and the Health runtime will use the same health procedure to service the request. For example, a `@Readiness` request returns the following response

```
1 http http://localhost:8080/health/ready
2
3 HTTP/1.1 200 OK
4 Content-Length: 169
5 Content-Type: application/json
6 Server: Payara Micro #badassfish
7 X-Frame-Options: SAMEORIGIN
8
9 {
10   "checks": [
11     {
12       "data": {
13         "from": "2020-03-13T18:21:48.148427",
14         "livenessStatus": "LIVE",
15         "readinessStatus": "READY"
16       },
17       "name": "Composite Health Check",
18       "status": "UP"
19     }
20   ],
21   "status": "UP"
22 }
23
```

Line 1 makes the `/health/ready` request to our MicroProfile application. The response so far is identical to what we have seen so far. Line 3 shows an HTTP status code of 200 and line 21 shows the status of UP for this health procedure. Now let us make a request to `health/live`.



```
1 http http://localhost:8080/health/live
2
3 HTTP/1.1 200 OK
4 Content-Length: 27
5 Content-Type: application/json
6 Server: Payara Micro #badassfish
7 X-Frame-Options: SAMEORIGIN
8
9 {
10     "checks": [],
11     "status": "UP"
12 }
13
```

Line 1 of the above image shows a request to the `/health/live` endpoint. Similar to the previous response, we get an HTTP 200 status code. However, the body of the response only shows the mandatory status, in this case UP. This shows that when you use composite health checks, the request for which the custom data added to the `HealthCheckResponse` object is returned is dependent upon the implementation, because the spec is silent on it. In this case, the Payara Server returns the data for a `/health/ready` request and only the status for a `/health/live` request.

@Health

The `@Health` annotation is a health check annotation that has been deprecated since [Version 2.0](#) of the Health API. It is available for backward compatibility reasons. You should not use it in your new MicroProfile projects. You might see it in older MicroProfile code in the wild. It functions similar to what we have seen so far. The API creators thought it is better to be more explicit and clearer

with the intent of the checks, among other reasons, and thus the creation of the @Liveness and @Readiness annotations to replace @Health.

Health check aggregation

So far you have seen different permutations of the use of the Health API. The question you may be asking is what happens if you have different procedures for liveness and readiness checks? Such as the code snippets below?

```
1 @Liveness
2 @ApplicationScoped
3 public class LivenessHealthChecker implements HealthCheck {
4     private String time;
5
6     @PostConstruct
7     private void init() {
8         time = LocalDateTime.now().toString();
9     }
10
11     @Override
12     public HealthCheckResponse call() {
13         return HealthCheckResponse.builder().name("Liveness check").up()
14             .withData("status", "ALIVE")
15             .withData("from", time)
16             .build();
17     }
18 }
```

The above image shows our liveness check, as discussed in a previous section.


```
1 @Readiness
2 @ApplicationScoped
3 public class ReadinessHealthChecker implements HealthCheck {
4
5     @Override
6     public HealthCheckResponse call() {
7         return HealthCheckResponse.named("Readiness check").up()
8             .withData("status", "READY")
9             .withData("from", LocalDateTime.now().toString())
10            .build();
11     }
12 }
```

The above code snippet shows a readiness health check procedure. This is the same code we discussed earlier. The only difference now is that we have a separate health procedure for liveness and another for a readiness check. So how do you make one request to get a response from both health procedures? Instead of making separate requests to `/health/live` and `/health/ready`?

To answer your questions, we make a HTTP GET to the base `/health` resource as shown below.

```
1 http http://localhost:8080/health
2
3 HTTP/1.1 200 OK
4 Content-Length: 231
5 Content-Type: application/json
6 Server: Payara Micro #badassfish
7 X-Frame-Options: SAMEORIGIN
8
9 {
10   "checks": [
11     {
12       "data": {
13         "from": "2020-03-14T06:40:04.085722",
14         "status": "READY"
15       },
16       "name": "Readiness check",
17       "status": "UP"
18     },
19     {
20       "data": {
21         "from": "2020-03-14T06:40:04.081363",
22         "status": "ALIVE"
23       },
24       "name": "Liveness check",
25       "status": "UP"
26     }
27   ],
28   "status": "UP"
29 }
30
31
```

Line 1 of the above snippet shows a request to the /health base path. The result is a HTTP 200 status and an overall health status of UP as shown on line 32. The main point of note in the above response is that the HealthCheckResponse objects of both health procedures have been aggregated into 1 response. Line 17 shows the status of the readiness check as UP and line 25 shows that of the liveness check as UP. These give an overall health status of UP. What do you think the overall status would be if 1 of the health procedures had returned a status of down? Let us see a /health request to our application when the readiness procedure returns a status of DOWN while the liveness status returns a status of UP.

```
1 http http://localhost:8080/health
2
3 HTTP/1.1 503 Service Unavailable
4 Connection: close
5 Content-Length: 234
6 Content-Type: application/json
7 Server: Payara Micro #badassfish
8 X-Frame-Options: SAMEORIGIN
9
10 {
11   "checks": [
12     {
13       "data": {
14         "from": "2020-03-14T06:51:09.778921",
15         "status": "ALIVE"
16       },
17       "name": "Liveness check",
18       "status": "UP"
19     },
20     {
21       "data": {
22         "from": "2020-03-14T06:51:09.778365",
23         "status": "down"
24       },
25       "name": "Readiness check",
26       "status": "DOWN"
27     }
28   ],
29   "status": "DOWN"
30 }
31
```

The above request to `/health` returns an HTTP status code of 503, with an overall health status of DOWN on line 29. Line 26 shows the status of the readiness check as DOWN. In aggregation, the Health runtime returned DOWN for the overall status. This shows that the overall status is only set to UP when all the health procedures are healthy. Also note that the spec does not declare any specific order in which multiple health procedures are to be invoked. In our case, the readiness or the liveness procedures can be invoked in any order by the runtime.

It makes sense to set the overall status to UP only when every health procedure returns UP because the goal of the Health API is to enable machine to machine probing. The cluster management system will use the result to make decisions. Imagine if the overall status in the above response had been UP even though the readiness health procedure had it said it was not ready. This would result in the cluster manager routing requests to our application when it had clearly stated that it was not ready to service requests.

Health Check Resource Paths

All through this guide, you have seen requests to `/health/ready`, `/health/live` and `/health`. You might have noted that the MicroProfile Health API endpoint is located on the root/base path of the server, that is `http://localhost:8080/health`. Note that there is no context path after the domain (and port). The MicroProfile Health runtime is hosted on the default “health” path at the base of the server.

What this implies is that you can have more than one application hosted on the server, each with their separate health check procedure. This is why the name of the `HealthCheckResponse` object is mandatory. In complex environments, the cluster manager can be configured to drill down to specific health response objects for their status.

Health Check and Security

So far we have seen health checks to unsecured endpoints. The MicroProfile Health API can be configured to be accessible only to authenticated clients. It can be further configured to allow access to clients with specific roles.

Health Check and CDI

So far we have seen the implementation of health procedures in individual classes. However, you can also simplify health procedures by the use of the CDI producer mechanism because health procedures are CDI beans. Because the Health API is integrated with CDI, the runtime would use your custom health producers to service requests. Let us rewrite our readiness and liveness through CDI producers.

```
1 @ApplicationScoped
2 public class HealthCheckProducer {
3
4     @Produces
5     @Liveness
6     HealthCheck livenessCheck() {
7         return () -> HealthCheckResponse.builder()
8             .up().name("Liveness check through CDI").build();
9     }
10
11     @Produces
12     @Readiness
13     HealthCheck readinessCheck() {
14         return () -> HealthCheckResponse.builder()
15             .up().name("Readiness check through CDI").build();
16     }
17 }
18 }
19 }
```

Line 2 of the above code snippet declares a CDI `@ApplicationScoped` bean `HealthCheckProducer`. Line 4 uses the CDI `@Produces` annotation to declare method (line 6) `livenessCheck()` as a CDI producer. The return type of the method is a `HealthCheck` implementation. Because `HealthCheck` is a functional interface, line 7 uses a lambda expression to construct and return an instance of the `HealthCheckResponse` object. Line 5 marks method `livenessCheck` as a liveness check. Except for the method names and health check annotation (line 12), the `readinessCheck()` method declared on line 13 is identical to the liveness check.

An HTTP invocation to the base `/health` resource returns the following response.

```
1 http http://localhost:8080/health
2
3 HTTP/1.1 200 OK
4 Content-Length: 151
5 Content-Type: application/json
6 Server: Payara Micro #badassfish
7 X-Frame-Options: SAMEORIGIN
8
9 {
10   "checks": [
11     {
12       "data": {},
13       "name": "Readiness check through CDI",
14       "status": "UP"
15     },
16     {
17       "data": {},
18       "name": "Liveness check through CDI",
19       "status": "UP"
20     }
21   ],
22   "status": "UP"
23 }
24
```

Line 1 makes an HTTP request to `http://localhost:8080/health`. The server returns with a response 200 response code on line 3 and overall health status of UP on line 22. The interesting part of this response is that it is from our earlier CDI produced HealthChecks. Line 13 shows the response from the readiness check and line 18 shows the response from the liveness check. Thanks to the tight integration between CDI and the Health API, we can use the CDI producer construct to simplify our code and gain the same result.

Custom Health Endpoint

The [Payara Server](#) full allows you to customize the /health endpoint for health checks. It also allows you to set whether health check endpoints are secured and which roles can access health checks. You can set these options through the server admin interface or through the use of the [Asadmin](#) CLI as shown below.

```
1 asadmin set-microprofile-healthcheck-configuration
2     [--enabled=true|false]
3     [--securityenabled=true|false]
4     [--roles=<role-list>]
5     [--endpoint=<context-root[default:health]>]
```

Summary

In this guide, we have looked at how you can use the MicroProfile Health API to enable your application to respond to probes about its health. You started by taking a look at the difference between the Health API and the Metrics API.

You then started by taking a look at implementing a @Readiness health procedure, followed by @Liveness. You saw how you can combine both checks to create composite health checks. You also learned that @Health annotation is deprecated in favor of the more semantic @Liveness and @Readiness health checks.

You then learned to get a health check response for all health check procedures by making a request to the /health endpoint. Then you learned how you can use the CDI produce mechanism to create compact health procedures. Finally you learned how you can customize certain aspects of the MicroProfile Health API through the Payara Server.

Conclusion

The MicroProfile Health API is a powerful and easy to use API designed to help your applications be able to respond to automated health probes. Developing cloud native enterprise Java applications requires that your application plays nicely with cluster management systems like Kubernetes. The MicroProfile Health API helps achieve that objective.

With the Payara Server Platform fully implementing the latest MicroProfile specification, you are assured of a powerful platform on which to run your mission critical enterprise Java workload.

Should you need further support or info about using or transitioning your enterprise Java workload to the Payara Server Platform, please don't hesitate to get in touch with us. You can always just say hi to us. We would love to hear from you. You can also keep in touch with us on our social media platforms.



sales@payara.fish



+44 207 754 0481



www.payara.fish

Payara Services Ltd 2021 All Rights Reserved. Registered in England and Wales; Registration Number 09998946
Registered Office: Malvern Hills Science Park, Geraldine Road, Malvern, United Kingdom, WR14 3SZ