Chapter 6: MicroProfile Health

Introduction

This chapter provides an in-depth exploration of MicroProfile Health, a critical component for ensuring the reliability and availability of microservices. This specification aims to enhance the observability of microservices in a cloud environment where automatic scaling, failover, and recovery are essential for maintaining service availability and reliability. In this chapter, we will learn about different types of health checks and standard health indicators provided by MicroProfile.

Topics to be covered:

- Overview of MicroProfile Health
- Types of Health Checks
- Standard Health Checks
- Implementing and Exposing Health Checks
- Logging and Reporting Health Checks
- Best Practices for Effective Health Checks

Overview of MicroProfile Health

The MicroProfile Health specification offers a standardized mechanism for microservices to report their health status. In the context of microservices, "health" refers to the ability of a microservice to perform its functions correctly and efficiently. The health check mechanism is crucial for operating microservices in a cloud or containerized environment where automated processes need to make decisions about whether to restart a failing service, reroute traffic away from an unhealthy service, or take other actions to maintain overall system reliability.

Let’s delve into the essentials of MicroProfile Health, its importance, and how it works.

Key Concepts

At its core, the MicroProfile Health specification defines a mechanism for microservices to report their health status via HTTP. These health checks can be used by external systems to verify the operational status of the services. This is crucial in modern cloud environments where automated processes

Health Check

A **health check** is a test that can be used to determine the health of an application or service. This mechanism is implemented via standard HTTP endpoints that respond with the health status of the service. These endpoints are typically exposed at predefined paths, such as `/health`, `/health/live` (for liveness), `/health/ready` (for readiness), and `/health/started` (for startup). Health status is communicated through a simple JSON format, which can be easily interpreted by humans and machines. Applications servers that support MicroProfile may offer built-in mechanisms or simplified configurations to define such health checks.

Types of Health Checks

MicroProfile Health Check defines three main types of health checks, each with its own annotation to indicate the MicroProfile Health runtime about the type of check being performed, allowing it to execute and report health check responses appropriately. These are:

Liveness Checks

Liveness checks help to determine if a microservice is in a state where it can perform its functions correctly. A failing liveness check suggests that the microservice is in a broken state, and the only way to recover might be to restart the microservice. This type of health check is crucial for detecting deadlocks, infinite loops, or any conditions that render the microservice unresponsive or dysfunctional. Liveness checks are annotated with `@Liveness`.

Readiness Checks

Readiness checks are used to determine if a microservice is ready to process requests. If a readiness check fails, it indicates that the microservice should not receive any inbound requests because it's not ready to handle them properly. This can be due to the application still initializing, waiting for dependencies, or any other condition that would prevent it from correctly processing incoming requests. Readiness checks are annotated with `@Readiness`.

Startup Checks

Startup checks are designed for verifying the microservice's health immediately after it has started. This type of check is useful for applications that require additional initialization time or need to perform certain actions before they are ready to serve requests. Including startup checks in the health checking mechanism is crucial because if we hit the liveness probe before the application is fully initialized, it could cause a continuous restart loop. Startup checks provide a mechanism to postpone other health checks until certain startup conditions are fulfilled. This ensures that readiness and liveness probes are not prematurely activated, allowing the microservice adequate time to complete its initialization processes, such as loading configurations, establishing database connections, or performing necessary pre-service tasks.These checks are annotated with `@Startup`.

Exposing Health Checks

Health checks are exposed via HTTP endpoints automatically without additional configuration needed from the developer's side. The runtime environment provides these endpoints:

- `/health`: Aggregates all health check responses.
- `/health/live`: Returns responses from liveness checks.
- `/health/ready`: Returns responses from readiness checks.
- `/health/started`: Returns responses from startup checks.

These endpoints return a JSON object containing the overall status (`UP` or `DOWN`) and individual health check responses, including their names, statuses, and optional data.

Example JSON Response

For example a `LivenessCheck`, if accessed via `/health/live`, the JSON response might look something like this when the service is healthy:

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

If the service is unhealthy, the "status" field would be "DOWN", and additional data might be provided to indicate the cause of the health check failure.

Each type of health check is implemented as a procedure annotated with the respective annotation. Each procedure returns a `HealthCheckResponse` indicating the health status (UP or DOWN) and optionally includes additional details. Implementing these health check types in microservices architecture ensures that services are only used when they are in a healthy state and can correctly process requests. This enhances the overall reliability and maintainability of applications.

Standard Health Check

Applications can implement multiple health checks of each kind. The overall health status reported by the application is a logical AND of all individual health checks. A special endpoint `/health` aggregates the results from all three types of checks.

Implementing and Exposing Health Check

To implement health checks for microservices using MicroProfile Health, you would generally follow a pattern to define health check procedures that align with the services' operational characteristics. The Health Check API allows us to expose information about the health of our application. This information can be used by load balancers and other tools to determine if an application is healthy.

The `HealthCheck` interface

The `HealthCheck` functional interface uses CDI beans with annotations (`@Liveness`, `@Readiness`, and, `@Startup`) to mark a class as a health checker for liveness, readiness and startup. They are automatically discovered and registered by the runtime. Implementations of this interface are expected to be provided by applications.

The Health Check API defines a contract for health check implementations. A health check is a Java class that implements the HealthCheck functional interface:

```
package org.eclipse.microprofile.health;

@FunctionalInterface
public interface HealthCheck {
  HealthCheckResponse call();
}
```

You can check out the actual code here - https://github.com/eclipse/microprofile-health/blob/main/api/src/main/java/org/eclipse/microprofile/health/HealthCheck.java

The `HealthCheckResponse` class

The `HealthCheckResponse` class is used to represent the result of a health check invocation. It contains information about the health check, such as name, state (up or down), and data that

The `call()` method of HealthCheck interface is used to perform the actual health check and return a `HealthCheckResponse` object:

```
package org.eclipse.microprofile.health;

public class HealthCheckResponse {

    // the name of the health check.
    private final String name;

    // the outcome of the health check
    private final Status status;

    // information about the health check.
```

```
    private final Optional<Map<String, Object>> data;

    // Status enum definition
    public enum Status {
        UP, DOWN
    }

   // Getters
    public String getName() {
        return name;
    }

    public Status getStatus() {
        return status;
    }

    public Optional<Map<String, Object>> getData() {
        return data;
    }

}
```

The provided code snippet offers a conceptual and simplified implementation of the `HealthCheckResponse` class to illustrate how health check responses can be structured within the MicroProfile Health framework. To view the actual `HealthCheckResponse` class source code, please visit: https://github.com/eclipse/microprofile-health/blob/main/api/src/main/java/org/eclipse/microprofile/health/HealthCheckResponse.java

The `HealthCheckResponseBuilder` class

The `HealthCheckResponseBuilder` abstract class provides a fluent API for constructing instances of HealthCheckResponse. This means you can chain method calls to set various properties of the response in a single statement, improving code readability and maintainability.

```
package org.eclipse.microprofile.health;

public abstract class HealthCheckResponseBuilder {

    // Sets the name of the health check response.
    public abstract HealthCheckResponseBuilder name(String name) {
        this.name = name;
    }

    // Sets the status of the health check to UP
    public abstract HealthCheckResponseBuilder up();

    // Sets the status of the health check to DOWN
    public abstract HealthCheckResponseBuilder down();
```

```
    // Adds additional string data to the health check response
    public HealthCheckResponseBuilder withData(String key, String value);

    // Adds additional numeric data to the health check response
    public HealthCheckResponseBuilder withData(String key, long value);

    // Sets the status of the health check response
    public abstract HealthCheckResponseBuilder status(boolean up);

    // Builds and returns the HealthCheckResponse instance
    public abstract HealthCheckResponse build();

}
```

The above code snippet offers a conceptual and simplified definition of the `HealthCheckResponseBuilder` abstract class to illustrate how health check responses can be structured within the MicroProfile Health framework. For the actual HealthCheckResponseBuilder abstract class source code, please visit: https://github.com/eclipse/microprofile-health/blob/main/api/src/main/java/org/eclipse/microprofile/health/HealthCheckResponseBuilder.java

Steps for Implementing Health Checks

Below are the steps for implementing Health Checks for each of the microservices:

Add MicroProfile Health Dependency: To utilize MicroProfile Health in a Java project, include the MicroProfile Health API dependency in your `pom.xml` or `build.gradle` file.

For maven, add:

```
<dependency>
  <groupId>org.eclipse.microprofile.health</groupId>
  <artifactId>microprofile-health-api</artifactId>
  <version>4.0.1</version>
</dependency>
```

For gradle, add:

```
implementation 'org.eclipse.microprofile.health:microprofile-health-api:4.0.1'
```

Note: When implementing MicroProfile Health checks, including the MicroProfile Health API dependency in your project is not enough. You need an actual implementation on the classpath. This could be a MicroProfile-compatible server runtime such as Open Liberty, Quarkus, Payara Micro, or WildFly. Without an

implementation present at runtime, the application will not be able to execute health checks.

The health information can be used by other tools to help keep our application running well.

Implementing Health Checks

Health checks in MicroProfile are implemented as CDI beans that implement the `HealthCheck` interface. Each health check procedure is a method that returns a `HealthCheckResponse`. You can define different types of health checks (readiness, liveness, and startup) depending on the type of check by annotating the health check class with `@Readiness`, `@Liveness`, or `@Startup`. These methods return a `HealthCheckResponse` object, which includes the health check status (UP or DOWN) and additional metadata about the health check.

Readiness Check:

```
package io.microprofile.tutorial.store.product.health;

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

import io.microprofile.tutorial.store.product.entity.Product;
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;

@Readiness
@ApplicationScoped
public class ProductServiceHealthCheck implements HealthCheck {

    @PersistenceContext
    EntityManager entityManager;

    @Override
    public HealthCheckResponse call() {
        if (isDatabaseConnectionHealthy()) {
            return HealthCheckResponse.named("ProductServiceReadinessCheck")
                    .up()
                    .build();
        } else {
            return HealthCheckResponse.named("ProductServiceReadinessCheck")
                    .down()
                    .build();
        }

    }
```

```
    private boolean isDatabaseConnectionHealthy(){
        try {
            // Perform a lightweight query to check the database connection
            entityManager.find(Product.class, 1L);
            return true;
        } catch (Exception e) {
            System.err.println("Database connection is not healthy: " +
e.getMessage());
            return false;
        }
    }
}
```

Liveness Check:

```
package io.microprofile.tutorial.store.product.health;

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

import jakarta.enterprise.context.ApplicationScoped;

@Liveness
@ApplicationScoped
public class ProductServiceLivenessCheck implements HealthCheck {

   @Override
   public HealthCheckResponse call() {
       Runtime runtime = Runtime.getRuntime();
       long maxMemory = runtime.maxMemory(); // Maximum amount of memory the JVM
will attempt to use
       long allocatedMemory = runtime.totalMemory(); // Total memory currently
allocated to the JVM
       long freeMemory = runtime.freeMemory(); // Amount of free memory within
the allocated memory
       long usedMemory = allocatedMemory - freeMemory; // Actual memory used
       long availableMemory = maxMemory - usedMemory; // Total available memory

       long threshold = 100 * 1024 * 1024; // threshold: 100MB

      // Including diagnostic data in the response
       HealthCheckResponseBuilder responseBuilder =
HealthCheckResponse.named("systemResourcesLiveness")
            .withData("FreeMemory", freeMemory)
            .withData("MaxMemory", maxMemory)
            .withData("AllocatedMemory", allocatedMemory)
```

```
            .withData("UsedMemory", usedMemory)
            .withData("AvailableMemory", availableMemory);

        if (availableMemory > threshold) {
            // The system is considered live
            responseBuilder = responseBuilder.up();
        } else {
            // The system is not live.
            responseBuilder = responseBuilder.down();
        }

        return responseBuilder.build();
    }
}
```

The above code uses the `HealthCheckResponseBuilder` to construct the response. Depending on the outcome of `checkDatabaseConnection()`, the health check response is marked either "up" or "down", and relevant data is added to the response using `.withData(key, value)`. This approach allows for rich, descriptive health check responses that can convey detailed status information, not just binary up/down states.

Startup Check:

```
package io.microprofile.tutorial.store.product.health;

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

import jakarta.ejb.Startup;
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.persistence.EntityManagerFactory;
import jakarta.persistence.PersistenceUnit;

@Startup
@ApplicationScoped
public class ProductServiceStartupCheck implements HealthCheck{

    @PersistenceUnit
    private EntityManagerFactory emf;

    @Override
    public HealthCheckResponse call() {
        if (emf != null && emf.isOpen()) {
            return HealthCheckResponse.up("ProductServiceStartupCheck");
        } else {
            return HealthCheckResponse.down("ProductServiceStartupCheck");
        }
    }
```

```
}
```

Integration with CDI

The specification also emphasizes the importance of integrating health checks with the application’s Context and Dependency Injection (CDI) context, enabling health check procedures to be automatically discovered and invoked by the runtime.

MicroProfile Health thus provides a robust and standardized way to implement health checks, facilitating the management and orchestration of microservices in a cloud environment.

Deployment and Access

Once defined, these health check procedures are automatically discovered and invoked by the MicroProfile Health runtime. They are accessible through standardized HTTP endpoints provided by MicroProfile Health (/health, /health/live, /health/ready, /health/started) and can be used by orchestration tools (like Kubernetes) or monitoring systems to manage and monitor the health of your microservices.

This approach allows you to tailor health checks to the operational specifics of each microservice, providing a robust mechanism for observing and managing your application’s health in a cloud-native environment.

Kubernetes Probe Configuration

Integrating MicroProfile Health checks with Kubernetes probes allows you to leverage Kubernetes' native capabilities to manage the lifecycle of your containers based on their current health status. Specifically, you can map Liveness, Readiness, and Startup probes in Kubernetes to the corresponding health check types defined by the MicroProfile Health specification.

Here's a basic overview of how each type of MicroProfile Health check maps to Kubernetes probes:

- **Liveness Probes:** Determine if a container is running and healthy. If a liveness probe fails, Kubernetes will kill the container and create a new one based on the restart policy.
- **Readiness Probes:** Determine if a container is ready to serve traffic. If a readiness probe fails, Kubernetes will stop sending traffic to that container until it passes again.
- **Startup Probes:** Determine if a container application has started. These are useful for applications that have a long startup time to prevent them from being killed by Kubernetes before they are up and running.

To configure these probes in your Kubernetes deployment, you can use the livenessProbe, readinessProbe, and startupProbe fields in your container specification. Here's an example of how you might define a readiness probe in your Kubernetes deployment configuration, that utilizes a MicroProfile Health endpoint:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-microprofile-application
spec:
  replicas: 2
  selector:
    matchLabels:
      app: my-microprofile-application
  template:
    metadata:
      labels:
        app: my-microprofile-application
    spec:
      containers:
      - name: my-microprofile-application
        image: myimage:v1
        ports:
        - containerPort: 8080
        readinessProbe:
          httpGet:
            path: /health/ready
            port: 8080
          initialDelaySeconds: 15
          timeoutSeconds: 2
          periodSeconds: 5
          failureThreshold: 3
```

In this example, the `readinessProbe` is configured to make an HTTP GET request to the `/health/ready` endpoint, which is the default endpoint provided by MicroProfile Health for readiness checks. Similarly, you can configure `livenessProbe` and `startupProbe` by specifying `/health/live` and `/health/startup` endpoints respectively.

It's important to adjust the `initialDelaySeconds`, `timeoutSeconds`, `periodSeconds`, and `failureThreshold` according to the specifics of your application to ensure that Kubernetes accurately reflects the state of your container based on its health checks.

Best Practices for Effective Health Checks

Here are some best practices for implementing and utilizing health checks effectively:

1. **Clearly Define Health Check Types:** Use readiness, liveness, and startup checks appropriately to reflect the state of your microservices. This helps in

accurately signaling the service's ability to handle traffic and its current operational state.
2. **Implement Meaningful Health Checks:** Ensure that your health checks meaningfully reflect the operational aspects they are intended to monitor. Avoid trivial checks that do not accurately represent the service's health.
3. **Utilize Health Check Responses:** Make effective use of the health check responses, including the UP/DOWN status and additional metadata. This information can be valuable for logging and reporting on the health state of your services.
4. **Secure Health Check Endpoints:** Consider the security of your health check endpoints, especially if they expose sensitive details about the application's state.
5. **Monitor Health Check Performance:** Health checks should be lightweight and not introduce significant overhead. Monitor the performance of your health checks and optimize as needed to prevent impacting the application's performance.
6. **Logging Health Check Results:** Implementing logging within your health check procedures can provide insights into the health status over time. Log entries can be made when health check statuses change or when significant health-related events occur.
7. **Reporting and Alerting:** Based on logged health check results, implement reporting mechanisms to visualize the health over time and set up alerting for when health checks fail. This could be integrated with existing monitoring and alerting tools.

By following these best practices, you can effectively implement and expose health checks in your MicroProfile applications, improving observability and reliability, especially in cloud-native environments.

Summary

This chapter provided a comprehensive overview of MicroProfile Health, emphasizing its critical role in enhancing the observability and reliability of microservices within cloud environments. Key topics included an introduction to the MicroProfile Health specification, detailed explanations of health check types (liveness, readiness, and startup checks), and guidance on implementing, exposing, and effectively utilizing these health checks.

The essence of MicroProfile Health lies in its standardized mechanism for microservices to report health status via HTTP endpoints, facilitating automated decision-making processes like scaling, failover, and recovery in cloud or containerized environments. The specification defines three primary types of health checks: liveness, readiness, and startup checks, each designed to assess different aspects of a microservice's operational status.

Implementing health checks involves creating procedures annotated with the respective health check annotations. These procedures return a HealthCheckResponse indicating the service's health status (UP or DOWN). These checks are automatically exposed via predefined HTTP endpoints, allowing easy integration with orchestration tools like Kubernetes.

The chapter also touched on best practices for effective health checks, including defining meaningful checks, utilizing health check responses, handling failures gracefully, and securing health check endpoints. In conclusion, MicroProfile Health offers a robust framework for monitoring and managing the health of microservices, ensuring that services remain reliable and available in dynamic cloud environments. By following the guidelines and best practices outlined in this chapter, developers can effectively implement and leverage health checks to maintain the overall health of their applications.