Universidad Central del Ecuador

Programación Distribuida

Nombre: Steven Ortiz

Fecha: 28/02/2023

Semestre: 8vo

MicroProfile Fault Tolerance

Fault Tolerance es parte del conjunto de especificaciones de MicroProfile. Esta API define principalmente anotaciones que mejoran la sólidez de la aplicación al brindar soporte para manejar convenientemente las condiciones de error (Fallas) que pueden ocurrir en las aplicaciones del mundo real. Los ejemplos incluyen reinicios de servicios, retrasos en la red, inestabilidades temporales de la infraestructura, etc.

Con MicroProfile Fault Tolerance, los desarrolladores pueden agregar Tolerancia a fallos a sus aplicaciones mediante la adición de anotaciones a sus metodos, estos mecanismos se utilizan para controlar y manejar las interacciones de la aplicación con servicios remotos o externos que pueden no estar disponibles, tener tiempos de respuesta lentos a experimentar errores. Esto permite a los desarrolladores identificar y solucionar problemas en la aplicación y mejorar la calidad del servicio.

Las anotaciones disponibles son @Retry, @Fallback, @CircuitBreaker, @Bulkhead, @Asynchronous y @Timeout.

@Retry: permite configurar una cantidad máxima de intentos de ejecución de un método antes de que falle. Los atributos de anotación se pueden usar para controlar el número de reintentos, la demora entre reintentos y las excepciones en las que se reintenta o cancela

Ejemplo

```
@Retry (
    maxRetries = 3,
    delay = 0,
    delayUnit = ChronoUnit.MILLIS,
    maxDuration = 180000,
    durationUnit = chronoUnit.MILLIS,
    jitter = 200,
    jitterDelayUnit = ChronoUnit.MILLIS,
    retryOn = {Exception.class},
    abortOn = {}
)
```

@Fallback: Establece un controlador que se ejecutará al encontrar un error de invocación. Un controlador es una clase que implementa FallbackHandler<T> o simplemente un método simple en la misma clase. Se utilizan propiedades adicionales para controlar las condiciones en las que se llama a estos controladores.

Ejemplo

```
@Fallback (
    value = DEFAULT.class,
    fallbackMethod = "",
    applyOn = {Throwable.class},
    skipOn = {}
)
```

@CircuitBreaker: Monitorea el número de errores en los métodos y, si se supera un umbral predefinido, interrumpe temporalmente el flujo del método para evitar una mayor propagación de errores. Un interruptor automático puede estar closed, open o half-open. En estado closed, un interruptor de circuito ejecutará la lógica normalmente. En estado open, un disyuntor evitará la ejecución de la lógica se ha visto fallar. Finalmente, en estado half-open, un interruptor automático permitirá ejecuciones de prueba en un intento de cambiar su estado interno a closed. Los otros parámetros de anotación se utilizan para controlar cómo se activan estas transiciones de estado.

Ejemplo

```
@CircuitBreaker (
    failOn = {Throwable.class},
    skipOn = {},
    delay = 5000,
    delayUnit = ChronoUnit.MILLIS,
    requestVolumeThreshold = 20,
    failureRation = .50,
    successThreshold = 1
)
```

@Bulkhead: limita la cantidad de recursos que se pueden asignar a un método para evitar que se agoten en caso de fallas simultáneas. Se utiliza una cola para estacionar tareas en espera de ejecución después de que se haya alcanzado el límite. Una cola solo está activa cuando las invocaciones son @Asynchronous

Ejemplo

```
@Bulkhead (
    value = 10,
    waitingTaskQueue = 10
)
```

@Asynchronous: Ejecuta una invocación de forma asincrónica sin bloquear el subproceso de llamada. El método anotado debe devolver Future o CompletionStage. Por lo general, se usa para evitar bloquear el subproceso de llamada en E/S a en un cálculo de ejecución prolongada.

@Timeout: establece un tiempo límite para la ejecución de un método. Si el método tarda más en completarse que el tiempo límite establecido se cancela la operación. El valor predeterminado es 1 segundo.

Ejemplo

```
@Timeout (
    value = 1000,
    unit = chronoUnit.MILLIS
)
```

La utilización de técnicas de Fault Tolerance es fundamental en sistemas distribuidos para garantizar su disponibilidad y confiabilidad, especialmente cuando se trata de aplicaciones empresariales críticas que requieren alta disponibilidad.

La incorporación de mecanismos de Fault Tolerance en una aplicación ayuda a mitigar los efetos negativos de posibles fallas y errores, permitiendo que la aplicación siga funcionando con un rendimiento aceptable, incluso en situaciones de fallos parciales o temporales de algunos componentes del sistema.

Además, la incorporación de técnicas de Fault Tolerance permite a las empresas ofrecer un mejor nivel de servicio a sos clientes y usuarios ya que les gurantiza que la aplicación siempre estará disponible y que sus datos estarán seguros y protegidos. También permite a las empresas ahorrar costos, ya que evita tiempos de inactividad prolongados y reducción de pérdidas financieras debido a la indisponibilidad del sistema.

Es importante destacar que esta herramienta está disponible en varios lenguajes de programación, incluyendo Java, Kotlin y Go. Además, la especificación de MicroProFile Fault Tolerance es independiente del proveedor, lo que significa que se puede utilizar en cualquier plataforma que compla con la especificación. Esto brinda a los desarrolladores una mayor fexibilidad en la elección de herramientas y plataformas para sus proyectos.
En resumen, la utilización de MicroProfile Fault Tolerance puede mejorar significativamente la tolerancia a fallos en las aplicaciones, proporcionando una mayor confiabilidad y estabilidad.

Dependencias

Para agregar las dependencias de MicroProfile Fault Tolerance en un proyecto, se pueden utilizar las herramientas de gestión de dependencias de Maven a Gradle.

Maven

```
<dependency>
    <groupId>io.helidon.microprofile</groupId>
    <artifactId>helidon-microprofile-fault-tolerance</artifactId>
</dependency>
```

Gradle

```
implementation 'io.helidon.microprofile:helidon-microprofile-fault-tolerance'
implementation 'io.helidon.microprofile.metrics:helidon-microprofile-metrics'
```

Es importante de que las versiones especificadas de las dependencias sean compatibles con la versión de MicroProfile utilizadas en el proyecto

Al implementar Fault Tolerance en un sistema, es importante tener en cuenta los siguientes aspectos:

1: Identificación de puntos de fallas: Es importante identificar los puntos críticos en el sistema que pueden fallar, ya sea por sobrecarga, falta de recursos, errores de comexión, entre otros. Una vez identificados, se pueden aplicar estrategias de tolerancia a fallas específicas.

2: Selección de estrategias de tolerancia a fallas: existen diversas estrategias de tolerancias a fallas que se pueden aplicar en función de las necesidades del sistema y del tipo de fallas que se espera. Es importante seleccionar la estrategia adecuada para cada punto crítico.

3.- Monitoreo y registro de fallas: Es importante contar con un mecanismo de monitoreo que permita detectar y registrar las fallas que se presentan en el sistema. Esto permitirá identificar patrones y tendencias que ayuden a mejorar las estrategias de tolerancia a fallas.

4.- Pruebas y simulaciones: Es importante realizar pruebas y simulaciones para evaluar la efectividad de las estrategias de tolerancia a fallas y para identificar posibles puntos críticos que puedan haberse pasado por alto.

5.- Mantenimiento y actualización: Las estrategias de tolerancia a fallas deben ser actualizadas y mantenidas constantemente para asegurar su efectividad y adaptabilidad a posibles cambios en el sistema o en el entorno.