Microprofile Fault Tolerance

Emily Jiang

1.1-RC4, May 18, 2018

Table of Contents

Specification: Microprofile Fault Tolerance

Version: 1.1-RC4

Status: Draft

Release: May 18, 2018

Chapter 1. Architecture

This specification defines an easy to use and flexible system for building resilient applications.

1.1. Rational

It is increasingly important to build fault tolerant microservices. Fault tolerance is about leveraging different strategies to guide the execution and result of some logic. Retry policies, bulkheads, and circuit breakers are popular concepts in this area. They dictate whether and when executions should take place, and fallbacks offer an alternative result when an execution does not complete successfully.

As mentioned above, the Fault Tolerance specification is to focus on the following aspects:

- `Timeout`: Define a duration for timeout
- `Retry`: Define a criteria on when to retry
- `Fallback`: provide an alternative solution for a failed execution.
- `CircuitBreaker`: offer a way of fail fast by automatically failing execution to prevent the system overloading and indefinite wait or timeout by the clients.
- `Bulkhead`: isolate failures in part of the system while the rest part of the system can still function.

The main design is to separate execution logic from execution. The execution can be configured with fault tolerance policies, such as RetryPolicy, fallback, Bulkhead and CircuitBreaker.

Hystrix and Failsafe are two popular libraries for handling failures. This specification is to define a standard API and approach for applications to follow in order to achieve the fault tolerance.

This specification introduces the following interceptor bindings:

- `Timeout`
- `Retry`
- `Fallback`
- `CircuitBreaker`
- `Bulkhead`
- `Asynchronous`

Refer to Interceptor Specification for more information on interceptor bindings.

Chapter 2. Relationship to other specifications

This specification defines a set of annotations to be used by classes or methods. The annotations are interceptor bindings. Therefore, this specification depends on the Java Interceptors and Contexts and Dependency Injection specifications define in Java EE platform.

2.1. Relationship to Contexts and Dependency Injection

The Contexts and Dependency Injection (CDI) specification defines a powerful component model to enable loosely coupled architecture design. This specification explores the rich SPI provided by CDI to register an interceptor so that the Fault Tolerance policies can be applied to the method invocation.

2.2. Relationship to Java Interceptors

The Java Interceptors specification defines the basic programming model and semantics for interceptors. This specification uses the typesafe interceptor bindings. The annotations `@Asynchronous`, `@Bulkhead`, `@CircuitBreaker`, `@Fallback`, `@Retry` and `@Timeout` are all interceptor bindings.

These annotations may be bound at the class level or method level. The annotations adhere to the interceptor binding rules defined by Java Interceptors specification.

For instance, if the annotation is bound to the class level, it applies to all business methods of the class. If the component class declares or inherits a class level interceptor binding, it must not be declared final, or have any static, private, or final methods. If a non-static, non-private method of a component class declares a method level interceptor binding, neither the method nor the component class may be declared final.

Since this specification depends on CDI and interceptors specifications, fault tolerance operations have the following restrictions:

- Fault tolerance interceptors bindings must applied on a bean class or bean class method otherwise it is ignored,
- invocation must be business method invocation as defined in CDI specification.
- if a method and its containing class don't have any fault tolerance interceptor binding, it won't be considered as a fault tolerance operation.

2.3. Relationship to MicroProfile Config

The MicroProfile config specification defines a flexible config model to enable microservice configurable and achieve the strict separation of config from code. All parameters on the annotations/interceptor bindings are config properties. They can be configured externally either

via other predefined config sources (e.g. environment variables, system properties or other sources). For an instance, the `maxRetries` parameter on the `@Retry` annotation is a configuration property. It can be configured externally.

Chapter 3. Execution

Use interceptor and annotation to specify the execution and policy configuration. An annotation of Asynchronous has to be specified for any asynchronous calls. Otherwise, synchronous execution is assumed.

Chapter 4. Asynchronous

`Asynchronous` means the execution of the client request will be on a separate thread. This thread should have the correct security context or naming context associated with.

4.1. Asynchronous Usage

A method or a class can be annotated with `@Asynchronous`, which means the method or the methods under the class will be invoked by a separate thread. The method annotated with `@Asynchronous` must return a `Future`, otherwise, `FaultToleranceDefinitionException` occurs.

```
@Asynchronous
public Future<Connection> serviceA() {
   Connection conn = null;
   counterForInvokingServiceA++;
   conn = connectionService();
   return CompletableFuture.completedFuture(conn);
}
```

The above code-snippet means the method serviceA applies the `Asynchronous` policy, which means the invocation will be done by a different thread.

The `@Asynchronous` annotation can be used together with `@Timeout`, `@Fallback`, `@Bulkhead` and `@Retry`. Method invocation will occur in a different thread.

Chapter 5. Timeout

`Timeout` prevents from the execution from waiting forever. It is recommended that a microservice invocation should have timeout associated with.

5.1. Timeout Usage

A method or a class can be annotated with `@Timeout`, which means the method or the methods under the class will have Timeout policy applied.

```
@Timeout(400) // timeout is 400ms
public Connection serviceA() {
   Connection conn = null;
   counterForInvokingServiceA++;
   conn = connectionService();
   return conn;
}
```

The above code-snippet means the method serviceA applies the `Timeout` policy, which is to fail the execution if the execution takes more than 400ms to complete even if it successfully returns.

When a timeout occurs, A `TimeoutException` must be thrown. The `@Timeout` annotation can be used together with `@Fallback`, `@CircuitBreaker`, `@Asynchronous`, `@Bulkhead` and `@Retry`.

When `@Timeout` is used without `@Asynchronous`, the current thread will be interrupted with a call to Thread.interrupt() on reaching the specified timeout duration. The interruption will only work in certain scenarios. The interruption will not work for the following situations:

- The thread is blocked on blocking I/O (database, file read/write), an exception is thrown only in case of waiting for a NIO channel
- The thread isn't waiting (CPU intensive task) and isn't checking for being interrupted
- The thread will catch the interrupted exception (with a general catch block) and will just continue processing, ignoring the interrupt

In the above situations, it is impossible to suspend the execution. The execution thread will finish its process. If the execution takes longer than the specified timeout, the `TimeoutException` will be thrown and the execution result will be discarded.

When `@Timeout` is used with `@Asynchronous`, then a separate thread will be spawned to perform the work in the annotated method or methods, while a `Future` is returned on the main thread. If the work on the spawned thread does timeout, then a get() call to the `Future` on the main thread should throw an `ExecutionException` that wraps a fault tolerance `TimeoutException`.

A `@Fallback` can be specified and it will be invoked if the `TimeoutException` is thrown. If `@Timeout` is used together with `@Retry`, the `TimoutException` will trigger the retry. When `@Timeout` is used with `@CircuitBreaker` and if a `TimeoutException` occurs, the failure will contribute towards the circuit open.

Chapter 6. Retry Policy

In order to recover from a brief network glitch, `@Retry` can be used to invoke the same operation again. The `Retry` policy allows to configure :

- `maxRetries`: the maximum retries
- `delay`: delays between each retry
- `delayUnit`: the delay unit
- `maxDuration`: maximum duration to perform the retry for.
- `durationUnit`: duration unit
- `jitter`: the random vary of retry delays
- `jitterDelayUnit`: the jitter unit
- `retryOn`: specify the failures to retry on
- `abortOn`: specify the failures to abort on

6.1. Retry usage

`@Retry` can be applied to the class or method level. If applied to a class, it means the all methods in the class will have the `@Retry` policy applied. If applied to a method, it means that method will have `@Retry` policy applied. If the `@Retry` policy applied on a class level and on a method level within that class, the method level `@Retry` will override the class-level `@Retry` policy for that particular method.

```
    /**
     * The configured the max retries is 90 but the max duration is 1000ms.
     * Once the duration is reached, no more retries should be performed,
     * even through it has not reached the max retries.
     */
    @Retry(maxRetries = 90, maxDuration= 1000)
    public void serviceB() {
        writingService();
    }

    /**
    * There should be 0-800ms (jitter is -400ms - 400ms) delays
    * between each invocation.
    * there should be at least 4 retries but no more than 10 retries.
    */
    @Retry(delay = 400, maxDuration= 3200, jitter= 400, maxRetries = 10)
    public Connection serviceA() {
        return connectionService();
    }

    /**
    * Sets retry condition, which means Retry will be performed on
    * IOException.
    */
    @Retry(retryOn = {IOException.class})
    public void serviceB() {
        writingService();
    }
```

The `@Retry` annotation can be used together with `@Fallback`, `@CircuitBreaker`, `@Asynchronous`, `@Bulkhead` and `@Timeout`. A `@Fallback` can be specified and it will be invoked if the `@Retry` still fails.

If `@Retry` is used with `@Timeout`, a retry will only be triggered if `TimeoutException` or one of its super classes are defined in the `retryOn` attribute of the `@Retry`.

Chapter 7. Fallback

Fallbacks are invoked once a Retry or CircuitBreaker has failed enough times.

For a Retry, Fallback is handled any time the Retry would exceed its maximum number of attempts.

For a CircuitBreaker, it is invoked any time the method invocation fails. When the Circuit is open, the Fallback is always invoked.

7.1. Fallback usage

A method can be annotated with `@Fallback`, which means the method will have Fallback policy applied. There are two ways to specify fallback:

- Specify a FallbackHandler class
- Specify the fallbackMethod

7.1.1. Specify a FallbackHandler class

If a FallbackHandler is registered for a method returning a different type than the FallbackHandler would return, then the container should treat as an error and deployment fails.

FallbackHandlers are meant to be CDI managed, and should follow the life cycle of the scope of the bean.

```
    @Retry(maxRetries = 1)
    @Fallback(StringFallbackHandler.class)
    public String serviceA() {
        counterForInvokingServiceA++;
        return nameService();
    }
```

The above code snippet means when the method failed and retry reaches its maximum retry, the fallback operation will be performed. The method `StringFallbackHandler.handle(ExecutionContext context)` will be invoked. The return type of `StringFallbackHandler.handle(ExecutionContext context)` must be `String`. Otherwise, the `FaultToleranceDefinitionException` exception will be thrown.

7.1.2. Specify the fallbackMethod

This is used if the `fallbackMethod` is on the same class as the method to call fall back.

```
    @Retry(maxRetries = 2)
    @Fallback(fallbackMethod= "fallbackForServiceB")
    public String serviceB() {
        counterForInvokingServiceB++;
       return nameService();
    }

    private String fallbackForServiceB() {
        return "myFallback";
    }
```

The above code snippet means when the method failed and retry reaches its maximum retry, the fallback operation will be performed. The method `fallbackForServiceB` will be invoked. The return type of `fallbackForServiceB` must be `String` and the argument list for `fallbackForServiceB` must be the same as `ServiceB`. Otherwise, the `FaultToleranceDefinitionException` exception will be thrown.

The parameter `value` and `fallbackMethod` on `@Fallback` cannot be specified at the same time. Otherwise, the `FaultToleranceDefinitionException` exception will be thrown.

The fallback should be triggered when an exception occurs. For instance, `BulkheadException`, `CircuitBreakerOpenException`, `TimeoutException` should trigger the fallback.

Chapter 8. Circuit Breaker

A Circuit Breaker prevents repeated failures, so that dysfunctional services or APIs fail fast. There are three circuit states:

- Closed: In normal operation, the circuit is closed. If a failure occurs, the Circuit Breaker records the event. In closed state the requestVolumeThreshold and failureRatio parameters may be configured in order to specify the conditions under which the breaker will transition the circuit to open. If the failure conditions are met, the circuit will be opened.
- Open: When the circuit is open, calls to the service operating under the circuit breaker will fail immediately. A delay may be configured for the Circuit Breaker. After the specified delay, the circuit transitions to half-open state.
- Half-open: In half-open state, trial executions of the service are allowed. By default one trial call to the service is permitted. If the call fails, the circuit will return to open state. The successThreshold parameter allows the configuration of the number of trial executions that must succeed before the circuit can be closed. After the specified number of successful executions, the circuit will be closed. If a failure occurs before the successThreshold is reached the circuit will transition to open.

Note that circuit state transitions will reset the Circuit Breaker's records. For example, when the circuit transitions to closed a new rolling failure window is created with the configured requestVolumeThreshold and failureRatio.

8.1. Circuit Breaker Usage

A method or a class can be annotated with `@CircuitBreaker`, which means the method or the methods under the class will have CircuitBreaker policy applied.

```
@CircuitBreaker(successThreshold = 10, requestVolumeThreshold = 4, failureRatio=0.75,
delay = 1000)
public Connection serviceA() {
   Connection conn = null;
   counterForInvokingServiceA++;
   conn = connectionService();
   return conn;
}
```

The above code-snippet means the method serviceA applies the `CircuitBreaker` policy, which is to open the circuit once 3 (4x0.75) failures occur among the rolling window of 4 consecutive invocation. The circuit will stay open for 1000ms and then back to half open. After 10 consecutive successful invocations, the circuit will be back to close again.

When a circuit is open, A `CircuitBreakerOpenException` must be thrown. The `@CircuitBreaker` annotation can be used together with `@Timeout`, `@Fallback`, `@Asynchronous`, `@Bulkhead` and `@Retry`. A `@Fallback` can be specified and it will be invoked if the `CircuitBreakerOpenException` is thrown. `@Timeout` can be configured to fail an operation if it takes longer than the `Timeout` value to complete.

Chapter 9. Bulkhead

The `Bulkhead` pattern is to prevent faults in one part of the system from cascading to the entire system, which might bring down the whole system. The implementation is to limit the number of concurrent requests accessing to an instance. Therefore, `Bulkhead` pattern is only effective when applying `@Bulkhead` to a component that can be accessed from multiple contexts.

9.1. Bulkhead Usage

A method or class can be annotated with `@Bulkhead`, which means the method or the methods under the class will have Bulkhead policy applied correspondingly. There are two different approaches to the bulkhead: thread pool isolation and semaphore isolation. When `@Bulkhead` is used with `@Asynchronous`, the thread pool isolation approach will be used. If `@Bulkhead` is used without `@Asynchronous`, the semaphore isolation approach will be used. The thread pool approach allows to configure the maximum concurrent requests together with the waiting queue size. The semaphore approach only allows the concurrent number of requests configuration.

9.1.1. Semaphore style Bulkhead

The below code-snippet means the method serviceA applies the `Bulkhead` policy, which is semaphore approach, limiting the maximum concurrent requests to 5.

```
@Bulkhead(5) // maximum 5 concurrent requests allowed
public Connection serviceA() {
   Connection conn = null;
   counterForInvokingServiceA++;
   conn = connectionService();
   return conn;
}
```

When using the semaphore approach, on reaching maximum request counter, the extra request will fail with `BulkheadException`.

9.1.2. Thread pool style Bulkhead

The below code-snippet means the method serviceA applies the `Bulkhead` policy, which is thread pool approach, limiting the maximum concurrent requests to 5 and the waiting queue size to 8.

```
// maximum 5 concurrent requests allowed, maximum 8 requests allowed in the waiting
queue
@Asynchronous
@Bulkhead(value = 5, waitingTaskQueue = 8)
public Future<Connection> serviceA() {
   Connection conn = null;
   counterForInvokingServiceA++;
   conn = connectionService();
   return CompletableFuture.completedFuture(conn);
}
```

When using the thread pool approach, when a request cannot be added to the waiting queue, `BulkheadException` will be thrown.

The `@Bulkhead` annotation can be used together with `@Fallback`, `@CircuitBreaker`, `@Asynchronous`, `@Timeout` and `@Retry`. If a `@Fallback` is specified, it will be invoked if the `BulkheadException` is thrown.

Chapter 10. Integration with Microprofile Metrics

When Microprofile Fault Tolerance and Microprofile Metrics are used together, metrics are automatically added for each of the methods annotated with a `@Retry`, `@Timeout`, `@CircuitBreaker`, `@Bulkhead` or `@Fallback` annotation.

10.1. Names

The automatically added metrics follow a consistent pattern which includes the fully qualified name of the annotated method. In the tables below, the placeholder `<name>` should be replaced by the fully qualified method name.

If two methods have the same fully qualified name then the metrics for those methods will be combined. The result of this combination is non-portable and may vary between implementations. For portable behavior, monitored methods in the same class should have unique names.

10.2. Metrics added for `@Retry`, `@Timeout`, `@CircuitBreaker`, `@Bulkhead` and `@Fallback`

Implementations must ensure that if any of these annotations are present on a method, then the following metrics are added only once for that method.

Name	Type	Unit	Description
`ft.<name>.invocations.total`	Counter	None	The number of times the method was called
`ft.<name>.invocations.failed.total`	Counter	None	The number of times the method was called and, after all Fault Tolerance actions had been processed, threw a `Throwable`

10.3. Metrics added for `@Retry`

Name	Type	Unit	Description
`ft.<name>.retry.callsSucceededNotRetried.total`	Counter	None	The number of times the method was called and succeeded without retrying
`ft.<name>.retry.callsSucceededRetried.total`	Counter	None	The number of times the method was called and succeeded after retrying at least once
`ft.<name>.retry.callsFailed.total`	Counter	None	The number of times the method was called and ultimately failed after retrying

Name	Type	Unit	Description
ft.<name>.retry.retries.total	Counter	None	The total number of times the method was retried

Note that the sum of ft.<name>.retry.callsSucceededNotRetried.total, ft.<name>.retry.callsSucceededRetried.total and ft.<name>.retry.callsFailed.total will give the total number of calls for which the Retry logic was run.

10.4. Metrics added for @Timeout

Name	Type	Unit	Description
ft.<name>.timeout.executionDuration	Histogram	Nanoseconds	Histogram of execution times for the method
ft.<name>.timeout.callsTimedOut.total	Counter	None	The number of times the method timed out
ft.<name>.timeout.callsNotTimedOut.total	Counter	None	The number of times the method completed without timing out

Note that the sum of ft.<name>.timeout.callsTimedOut.total and ft.<name>.timeout.callsNotTimedOut.total will give the total number of calls for which the Timeout logic was run. This may be larger than the total number of invocations of the method it's also annotated with @Retry because the Timeout logic is applied to each Retry attempt, not to the whole method invocation time.

10.5. Metrics added for @CircuitBreaker

Name	Type	Unit	Description
ft.<name>.circuitbreaker.callsSucceeded.total	Counter	None	Number of calls allowed to run by the circuit breaker that returned successfully
ft.<name>.circuitbreaker.callsFailed.total	Counter	None	Number of calls allowed to run by the circuit breaker that then failed
ft.<name>.circuitbreaker.callsPrevented.total	Counter	None	Number of calls prevented from running by an open circuit breaker
ft.<name>.circuitbreaker.open.total	Gauge<Long>	Nanoseconds	Amount of time the circuit breaker has spent in open state
ft.<name>.circuitbreaker.halfOpen.total	Gauge<Long>	Nanoseconds	Amount of time the circuit breaker has spent in half-open state
ft.<name>.circuitbreaker.closed.total	Gauge<Long>	Nanoseconds	Amount of time the circuit breaker has spent in closed state
ft.<name>.circuitbreaker.opened.total	Counter	None	Number of times the circuit breaker has moved from closed state to open state

Note that the sum of ft.<name>.circuitbreaker.callsSucceeded.total,

`ft.<name>.circuitbreaker.callsFailed.total` and `ft.<name>.circuitbreaker.callsPrevented.total` will give the total number of calls for which the Bulkhead logic was run.

Similarly, the sum of `ft.<name>.circuitbreaker.open.total`, `ft.<name>.circuitbreaker.halfOpen.total` and `ft.<name>.circuitbreaker.closed.total` will give the total time that the circuit breaker has been active for.

10.6. Metrics added for `@Bulkhead`

Name	Type	Unit	Description
`ft.<name>.bulkhead.concurrentExecutions`	Gauge<Long>	None	Number of currently running executions
`ft.<name>.bulkhead.callsAccepted.total`	Counter	None	Number of calls accepted by the bulkhead
`ft.<name>.bulkhead.callsRejected.total`	Counter	None	Number of calls rejected by the bulkhead
`ft.<name>.bulkhead.executionDuration`	Histogram	Nanoseconds	Histogram of method execution times. This does not include any time spent waiting in the bulkhead queue.
`ft.<name>.bulkhead.waitingQueue.population`*	Gauge<Long>	None	Number of executions currently waiting in the queue
`ft.<name>.bulkhead.waiting.duration`*	Histogram	Nanoseconds	Histogram of the time executions spend waiting in the queue

*Only added if the method is also annotated with `@Asynchronous`

10.7. Metrics added for `@Fallback`

Name	Type	Unit	Description
`ft.<name>.fallback.calls.total`	Gauge<Long>	None	Number of times the fallback handler or method was called

Chapter 11. Notes

Metrics added by this specification will appear as application metrics for the application which uses the Fault Tolerance annotations.

Future versions of this specification may change the definitions of the metrics which are added to take advantage of enhancements in the MicroProfile Metrics specification.

If more than one annotation is applied to a method, the metrics associated with each annotation will be added for that method.

All of the counters are monotonic and count the number of events which occurred since the application started. It is expected that these counters will be sampled regularly by monitoring software which is then able to compute deltas or moving averages from the gathered samples.

11.1. Annotation Example

```
package com.exmaple;

@Timeout(1000)
public class MyClass {

    @Retry
    public void doWork() {
        // work
    }

}
```

This class would result in the following metrics being added.

- `ft.com.example.MyClass.doWork.invocations.total`
- `ft.com.example.MyClass.doWork.invocations.failed`
- `ft.com.example.MyClass.doWork.retry.callsSucceededNotRetried.total`
- `ft.com.example.MyClass.doWork.retry.callsSucceededRetried.total`
- `ft.com.example.MyClass.doWork.retry.callsFailed.total`
- `ft.com.example.MyClass.doWork.retry.retries.total`
- `ft.com.example.MyClass.doWork.timeout.executionDuration`
- `ft.com.example.MyClass.doWork.timeout.callsTimedOut.total`
- `ft.com.example.MyClass.doWork.timeout.callsNotTimedOut.total`

Now imagine the `doWork()` method is called and the invocation goes like this: * On the first attempt, the invocation takes more than 1000ms and times out * On the second attempt, something goes wrong and the method throws an `IOException` * On the third attempt, the method returns successfully and the result of this attempt is returned to the user

After this sequence, the value of these metrics would be as follows:

`ft.com.example.MyClass.doWork.invocations.total = 1`
The method has been called once.

`ft.com.example.MyClass.doWork.invocations.failed = 0`
No exceptions were propagated back to the caller.

`ft.com.example.MyClass.doWork.retry.callsSucceededNotRetried.total = 0`
`ft.com.example.MyClass.doWork.retry.callsSucceededRetried.total = 1`
`ft.com.example.MyClass.doWork.retry.callsFailed.total = 0`
Only one call was made, and it succeeded after some retries.

`ft.com.example.MyClass.doWork.retry.retries.total = 2`
Two retries were made during the invocation.

`ft.com.example.MyClass.doWork.timeout.executionDuration`
The `Histogram` will have been updated with the length of time taken for each attempt. It will show a count of `3` and will have calculated averages and percentiles from the execution times.

`ft.com.example.MyClass.doWork.timeout.callsTimedOut.total = 1`
One of the attempts timed out.

`ft.com.example.MyClass.doWork.timeout.callsNotTimedOut.total = 2`
Two of the attempts did not time out.

Chapter 12. Fault Tolerance configuration

This specification defines the programming model to build a resilient microservice. Microservices developed using this feature are guaranteed to be resilient despite of running environments. In some service mash platform, e.g. Istio, has its own Fault Tolerance policy. The operation team might want to use the platform Fault Tolerance. In order to fulfil the requirement, MicroProfile Fault Tolerance provides a capability to have its resilient functionalities except `fallback` disabled. The reason `fallback` is special as the `fallback` business logic can only be defined by microservices not by any other platforms.

Set the config property of `MP_Fault_Tolerance_NonFallback_Enabled` with the value of `false` means the Fault Tolerance is disabled, except `@Fallback`. If the property is absent or with the value of `true`, it means that MicroProfile Fault Tolerance is enabled if any annotations are specified. For more information about how to set config properties, refer to MicroProfile Config specification.

In order to prevent from any unexpected behaviours, the property `MP_Fault_Tolerance_NonFallback_Enabled` will only be read on application starting. Any dynamic changes afterwards will be ignored until the application restarting.

12.1. Config Fault Tolerance parameters

This specification defines the annotations: `@Asynchronous`, `@Bulkhead`, `@CircuitBreaker`, `@Fallback`, `@Retry` and `@Timeout`. Each annotation except `@Asynchronous` has parameters. All of the parameters are configurable. The value of each parameter can be overridden individually or globally.

- Override individual parameters The annotation parameters can be overwritten via config properties in the naming convention of `<classname>/<methodname>/<annotation>/<parameter>`.

The `<classname>` and `<methodname>` must be the class name and method name where the annotation is declared upon.

In the following code snippet, in order to override the `maxRetries` for serviceB invocation to 100, set the config property `com.acme.test.MyClient/serviceB/Retry/maxRetries=100` Similarly to override the `maxDuration` for ServiceA, set the config property

```
com.acme.test.MyClient/serviceA/Retry/maxDuration=3000
```

- Override parameters globally

If the parameters for a particular annotation need to be configured with the same value for a particular class, use the config property `<classname>/<annotation>/<parameter>` for configuration. For an instance, use the following config property to override all `maxRetries` for `Retry` specified on the class `MyClient` to 100.

```
com.acme.test.MyClient/Retry/maxRetries=100
```

Sometimes, the parameters need to be configured with the same value for the whole microservice. For an instance, all `Timeout` needs to be set to `100ms`. It can be cumbersome to override each occurrence of `Timeout`. In this circumstance, the config property `<annotation>/<parameter>` overrides the corresponding parameter value for the specified annotation. For instance, in order to override

the `maxRetries` for the `Retry` to be `30`, specify the config property `Retry/maxRetries=30`.

When multiple config properties are present, the property `<classname>/<methodname>/<annotation>/<parameter>` takes precedence over `<classname>/<annotation>/<parameter>`, which is followed by `<annotation>/<parameter>`.

The override just changes the value of the corresponding parameter specified in the microservice and nothing more. If no annotation matches the specified parameter, the property will be ignored. For instance, if the annotation `Retry` is specified on the class level for the class `com.acme.ClassA`, which has method methodB, the config property `com.acme.ClassA/methodB/Retry/maxRetries` will be ignored. In order to override the property, the config property `com.acme.ClassA/Retry/maxRetries` or `Retry/maxRetries` needs to be specified.

```
package come.acme.test;
public class MyClient{
    /**
     * The configured the max retries is 90 but the max duration is 1000ms.
     * Once the duration is reached, no more retries should be performed,
     * even through it has not reached the max retries.
     */
    @Retry(maxRetries = 90, maxDuration= 1000)
    public void serviceB() {
        writingService();
    }

    /**
    * There should be 0-800ms (jitter is -400ms - 400ms) delays
    * between each invocation.
    * there should be at least 4 retries but no more than 10 retries.
    */
    @Retry(delay = 400, maxDuration= 3200, jitter= 400, maxRetries = 10)
    public Connection serviceA() {
        return connectionService();
    }

    /**
    * Sets retry condition, which means Retry will be performed on
    * IOException.
    */
    @Retry(retryOn = {IOException.class})
    public void serviceB() {
        writingService();
    }

}
```

If an annotation is not present, the configured properties are ignored. For instance, the property `com.acme.ClassA/methodB/Retry/maxRetries` will be ignored if `@Retry` annotation is not specified on the `methodB` of `com.acme.ClassA`. Similarly, the property `com.acme.ClassA/Retry/maxRetries` will be

ignored if `@Retry` annotation is not specified on the class `com.acme.ClassA` as a class-level annotation.

12.2. Disabled Fault Tolerance policies with configuration

Fault Tolerance policies can be disabled with configuration at method level, class level or globally for all deployment. If multiple configurations are specified, method-level configuration overrides class-level configuration, which then overrides global configuration. e.g. * `com.acme.test.MyClient/methodA/CircuitBreaker/enabled=false` * `com.acme.test.MyClient/CircuitBreaker/enabled=true` * `CircuitBreaker/enabled=false` For the above scenario, all occurrences of `CircuitBreaker` for the application are disabled except for those on the class `com.acme.test.MyClient`. All occurrences of `CircuitBreaker` on `com.acme.test.MyClient` are enabled, except for the one on `methodA` which is disabled.

Each policy can be disabled by using its annotation name.

- Disabling a policy at Method level

A policy can be disabled at method level with the following config property and value:

`<classname>/<methodname>/<annotation>/enabled=false`

For instance the following config will disable circuit breaker policy on `methodA` of `com.acme.test.MyClient` class:

`com.acme.test.MyClient/methodA/CircuitBreaker/enabled=false`

Policy will be disabled even if the policy is also defined at class level

- Disabling a policy at class level

A policy can be disabled at class level with the following config property and value:

`<classname>/<annotation>/enabled=false`

For instance the following config will disable fallback policy on `com.acme.test.MyClient` class:

`com.acme.test.MyClient/Fallback/enabled=false`

Policy will be disabled on all class methods even if a method has the policy.

- Disabling a policy globally

A policy can be disabled globally with the following config property and value:

`<annotation>/enabled=false`

For instance the following config will disable bulkhead policy globally:

`Bulkhead/enabled=false`

Policy will be disabled everywhere ignoring existing policy annotations on methods and classes.

If the above configurations patterns are used with a value other than `true` or `false` (i.e. `<classname>/<methodname>/<annotation>/enabled=whatever`) non-portable behaviour results.

When the above property is used together with the property `MP_Fault_Tolerance_NonFallback_Enabled`, the propoerty `MP_Fault_Tolerance_NonFallback_Enabled` has the lowest priority. `MP_Fault_Tolerance_NonFallback_Enabled=true Bulkhead/enabled=true` All other annotations except `Fallback` and `Bulkhead` will not be enabled.