Microprofile Fault Tolerance

Emily Jiang, Antoine Sabot-Durant, Andrew Rouse

2.0.4-RC1, April 08, 2020
11.9. Annotation Example ................................................................. 23
12. Fault Tolerance configuration .................................................. 25
  12.1. Config Fault Tolerance parameters .................................... 25
  12.2. Disable a group of Fault Tolerance annotations on the global level ..... 26
  12.3. Disabled individual Fault Tolerance policy ............................. 27
  12.4. Configuring Metrics Integration .......................................... 28
  13.1. API/SPI Changes ................................................................. 29
  13.2. Functional Changes .......................................................... 29
  13.3. Specification Changes ....................................................... 29
  13.4. Other changes ................................................................. 29
  14.1. API/SPI Changes ................................................................. 30
  14.2. Functional Changes .......................................................... 30
  14.3. Specification Changes ....................................................... 30
  14.4. Other changes ................................................................. 30
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.
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.

2.4. Relationship to MicroProfile Metrics

The MicroProfile Metrics specification provides a way to monitor microservice invocations. It is also important to find out how Fault Tolerance policies are operating, e.g.

- When Retry is used, it is useful to know how many times a method was called and succeeded after retrying at least once.
- When Timeout is used, you would like to know how many times the method timed out.

Because of this requirement, 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.
Chapter 3. Fault Tolerance Interceptor(s)

The implementor of the MicroProfile Fault Tolerance specification must provide one or more Fault Tolerance interceptors. The interceptor(s) provide the functionality for Fault Tolerance annotations. The interceptor(s) will be called if one or more Fault Tolerance annotations are specified. For instance, a Fault Tolerance interceptor will retry the specified operation if the `Retry` annotation is specified on that operation. The base priority of the lowest priority Fault Tolerance interceptor is `Priority.PLATFORM_AFTER+10`, which is `4010`. If more than one Fault Tolerance interceptor is provided by an implementation, the priority number taken by Fault Tolerance interceptor(s) should be in the range of `[base, base+40]`.

The Fault Tolerance interceptor base priority can be configured via MicroProfile Config with the property name of `mp.fault.tolerance.interceptor.priority`. The property value will only be read at application startup. Any subsequent value changes will not take effect until the application restarts.

A method, annotated with any of the Fault Tolerance interceptor bindings, may also be annotated with other interceptor bindings. The bound interceptors will be invoked in ascending order of interceptor priority, as specified by Interceptor Specification. If the application interceptors are enabled via beans.xml, the interceptors enabled via beans.xml will be invoked after the Fault Tolerance interceptor. For more details, refer to Interceptor ordering in CDI specification.

For instance, in the following example, `MyLogInterceptor` will be invoked first, followed by a Fault Tolerance interceptor that does `Retry` capability, and then `MyPrintInterceptor`.

```java
@Retry
@MyLog
@MyPrint
public void myInvoke() {
    //do something
}

@Priority(3000)
@MyLog
public class MyLogInterceptor{
    //do logging
}

@Priority(5000)
@MyPrint
public class MyPrintInterceptor{
    //do printing
}
```
Chapter 4. Execution

Use interceptor and annotation to specify the execution and policy configuration. The annotation *Asynchronous* has to be specified for any asynchronous calls. Otherwise, synchronous execution is assumed.
Chapter 5. 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 it.

5.1. 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 or a CompletionStage from the java.util.concurrent package. Otherwise, a FaultToleranceDefinitionException occurs.

When a method annotated with @Asynchronous is invoked, it immediately returns a Future or CompletionStage. The execution of the any remaining interceptors and the method body will then take place on a separate thread.

- Until the execution has finished, the Future or CompletionStage which was returned will be incomplete.
- If the execution throws an exception, the Future or CompletionStage will be completed with that exception. (I.e. Future.get() will throw an ExecutionException which wraps the thrown exception and any functions passed to CompletionStage.exceptionally() will run.)
- If the execution completes normally and returns a value, the Future or CompletionStage will then delegate to the returned value.

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

The above code-snippet means that the Asynchronous policy is applied to the serviceA method, which means that a call to serviceA will return a CompletionStage immediately and that execution of the method body will be done on a different thread.

5.2. Interactions with other Fault Tolerance annotations

The @Asynchronous annotation can be used together with @Timeout, @Fallback, @Bulkhead, @CircuitBreaker and @Retry. In this case, the method invocation and any fault tolerance processing will occur in a different thread. The returned Future or CompletionStage will be completed with the final result once all other Fault Tolerance processing has been completed. However, the two different return types have some differences.
5.2.1. Interactions when returning a Future

If a method returns a Future, the other Fault Tolerance annotations are applied only around the method invocation regardless of whether the returned Future completes exceptionally or not. In more detail:

- If the method invocation throws an exception, this will trigger other specified Fault Tolerance policies to be applied.
- If the method returns a Future, then the method call is considered to be successful, which will not trigger other Fault Tolerance policies to be applied even if specified.

In the following example, the Retry will not be triggered as the method invocation returns normally.

```java
@Asynchronous
@Retry
public Future<Connection> serviceA() {
    CompletableFuture<U> future = new CompletableFuture<>();
    future.completeExceptionally(new RuntimeException("Failure"));
    return future;
}
```

5.2.2. Interactions when returning a CompletionStage

If the method returns CompletionStage, the other specified Fault Tolerance annotations will be triggered if either an exception is thrown from the method call or the returned CompletionStage completes exceptionally. In more detail:

- If the method invocation throws an exception, this will trigger other specified Fault Tolerance policies to be applied.
- If the method returns a CompletionStage, then the method call is considered to be successful only if the CompletionStage completes successfully. If an exceptionally completed CompletionStage is returned, or if an incomplete CompletionStage is returned which later completes exceptionally, then this will cause other specified Fault Tolerance policies to be applied.

In the following example, the Retry will be triggered as the returned CompletionStage completes exceptionally.

```java
@Asynchronous
@Retry
public CompletionStage<Connection> serviceA() {
    CompletableFuture<U> future = new CompletableFuture<>();
    future.completeExceptionally(new RuntimeException("Failure"));
    return future;
}
```

The above behaviour makes it easier to apply Fault Tolerance logic around a CompletionStage which
was returned by another component, e.g. applying `@Asynchronous`, `@Retry` and `@Timeout` to a JAX-RS client call.

It is apparent that when using `@Asynchronous`, it is much more desirable to specify the return type `CompletionStage` over `Future` to maximise the usage of Fault Tolerance.

### 5.3. Exception Handling

A call to a method annotated with `@Asynchronous` will never throw an exception directly. Instead, the returned `Future` or `CompletionStage` will report that its task failed with the exception which would have been thrown.

For example, if `@Asynchronous` is used with `@Bulkhead` on a method which returns a `Future` and the bulkhead queue is full when the method is called, the method will return a `Future` where calling `isDone()` returns `true` and calling `get()` will throw an `ExecutionException` which wraps a `BulkheadException`. 
Chapter 6. Timeout

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

6.1. 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.

```java
@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.

If a timeout occurs, the thread interrupted status must be cleared when the method returns.

If @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 or CompletionStage is returned on the main thread. If the work on the spawned thread does time out, then a get() call to the Future on the main thread will throw an Exception that wraps a fault tolerance TimeoutException.

If @Timeout is used with @Fallback then the fallback method or handler will be invoked if a TimeoutException is thrown (unless the exception is handled by another fault tolerance component).
If @Timeout is used with @Retry, a TimeoutException may trigger a retry, depending on the values of retryOn and abortOn of the @Retry annotation. The timeout is restarted for each retry. If @Asynchronous is also used and the retry is the result of a TimeoutException, the retry starts after any delay period, even if the original attempt is still running.

If @Timeout is used with @CircuitBreaker, a TimeoutException may be counted as a failure by the circuit breaker and contribute towards opening the circuit, depending on the value of failOn on the @CircuitBreaker annotation.

If @Timeout is used with @Bulkhead and @Asynchronous, the execution time measured by @Timeout should be the period starting when the execution is added to the Bulkhead queue, until the execution completes. If a timeout occurs while the execution is still in the queue, it must be removed from the queue and must not be started. If a timeout occurs while the method is executing, the thread where the method is executing must be interrupted but the method must still count as a running concurrent request for the Bulkhead until it actually returns.
Chapter 7. 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

7.1. 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.
@Retry(maxRetries = 90, maxDuration= 1000)
public void serviceB()
{
    writingService();
}

@Retry(delay = 400, maxDuration= 3200, jitter= 400, maxRetries = 10)
public Connection serviceA()
{
    return connectionService();
}

@Retry(delay = 0, maxDuration= 3200, jitter= 400, maxRetries = 10)
public Connection serviceA()
{
    return connectionService();
}

@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 method still fails after any retries have been run.

If @Retry is used with @Asynchronous and a retry is required, the new retry attempt may be run on the same thread as the previous attempt, or on a different thread.
Chapter 8. Fallback

A Fallback method is invoked if a method annotated with @Fallback completes exceptionally.

The Fallback annotation can be used on its own or together with other Fault Tolerance annotations. The fallback is invoked if an exception would be thrown after all other Fault Tolerance processing has taken place.

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.

8.1. 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

8.1.1. 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.

```java
@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.

8.1.2. Specify the fallbackMethod

This is used to specify that a named method should be called if a fallback is required.
@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.

When fallbackMethod is used a FaultToleranceDefinitionException will be thrown if any of the following constraints are not met:

• The named fallback method must be on the same class, a superclass or an implemented interface of the class which declares the annotated method

• The named fallback method must have the same parameter types as the annotated method (after resolving any type variables)

• The named fallback method must have the same return type as the annotated method (after resolving any type variables)

• The named fallback method must be accessible from the class which declares the annotated method

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 9. 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. The circuit state will only be assessed when the rolling window reaches the requestVolumeThreshold.

```java
@CircuitBreaker(successThreshold = 10, requestVolumeThreshold = 4, failureRatio=0.5, delay = 1000)
public Connection serviceA() {
    Connection conn = null;
    counterForInvokingServiceA++;
    conn = connectionService();
    return conn;
}
```

Below are a couple scenarios to demonstrate when the circuit opens for the above code snippet.

- **Scenario 1**
  - Request 1 - success
  - Request 2 - failure
  - Request 3 - failure
  - Request 4 - success
  - Request 5 - CircuitBreakerOpenException

The Request 4 will trigger the circuit to open. The Request 5 will hit the exception.

- **Scenario 2**
- Request 1 - success
- Request 2 - failure
- Request 3 - success
- Request 4 - success
- Request 5 - failure
- Request 6 - CircuitBreakerOpenException

The Request 5 will trigger the circuit to open. The Request 6 will hit the exception.

9.1. 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.

```java
@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.

If @Fallback is used with @CircuitBreaker, the fallback method or handler will be invoked if a CircuitBreakerOpenException is thrown.

If @Retry is used with @CircuitBreaker, each retry attempt is processed by the circuit breaker and recorded as either a success or a failure. If a CircuitBreakerOpenException is thrown, the execution may be retried, depending on how the @Retry is configured.

If @Bulkhead is used with @CircuitBreaker, the circuit breaker is checked before attempting to enter the bulkhead. If attempting to enter the bulkhead results in a BulkheadException, this may be counted as a failure, depending on the value of the circuit breaker failOn attribute.
Chapter 10. 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.

10.1. 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.

10.1.1. Semaphore style Bulkhead

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

```java
@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.

10.1.2. Thread pool style Bulkhead

The below code-snippet means the method serviceA applies the Bulkhead policy with the 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.

If @Retry is used with @Bulkhead, when an invocation fails due to a BulkheadException it is retried after waiting for the delay configured on @Retry. If an invocation is permitted to run by the bulkhead but then throws another exception which is handled by @Retry, it first leaves the bulkhead, reducing the count of running concurrent requests by 1, waits for the delay configured on @Retry, and then attempts to enter the bulkhead again. At this point, it may be accepted, queued (if the method is also annotated with @Asynchronous) or fail with a BulkheadException (which may result in further retries).
Chapter 11. 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.

11.1. 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.

11.2. 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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Unit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ft.&lt;name&gt;.invocations.total</td>
<td>Counter</td>
<td>None</td>
<td>The number of times the method was called</td>
</tr>
<tr>
<td>ft.&lt;name&gt;.invocations.failed.total</td>
<td>Counter</td>
<td>None</td>
<td>The number of times the method was called and, after all Fault Tolerance actions had been processed, threw a Throwable</td>
</tr>
</tbody>
</table>

11.3. Metrics added for @Retry

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Unit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ft.&lt;name&gt;.retry.callsSucceeded NotRetried.total</td>
<td>Counter</td>
<td>None</td>
<td>The number of times the method was called and succeeded without retrying</td>
</tr>
<tr>
<td>ft.&lt;name&gt;.retry.callsSucceeded Retried.total</td>
<td>Counter</td>
<td>None</td>
<td>The number of times the method was called and succeeded after retrying at least once</td>
</tr>
<tr>
<td>ft.&lt;name&gt;.retry.callsFailed.total</td>
<td>Counter</td>
<td>None</td>
<td>The number of times the method was called and ultimately failed after retrying</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Unit</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>----------</td>
<td>--------</td>
<td>---------------------------------------------------------------</td>
</tr>
<tr>
<td><code>ft.&lt;name&gt;.retry.retries.total</code></td>
<td>Counter</td>
<td>None</td>
<td>The total number of times the method was retried</td>
</tr>
</tbody>
</table>

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.

### 11.4. Metrics added for @Timeout

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Unit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ft.&lt;name&gt;.timeout.executionDuration</code></td>
<td>Histogram</td>
<td>Nanoseconds</td>
<td>Histogram of execution times for the method</td>
</tr>
<tr>
<td><code>ft.&lt;name&gt;.timeout.callsTimedOut.total</code></td>
<td>Counter</td>
<td>None</td>
<td>The number of times the method timed out</td>
</tr>
<tr>
<td><code>ft.&lt;name&gt;.timeout.callsNotTimedOut.total</code></td>
<td>Counter</td>
<td>None</td>
<td>The number of times the method completed without timing out</td>
</tr>
</tbody>
</table>

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 if it's also annotated with `@Retry` because the Timeout logic is applied to each Retry attempt, not to the whole method invocation time.

### 11.5. Metrics added for @CircuitBreaker

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Unit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ft.&lt;name&gt;.circuitbreaker.callsSucceeded.total</code></td>
<td>Counter</td>
<td>None</td>
<td>Number of calls which were allowed to run and were considered a success by the circuit breaker</td>
</tr>
<tr>
<td><code>ft.&lt;name&gt;.circuitbreaker.callsFailed.total</code></td>
<td>Counter</td>
<td>None</td>
<td>Number of calls which were allowed to run and were considered a failure by the circuit breaker</td>
</tr>
<tr>
<td><code>ft.&lt;name&gt;.circuitbreaker.callsPrevented.total</code></td>
<td>Counter</td>
<td>None</td>
<td>Number of calls which were prevented from running by the circuit breaker</td>
</tr>
<tr>
<td><code>ft.&lt;name&gt;.circuitbreaker.open.total</code></td>
<td>Gauge&lt;<code>Long</code>&gt;</td>
<td>Nanoseconds</td>
<td>Amount of time the circuit breaker has spent in open state</td>
</tr>
<tr>
<td><code>ft.&lt;name&gt;.circuitbreaker.halfOpen.total</code></td>
<td>Gauge&lt;<code>Long</code>&gt;</td>
<td>Nanoseconds</td>
<td>Amount of time the circuit breaker has spent in half-open state</td>
</tr>
<tr>
<td><code>ft.&lt;name&gt;.circuitbreaker.closed.total</code></td>
<td>Gauge&lt;<code>Long</code>&gt;</td>
<td>Nanoseconds</td>
<td>Amount of time the circuit breaker has spent in closed state</td>
</tr>
<tr>
<td><code>ft.&lt;name&gt;.circuitbreaker.opened.total</code></td>
<td>Counter</td>
<td>None</td>
<td>Number of times the circuit breaker has moved from closed state to open state</td>
</tr>
</tbody>
</table>

21
Note that the sum of \( \text{ft.<name>.circuitbreaker.callsSucceeded.total} \), \( \text{ft.<name>.circuitbreaker.callsFailed.total} \) and \( \text{ft.<name>.circuitbreaker.callsPrevented.total} \) will give the total number of calls for which the circuit breaker logic was run.

Similarly, the sum of \( \text{ft.<name>.circuitbreaker.open.total} \), \( \text{ft.<name>.circuitbreaker.halfOpen.total} \) and \( \text{ft.<name>.circuitbreaker.closed.total} \) will give the total time since the circuit breaker was initialized.

### 11.6. Metrics added for @Bulkhead

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Unit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{ft.&lt;name&gt;.bulkhead.concurrentExecutions} )</td>
<td>Gauge&lt;Long&gt;</td>
<td>None</td>
<td>Number of currently running executions</td>
</tr>
<tr>
<td>( \text{ft.&lt;name&gt;.bulkhead.callsAccepted.total} )</td>
<td>Counter</td>
<td>None</td>
<td>Number of calls accepted by the bulkhead</td>
</tr>
<tr>
<td>( \text{ft.&lt;name&gt;.bulkhead.callsRejected.total} )</td>
<td>Counter</td>
<td>None</td>
<td>Number of calls rejected by the bulkhead</td>
</tr>
<tr>
<td>( \text{ft.&lt;name&gt;.bulkhead.executionDuration} )</td>
<td>Histogram</td>
<td>Nanoseconds</td>
<td>Histogram of the time that executions spend holding a semaphore permit or using one of the threads from the thread pool</td>
</tr>
<tr>
<td>( \text{ft.&lt;name&gt;.bulkhead.waitingQueue.population*} )</td>
<td>Gauge&lt;Long&gt;</td>
<td>None</td>
<td>Number of executions currently waiting in the queue</td>
</tr>
<tr>
<td>( \text{ft.&lt;name&gt;.bulkhead.waiting.duration*} )</td>
<td>Histogram</td>
<td>Nanoseconds</td>
<td>Histogram of the time executions spend waiting in the queue</td>
</tr>
</tbody>
</table>

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

### 11.7. Metrics added for @Fallback

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Unit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{ft.&lt;name&gt;.fallback.calls.total} )</td>
<td>Counter</td>
<td>None</td>
<td>Number of times the fallback handler or method was called</td>
</tr>
</tbody>
</table>

### 11.8. 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 count the number of events which occurred since the application started, and
therefore never decrease. 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.

11.9. Annotation Example

```java
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
```
Only one call was made, and it succeeded after some retries.

Two retries were made during the invocation.

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.

One of the attempts timed out.

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.

This programming model is very flexible. All the annotation parameters are configurable and the annotations can be disabled as well.

12.1. Config Fault Tolerance parameters

This specification defines the annotations: \texttt{@Asynchronous}, \texttt{@Bulkhead}, \texttt{@CircuitBreaker}, \texttt{@Fallback}, \texttt{@Retry} and \texttt{@Timeout}. Each annotation except \texttt{@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 \texttt{<classname>/<methodname>/<annotation>/<parameter>}. The \texttt{<classname>} and \texttt{<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 \texttt{maxRetries} for serviceB invocation to 100, set the config property \texttt{com.acme.test.MyClient/serviceB/Retry/maxRetries=100} Similarly to override the \texttt{maxDuration} for ServiceA, set the config property \texttt{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 \texttt{<classname>/<annotation>/<parameter>} for configuration. For an instance, use the following config property to override all \texttt{maxRetries} for \texttt{Retry} specified on the class \texttt{MyClient} to 100.

\texttt{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 \texttt{Timeout} needs to be set to \texttt{100ms}. It can be cumbersome to override each occurrence of \texttt{Timeout}. In this circumstance, the config property \texttt{<annotation>/<parameter>} overrides the corresponding parameter value for the specified annotation. For instance, in order to override the \texttt{maxRetries} for the \texttt{Retry} to be 30, specify the config property \texttt{Retry/maxRetries=30}.

When multiple config properties are present, the property \texttt{<classname>/<methodname>/<annotation>/<parameter>} takes precedence over \texttt{<classname>/<annotation>/<parameter>}, which is followed by \texttt{<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 \texttt{Retry} is specified on the class level for the class \texttt{com.acme.ClassA}, which has method \texttt{methodB}, the config property \texttt{com.acme.ClassA/methodB/Retry/maxRetries} will be ignored. In order to override the property, the config property \texttt{com.acme.ClassA/Retry/maxRetries} or
package com.acme.test;
public class MyClient{
    /**
     * The configured 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. Disable a group of Fault Tolerance annotations on the global level

Some service mesh platforms, e.g. Istio, have their 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 disabled except fallback. The reason fallback is special is that the fallback business logic can only be defined by microservices and not by any other platforms.
Setting 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.

### 12.3. Disabled individual Fault Tolerance policy

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:

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

For instance the following config will disable bulkhead policy globally:

```xml
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 property `MP_Fault_Tolerance_NonFallback_Enabled` has the lowest priority. e.g.

- `MP_Fault_Tolerance_NonFallback_Enabled=true`
- `Bulkhead/enabled=true`

In the above example, only `Fallback` and `Bulkhead` are enabled while the others are disabled.

### 12.4. Configuring Metrics Integration

The integration with MicroProfile Metrics can be disabled by setting a config property named `MP_Fault_Tolerance_Metrics_Enabled` to the value `false`. If this property is absent or set to `true` then the integration with MicroProfile Metrics will be enabled and the metrics listed earlier in this specification will be added automatically for every method annotated with a `@Retry`, `@Timeout`, `@CircuitBreaker`, `@Bulkhead` or `@Fallback` annotation.

In order to prevent any unexpected behaviour, the property `MP_Fault_Tolerance_Metrics_Enabled` will only be read when the application starts. Any dynamic changes afterwards will be ignored until the application is restarted.
Chapter 13. Release Notes for MicroProfile Fault Tolerance 1.1

The following changes occurred in the 1.1 release, compared to 1.0

A full list of changes may be found on the MicroProfile Fault Tolerance 1.1 Milestone

13.1. API/SPI Changes

- The ExecutionContext interface has been extended with a getFailure method that returns the execution failure(#224).

13.2. Functional Changes

- Implementations must implement the new method of ExecutionContext.getFailure(#224).
- Added metrics status automatically for FT (#234)
- Disable individual Fault Tolerance annotation using external config (#109)
- Define priority when multiple properties declared (link: #278)

13.3. Specification Changes

- Implementations must implement the new method of ExecutionContext.getFailure(#224).
- Added metrics status automatically for FT (#234)
- Disable individual Fault Tolerance annotation using external config (#109)
- Define priority when multiple properties declared (link: #278)
- Clarify fallback (#177)

13.4. Other changes

- Bulkhead TCK changes (#227)
- Add standalone async test (#194)
- Add more configuration test (#182)
- Circuit Breaker Rolling window behaviour test (#197)
- Improve Bulkhead test (#198)
Chapter 14. Release Notes for MicroProfile Fault Tolerance 2.0

This release is a major release of Fault Tolerance. The reason for increasing the release version to 2.0 is that this release upgrades its CDI dependency from CDI 1.2 to CDI 2.0, in order to use the new features introduced by CDI 2.0. Therefore, this specification is not compatible with Java EE7 but is compatible with Java EE8. Other than this, there are no backward incompatible changes introduced in this specification.

The following changes occurred in the 2.0 release, compared to 1.1.

A full list of changes may be found on the MicroProfile Fault Tolerance 2.0 Milestone

14.1. API/SPI Changes

• Add support of the CompletionStage return type when annotated with @Asynchronous (#110).

14.2. Functional Changes

• Specify the invocation sequence of MicroProfile Fault Tolerance annotations when used together (#291)

• Clarify how the Fault Tolerance annotations interact with other application defined interceptors (#313)

14.3. Specification Changes

• Clarify whether other Fault Tolerance functionalities will be triggered on an exceptional returned Future (#246).

• Specify the sequence of MicroProfile Fault Tolerance annotations when used together (#291)

• Clarify how the Fault Tolerance annotations interact with other application defined interceptors (#313)

14.4. Other changes

• Clarify failOn() on CircuitBreaker and Fallback (#240)

• Circuit Breaker - clarify how requestVolumeThreshold() and rolling window work (#342)

• other smaller fixes (#341) (#252) (#306)