Microprofile Fault Tolerance

Emily Jiang, Antoine Sabot-Durant, Andrew Rouse

4.1-RC3, July 31, 2024: Draft

Table of Contents

Specification: Microprofile Fault Tolerance

Version: 4.1-RC3

Status: Draft

Release: July 31, 2024

Copyright

Eclipse Foundation Specification License

Disclaimers

Architecture

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

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.

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 Jakarta Interceptors and Contexts and Dependency Injection specifications defined in Jakarta EE platform.

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.

Relationship to Jakarta Interceptors

The Jakarta 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 Jakarta 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.

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.

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.

Relationship to MicroProfile Telemetry

The MicroProfile Telemetry 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 Telemetry are used together, metrics are automatically added for each of the methods annotated with a `@Retry`, `@Timeout`, `@CircuitBreaker`, `@Bulkhead` or `@Fallback` annotation.

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`.

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

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

@Priority(5000)
@MyPrint
public class MyPrintInterceptor {
  // do printing
}
```

Custom throwables

Throwing custom throwables from business methods annotated with any of the Fault Tolerance interceptor bindings results in non-portable behavior. The term "custom throwable" means: any

class that is a subtype of `Throwable`, but isn't a subtype of `Error` or `Exception`. This includes `Throwable` itself, and *doesn't* include `Error` and `Exception`.

NOTE

Some Fault Tolerance annotations allow configuring a set of exception types for various purposes. For example, `@Retry` includes the `retryOn` attribute which configures the set of exceptions on which retry will be performed. In these cases, it is possible to specify `Throwable` and it is guaranteed to cover all `Error`s and `Exception`s.

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.

Asynchronous

`Asynchronous` means the execution of the client request will be on a separate thread.

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 context for `RequestScoped` must be active during the asynchronous method invocation. 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 ends normally and returns a value, the `Future` or `CompletionStage` will be behaviorally equivalent to the return value (which, itself, is a `Future` or `CompletionStage`).

```
@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.

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.

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.

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

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 not considered to have completed until the returned `CompletionStage` completes.
  - The method 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.

> **TIP**
>
> As a consequence of these rules:
>
> - `@Timeout` does not consider the method to have completed until the returned `CompletionStage` completes
> - `@Bulkhead` considers the method to still be running until the returned `CompletionStage` completes.

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

```
@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.

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`.

Timeout

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

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.

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 `ExecutionException` 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.

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

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.

When a method returns and the retry policy is present, the following rules are applied:

- If the method returns normally (doesn't throw), the result is simply returned.
- Otherwise, if the thrown object is assignable to any value in the `abortOn` parameter, the thrown object is rethrown.
- Otherwise, if the thrown object is assignable to any value in the `retryOn` parameter, the method call is retried.
- Otherwise the thrown object is rethrown.

For example, to retry on all exceptions except for IO exceptions, one would write:

```
    /**
     * In case the underlying service throws an exception, it will be retried,
     * unless the thrown exception was an IO exception.
     */
    @Retry(retryOn = Exception.class, abortOn = IOException.class)
    public void service() {
        underlyingService();
    }
```

If a method throws a `Throwable` which is not an `Error` or `Exception`, non-portable behavior results.

```
    /**
     * 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();
    }

    /**
    * There should be 0-400ms delays between each invocation.
    * The effective delay will be between:
    * [delay - jitter, delay + jitter] and always >= 0. Negative effective delays will
be 0.
    * There should be at least 8 retries but no more than 10 retries.
    */
    @Retry(delay = 0, 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 method still fails after any retires 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. (However, note that if `@Retry` is used with `@Timeout` and `@Asynchronous`, and a `TimeoutException` results in a new retry attempt, the

new retry attempt must start after the configured delay period, even if the previous retry attempt has not finished. See Timeout Usage.)

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.

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

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.

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.

Specify the criteria for triggering Fallback

The fallback might be triggered when an exception occurs, including the ones defined in this spec (e.g. `BulkheadException`, `CircuitBreakerOpenException`, `TimeoutException`, etc), detailed below. When a method returns and the Fallback policy is present, the following rules are applied:

- If the method returns normally (doesn't throw an exception), the result will be simply returned.
- Otherwise, if the thrown object is assignable to any value in the `skipOn` parameter, the thrown object will be rethrown.
- Otherwise, if the thrown object is assignable to any value in the `applyOn` parameter, the specified fallback will be triggered.
- Otherwise the thrown object will be rethrown. In the following example, all exceptions assignable to `ExceptionA` and `ExceptionB`, except the ones assignable to `ExceptionBSub` will trigger the fallback operation.

```
    @Retry(maxRetries = 2)
    @Fallback(applyOn={ExceptionA.class, ExceptionB.class}, skipOn=ExceptionBSub.
class, fallbackMethod= "fallbackForServiceB")
```

```
    public String serviceB() {
       return nameService();
    }

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

If a method throws a `Throwable` which is not an `Error` or `Exception`, non-portable behavior results.

Circuit Breaker

A Circuit Breaker prevents repeated failures, so that dysfunctional services or APIs fail fast. If a service is failing frequently, the circuit breaker opens and no more calls to that service are attempted until a period of time has passed.

There are three circuit states:

- **Closed**: In normal operation, the circuit breaker is closed. The circuit breaker records whether each call is a success or failure and keeps track of the most recent results in a rolling window. Once the rolling window is full, if the proportion of failures in the rolling window rises above the `failureRatio`, the circuit breaker will be opened.
- **Open**: When the circuit breaker is open, calls to the service operating under the circuit breaker will fail immediately with a `CircuitBreakerOpenException`. After a configurable delay, the circuit breaker transitions to half-open state.
- **Half-open**: In half-open state, a configurable number of trial executions of the service are allowed. If any of them fail, the circuit breaker transitions back to open state. If all the trial executions succeed, the circuit breaker transitions to the closed state.

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.

Configuring when the circuit opens and closes

The following parameters control when the circuit breaker opens and closes.

- `requestVolumeThreshold` controls the size of the rolling window used when the circuit breaker is closed
- `failureRatio` controls the proportion of failures within the rolling window which will cause the circuit breaker to open
- `successThreshold` controls the number of trial calls which are allowed when the circuit breaker is half-open
- `delay` and `delayUnit` control how long the circuit breaker stays open

Circuit breaker state transitions will reset the Circuit Breaker's records. For example, when the circuit breaker 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`.

The following example and scenarios demonstrate when the circuit opens.

```
@CircuitBreaker(successThreshold = 10, requestVolumeThreshold = 4, failureRatio=0.5,
delay = 1000)
public Connection serviceA() {
```

```
    Connection conn = null;
    counterForInvokingServiceA++;
    conn = connectionService();
    return conn;
}
```

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

In this scenario, request 5 will trigger the circuit to open because out of the last four requests (the `requestVolumeThreshold`), two failed which reaches the `failureRatio` of `0.5`. Request 6 will therefore hit the `CircuitBreakerOpenException`.

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

In this scenario, request 4 will cause the circuit to open. Request 5 will hit the `CircuitBreakerOpenException`. Note that request 3 does not cause the circuit to open because the rolling window has not yet reached the `requestVolumeThreshold`.

Configuring which exceptions are considered a failure

The `failOn` and `skipOn` parameters are used to define which exceptions are considered failures for the purpose of deciding whether the circuit breaker should open.

When a method returns a result, the following rules are applied to determine whether the result is a success or a failure:

- If the method does not throw a `Throwable`, it is considered a success
- Otherwise, if the thrown object is assignable to any value in the `skipOn` parameter, is is considered a success
- Otherwise, if the thrown object is assignable to any value in the `failOn` parameter, it is considered a failure
- Otherwise it is considered a success

If a method throws a `Throwable` which is not a subclass of either `Error` or `Exception`, non-portable behavior results.

In the following example, all exceptions assignable to `ExceptionA` and `ExceptionB`, except the ones assignable to `ExeptionBSub` will be considered failures. `ExceptionBSub` and all other exceptions will not be considered failures for the purpose of deciding whether the circuit breaker should open.

```
@CircuitBreaker(failOn = {ExceptionA.class, ExceptionB.class}, skipOn = ExceptionBSub
.class)
public void service() {
    underlyingService();
}
```

Lifecycle

Circuit breaker needs to maintain some state between invocations: the number of recent successful and failed invocations, or how long has the circuit breaker been open. This state is a singleton, irrespective of the lifecycle of the bean that uses the `@CircuitBreaker` annotation.

More specifically, the circuit breaker state is uniquely identified by the combination of the bean class (`java.lang.Class`) and the method object (`java.lang.reflect.Method`) representing the guarded method.

For example, if there's a guarded method `doWork` on a bean which is `@RequestScoped`, each request will have its own instance of the bean, but all invocations of `doWork` will share the same circuit breaker state.

Interactions with other annotations

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.

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 an instance. Therefore, `Bulkhead` pattern is only effective when applying `@Bulkhead` to a component that can be accessed from multiple contexts.

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.

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.

```
@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`.

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.

Lifecycle

Bulkhead needs to maintain some state between invocations: the number of currently running executions, or the queue of waiting executions. This state is a singleton, irrespective of the lifecycle of the bean that uses the `@Bulkhead` annotation.

More specifically, the bulkhead state is uniquely identified by the combination of the bean class (`java.lang.Class`) and the method object (`java.lang.reflect.Method`) representing the guarded method.

For example, if there's a guarded method `doWork` on a bean which is `@RequestScoped`, each request will have its own instance of the bean, but all invocations of `doWork` will share the same bulkhead state.

Interactions with other annotations

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).

Integration with MicroProfile Metrics and MicroProfile Telemetry

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

If all three of MicroProfile Fault Tolerance, MicroProfile Metrics, and MicroProfile Telemetry are used together then MicroProfile Fault Tolerance exports metrics to both MicroProfile Metrics and MicroProfile Telemetry.

Names

The automatically added metrics follow a consistent pattern which includes the fully qualified name of the annotated method.

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.

Scope

In MicroProfile Metrics, metrics added by this specification will appear in the `base` MicroProfile Metrics scope.

Registration

All metrics added by this specification for a particular method are registered with each applicable combination of tags either on startup or on first call of the annotated method. Policies that have been disabled through configuration do not cause registration of the corresponding metrics.

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	`ft.invocations.total`
Type in MP Metrics	`Counter`
Type in MP Telemetry	A counter that emits long
Unit	None
Description	The number of times the method was called

Name	`ft.invocations.total`
Tags	• `method` - the fully qualified method name • `result` = `[valueReturned|exceptionThrown]` - whether the invocation returned a value or threw an exception • `fallback` = `[applied|notApplied|notDefined]` - `applied` if fallback was used, `notApplied` if a fallback is configured but was not used, `notDefined` if a fallback is not configured

Metrics added for `@Retry`

Name	`ft.retry.calls.total`
Type in MP Metrics	`Counter`
Type in MP Telemetry	A counter that emits long
Unit	None
Description	The number of times the retry logic was run. This will always be once per method call.
Tags	• `method` - the fully qualified method name • `retried` = `[true|false]` - whether any retries occurred • `retryResult` = `[valueReturned|exceptionNotRetryable|maxRetriesReached|maxDurationReached]` - the reason that last attempt to call the method was not retried

Name	`ft.retry.retries.total`
Type in MP Metrics	`Counter`
Type in MP Telemetry	A counter that emits long
Unit	None
Description	The number of times the method was retried
Tags	• `method` - the fully qualified method name

Metrics added for `@Timeout`

Name	`ft.timeout.calls.total`
Type in MP Metrics	`Counter`
Type in MP Telemetry	A counter that emits long
Unit	None

Name	`ft.timeout.calls.total`
Description	The number of times the timeout logic was run. This will usually be once per method call, but may be zero times if the circuit breaker prevents execution or more than once if the method is retried.
Tags	• `method` - the fully qualified method name • `timedOut` = `[true|false]` - whether the method call timed out

Name	`ft.timeout.executionDuration`
Type in MP Metrics	`Histogram`
Unit in MP Metrics	Nanoseconds
Type in MP Telemetry	A Histogram that records `double` values with explicit bucket boundaries `[ 0.005, 0.01, 0.025, 0.05, 0.075, 0.1, 0.25, 0.5, 0.75, 1, 2.5, 5, 7.5, 10 ]`
Unit in MP Telemetry	Seconds
Description	Histogram of execution times for the method
Tags	• `method` - the fully qualified method name

Metrics added for `@CircuitBreaker`

Name	`ft.circuitbreaker.calls.total`
Type in MP Metrics	`Counter`
Type in MP Telemetry	A counter that emits long
Unit	None
Description	The number of times the circuit breaker logic was run. This will usually be once per method call, but may be more than once if the method call is retried.
Tags	• `method` - the fully qualified method name • `circuitBreakerResult` = `[success|failure|circuitBreakerOpen]` - the result of the method call, as considered by the circuit breaker according to the rules in Configuring which exceptions are considered a failure ◦ `success` - the method ran and was successful ◦ `failure` - the method ran and failed ◦ `circuitBreakerOpen` - the method did not run because the circuit breaker was in open or half-open state

Name	`ft.circuitbreaker.state.total`
Type in MP Metrics	`Gauge<Long>`

Name	`ft.circuitbreaker.state.total`
Type in MP Telemetry	A counter that emits long
Unit	Nanoseconds
Description	Amount of time the circuit breaker has spent in each state
Tags	• `method` - the fully qualified method name • `state` = `[open|closed|halfOpen]` - the circuit breaker state
Notes	Although this metric is a `Gauge`, its value increases monotonically.

Name	`ft.circuitbreaker.opened.total`
Type in MP Metrics	`Counter`
Type in MP Telemetry	A counter that emits long
Unit	None
Description	Number of times the circuit breaker has moved from closed state to open state
Tags	• `method` - the fully qualified method name

Metrics added for `@Bulkhead`

Name	`ft.bulkhead.calls.total`
Type in MP Metrics	`Counter`
Type in MP Telemetry	A counter that emits long
Unit	None
Description	The number of times the bulkhead logic was run. This will usually be once per method call, but may be zero times if the circuit breaker prevented execution or more than once if the method call is retried.
Tags	• `method` - the fully qualified method name • `bulkheadResult` = `[accepted|rejected]` - whether the bulkhead allowed the method call to run

Name	`ft.bulkhead.executionsRunning`
Type in MP Metrics	`Gauge<Long>`
Type in MP Telemetry	An UpDownCounter that emits long
Unit	None
Description	Number of currently running executions
Tags	• `method` - the fully qualified method name

Name	`ft.bulkhead.executionsWaiting`
Type in MP Metrics	`Gauge<Long>`
Type in MP Telemetry	An UpDownCounter that emits long
Unit	None
Description	Number of executions currently waiting in the queue
Tags	• `method` - the fully qualified method name
Notes	Only added if the method is also annotated with `@Asynchronous`

Name	`ft.bulkhead.runningDuration`
Type in MP Metrics	`Histogram`
Unit in MP Metrics	Nanoseconds
Type in MP Telemetry	A Histogram that records `double` values with explicit bucket boundaries `[ 0.005, 0.01, 0.025, 0.05, 0.075, 0.1, 0.25, 0.5, 0.75, 1, 2.5, 5, 7.5, 10 ]`
Unit in MP Telemetry	Seconds
Description	Histogram of the time that method executions spent running
Tags	• `method` - the fully qualified method name

Name	`ft.bulkhead.waitingDuration`
Type in MP Metrics	`Histogram`
Unit in MP Metrics	Nanoseconds
Type in MP Telemetry	A Histogram that records `double` values with explicit bucket boundaries `[ 0.005, 0.01, 0.025, 0.05, 0.075, 0.1, 0.25, 0.5, 0.75, 1, 2.5, 5, 7.5, 10 ]`
Unit in MP Telemetry	Seconds
Description	Histogram of the time that method executions spent waiting in the queue
Tags	• `method` - the fully qualified method name
Notes	Only added if the method is also annotated with `@Asynchronous`

Notes

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

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.invocations.total{method="com.example.MyClass.doWork", result="valueReturned",
fallback="notDefined"}
ft.invocations.total{method="com.example.MyClass.doWork", result="exceptionThrown",
fallback="notDefined"}
ft.retry.calls.total{method="com.example.MyClass.doWork", retried="true",
retryResult="valueReturned"}
ft.retry.calls.total{method="com.example.MyClass.doWork", retried="true",
retryResult="exceptionNotRetryable"}
ft.retry.calls.total{method="com.example.MyClass.doWork", retried="true",
retryResult="maxRetriesReached"}
ft.retry.calls.total{method="com.example.MyClass.doWork", retried="true",
retryResult="maxDurationReached"}
ft.retry.calls.total{method="com.example.MyClass.doWork", retried="false",
retryResult="valueReturned"}
ft.retry.calls.total{method="com.example.MyClass.doWork", retried="false",
retryResult="exceptionNotRetryable"}
ft.retry.calls.total{method="com.example.MyClass.doWork", retried="false",
retryResult="maxRetriesReached"}
ft.retry.calls.total{method="com.example.MyClass.doWork", retried="false",
retryResult="maxDurationReached"}
ft.retry.retries.total{method="com.example.MyClass.doWork"}
ft.timeout.calls.total{method="com.example.MyClass.doWork", timedOut="true"}
ft.timeout.calls.total{method="com.example.MyClass.doWork", timedOut="false"}
ft.timeout.executionDuration{method="com.example.MyClass.doWork"}
```

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

- The invocation is retried but something goes wrong and the method throws an `IOException`
- The invocation is retried again and this time the method returns successfully and the result of this attempt is returned to the user

After this sequence, the following metrics would have new values:

```
ft.invocations.total{method="com.example.MyClass.doWork", result="valueReturned",
fallback="notDefined"} = 1
```

The method has been called successfully once and it returned a value.

```
ft.retry.calls.total{method="com.example.MyClass.doWork", retried="true",
retryResult="valueReturned"} = 1
```

One call was made and, after some retries, it returned a value.

```
ft.retry.retries.total{method="com.example.MyClass.doWork"} = 2
```

Two retries were made during the invocation.

```
ft.timeout.executionDuration{method="com.example.MyClass.doWork"}
```

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.timeout.calls.total{method="com.example.MyClass.doWork", timedOut="true"} = 1
```

One of the attempts timed out.

```
ft.timeout.calls.total{method="com.example.MyClass.doWork", timedOut="false"} = 2
```

Two of the attempts did not time out.

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.

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.

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.

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:

`<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 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.

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.

Recommendations for Optional Container Integration

This section describes the expected behaviors when the implementation runs in a Jakarta EE container.

`@Asynchronous`

Threads that are servicing `@Asynchronous` invocations should, for the duration of the invocation, have the correct security context and naming context associated.

Release Notes for MicroProfile Fault Tolerance 4.1

A full list of changes delivered in the 4.1 release can be found at MicroProfile Fault Tolerance 4.1 Milestone.

Incompatible Changes

No.

API/SPI Changes

No.

Specification changes

- Work with MicroProfile Telemetry Metrics (#622)

Other Changes

- Removed unmanaged threads in TCK (#634)

Release Notes for MicroProfile Fault Tolerance 4.0

A full list of changes delivered in the 4.0 release can be found at MicroProfile Fault Tolerance 4.0 Milestone.

Incompatible Changes

This release aligns with Jakarta EE 9.1, so it won't work with earlier versions of Jakarta or Java EE.

API/SPI Changes

There are no functional changes introduced in this release, except the dependency updating from javax to jakarta.

Release Notes for MicroProfile Fault Tolerance 3.0

This release is a major release of Fault Tolerance which includes backward incompatible changes.

A full list of changes can be found on the MicroProfile Fault Tolerance 3.0 Milestone

Backward incompatible changes

Metric names and scopes changed

The metrics added automatically by MicroProfile Fault Tolerance have been updated to take advantage of support for metric tags which was added to MicroProfile Metrics in version 2.0. As a result, some information which was previously contained in the metric name is now instead included in tags.

In addition, metrics have moved from the `application:` scope to the `base:` scope for consistency with other MicroProfile specifications. Note that this means:

- Metrics are now exported under `/metrics` and `/metrics/base`, instead of `/metrics` and `/metrics/application` as in previous versions.
- In the JSON format, when metrics are retrieved from `/metrics` they appear in the `base` object rather than the `application` object.
- In the OpenMetrics format, the names are prefixed with `base_` instead of `application_`.

Example

Old metric

```
application:ft.<name>.timeout.callsTimedOut.total
```

New metric

```
base:ft.timeout.calls.total{method="<name>", timedOut="true"}
```

These changes mean that existing dashboards and queries which use metrics provided by MicroProfile Fault Tolerance will need to be updated to use the new metrics listed in Integration with MicroProfile Metrics and MicroProfile Telemetry.

Lifecycle of circuit breakers and bulkheads is now specified

In previous versions of MicroProfile Fault Tolerance, the lifecycle of circuit breakers and bulkheads was not specified. These fault tolerance strategies hold state between invocations, so their lifecycle is important for correct functioning.

The specification now requires that circuit breakers and bulkheads are singletons, identified by the bean class and the guarded method. For example, if a `@RequestScoped` bean has a `@CircuitBreaker` method, all invocations of that method will share the same circuit breaker state, even though each request has a different instance of the bean.

API/SPI changes

Functional changes

- Updated metrics to use tags (#401)
- Moved metrics into the base scope (#499)
- Specified lifecycle of circuit breakers and bulkheads (#479])

Specification changes

- Updated metrics to use tags (#401)
- Moved metrics into the base scope (#499)
- Specified lifecycle of circuit breakers and bulkheads (#479])

Release Notes for MicroProfile Fault Tolerance 2.1

This release is a minor release of Fault Tolerance. It includes a number of new features and clarifications, as well as TCK improvements. The following changes occurred in the 2.1 release, compared to 2.0.

A full list of changes can be found on the MicroProfile Fault Tolerance 2.1 Milestone

API/SPI changes

- The `Retry.retryOn` and `abortOn` attributes no longer ignore `Throwable.class` (#449).
- Added `CircuitBreaker.skipOn` (#418).
- Added `Fallback.applyOn` and `skipOn` (#417).

Functional changes

- The `Retry.retryOn` and `abortOn` attributes no longer ignore `Throwable.class` (#449).
- Added `CircuitBreaker.skipOn` (#418).
- Added `Fallback.applyOn` and `skipOn` (#417).
- Specified the meaning of overlapping `Retry.retryOn` and `abortOn` (#442).
- Relaxed the requirements on `Future` and `CompletionStage` implementations (#425).

Specification changes

- Specified the meaning of overlapping `Retry.retryOn` and `abortOn` (#442).
- Specified that throwing custom throwables is non-portable (#441).
- Specified that the CDI request context must be active during the execution of methods annotated with `Asynchronous` (#274)
- Relaxed the requirements on `Future` and `CompletionStage` implementations (#425).
- Clarified that when a method returning `CompletionStage` is annotated with both `Bulkhead` and `Asynchronous`, the bulkhead considers the method to be executing until the `CompletionStage` returned by the method completes. (#484)
- Clarified the retry metrics specification (#491).

Other changes

- Time values in TCK tests are now configurable (#399).
- Transitive dependency on `jakarta.el-api` has been excluded (#439).

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 can be found on the MicroProfile Fault Tolerance 2.0 Milestone

API/SPI changes

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

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)

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)

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)

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 can be found on the MicroProfile Fault Tolerance 1.1 Milestone

API/SPI changes

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

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)

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)

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)