payara®

MICROPROFILE™
OPTIMIZING ENTERPRISE JAVA

Build Resilient Cloud Native Applications With MicroProfile Fault Tolerance

The Payara® Platform - Production-Ready, Cloud Native and Aggressively Compatible.

User Guide

Contents

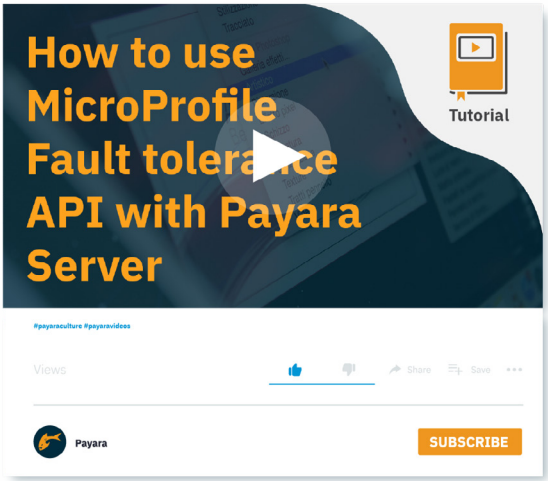
Guide Updated: **April 2023**

Modern cloud-native applications are generally crafted as a set of microservices that communicate with each other, mostly over the network. Users of such applications have come to expect applications to always be available. However, there are a number of things that can always go wrong during the runtime of an application instance.

The network may be unavailable, a database service might take too long to respond, an application might go viral overnight and have an unexpected surge in concurrent users, a web service might be offline and a multitude of other potential problems. For an application to be able to weather such inevitable mishaps, it needs to be fault-tolerant and resilient.

The MicroProfile Fault Tolerance specification provides a number of declarative API constructs that can be used to create fault-tolerant and resilient modern, cloud-native applications. The goal of this guide is to show you how you can use the MicroProfile Fault Tolerance API to add retry policies, bulkheads, circuit breakers, timeouts and fallbacks to your microservices, transforming them into much more resilient cloud-native deployments.

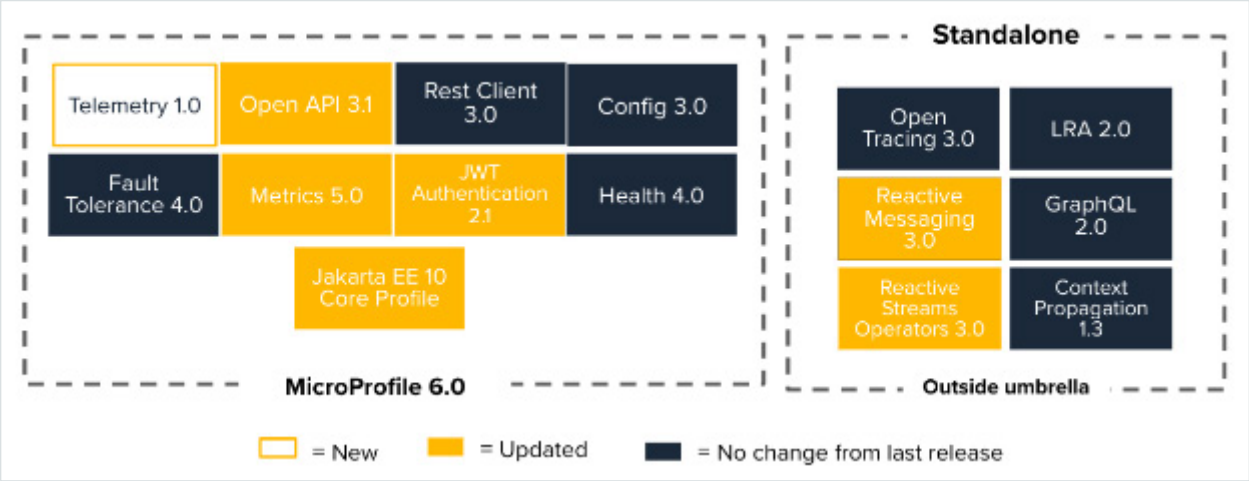
Also, watch our video to learn how to use the Fault Tolerance API with Payara Server.



What is MicroProfile?

MicroProfile is a community driven initiative, built on top of the Jakarta EE Core Profile, that is a collection of abstract specs that form a complete solution to developing cloud native, Jakarta EE microservices. The goal is to create a set of APIs that abstracts you from their implementations so that you can create highly portable microservices across vendors.

The current release is version 6.0 which is a major release that includes MicroProfile Config 3.0, MicroProfile Fault Tolerance 4.0, MicroProfile Health 4.0, MicroProfile Metrics 5.0, and MicroProfile Rest Client 3.0. MicroProfile 6.0 is built on top of the Core Profile of Jakarta EE, a slimmed down version of Jakarta EE “that contains a set of Jakarta EE Specifications targeting smaller runtimes suitable for microservices and ahead-of-time compilation.” The Core Profile of Jakarta EE was released as part of Jakarta EE 10.



As abstract specifications, the various implementations are free to implement the base specs and add custom features on top. Payara Server is one of the popular implementations of the MicroProfile spec and adds quite a number of custom features on top of the base specs. You can download a free trial of Payara Enterprise here to follow along with the rest of the guide.

Getting Started with MicroProfile

To get started with the MicroProfile API, you need to include it as a dependency in your project as shown below.

```
    <dependency>
        <groupId>org.eclipse.microprofile</groupId>
        <artifactId>microprofile</artifactId>
        <version>6.0</version>
        <type>pom</type>
        <scope>provided</scope>
    </dependency>
```

With the MicroProfile API dependency in place, you have access to all the APIs of the project. In our case, the Payara Server will provide the implementation for us.

What is Fault Tolerance?

Fault Tolerance is a set of adaptable patterns for writing resilient applications. The failure of one component of a fault tolerant application does not bring the entire application down. In a cloud native era, designing applications that can scale both up and down is vital.

MicroProfile Fault Tolerance builds on top of Jakarta Contexts and Dependency Injection's interceptor construct. This allows the various Fault Tolerance annotations to be able to intercept method calls and react based on their purpose.

The MicroProfile Fault Tolerance API defines a set of annotations that together cover all that you will need to write more resilient applications. They are:

- Fallback
- Timeout
- Retry Policy
- Circuit Breaker
- Asynchronous
- Bulkhead

Let us examine each of them in detail.

Fallback

The *@Fallback* annotation, or technically, interceptor binding, is used to tell the runtime to fall back on a designated method if an exception is thrown. Let us take a look at this in the code snippet below.

```
@Fallback(fallbackMethod = "defaultCurrencyInfo")
public CurrencyInfo getCurrencyInfo(String country) {
        return geoFetch.getCurrencyInfo(country);
    }
```

The *@Fallback* annotation is passed a String with the value "`defaultCurrencyInfo`" to the fallbackMethod parameter. This tells the runtime to look for a method with the name defaultCurrencyInfo in the same class that the currently annotated method is located and invoke that method should an exception be thrown in method `getCurrencyInfo`.

The fallback method should have the same signature as the method for which it is acting as a fallback. Method `defaultCurencyInfo` is shown below.

```
public CurrencyInfo defaultCurrencyInfo(String country) {
        CurrencyInfo currencyInfo = new CurrencyInfo();
        currencyInfo.name = "British Pound";
        currencyInfo.code = "GBP";
        currencyInfo.symbol = "£";
        return currencyInfo;
    }
```

Method `defaultCurrencyInfo` has the same signature as method `getCurrencyInfo` as shown earlier. In the event of an exception thrown in the `getCurrencyInfo` method, the MicroProfile runtime will invoke `defaultCurrencyInfo`.

The *@Fallback* annotation can also be passed a class that implements the `org.eclipse.microprofile.faulttolerance.FallbackHandler<T>` interface. This interface takes the return type of the fallback method as a parameter. It has only one method called handle that is passed an `org.eclipse.microprofile.faulttolerance.ExecutionContext` and returns whatever type is passed in.

Let us examine the use of a Fallback handler interface starting with an implementation as shown below.

```
public class CurrencyInfoFallbackHandler implements
FallbackHandler<CurrencyInfo> {

    @Override
    public CurrencyInfo handle(ExecutionContext context) {
        CurrencyInfo currencyInfo = new CurrencyInfo();
        currencyInfo.name = "British Pound";
        currencyInfo.code = "GBP";
        currencyInfo.symbol = "£";
        return currencyInfo;
    }
}
```

The above code shows a class called `CurrencyInfoFallbackHandler` that implements the FallbackHandler interface, passing in `CurrencyInfo` as the concrete type. The handle method also returns a `CurrencyInfo`. Now let us take a look at how to use a `FallbackHandler` class.

```
@Fallback(CurrencyInfoFallbackHandler.class)
 public CurrencyInfo fetchCurrencyInfo(String country) {
     return geoFetch.getCurrencyInfo(country);
 }
```

The fetchCurrencyInfo is annotated with the @Fallback annotation, but this time around it is passed the class of the FallbackHandler - the CurrencyInfoFallbackHandler. The method has the same signature as the one seen before.

The exceptions that trigger the fallback can be customized using the applyOn or skipOn parameters of the @Fallback annotation. The applyOn parameter takes an array of exceptions that extend java.lang. Throwable. Any of the passed exceptions thrown will cause the runtime to invoke the fallback method. The skipOn annotation also takes an array of exceptions deriving from the same Throwable class. However, any skipOn exception thrown will cause the runtime to skip calling the fallback method.

The Fallback Fault Tolerance interceptor binding helps you write applications and services that are resilient to failure. Returning default responses in the event of an exception is a much more robust way to prevent unforeseen errors from bringing down the whole application.

Timeout

The *@Timeout* annotation of the Fault Tolerance API is used to stop the execution of a method after a set timeout. It is important that cloud native applications do not take a lot of time per given execution. Doing so could cost you money in terms of expensive resource consumption or result in poor performance and latency.

You can use the *@Timeout* annotation to tell the MicroProfile runtime to timeout the execution of a method or methods. The annotation can be used on a method or a class. When used at the class level, all methods in the class will timeout at the specified period. *@Timeout* can be used together with the other Fault Tolerance API constructs like the previously discussed *@Fallback*. Let us add a timeout to the `getCurrencyInfo` method.

```
@Fallback(fallbackMethod = "defaultCurrencyInfo")
@Timeout(500)
    public CurrencyInfo getCurrencyInfo(String country) {
    return geoFetch.getCurrencyInfo(country);
}
```

The above code snippet uses the *@Timeout* annotation with the value of 500. This means the `getCurrencyInfo` method will timeout after 500 milliseconds if it has not returned. It is possible to set the unit of time after which the timeout should occur. The default is `ChronoUnit.MILLIS`.

```
@Fallback(fallbackMethod = "defaultCurrencyInfo")
    @Timeout(value = 5, unit = ChronoUnit.SECONDS)
    @Bulkhead(5)
    public CurrencyInfo getCurrencyInfo(String country) {
        return geoFetch.getCurrencyInfo(country);
    }
```

In the above code snippet, the *@Timeout* annotation takes 2 parameters - the value being the number of units of time and the unit which is the unit of time, in this case seconds. So `getCurrencyInfo` will timeout after 5 seconds.

Using the *@Timeout* and *@Fallback* as shown above, the runtime will invoke the fallback method declared in the *@Fallback* annotation after the method times out. This is because the *@Timeout* annotation causes a org.eclipse.microprofile.faulttolerance.exceptions.`TimeoutException` to be thrown.

Retry Policy

Microservices have to connect and communicate with each other over the network. There are times when the network might not be very reliable and thus one service might need to retry again to connect to another network. The *@Retry* annotation from the Fault Tolerance API helps you achieve just that.

The *@Retry* annotation can take any combination of the following parameters:

- 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 variation to introduce into retry delays
- jitterDelayUnit: the jitter unit
- retryOn: specify the failures to retry on
- abortOn: specify the failures to abort on

Let us add a retry to our `fetchCurrencyInfo` method to make it more reliable should the network latency turn against us.

```
@Fallback(CurrencyInfoFallbackHandler.class)
    @Retry(retryOn = RuntimeException.class, delay = 400, maxDuration =
3200, jitter = 150, maxRetries = 10)
        public CurrencyInfo fetchCurrencyInfo(String country) {
        return geoFetch.getCurrencyInfo(country);
    }
```

The above code snippet uses the *@Retry* annotation with some parameters. The first one is the `retryOn`, `specifying RuntimeException.class`. This means the method invocation should only be retried when a RuntimeException is thrown. The `geoFetch.getCurrencyInfo` method can throw such an exception.

The second parameter is the delay annotation which we set 400. This means there could be 400 milliseconds between each retry. It wouldn't make that much sense to keep endlessly retrying, so `maxDuration` is set to 3200. This means the maximum duration for the entire retry operation is 3200 milliseconds.

The jitter value is set to 150 in the above code. The jitter is used to attain the effective delay between retries, to give it some kind of randomness. The effective delay will be between [delay - jitter, delay + jitter]. From the above code snippet, the effective delay is between 250 and 550 milliseconds.

Finally, the maxRetries is set to 10. This means the runtime will retry the `fetchCurrencyInfo` method 10 times with a delay of 400 and a jitter of 150, resulting in an effective delay of 250 to 550 milliseconds.

Finally, as the *@Fallback* annotation is also present, after all the retries are over, the runtime will pass execution to the handle method of the CurrencyInfoFallbackHandler class.

The *@Retry* annotation can be used on a class or a method. When used on a class, all methods of the class will be retried. When used on both a class and a method in the same class, that of the method will take precedence over that of the class.

Circuit Breaker

The circuit breaker is a way to protect system resources from being unnecessarily tied up on timeouts and waiting. It is a way to help systems fail fast. The *@CircuitBreaker* annotation can be placed on a class or a method. When placed on a class, then the circuit breaker will apply to all methods in the class. When placed on a class and a method in the same class, that of the method takes precedence. The *@CircuitBreaker* annotation can also be used in addition to the other Fault Tolerance annotations.

The circuit breaker as defined in the Fault Tolerance spec goes through 3 stages - open, half-open and closed.

- **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 the 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 success Threshold is reached the circuit will transition to open.

Let us add a circuit breaker to our fetchCurrencyInfo method, which we anticipate may come under heavy load at peak times.

```
    @Fallback(CurrencyInfoFallbackHandler.class)
    @Retry(retryOn = RuntimeException.class, delay = 400, maxDuration =
3200, jitter = 150, maxRetries = 10)
    @CircuitBreaker(successThreshold = 10, requestVolumeThreshold = 4,
failureRatio = 0.75, delay = 1000)
    public CurrencyInfo fetchCurrencyInfo(String country) {
        return geoFetch.getCurrencyInfo(country);
    }
```

The above code snippet declares the *@CircuitBreaker* annotation. As you can see, the *@CircuitBreaker* annotation is being used in addition to the *@Fallback* and *@Retry* annotations.

The first parameter to the *@CircuitBreaker* annotation is `successThreshold`, which is set to 10. This is the number of success cases that will return the circuit to a closed state. The requestVolumeThreshold, set to 4, is the number of consecutive requests in a rolling window that will trip the circuit. The failureRatio in the above code snippet is set to 0.75, meaning 75% of the requestVolumeThreshold of 4 failing will trigger the opening of the circuit. The delay is set to 1000 milliseconds, meaning after a delay of that span of time in the open state, the circuit will transition to a half-open state. Once in the half-open state, 10 (`successThreshold`) successful requests will get the circuit back in the closed state.

The *@CircuitBreaker* is a powerful component that can help you write microservices that are more resilient to sudden spikes in requests. Combining this with the other Fault Tolerance components above results in resilient cloud-native applications.

Asynchronous

The Asynchronous annotation allows the execution of a client request on a thread other than the one to which the request was sent. This means using a new thread to run the annotated *@Asynchronous* method. The annotation can be used both on a class and on a method. When used on a class, all methods in the class will be invoked asynchronously. When used on both a class and a method in the same class, that of the method takes precedence.

A method annotated with *@Asynchronous* must return a Future or a `CompletionStage` from the `java.util.concurrent` package. Otherwise, a `org.eclipse.microprofile.faulttolerance.exceptions.FaultToleranceDefinitionException` occurs.

Even though you can return either of Future or `CompletionStage` from a method annotated with *@Asynchronous*, the Fault Tolerance spec recommends the use of `CompletionStage` because it does not impact other Fault Tolerance annotations used on the same method. Let us examine the code snippet below which we will like to be invoked on a different thread.

```
    public CompletionStage<List<String>> getCountries() {
        final List<String> countryList = new ArrayList<>();
        Response response =
                webTarget.path("all").queryParam("fields", "name").
request(MediaType.APPLICATION_JSON).get();
        if (response.getStatus() != 200) {
            throw new RuntimeException("No countries found or an error
occurred. Sorry. Please try again");
        }
        JsonArray entity = response.readEntity(JsonArray.class);

        entity.getValuesAs(JsonObject.class).forEach(o -> countryList.
add(o.getString("name")));
        return CompletableFuture.completedFuture(countryList);

    }
```

Method getCountries returns a List of Strings wrapped in a CompletionStage. The rest of the method uses the JAX-RS client to make a request to an API for a list of countries in the world. This operation could be time consuming and therefore is something better run on a different thread.

```
    @Asynchronous
    @Timeout(value = 7, unit = ChronoUnit.SECONDS)
    @Fallback(fallbackMethod = "fallbackCountryList")
        public CompletionStage<List<String>> getCountryList() {
        return geoFetch.getCountries();
    }
```

The above code snippet shows the use of the `getCountries` method. Method `getCountryList` returns a `CompletionStage` wrapped List of Strings. This method is annotated with the *@Asynchronous* annotation. It is also annotated with *@Timeout* set to 7 seconds and *@Fallback*.

When method `getCountryList` is invoked, it will return immediately with a `CompletionStage` and the rest of the method will be run on a different thread. Because the method returns a `CompletionStage`, if an exception gets thrown or the `CompletionStage` returns exceptionally, the fallback method `fallBackCountryList` declared in the *@Fallback* annotation will be invoked. This method simply returns an empty List of Strings wrapped in a `CompletionStage` as shown below.

```
    public CompletionStage<List<String>> fallbackCountryList() {
        List<String> countryList = new ArrayList<>();

        return CompletableFuture.completedFuture(countryList);
    }
```

The signature of method `fallbackCountryList` is the same as the method `getCountryList` because a fallback method must have the same signature as the method for which it is acting as a fallback. Asynchronous method `getCountryList` will only complete successfully when the `CompletionStage` returned completes successfully.

Alternative @Asynchronous Annotations

Payara Server allows you to specify other annotations that should have the same effect as the *@Asynchronous* annotation from the Fault Tolerance API. These annotations do not have to be interceptor bindings. This is achieved by setting the value of the Payara Server Config property `MP_Fault_Tolerance_Alternative_Asynchronous_Annotations` to the fully qualified class name of the alternative annotation you would like to have the same effect as that of the Fault Tolerance *@Asynchronous.*

As an example, assuming you would want the *@Asynchronous* annotation from the EJB API in Jakarta EE to have the same effect as the Fault Tolerance *@Asynchronous,* you can do so as follows.

```
MP_Fault_Tolerance_Alternative_Asynchronous_Annotations=javax.ejb.Asynchronous
```

After setting the above Config property, the *@Asynchronous* annotation of the EJB API will have the same effect as the identically named annotation from the MicroProfile Fault Tolerance API.

Bulkhead

The bulkhead pattern is used to prevent a fault in one part of the system from cascading to the whole system. Using the *@Bulkhead* annotation, you can limit the number of concurrent requests that hit a component. The *@Bulkhead* annotation can be used together with the other Fault Tolerance annotations like @Asynchronous and *@Fallback*.

The annotation can be used on a class or a method. When declared on a class, all methods of the class will have the bulkhead policy applied. When applied to a class and a method in the same class, the bulkhead policy of the method takes precedence. The bulkhead pattern is only effective when applying *@Bulkhead* to a component that can be accessed from multiple contexts.

There are 2 approaches via which the bulkhead pattern is realized in Fault Tolerance using the *@Bulkhead* annotation. The first is thread pool isolation and the second is semaphore isolation.

Let us examine bulkhead through thread pool isolation in the following code snippet.

```
@Asynchronous
@Timeout(value = 7, unit = ChronoUnit.SECONDS)
@Fallback(fallbackMethod = "fallbackCountryList")
@Bulkhead(value = 5, waitingTaskQueue = 10)
public CompletionStage<List<String>> getCountryList() {
    return geoFetch.getCountries();
}
```

The above code declares the *@Bulkhead* annotation on method `getCountryList`. The *@Bulkhead* annotation is passed two parameters - value set to 5 and `waitingTaskQueue` set to 10. Also note that the method is annotated with *@Asynchronous.*

Thread pool isolation with *@Bulkhead* annotation can only be realized with a method annotated *@Asynchronous* as shown above. The value parameter of the *@Bulkhead* annotation specifies the number of concurrent requests that can access an instance of the annotated component. In this case there will be only 5 concurrent requests to `getCountryList` at any given time. The `waitingTaskQueue` refers to the maximum number of requests that will be available in the waiting queue. In the above code snippet, the task queue is set to 10.

The *@Bulkhead* use above, together with the *@Asynchronous* annotation will result in the use of the thread pool isolation method for attaining the bulkhead. When a request cannot be added to the waiting queue, a `org.eclipse.microprofile.faulttolerance.exceptions.BulkheadException` will be thrown. With the declaration of the *@Fallback* annotation on method `getCountryList` as shown above, when the `BulkheadException` gets thrown, the fallback method declared in that annotation will be invoked.

The semaphore isolation bulkhead approach can be used just by annotating a non Asynchronous method with the *@Bulkhead* annotation as shown below.

```
@Fallback(fallbackMethod = "defaultCurrencyInfo")
    @Timeout(value = 5, unit = ChronoUnit.SECONDS)
    @Bulkhead(5)
    public CurrencyInfo getCurrencyInfo(String country) {
        return geoFetch.getCurrencyInfo(country);
    }
```

The above code snippet shows the *@Bulkhead* annotation with a value of 5 on method `getCurrencyInfo`. The effect of this bulkhead usage is that only 5 concurrent requests will be allowed without a waiting task queue. This is the semaphore approach to realizing the bulkhead pattern.

Configuring MicroProfile Fault Tolerance

The Fault Tolerance API has tight integration with all the other APIs of the MicroProfile platform. All the Fault Tolerance APIs - *@Asynchronous, @Bulkhead, @CircuitBreaker, @Fallback, @Retry* and *@Timeout*- with the exception of *@Asynchronous,* do have parameters that you can override or set using the Config API.

Overriding Parameters Individually

You can set or override the parameters of individual Fault Tolerance API components using Config properties. The format for the property is `<classname>/<methodname>/<annotation>/<parameter>` where the class name refers to the fully qualified class name of the class containing the method with method name in it. Annotation refers to the annotation whose parameter you wish to set or override.

For example, let us override the value parameter of the *@Timeout* annotation of method `getCurrencyInfo` reproduced below.

```
@Fallback(fallbackMethod = "defaultCurrencyInfo")
 @Timeout(value = 5, unit = ChronoUnit.SECONDS)
 public CurrencyInfo getCurrencyInfo(String country) {
     return geoFetch.getCurrencyInfo(country);
 }
```

To do so, we set the property `fish.payara.control.Controller/getCurrencyInfo/Timeout/value` in any valid Config source, in this case `microprofile-config.properties` file as shown below.

```
fish.payara.control.Controller/getCurrencyInfo/Timeout/value=7
```

Setting the value of the value parameter of the *@Timeout* annotation this way will override what is specified in the class. Given the ordinal algorithm used by the Config API, you can set these same properties in different `ConfigSources` and have a very dynamic *@Timeout* annotation depending on the environment.

Overriding Parameters Globally

There are two ways you can globally set or override the parameter of Fault Tolerance annotations. They are the class level and the microservice level. To set or override the parameter of the *@Timeout* annotation for all occurrences in class Controller, you use the property `<classname>/<annotation>/<parameter>` as shown below.

```
fish.payara.control.Controller/Timeout/value=7
```

All you have to do to attain a class level setting or overriding of an annotation is to omit method name from the property as shown in the above properties file snippet. To set the value parameter of the *@Timeout* annotation for the entire microservice, set a value to the property without the class name as show below.

```
Timeout/value=7
```

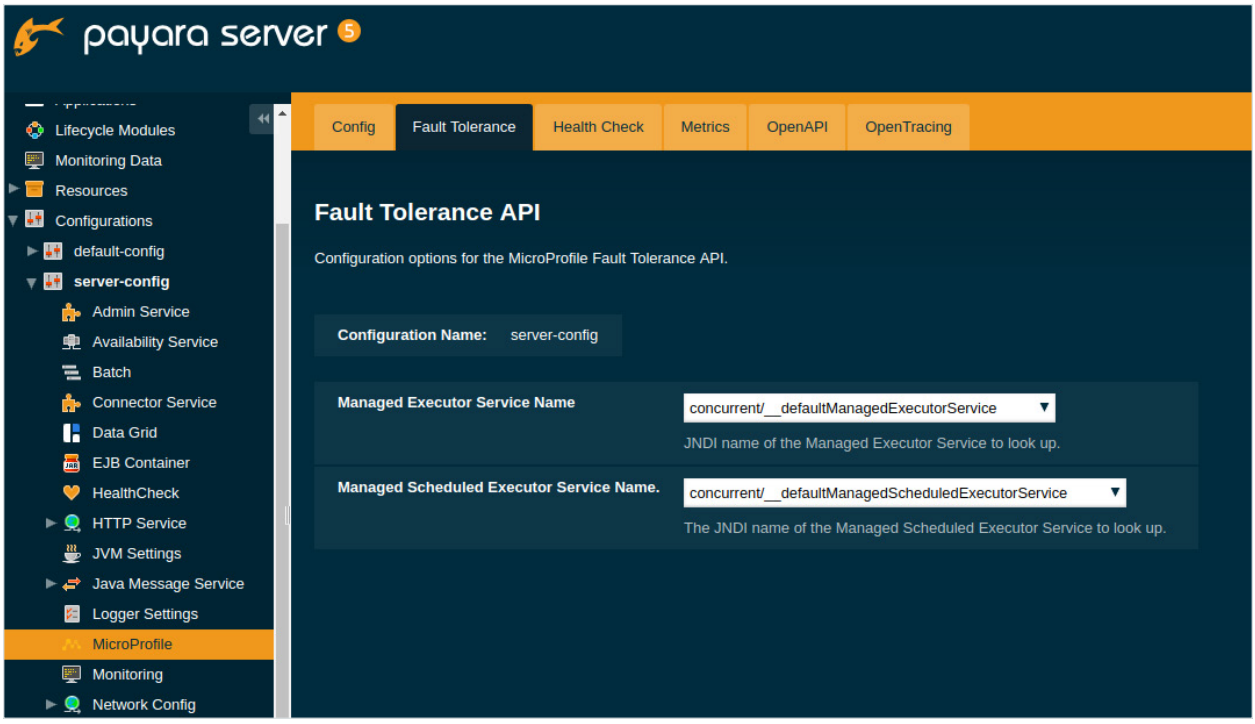
The above snippet sets - at the microservice level - the value parameter of the @Timeout annotation to 7. Configurations closest to a component take precedence over those farthest from it. For example, the value 10 in the property below will take precedence over the value 7 as set above.

```
fish.payara.control.Controller/getCurrencyInfo/Timeout/value=10
```

Payara Server Fault Tolerance Configuration

The Payara implementation of the Fault Tolerance specifications is deeply integrated into the application server.

You can choose the `ExecutorService` that will be used to run asynchronous components in the admin interface of the Payara Server as shown below.



Summary

In this guide, you have seen how you can use the six Fault Tolerance API components of *@Asynchronous, @Bulkhead, @CircuitBreaker, @Fallback, @Retry* and *@Timeout* to build highly resilient and fault tolerant microservices on the Payara Server platform.

You can use *@Asynchronous* for running a component on a different thread. *@Bulkhead* for isolating and preventing faults from cascading to other parts of your microservices. *@CircuitBreaker* to act as a breaker to manage sudden spikes in requests and to fail fast. *@Fallback* to declare an alternative method to call in the event of an exception. *@Retry* for retrying requests and *@Timeout* to prevent a given component from spending too much computer resources satisfying a given request. All these components work together to help you build powerful, cloud native Java Enterprise microservices.

MicroProfile is a very compelling community effort aimed at increasing the pace of innovation in enterprise Java software development. MicroProfile itself is an abstract spec, making it possible for various vendors to implement the base spec and add other custom features on top.

Payara Platform is a full Jakarta EE implementation that also implements the MP API and adds quite a number of features on top of it.

Should you need further support or info about using or transitioning your enterprise Java workload to the Payara Server, please don't hesitate to get in touch with us. You can always just say hi to us. We would love to hear from you.

Payara Enterprise is the fully supported and stable Edition of Payara Payara Platform with support provided directly by Payara's engineers.

You can also keep in touch with us on our social media platforms - Twitter, YouTube, GitHub.

sales@payara.fish

UK: +44 800 538 5490
Intl: +1 888 239 8941

www.payara.fish