

Quarkus - Fault Tolerance

One of the challenges brought by the distributed nature of microservices is that communication with external systems is inherently unreliable. This increases demand on resiliency of applications. To simplify making more resilient applications, Quarkus contains an implementation of the MicroProfile Fault Tolerance specification.

In this guide, we demonstrate usage of MicroProfile Fault Tolerance annotations such as `@Timeout`, `@Fallback`, `@Retry` and `@CircuitBreaker`.

Prerequisites

To complete this guide, you need:

- less than 15 minutes
- an IDE
- JDK 1.8+ installed with `JAVA_HOME` configured appropriately
- Apache Maven 3.6.3

The Scenario

The application built in this guide simulates a simple backend for a gourmet coffee e-shop. It implements a REST endpoint providing information about coffee samples we have on store.

Let's imagine, although it's not implemented as such, that some of the methods in our endpoint require communication to external services like a database or an external microservice, which introduces a factor of unreliability.

Solution

We recommend that you follow the instructions in the next sections and create the application step by step. However, you can go right to the completed example.

Clone the Git repository: `git clone https://github.com/quarkusio/quarkus-quickstarts.git`, or download an [archive](#).

The solution is located in the `microprofile-fault-tolerance-quickstart` directory.

Creating the Maven Project

First, we need a new project. Create a new project with the following command:

```
mvn io.quarkus:quarkus-maven-plugin:1.4.1.Final:create \
    -DprojectId=org.acme \
    -DprojectArtifactId=microprofile-fault-tolerance-quickstart \

-DclassName="org.acme.microprofile.faulttolerance.CoffeeResource" \
    -Dpath="/coffee" \
    -Dextensions="smallrye-fault-tolerance, resteasy-jsonb"
cd microprofile-fault-tolerance-quickstart
```

This command generates a Maven structure, importing the extensions for RESTEasy/JAX-RS and Smallrye Fault Tolerance, which is an implementation of the MicroProfile Fault Tolerance spec that Quarkus uses.

Preparing an Application: REST Endpoint and CDI Bean

In this section we create a skeleton of our application, so that we have something that we can extend and to which we can add fault tolerance features later on.

First, create a simple entity representing a coffee sample in our store:

```
package org.acme.microprofile.faulttolerance;

public class Coffee {

    public Integer id;
    public String name;
    public String countryOfOrigin;
    public Integer price;

    public Coffee() {
    }

    public Coffee(Integer id, String name, String countryOfOrigin,
Integer price) {
        this.id = id;
        this.name = name;
        this.countryOfOrigin = countryOfOrigin;
        this.price = price;
    }
}
```

Let's continue with a simple CDI bean, that would work as a repository of our coffee samples.

```

package org.acme.microprofile.faulttolerance;

import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;
import javax.enterprise.context.ApplicationScoped;

@ApplicationScoped
public class CoffeeRepositoryService {

    private Map<Integer, Coffee> coffeeList = new HashMap<>();

    public CoffeeRepositoryService() {
        coffeeList.put(1, new Coffee(1, "Fernandez Espresso",
"Colombia", 23));
        coffeeList.put(2, new Coffee(2, "La Scala Whole Beans",
"Bolivia", 18));
        coffeeList.put(3, new Coffee(3, "Dak Lak Filter", "Vietnam",
25));
    }

    public List<Coffee> getAllCoffeees() {
        return new ArrayList<>(coffeeList.values());
    }

    public Coffee getCoffeeById(Integer id) {
        return coffeeList.get(id);
    }

    public List<Coffee> getRecommendations(Integer id) {
        if (id == null) {
            return Collections.emptyList();
        }
        return coffeeList.values().stream()
            .filter(coffee -> !id.equals(coffee.id))
            .limit(2)
            .collect(Collectors.toList());
    }
}

```

Finally, edit the `org.acme.microprofile.faulttolerance.CoffeeResource` class as follows:

```

package org.acme.microprofile.faulttolerance;

import java.util.List;
import java.util.Random;
import java.util.concurrent.atomic.AtomicLong;
import javax.inject.Inject;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

import org.jboss.logging.Logger;

@Path("/coffee")
@Produces(MediaType.APPLICATION_JSON)
public class CoffeeResource {

    private static final Logger LOGGER = Logger.getLogger(
(CoffeeResource.class);

    @Inject
    private CoffeeRepositoryService coffeeRepository;

    private AtomicLong counter = new AtomicLong(0);

    @GET
    public List<Coffee> coffees() {
        final Long invocationNumber = counter.getAndIncrement();

        maybeFail(String.format("CoffeeResource#coffees()
invocation #%d failed", invocationNumber));

        LOGGER.infof("CoffeeResource#coffees() invocation #%d
returning successfully", invocationNumber);
        return coffeeRepository.getAllCoffees();
    }

    private void maybeFail(String failureLogMessage) {
        if (new Random().nextBoolean()) {
            LOGGER.error(failureLogMessage);
            throw new RuntimeException("Resource failure.");
        }
    }
}

```

At this point, we expose a single REST method that will show a list of coffee samples in a JSON format. Note that we introduced some fault making code in our `CoffeeResource#maybeFail()` method, which is going to cause failures in the `CoffeeResource#coffees()` endpoint method in about 50

% of requests.

Why not check that our application works? Run the Quarkus development server with:

```
./mvnw compile quarkus:dev
```

and open <http://localhost:8080/coffee> in your browser. Make couple of requests (remember, some of them we expect to fail). At least some of the requests should show us the list of our coffee samples in JSON, the rest will fail with a `RuntimeException` thrown in `CoffeeResource#maybeFail()`.

Congratulations, you've just made a working (although somewhat unreliable) Quarkus application!

Adding Resiliency: Retries

Let the Quarkus development server running and in your IDE add the `@Retry` annotation to the `CoffeeResource#coffees()` method as follows and save the file:

```
import org.eclipse.microprofile.faulttolerance.Retry;
...

public class CoffeeResource {
    ...
    @GET
    @Retry(maxRetries = 4)
    public List<Coffee> coffees() {
        ...
    }
    ...
}
```

Hit refresh in your browser. The Quarkus development server will automatically detect the changes and recompile the app for you, so there's no need to restart it.

You can hit refresh couple more times. Practically all requests should now be succeeding. The `CoffeeResource#coffees()` method is still in fact failing in about 50 % of time, but every time it happens, the platform will automatically retry the call!

To see that that the failures still happen, check the output of the development server. The log messages should be similar to these:

```
2019-03-06 12:17:41,725 INFO [org.acm.fau.CoffeeResource] (XNIO-1
task-1) CoffeeResource#coffees() invocation #5 returning
successfully
2019-03-06 12:17:44,187 INFO [org.acm.fau.CoffeeResource] (XNIO-1
task-1) CoffeeResource#coffees() invocation #6 returning
successfully
2019-03-06 12:17:45,166 ERROR [org.acm.fau.CoffeeResource] (XNIO-1
task-1) CoffeeResource#coffees() invocation #7 failed
2019-03-06 12:17:45,172 ERROR [org.acm.fau.CoffeeResource] (XNIO-1
task-1) CoffeeResource#coffees() invocation #8 failed
2019-03-06 12:17:45,176 INFO [org.acm.fau.CoffeeResource] (XNIO-1
task-1) CoffeeResource#coffees() invocation #9 returning
successfully
```

You can see that every time an invocation fails, it's immediately followed by another invocation, until one succeeds. Since we allowed 4 retries, it would require 5 invocations to fail in a row, in order for the user to be actually exposed to a failure. Which is fairly unlikely to happen.

Adding Resiliency: Timeouts

So what else have we got in MicroProfile Fault Tolerance? Let's look into timeouts.

Add following two methods to our `CoffeeResource` endpoint. Again, no need to restart the server, just paste the code and save the file.

```

import org.jboss.resteasy.annotations.jaxrs.PathParam;
import org.eclipse.microprofile.faulttolerance.Timeout;
...
public class CoffeeResource {
    ...
    @GET
    @Path("/{id}/recommendations")
    @Timeout(250)
    public List<Coffee> recommendations(@PathParam int id) {
        long started = System.currentTimeMillis();
        final long invocationNumber = counter.getAndIncrement();

        try {
            randomDelay();
            LOGGER.infof("CoffeeResource#recommendations()
invocation #%d returning successfully", invocationNumber);
            return coffeeRepository.getRecommendations(id);
        } catch (InterruptedException e) {
            LOGGER.errorf("CoffeeResource#recommendations()
invocation #%d timed out after %d ms",
                invocationNumber, System.currentTimeMillis() -
started);
            return null;
        }
    }

    private void randomDelay() throws InterruptedException {
        Thread.sleep(new Random().nextInt(500));
    }
}

```

We added some new functionality. We want to be able to recommend some related coffees based on a coffee that a user is currently looking at. It's not a critical functionality, it's a nice-to-have. When the system is overloaded and the logic behind obtaining recommendations takes too long to execute, we would rather time out and render the UI without recommendations.

Note that the timeout was configured to 250 ms, and a random artificial delay between 0 to 500 ms was introduced into the `CoffeeResource#recommendations()` method.

In your browser, go to <http://localhost:8080/coffee/2/recommendations> and hit refresh a couple of times.

You should see some requests time out with `org.eclipse.microprofile.faulttolerance.exceptions.TimeoutException`. Requests that do not time out should show two recommended coffee samples in JSON.

Adding Resiliency: Fallbacks

Let's make our recommendations feature even better by providing a fallback (and presumably faster) way of getting related coffees.

Add a fallback method to `CoffeeResource` and a `@Fallback` annotation to `CoffeeResource#recommendations()` method as follows:

```
import java.util.Collections;
import org.eclipse.microprofile.faulttolerance.Fallback;
...
public class CoffeeResource {
    ...
    @Fallback(fallbackMethod = "fallbackRecommendations")
    public List<Coffee> recommendations(@PathParam int id) {
        ...
    }

    public List<Coffee> fallbackRecommendations(int id) {
        LOGGER.info("Falling back to
RecommendationResource#fallbackRecommendations()");
        // safe bet, return something that everybody likes
        return Collections.singletonList(coffeeRepository
.getCoffeeById(1));
    }
    ...
}
```

Hit refresh several times on <http://localhost:8080/coffee/2/recommendations>. The `TimeoutException` should not appear anymore. Instead, in case of a timeout, the page will display a single recommendation that we hardcoded in our fallback method `fallbackRecommendations()`, rather than two recommendations returned by the original method.

Check the server output to see that fallback is really happening:

```
2020-01-09 13:21:34,250 INFO [org.acm.fau.CoffeeResource]
(executor-thread-1) CoffeeResource#recommendations() invocation #1
returning successfully
2020-01-09 13:21:36,354 ERROR [org.acm.fau.CoffeeResource]
(executor-thread-1) CoffeeResource#recommendations() invocation #2
timed out after 250 ms
2020-01-09 13:21:36,355 INFO [org.acm.fau.CoffeeResource]
(executor-thread-1) Falling back to
RecommendationResource#fallbackRecommendations()
```




The fallback method is required to have the same parameters as the original method.

Adding Resiliency: Circuit Breaker

A circuit breaker is useful for limiting number of failures happening in the system, when part of the system becomes temporarily unstable. The circuit breaker records successful and failed invocations of a method, and when the ratio of failed invocations reaches the specified threshold, the circuit breaker *opens* and blocks all further invocations of that method for a given time.

Add the following code into the `CoffeeRepositoryService` bean, so that we can demonstrate a circuit breaker in action:

```
import java.util.concurrent.atomic.AtomicLong;
import org.eclipse.micrometer.faulttolerance.CircuitBreaker;
...

public class CoffeeRepositoryService {
    ...

    private AtomicLong counter = new AtomicLong(0);

    @CircuitBreaker(requestVolumeThreshold = 4)
    public Integer getAvailability(Coffee coffee) {
        maybeFail();
        return new Random().nextInt(30);
    }

    private void maybeFail() {
        // introduce some artificial failures
        final Long invocationNumber = counter.getAndIncrement();
        if (invocationNumber % 4 > 1) { // alternate 2 successful
and 2 failing invocations
            throw new RuntimeException("Service failed.");
        }
    }
}
```

And inject the code bellow into the `CoffeeResource` endpoint:

```

public class CoffeeResource {
    ...
    @Path("/{id}/availability")
    @GET
    public Response availability(@PathParam int id) {
        final Long invocationNumber = counter.getAndIncrement();

        Coffee coffee = coffeeRepository.getCoffeeById(id);
        // check that coffee with given id exists, return 404 if
not
        if (coffee == null) {
            return Response.status(Response.Status.NOT_FOUND).
build();
        }

        try {
            Integer availability = coffeeRepository.
getAvailability(coffee);
            LOGGER.infof("CoffeeResource#availability() invocation
            #%d returning successfully", invocationNumber);
            return Response.ok(availability).build();
        } catch (RuntimeException e) {
            String message = e.getClass().getSimpleName() + ": " +
e.getMessage();
            LOGGER.errorf("CoffeeResource#availability() invocation
            #%d failed: %s", invocationNumber, message);
            return Response.status(Response.Status
.INTERNAL_SERVER_ERROR)
                .entity(message)
                .type(MediaType.TEXT_PLAIN_TYPE)
                .build();
        }
    }
    ...
}

```

We added another functionality - the application can return the amount of remaining packages of given coffee on our store (just a random number).

This time an artificial failure was introduced in the CDI bean: the `CoffeeRepositoryService#getAvailability()` method is going to alternate between two successful and two failed invocations.

We also added a `@CircuitBreaker` annotation with `requestVolumeThreshold = 4`. `CircuitBreaker.failureRatio` is by default 0.5, and `CircuitBreaker.delay` is by default 5 seconds. That means that a circuit breaker will open when 2 of the last 4 invocations failed and it will stay open for 5 seconds.

To test this out, do the following:

1. Go to <http://localhost:8080/coffee/2/availability> in your browser. You should see a number being returned.
2. Hit refresh, this second request should again be successful and return a number.
3. Refresh two more times. Both times you should see text "RuntimeException: Service failed.", which is the exception thrown by `CoffeeRepositoryService#getAvailability()`.
4. Refresh a couple more times. Unless you waited too long, you should again see exception, but this time it's "CircuitBreakerOpenException: getAvailability". This exception indicates that the circuit breaker opened and the `CoffeeRepositoryService#getAvailability()` method is not being called anymore.
5. Give it 5 seconds during which circuit breaker should close and you should be able to make two successful requests again.

Conclusion

MicroProfile Fault Tolerance allows to improve resiliency of your application, without having an impact on the complexity of our business logic.

All that is needed to enable the fault tolerance features in Quarkus is:

- adding the `smallrye-fault-tolerance` Quarkus extension to your project using the `quarkus-maven-plugin`:

```
./mvnw quarkus:add-extension -Dextensions="smallrye-fault-tolerance"
```

- or simply adding the following Maven dependency:

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-smallrye-fault-tolerance</artifactId>
</dependency>
```