Context Propagation in Quarkus

Traditional blocking code uses `ThreadLocal` variables to store contextual objects in order to avoid passing them as parameters everywhere. Many Quarkus extensions require those contextual objects to operate properly: RESTEasy, ArC and Transaction for example.

If you write reactive/async code, you have to cut your work into a pipeline of code blocks that get executed "later", and in practice after the method you defined them in have returned. As such, `try/finally` blocks as well as `ThreadLocal` variables stop working, because your reactive code gets executed in another thread, after the caller ran its `finally` block.

MicroProfile Context Propagation was made to make those Quarkus extensions work properly in reactive/async settings. It works by capturing those contextual values that used to be in thread-locals, and restoring them when your code is called.

Setting it up

If you are using SmallRye Reactive Streams Operators (the `quarkus-smallrye-reactive-streams-operators` module), you get automatic context propagation for all your streams, because the `quarkus-smallrye-context-propagation` module is automatically imported.

If you are not using SmallRye Reactive Streams Operators, add the following to your `pom.xml`:

```xml
<dependencies>
    <!-- Context Propagation extension -->
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-smallrye-context-propagation</artifactId>
    </dependency>
</dependencies>
```

With this, you will get context propagation for ArC, RESTEasy and transactions, if you are using them.

Usage example with SmallRye Reactive Streams Operators

If you are using SmallRye Reactive Streams Operators (the `quarkus-smallrye-reactive-streams-operators` module), you get context propagation out of the box, so, for example, let's write a REST endpoint that reads the next 3 items from a Kafka topic, stores them in a database using Hibernate ORM with Panache (all in the same transaction) before returning them to the client, you can do it like this:

```
    // Get the price-stream Kafka topic
    @Inject
    @Channel("price-stream") Publisher<Double> prices;

    @Transactional
    @GET
    @Path("/prices")
    public Publisher<Double> prices() throws SystemException {
        // get the next three prices from the price stream
        return ReactiveStreams.fromPublisher(prices)
                .limit(3)
                .map(price -> {
                    // store each price before we send them
                    Price priceEntity = new Price();
                    priceEntity.value = price;
                    // here we are all in the same transaction
                    // thanks to context propagation
                    priceEntity.persist();

                    return price;
                })
                .buildRs();
    }
```

Notice that thanks to automatic support for context propagation with Reactive Streams Operators this works out of the box.

Usage example for `CompletionStage`

If you are using `CompletionStage` you need manual context propagation. You can do that by injecting a `ThreadContext` or `ManagedExecutor` that will propagate every context. For example, here we use the Vert.x Web Client to get the list of Star Wars people, then store them in the database using Hibernate ORM with Panache (all in the same transaction) before returning them to the client:

```java
    @Inject ThreadContext threadContext;
    @Inject Vertx vertx;

    @Transactional
    @GET
    @Path("/people")
    public CompletionStage<Person> people() throws SystemException
 {
        // Create a REST client to the Star Wars API
        WebClient client = WebClient.create(vertx,
                        new WebClientOptions()
                         .setDefaultHost("swapi.co")
                         .setDefaultPort(443)
                         .setSsl(true));
        // get the list of Star Wars people, with context capture
        return threadContext.withContextCapture(client.get(
"/api/people/").send())
                .thenApply(response -> {
                    JsonObject json = response.bodyAsJsonObject();
                    List<Person> persons = new ArrayList<>(json
.getInteger("count"));
                    // Store them in the DB
                    // Note that we're still in the same
transaction as the outer method
                    for (Object element : json.getJsonArray(
"results")) {
                        Person person = new Person();
                        person.name = ((JsonObject)element)
.getString("name");
                        person.persist();
                        persons.add(person);
                    }
                    return persons;
                });
    }
```

Using `ThreadContext` or `ManagedExecutor` you can wrap most useful functional types and `CompletionStage` in order to get context propagated.

ℹ️ The injected `ManagedExecutor` uses the Quarkus thread pool.

Adding support for RxJava2

If you use Reactive Streams Operators (the `quarkus-smallrye-reactive-streams-operators` module), you get support for RxJava2 context propagation automatically, but if you don't, you may want to include the following modules to get RxJava2 support:

```xml
<dependencies>
    <!-- Automatic context propagation for RxJava2 -->
    <dependency>
        <groupId>io.smallrye</groupId>
        <artifactId>smallrye-context-propagation-propagators-
rxjava2</artifactId>
    </dependency>
    <!--
    Required if you want transactions extended to the end of
methods returning
    an RxJava2 type.
     -->
    <dependency>
        <groupId>io.smallrye.reactive</groupId>
        <artifactId>smallrye-reactive-converter-
rxjava2</artifactId>
    </dependency>
    <!-- Required if you return RxJava2 types from your REST
endpoints -->
    <dependency>
        <groupId>org.jboss.resteasy</groupId>
        <artifactId>resteasy-rxjava2</artifactId>
    </dependency>
</dependencies>
```