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 Mutiny (the `quarkus-mutiny` extension), you just need to add the the `quarkus-smallrye-context-propagation` extension to enable context propagation.

In other words, add the following dependencies to your `pom.xml`:

```xml
<dependencies>
    <!-- Mutiny and RestEasy support extensions if not already
included -->
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-mutiny</artifactId>
    </dependency>
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-resteasy-mutiny</artifactId>
    </dependency>
    <!-- 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 Mutiny

> *Mutiny*
>
> This section uses Mutiny reactive types, if you're not familiar with them, read the [Getting Started with Reactive guide](#) first.

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:

```java
// Get the prices stream
@Inject
@Channel("prices") Publisher<Double> prices;

@Transactional
@GET
@Path("/prices")
@Produces(MediaType.SERVER_SENT_EVENTS)
@SseElementType(MediaType.TEXT_PLAIN)
public Publisher<Double> prices() {
    // get the next three prices from the price stream
    return Multi.createFrom().publisher(prices)
            .transform().byTakingFirstItems(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;
                // the transaction is committed once the stream
completes
            });
}
```

Notice that thanks to Mutiny support for context propagation, this works out of the box. The 3 items are persisted using the same transaction and this transaction is committed when the stream completes.

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 as

JSON using JSON-B or Jackson:

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

    @Transactional
    @GET
    @Path("/people")
    @Produces(MediaType.APPLICATION_JSON)
    public CompletionStage<List<Person>> people() throws
SystemException {
        // Create a REST client to the Star Wars API
        WebClient client = WebClient.create(vertx,
                        new WebClientOptions()
                          .setDefaultHost("swapi.dev")
                          .setDefaultPort(443)
                          .setSsl(true));
        // get the list of Star Wars people, with context capture
        return
threadContext.withContextCapture(client.get("/api/people/").send())
                .thenApplyAsync(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;
                }, managedExecutor);
    }
```

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

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

Adding support for RxJava2

You need 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>
```