LAB 13: QUARKUS REACTIVE REVIEW

Autor: José Díaz

Github Repo: https://github.com/joedayz/quarkus-bcp-2025.git

Abre el proyecto **reactive-review-start**.

1. Open the application located in the `~/DO378/reactive-review` directory with an editor, such as VSCodium or vim.

   1.1. Navigate to the `~/DO378/reactive-review` directory.

```
[student@workstation ~]$ cd ~/DO378/reactive-review
```

   1.2. Open the project with an editor, such as VSCodium or vim.

```
[student@workstation reactive-review]$ codium .
```

2. Add the required dependencies to create a reactive endpoint that stores data in a PostgreSQL database, and sends events to Apache Kafka.
3. Configure the application to use four channels.

   - An incoming channel that consumes `SpeakerWasCreated` events from the `new-speakers-in` channel. Use the `speaker-was-created` Kafka topic to consume events. Set the `offset.reset` property of the incoming channel to `earliest`, and deserialize the incoming messages with the `com.redhat.training.serde.SpeakerWasCreatedDeserializer` class.

- An outgoing channel that publishes `SpeakerWasCreated` events to the `new-speakers-out` channel. Use the `speaker-was-created` Kafka topic to publish events.
- An outgoing channel that publishes `EmployeeSignedUp` events to the `employees-out` channel . Use the `employees-signed-up` Kafka topic to publish events.
- An outgoing channel that publishes `UpstreamMemberSignedUp` events to the `upstream-members-out` channel. Use the `upstream-members-signed-up` Kafka topic to publish events.

4. Create a reactive POST endpoint with the following requisites:
   - Receives a `Speaker` object as payload.
   - Stores the payload in the database by using a transaction.
   - Sends a `SpeakerWasCreated` event to the `new-speakers-out` channel.
   - Returns a `201` HTTP response that includes in the `location` response header the URI of the inserted element. The URI must follow the `/speakers/{id}` pattern.
5. Create an event processor that consumes and filters `SpeakerWasCreated` events.
   - If the affiliation is `RED_HAT`, then send a `EmployeeSignedUp` event to the `employees-out` channel.
   - If the affiliation is `GNOME_FOUNDATION`, then send a `UpstreamMemberSignedUp` event to the `upstream-members-out` channel.
   - Always acknowledge the message.
6. Execute the tests to validate the code changes. Optionally, you can use the Swagger UI to manually validate the changes before executing the tests.

Evaluation

As the `student` user on the `workstation` machine, use the `lab` command to grade your work. Correct any reported failures and rerun the command until successful.

```
[student@workstation reactive-review]$ lab grade reactive-review
```

Finish

Solución:

Instructions

The application for this exercise keeps records of conference speakers on a database, and uses reactive messaging to publish events about actions occurred in the application. The application uses Dev Services to start a PostgreSQL database and an Apache Kafka instance.

1. Open the application located in the `~/D0378/reactive-review` directory with an editor, such as VSCodium or vim.

    1.1. Navigate to the `~/D0378/reactive-review` directory.

```
[student@workstation ~]$ cd ~/D0378/reactive-review
```

    1.2. Open the project with an editor, such as VSCodium or vim.

```
[student@workstation reactive-review]$ codium .
```

2. Add the required dependencies to create a reactive endpoint that stores data in a PostgreSQL database, and sends events to Apache Kafka.

    2.1. Return to the terminal window, and use the Maven command to install the `quarkus-resteasy-reactive`, `quarkus-smallrye-reactive-messaging-kafka`, `quarkus-hibernate-reactive-panache` and `quarkus-reactive-pg-client` extensions.

```
[student@workstation reactive-review]$ mvn quarkus:add-extensions \
-Dextensions="resteasy-reactive,smallrye-reactive-messaging-kafka, \
 hibernate-reactive-panache,reactive-pg-client"
...output omitted...
[INFO] [SUCCESS] ... Extension io.quarkus:quarkus-reactive-pg-client has been
 installed
[INFO] [SUCCESS] ... Extension io.quarkus:quarkus-smallrye-reactive-messaging-
kafka has been installed
[INFO] [SUCCESS] ... Extension io.quarkus:quarkus-resteasy-reactive has been
 installed
[INFO] [SUCCESS] ... Extension io.quarkus:quarkus-hibernate-reactive-panache has
 been installed
...output omitted...
```

3. Configure the application to use four channels.

   - An incoming channel that consumes `SpeakerWasCreated` events from the `new-speakers-in` channel. Use the `speaker-was-created` Kafka topic to consume events. Set the `offset.reset` property of the incoming channel to `earliest`, and deserialize the incoming messages with the `com.redhat.training.serde.SpeakerWasCreatedDeserializer` class.
   - An outgoing channel that publishes `SpeakerWasCreated` events to the `new-speakers-out` channel. Use the `speaker-was-created` Kafka topic to publish events.
   - An outgoing channel that publishes `EmployeeSignedUp` events to the `employees-out` channel . Use the `employees-signed-up` Kafka topic to publish events.
   - An outgoing channel that publishes `UpstreamMemberSignedUp` events to the `upstream-members-out` channel. Use the `upstream-members-signed-up` Kafka topic to publish events.

   3.1. Open the `src/main/resources/application.properties` file, and then configure the `new-speakers-in` incoming channel.

   - Set the name of the incoming channel to `new-speakers-in`.
   - Use the `speaker-was-created` Kafka topic to consume events.
   - Set the `offset.reset` property of the incoming channel to `earliest`.
   - Deserialize the incoming messages with the `com.redhat.training.serde.SpeakerWasCreatedDeserializer` class.

```
...code omitted...

# Incoming Channels
mp.messaging.incoming.new-speakers-in.connector = smallrye-kafka
mp.messaging.incoming.new-speakers-in.topic = speaker-was-created
mp.messaging.incoming.new-speakers-in.auto.offset.reset = earliest
mp.messaging.incoming.new-speakers-in.value.deserializer =
 com.redhat.training.serde.SpeakerWasCreatedDeserializer
```

   3.2. Configure the `new-speakers-out` outgoing channel.

- Set the name of the outgoing channel to `new-speakers-out`.
- Use the `speaker-was-created` Kafka topic to publish events.
- Serialize the outgoing messages with the `io.quarkus.kafka.client.serialization.ObjectMapperSerializer` class.

```
...code omitted...

# Outgoing Channels
mp.messaging.outgoing.new-speakers-out.connector = smallrye-kafka
mp.messaging.outgoing.new-speakers-out.topic = speaker-was-created
mp.messaging.outgoing.new-speakers-out.value.serializer =
 io.quarkus.kafka.client.serialization.ObjectMapperSerializer
```

3.3. Configure the `employees-out` outgoing channel.

- Set the name of the outgoing channel to `employees-out`.
- Use the `employees-signed-up` Kafka topic to publish events.
- Serialize the outgoing messages with the `io.quarkus.kafka.client.serialization.ObjectMapperSerializer` class.

```
...code omitted...

# Outgoing Channels
...code omitted...
mp.messaging.outgoing.employees-out.connector = smallrye-kafka
mp.messaging.outgoing.employees-out.topic = employees-signed-up
mp.messaging.outgoing.employees-out.value.serializer =
 io.quarkus.kafka.client.serialization.ObjectMapperSerializer
```

3.4. Configure the `upstream-members-out` outgoing channel.

- Set the name of the outgoing channel to `upstream-members-out`.
- Use the `upstream-members-signed-up` Kafka topic to publish events.
- Serialize the outgoing messages with the `io.quarkus.kafka.client.serialization.ObjectMapperSerializer` class.

```
...code omitted...

# Outgoing Channels
...code omitted...
mp.messaging.outgoing.upstream-members-out.connector = smallrye-kafka
mp.messaging.outgoing.upstream-members-out.topic = upstream-members-signed-up
mp.messaging.outgoing.upstream-members-out.value.serializer =
 io.quarkus.kafka.client.serialization.ObjectMapperSerializer
```

4. Create a reactive POST endpoint with the following requisites:

   - Receives a `Speaker` object as payload.
   - Stores the payload in the database by using a transaction.
   - Sends a `SpeakerWasCreated` event to the `new-speakers-out` channel.
   - Returns a `201` HTTP response that includes in the `location` response header the URI of the inserted element. The URI must follow the `/speakers/{id}` pattern.

   4.1. Open the `SpeakerResource` class, and then add an `Emitter` variable to send `SpeakerWasCreated` events to the `new-speakers-out` channel.

```
package com.redhat.training.resource;

...code omitted...

@Path("/speakers")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class SpeakerResource {

    @Channel("new-speakers-out")
    Emitter<SpeakerWasCreated> emitter;

    ...code omitted...
}
```

   4.2. Create a POST endpoint that receives `Speaker` instances and returns a `Uni<Response>` instance.

```
...code omitted...

@Path("/speakers")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class SpeakerResource {

    @Channel("new-speakers-out")
    Emitter<SpeakerWasCreated> emitter;

    @POST
    public Uni<Response> create(Speaker newSpeaker) {
    }

...code omitted...
```

   4.3. Create the endpoint logic. Persist the `Speaker` instance within a transaction. On the inserted item, use the emitter to send a `SpeakerWasCreated` event, and then return a *created* response that includes a URI to the inserted element.

```
@POST
public Uni<Response> create(Speaker newSpeaker) {
    return Panache
        .<Speaker>withTransaction(newSpeaker::persist)
        .onItem()
        .transform(
            inserted -> {
                emitter.send(
                    new SpeakerWasCreated(
                        inserted.id,
                        newSpeaker.fullName,
                        newSpeaker.affiliation,
                        newSpeaker.email
                    )
                );

                return Response.created(
                    URI.create("/speakers/" + inserted.id)
                ).build();
            }
        );
}
```

5. Create an event processor that consumes and filters `SpeakerWasCreated` events.
   - If the affiliation is `RED_HAT`, then send a `EmployeeSignedUp` event to the `employees-out` channel.
   - If the affiliation is `GNOME_FOUNDATION`, then send a `UpstreamMemberSignedUp` event to the `upstream-members-out` channel.
   - Always acknowledge the message.

   5.1. Create a deserializer that transforms event messages from Apache Kafka to `SpeakerWasCreated` instances.
      - Call the class `SpeakerWasCreatedDeserializer`.
      - Create the entity in the `com.redhat.training.serde` package.

```
package com.redhat.training.serde;

import com.redhat.training.event.SpeakerWasCreated;
import io.quarkus.kafka.client.serialization.ObjectMapperDeserializer;

public class SpeakerWasCreatedDeserializer
        extends ObjectMapperDeserializer<SpeakerWasCreated> {
    public SpeakerWasCreatedDeserializer() {
        super(SpeakerWasCreated.class);
    }
}
```

   5.2. Open the `com.redhat.training.reactive.NewSpeakersProcessor` class, and then add two emitters.

- An emitter called `employeeEmitter` to send `EmployeeSignedUp` events to the `employees-out` channel.
- An emitter called `upstreamEmitter` to send `UpstreamMemberSignedUp` events to the `upstream-members-out` channel.

```
...code omitted...

@ApplicationScoped
public class NewSpeakersProcessor {
    private static final Logger LOGGER =
 Logger.getLogger(NewSpeakersProcessor.class);

    @Channel("employees-out")
    Emitter<EmployeeSignedUp> employeeEmitter;

    @Channel("upstream-members-out")
    Emitter<UpstreamMemberSignedUp> upstreamEmitter;

    ...code omitted...
}
```

5.3. Add a method called `sendEventNotifications` that processes `SpeakerWasCreated` messages, and returns a `CompletionStage<Void>` value.

- Set `new-speakers-in` as the incoming channel.
- If the speaker affiliation of the incoming event is `RED_HAT`, then send a `EmployeeSignedUp` event to the `employees-out` channel.
- If the speaker affiliation of the incoming event is `GNOME_FOUNDATION`, then send a `UpstreamMemberSignedUp` event to the `upstream-members-out` channel.
- Acknowledge the events.
- Use the `logEmitEvent()`, and `logProcessEvent()` methods to debug the logic.

```
@Incoming("new-speakers-in")
public CompletionStage<Void> sendEventNotifications(
        Message<SpeakerWasCreated> message
) {
    SpeakerWasCreated event = message.getPayload();

    logProcessEvent(event.id);

    if (event.affiliation == Affiliation.RED_HAT) {
        logEmitEvent("EmployeeSignedUp", event.affiliation);
        employeeEmitter.send(
                new EmployeeSignedUp(event.id, event.fullName, event.email)
        );
    } else if (event.affiliation == Affiliation.GNOME_FOUNDATION) {
        logEmitEvent("UpstreamMemberSignedUp", event.affiliation);
        upstreamEmitter.send(
                new UpstreamMemberSignedUp(event.id, event.fullName, event.email)
```

```
            );
        }

        return message.ack();
    }
```

6. Execute the tests to validate the code changes. Optionally, you can use the Swagger UI to manually validate the changes before executing the tests.

    6.1. Return to the terminal window, and then use the Maven command to execute the tests.

```
[student@workstation reactive-review]$ mvn clean test
...output omitted...
[INFO]
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0
[INFO]
...output omitted...
```

Evaluation

As the `student` user on the `workstation` machine, use the `lab` command to grade your work. Correct any reported failures and rerun the command until successful.

```
[student@workstation reactive-review]$ lab grade reactive-review
```

Finish

Run the `lab finish` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish reactive-review
```

This concludes the section.

enjoy!

Jose