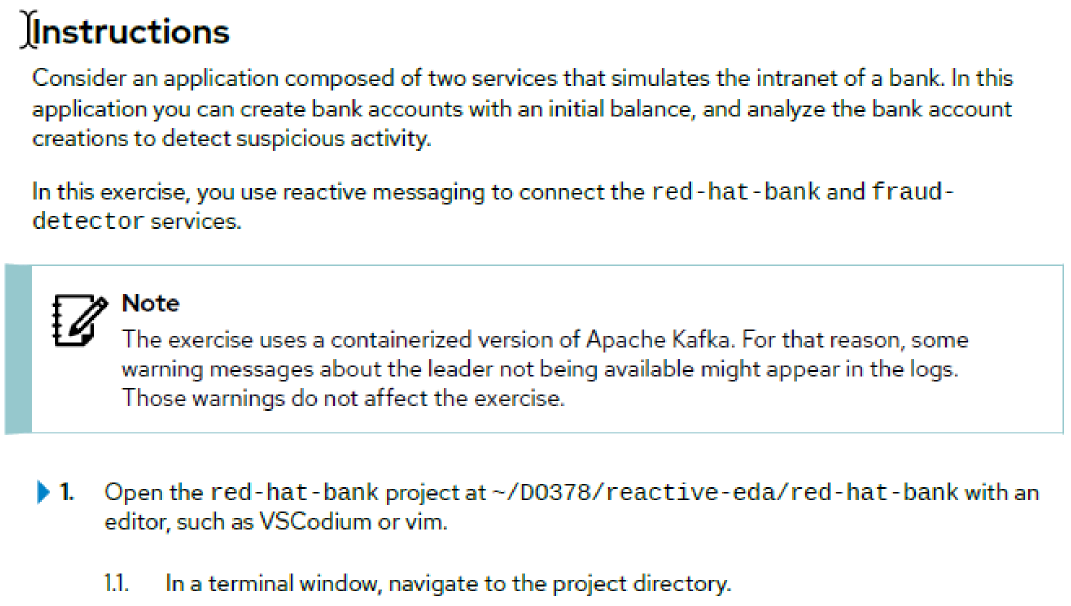
LAB 12: QUARKUS REACTIVE MESSAGING

Autor: José Díaz

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

1. Abre el proyecto **reactive-develop-start**.



Instructions

Consider an application composed of two services that simulates the intranet of a bank. In this application you can create bank accounts with an initial balance, and analyze the bank account creations to detect suspicious activity.

In this exercise, you use reactive messaging to connect the `red-hat-bank` and `fraud-detector` services.

Note
The exercise uses a containerized version of Apache Kafka. For that reason, some warning messages about the leader not being available might appear in the logs. Those warnings do not affect the exercise.

1. Open the `red-hat-bank` project at `~/DO378/reactive-eda/red-hat-bank` with an editor, such as VSCodium or vim.

   1.1. In a terminal window, navigate to the project directory.

```
[student@workstation ~]$ cd ~/DO378/reactive-eda/red-hat-bank
```

1.2. Open the project with your editor and examine the files.

- The `com.redhat.training.model.BankAccount` class is a Panache entity that models a bank account.
- The `com.redhat.training.resource.BankAccountsResource` class exposes two REST endpoints. One to get all the bank accounts from the database, and another that creates new bank accounts.

2. Include the Quarkus extension required to use SmallRye Reactive Messaging with Apache Kafka, and then configure the application.

2.1. Return to the terminal window, and use the Maven command to install the `quarkus-smallrye-reactive-messaging-kafka` extension.

```
[student@workstation red-hat-bank]$ mvn quarkus:add-extension \
-Dextensions=smallrye-reactive-messaging-kafka
...output omitted...
[INFO] [SUCCESS] ... Extension io.quarkus:quarkus-smallrye-reactive-messaging-
kafka has been installed
...output omitted...
```

2.2. Open the `src/main/resources/application.properties` file, and then set `localhost:9092` as the value for the `kafka.bootstrap.servers` configuration property.

```
...code omitted...

# Kafka Settings
kafka.bootstrap.servers = localhost:9092

...code omitted...
```

2.3. Configure an incoming channel.

- Set the name of the incoming channel to `new-bank-accounts-in`.
- Use the `bank-account-was-created` Kafka topic to receive events.
- Set the `offset.reset` property of the incoming channel to `earliest`.
- Deserialize the incoming messages with the `com.redhat.training.serde.BankAccountWasCreatedDeserializer` class.

```
...code omitted...

# Incoming Channels
mp.messaging.incoming.new-bank-accounts-in.connector = smallrye-kafka
mp.messaging.incoming.new-bank-accounts-in.topic = bank-account-was-created
mp.messaging.incoming.new-bank-accounts-in.auto.offset.reset = earliest
mp.messaging.incoming.new-bank-accounts-in.value.deserializer =
 com.redhat.training.serde.BankAccountWasCreatedDeserializer

...code omitted...
```

2.4. Configure an outgoing channel.

- Set the name of the outgoing channel to `new-bank-accounts-out`.
- Use the `bank-account-was-created` Kafka topic to send events.
- Set the Quarkus `ObjectMapperSerializer` class as the serializer for the outgoing messages.

```
...code omitted...

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

3. Create a class called `BankAccountWasCreated` that represents the event of creating a new account, and a deserializer for that event.

The event includes two fields:

- `id`: A `Long` type field that identifies the bank account.
- `balance`: A `Long` type field that indicates the bank account balance at the moment of creating the bank account.

3.1. Create a class that represents the event of creating a bank account:

- Call the class `BankAccountWasCreated`.
- Create the entity in the `com.redhat.training.event` package.
- The event must have a `Long id` and `Long balance` attributes.

```
package com.redhat.training.event;

public class BankAccountWasCreated {
    public Long id;
    public Long balance;

    public BankAccountWasCreated() {}
    public BankAccountWasCreated(Long id, Long balance) {
        this.id = id;
        this.balance = balance;
    }
}
```

3.2. Create a deserializer that transforms event messages from Apache Kafka to `BankAccountWasCreated` instances.

- Call the class `BankAccountWasCreatedDeserializer`.
- Create the entity in the `com.redhat.training.serde` package.

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

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

public class BankAccountWasCreatedDeserializer
        extends ObjectMapperDeserializer<BankAccountWasCreated> {
    public BankAccountWasCreatedDeserializer() {
        super(BankAccountWasCreated.class);
    }
}
```

4. Update the POST `/accounts` endpoint to send a `BankAccountWasCreated` event to the `new-bank-accounts-out` channel.

4.1. Open the `BankAccountsResource` class, and then add an `Emitter` variable to send `BankAccountWasCreated` events to the `new-bank-accounts-out` channel.

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

...code omitted...

@Path("/accounts")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class BankAccountsResource {

    @Channel("new-bank-accounts-out")
    Emitter<BankAccountWasCreated> emitter;

    @GET
    public Uni<List<BankAccount>> get() {
        return BankAccount.listAll(Sort.by("id"));
    }

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

4.2. Update the `sendBankAccountEvent()` method to use the emitter, and send `BankAccountWasCreated` events to Apache Kafka.

–

```
public void sendBankAccountEvent(Long id, Long balance) {
    emitter.send(new BankAccountWasCreated(id, balance));
}
```

4.3. Update the `create()` method to send `BankAccountWasCreated` events after inserting new records in the database.

```
@POST
public Uni<Response> create(BankAccount bankAccount) {
    return Panache
            .<BankAccount>withTransaction(bankAccount::persist)
            .onItem()
            .transform(
                inserted -> {
                    sendBankAccountEvent(inserted.id, inserted.balance);

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

5. Create a consumer of `BankAccountWasCreated` events to set the type of the bank account.

- If the balance is lower than `100000`, then the account type must be `regular`.
- Otherwise, the type must be `premium`.

You can use the `logEvent()` method to debug the processed events.

5.1. Open the `com.redhat.training.reactive.AccountTypeProcessor` class.

5.2. Update the `calculateAccountType()` method to return `premium` on balances higher than or equal to `100000`, and `regular`.

```
public String calculateAccountType(Long balance) {
    return balance >= 100000 ? "premium" : "regular";
}
```

5.3. Add a method called `processNewBankAccountEvents` that processes `BankAccountWasCreated` events, and returns `Uni<Void>` values.

- Set `new-bank-accounts-in` as the incoming channel.
- Annotate the method with the `@ActivateRequestContext` tag.
- Find records in the database by using the event ID.
- Use a session transaction to close the connection to the database after updating the records.

```
...code omitted...

@ApplicationScoped
public class AccountTypeProcessor {
    ...code omitted...

    @Incoming("new-bank-accounts-in")
    @ActivateRequestContext
    public Uni<Void> processNewBankAccountEvents(BankAccountWasCreated event) {
        String assignedAccountType = calculateAccountType(event.balance);

        logEvent(event, assignedAccountType);

        return session.withTransaction(
            t -> BankAccount.<BankAccount>findById(event.id)
                .onItem()
                .ifNotNull()
                .invoke(
                    entity -> entity.type = assignedAccountType
                ).replaceWithVoid()
        ).onTermination().call(() -> session.close());
    }

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

5.4. Return to the terminal window, and then use the command `mvn quarkus:dev` to start the application.

```
[student@workstation red-hat-bank]$ mvn quarkus:dev
...output omitted...
... INFO [io.quarkus] ... Listening on: http://localhost:8080
...output omitted...
```

6. Open the `fraud-detector` project at `~/DO378/reactive-eda/fraud-detector` with an editor, such as VSCodium or vim.

6.1. Open a new terminal window and navigate to the project directory.

```
[student@workstation ~]$ cd ~/DO378/reactive-eda/fraud-detector
```

6.2. Open the project with your editor and examine the files.

- The `com.redhat.training.event.BankAccountWasCreated` class represents events of bank account creations.
- The `com.redhat.training.event.FraudScoreWasCalculated` class represents the event of calculating a fraud score for a new bank account, based on the initial balance.
- The `com.redhat.training.event.HighRiskAccountWasDetected` class represents a high risk event.

- The `com.redhat.training.event.LowRiskAccountWasDetected` class represents a low risk event.
- The `com.redhat.training.serde.BankAccountWasCreatedDeserializer` class is a deserializer for the `BankAccountWasCreated` events.
- The `application.properties` file defines the configuration properties for the `new-bank-accounts-in` incoming channel, and for the `low-risk-alerts-out` and `high-risk-alerts-out` outgoing channels.

7. Create a fraud detection system that processes `BankAccountWasCreated` events.

- Process the incoming `BankAccountWasCreated` events.
- Calculate a fraud score for the incoming `BankAccountWasCreated` events.
- Use the fraud score to send `HighRiskAccountWasDetected` or `LowRiskAccountWasDetected` events to Kafka.

7.1. Open the `com.redhat.training.reactive.FraudProcessor` class, and then add an `Emitter` variable to send `LowRiskAccountWasDetected` events to the `low-risk-alerts-out` channel.

```
package com.redhat.training.reactive;

...code omitted...

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

    @Channel("low-risk-alerts-out")
    Emitter<LowRiskAccountWasDetected> lowRiskEmitter;

    private Integer calculateFraudScore(Long amount) {
        ...code omitted...;
    }

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

7.2. Add an `Emitter` variable to send `HighRiskAccountWasDetected` events to the `high-risk-alerts-out` channel.

```
package com.redhat.training.reactive;

...code omitted...

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

    @Channel("low-risk-alerts-out")
    Emitter<LowRiskAccountWasDetected> lowRiskEmitter;

    @Channel("high-risk-alerts-out")
    Emitter<HighRiskAccountWasDetected> highRiskEmitter;

    private Integer calculateFraudScore(Long amount) {
        ...code omitted...
    }

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

7.3. Add a method called `sendEventNotifications` that processes `FraudScoreWasCalculated` events, and returns `CompletionStage<Void>` values.

- Set `new-bank-accounts-in` as the incoming channel.
- If the fraud score of the incoming event is greater than `50`, then send a `HighRiskAccountWasDetected` event to the `high-risk-alerts-out` channel.
- If the fraud score of the incoming event is greater than `20` and lower than or equal to `50`, then send a `LowRiskAccountWasDetected` event to the `low-risk-alerts-out` channel.
- Otherwise, ignore the event.
- Use the `logBankAccountWasCreatedEvent()`, `logFraudScore()`, and `logEmitEvent()` methods to debug the logic.

```
package com.redhat.training.reactive;

...code omitted...

@ApplicationScoped
public class FraudProcessor {
    ...code omitted...

    @Incoming("new-bank-accounts-in")
    public CompletionStage<Void> sendEventNotifications(
            Message<BankAccountWasCreated> message
    ) {
        BankAccountWasCreated event = message.getPayload();

        logBankAccountWasCreatedEvent(event);

        Integer fraudScore = calculateFraudScore(event.balance);

        logFraudScore(event.id, fraudScore);

        if (fraudScore > 50) {
            logEmitEvent("HighRiskAccountWasDetected", event.id);
            highRiskEmitter.send(
                new HighRiskAccountWasDetected(event.id)
            );
        } else if (fraudScore > 20) {
            logEmitEvent("LowRiskAccountWasDetected", event.id);
            lowRiskEmitter.send(
                new LowRiskAccountWasDetected(event.id)
            );
        }

        return message.ack();
    }

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

7.4. Open a new terminal window, and then use the command `mvn quarkus:dev` to start the application.

```
[student@workstation fraud-detector]$ mvn quarkus:dev
...output omitted...
... INFO [io.quarkus] ... Listening on: http://localhost:8081
...output omitted...
```

8. Do a manual end-to-end test to verify the logic of the application.

Note
Due to eventual consistency, the event processor running in the back end might process events faster than the front end makes calls to the back end. For that reason, in some steps you might see the account type assigned immediately in the front end.

8.1. Open a browser and navigate to the URL http://localhost:8080.

8.2. In the Create a Bank Account area, type `5000` in the Initial Balance field, and click Create. Notice that the front end displays the created account at the bottom of the page without a type.

8.3. In the Create a Bank Account area, type `200000` in the Initial Balance field, and click Create. Notice that the front end displays the created account at the bottom of the page without a type.

8.4. Refresh the page, and verify that the processor updated the type of account in the background, and now the front end displays the account type.

The account with a balance of `5000` has the type `regular` type assigned, and the other account has the type `premium` assigned.

8.5. Return to the command line terminal that runs the `fraud-detector` service, and verify that the processor executed the logic to detect suspicious accounts.

```
...output omitted...
... [...FraudProcessor] (...) Received BankAccountWasCreated - ID: 1 Balance:
 5,000
... [...FraudProcessor] (...) Fraud score was calculated - ID: 1 Score: 25
... [...FraudProcessor] (...) Sending a LowRiskAccountWasDetected event for bank
 account #1
...output omitted...
... [...FraudProcessor] (...) Received BankAccountWasCreated - ID: 2 Balance:
 200,000
... [...FraudProcessor] (...) Fraud score was calculated - ID: 2 Score: 75
... [...FraudProcessor] (...) Sending a HighRiskAccountWasDetected event for bank
 account #2
...output omitted...
```

8.6. Press q, and close the terminal.

8.7. Return to the terminal that runs the `red-hat-bank` service, and then press q to stop the application.

Finish

On the `workstation` machine, use the `lab` 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-eda
```

This concludes the section.

enjoy!

Jose