

LAB 12: QUARKUS REACTIVE MESSAGING

Autor: José Díaz

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

1. Abre el proyecto **09-reactive-eda-start**.

Instrucciones

Considera una aplicación compuesta de dos servicios que simulan la intranet de un banco. En esta aplicación tú puedes crear cuentas de banco con un balance inicial, y analizar la creación de cuentas bancarias para detectar actividad sospechosa.

En este ejercicio, tu puedes usar **reactive messaging** para conectar los **servicios joedayz-bank** y **fraud-detector**.

NOTA: El servicio uso una versión contenerizada de Kafka, por esa razón, algunos mensajes de warning acerca de que el leader no está disponible, pueden aparecer en los logs. Estos warnings no afectan el ejercicio.

1. Abre el proyecto joedayz-bank con tu editor favorito.
 - 1.1. En una terminal de Windows, navega al directorio del proyecto

```
cd joedayz-bank
```
 - 1.2. Abre el proyecto con tu editor y examina los archivos
 - La clase **com.bcp.training.model.BankAccount** es una entidad Panache que modela una cuenta bancaria.
 - La clase **com.bcp.training.resource.BankAccountsResource** expone dos REST endpoints. Uno para obtener todas las cuentas de la base de datos, y otra que crea nuevas cuentas bancarias.
2. Agrega la dependencia requerida para usar SmallRye Reactive Messaging con Apache Kafka y luego configura la aplicación.
 - 2.1. Retorna a la terminal de Windows y usa el comando Maven para instalar la extensión quarkus-messaging-kafka.

mvn quarkus:add-extension -Dextensions=quarkus-messaging-kafka

- 2.2. Abre el `src/main/resources/application.properties` y configura el <http://localhost:9092> como el valor para la propiedad `kafka.bootstrap.servers`

```
...code omitted...  
  
# Kafka Settings  
kafka.bootstrap.servers = localhost:9092  
  
...code omitted...
```

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

- 2.3. Configura un incoming channel

- Establece el nombre del incoming channel a **new-bank-accounts-in**.
- Usa el kafka topic **bank-account-was-created** para recibir eventos
- Establece la propiedad **offset.reset** del incoming channel a **earliest**.
- Deserializa los mensajes entrantes con la clase **com.bcp.training.serde.BankAccountWasCreatedDeserializer**.

```
# 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.bcp.training.serde.BankAccountWasCreatedDeserializer
```

- 2.4. Configurar un outgoing channel

- Establece el nombre del outgoing channel a **new-bank-accounts-out**.
- Usa el t3pico **bank-account-was-created** para enviar eventos
- Establece la clase de quarkus **ObjectMapperSerializer** como el serializer para los mensajes outgoing.

```
# 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. Crea una clase llamada `BankAccountWasCreated` que representa el evento de crear una nueva cuenta y un deserializer para ese evento.

El evento incluye dos campos:

- `id`: un tipo `Long` que identifica la cuenta bancaria
- `balance`: un tipo `Long` que indica el balance de la cuenta al momento de la creación.

- 3.1. Crea una clase que representa el evento de crear una cuenta bancaria:
- Llama a la case BankAccountWasCreated
 - Crea la entidad en el paquete com.bcp.training.event
 - El evento debe tener un Long id y Long balance como atributos.

```
package com.bcp.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. Crear un deserializer que transforma mensajes tipo evento desde Apache Kafka a instancias BankAccountWasCreated.

- Llama a la clase BankAccountWasCreatedDeserializer
- Crea la entidad en el paquete com.bcp.training.serde.

```
package com.bcp.training.serde;

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

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

4. Actualiza el endpoint POST /accounts para enviar un evento BankAccountWasCreated al channel new-bank-accounts-out.

- 4.1. Abre la clase BankAccountsResource y luego agrega una variable Emitter para enviar un evento BankAccountWasCreated al channel new-bank-accounts-out.

```

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

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

```

- 4.2. Actualiza el método `sendBankAccountEvent()` para usar el emitter, y enviar eventos `BankAccountWasCreated` al Apache Kafka.

```

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

```

- 4.3. Actualizar el método `create()` para enviar eventos `BankAccountWasCreated` después de insertar nuevos registros a la base de datos.

```

@POST
public Uni<Response> create(@Valid BankAccount bankAccount) {
    // Validar que el balance no sea null
    if (bankAccount.balance == null) {
        return Uni.createFrom().item(
            Response.status(Response.Status.BAD_REQUEST)
                .entity("{\"error\": \"El balance no puede ser null\"}")
                .build()
        );
    }

    // Validar que el balance sea positivo
    if (bankAccount.balance <= 0) {
        return Uni.createFrom().item(
            Response.status(Response.Status.BAD_REQUEST)
                .entity("{\"error\": \"El balance debe ser un número positivo\"}")
                .build()
        );
    }

    return Panache
        .<BankAccount>withTransaction(bankAccount::persist)
        .onItem()
        .transform(
            inserted -> {
                sendBankAccountEvent(inserted.id,
inserted.balance);
                return Response.created(
                    URI.create("/accounts/" +
inserted.id)

```

```
        ).build();
    }

    );
}
```

5. Crea un consumidor de eventos `BankAccountWasCreated` para establecer el tipo de bank account.

- Si el balance es menor a 100000, entonces el tipo de cuenta debe ser regular.
- Caso contrario, el tipo debe ser premium.

Tu puedes usar el método `logEvent()` para debuggear los eventos procesados.

- 5.1. Abre la clase `com.bcp.training.reactive.AccountTypeProcessor`.
- 5.2. Actualiza el método `calculateAccountType()` para retornar `premium` cuando el balance es mayor o igual a 100000, y `regular` en caso contrario.

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

- 5.3. Agrega un método llamado `processNewBankAccountEvents` que procesa eventos `BankAccountWasCreated` y retorna valores `Uni<Void>`.

- Establece el incoming channel a `new-bank-accounts-in`.
- Anota el método con la anotación `@ActivateRequestContext`.
- Encuentra registros en la base de datos usando el event ID.
- Usa un `session transaction` para cerrar la conexión a la base de datos después de actualizar los registros.

```
@Incoming("new-bank-accounts-in")  
@ActivateRequestContext  
public Uni<Void> processNewBankAccountEvents(BankAccountWasCreated  
event){  
    String assignedAccountType = calculateAccountType(event.balance);  
  
    logEvent(event, assignedAccountType);  
  
    return session.withTransaction(  
        s -> BankAccount.<BankAccount>findById(event.id)  
            .onItem()  
            .ifNotNull()  
            .invoke(  
                entity -> entity.type =  
assignedAccountType)  
            .replaceWithVoid()  
    );  
}
```

- 5.4. Retorna a la terminal windows, y luego usa el comando `mvn quarkus:dev` para iniciar la aplicación.

mvn quarkus:dev

Listening on: `http://localhost:8080`

6. Abre el proyecto `fraud-detector` en `~/09-reactive-eda-start/fraud-detector` con tu editor favorito.

6.1. Abre una nueva terminal y navega al directorio de proyecto.

```
cd ~/09-reactive-eda-start/fraud-detector
```

6.2. Abre el Proyecto con tu editor y examina los archivos.

- La clase `com.bcp.training.event.BankAccountWasCreated` representa eventos de creación de bank accounts.
- La clase `com.bcp.training.event.HighRiskAccountWasDetected` representa un evento de alto riesgo.
- La clase `com.bcp.training.event.LowRiskAccountWasDetected` representa un evento de bajo riesgo.
- La clase `com.bcp.training.serde.BankAccountWasCreatedDeserializer` es un deserializer para los eventos `BankAccountWasCreated`.
- El archivo `application.properties` define la configuración para los channels de incoming como `new-bank-accounts-in` y para outgoing channels como `low-risk-alerts-out` y `high-risk-alerts-out`.

7. Crear un Sistema de detección de fraude que procesa eventos `BankAccountWasCreated`.

- Procesar los eventos incoming `BankAccountWasCreated`
- Calcular un fraud score para los eventos incoming `BankAccountWasCreated`
- Usar el fraud score para enviar eventos `HighRiskAccountWasDetected` o `LowRiskAccountWasDetected` a Kafka.

7.1. Abre la clase `com.bcp.training.reactive.FraudProcessor` y luego agrega una variable `Emitter` para enviar eventos `LowRiskAccountWasDetected` al channel de salida `low-risk-alerts-out`.

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

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

7.2. Agrega un variable `Emitter` para enviar eventos `HighRiskAccountWasDetected` al channel `high-risk-alerts-out`.

```
@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;
```

7.3. Agrega un método llamado `sendEventNotifications` que procesa eventos `FraudScoreWasCalculated` y retorna `CompletionStage<Void>`.

- Establece `new-bank-accounts-in` como el `incoming channel`.
- Si el `fraud score` del evento `incoming` es mayor a 50, entonces enviar un evento `HighRiskAccountWasDetected` al `channel high-risk-alerts-out`.
- Si el `fraud score` del evento `incoming` es mayor a 20 y menor o igual a 50, entonces enviar un evento `LowRiskAccountWasDetected` al `channel low-risk-alerts-out`.
- Caso contrario, ignorar el evento.
- Usa el `logBankAccountWasCreatedEvent()`, `logFraudScore()` y `logEmitEvent()` para debuggear la lógica.

```
@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();
}
```

- 7.4. Abre la terminal de windows, y luego usa el comando `mvn quarkus:dev` para iniciar la aplicación.

mvn quarkus:dev

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

8. Finalmente hagamos un test end-to-end para verificar la lógica de la aplicación.
 - 8.1. Abre el navegador y ve a el URL <http://localhost:8080>
 - 8.2. En el area Create Bank Account, ingresa 5000 en el campo Initial Balance, y clic en Create. Observa que el front end muestra la cuenta creada en la sección de abajo, pero, sin tipo de cuenta.
 - 8.3. En el area Create Bank Account, ingresa 200000 en el campo Initial Balance y clic en Create. Observa que el front end muestra la cuenta creada en la sección de abajo sin tipo de cuenta.
 - 8.4. Refresca la página y verifica que el processor actualizo el tipo de cuenta por detrás, y ahora el front end muestra los tipos de cuenta.
La cuenta con balance de 5000 tiene el tipo regular asignado, y el otro tipo de cuenta tiene el tipo premium asignado.
 - 8.5. Retorna a la terminal de linea de comandos que ejecuta el servicio fraud-detector y verifica que el processor ejecutó la lógica para detectar cuentas sospechosas.

```
...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. Presiona la letra q para salir y cierra la terminal.
- 8.7. Retorna a la terminal que ejecuta el servicio joedayz-bank y luego presiona q para detener la aplicación.

Felicitaciones has terminado el laboratorio.

Jose