User Guide

Develop Resilient Applications with Event-Driven Microservices

David Heffelfinger

The Payara® Platform - Production-Ready, Cloud Native and Aggressively Compatible.

Contents

Developing applications by following a microservices architecture has been a popular way of developing software for a while now. During this time, a series of design patterns and best practices have emerged. These new design patterns can be applied to avoid common microservices pitfalls, particularly, event-driven architectures have become a popular way to develop distributed systems with microservices.

In this guide, we will cover some of these common microservices design patterns and best practices, then we will see how these best practices can be implemented in Java EE with Payara Micro.

Design Patterns Typically Used in an Event-Driven Architecture

An event-driven architecture offers some advantages over traditional REST invocations, such as loose coupling and higher resilience. Loose coupling is achieved by communicating via an event store or message broker, microservices don't need to know about each other's existence. Resilience is also achieved by communicating via an event store or message broker, if the microservice meant to receive the message is down at the time the message is sent, the message will be stored in a message store, then received and processed when the receiving microservice comes back up.

When adopting a microservices architectural style, a series of design patterns have emerged that facilitate an event-driven architecture.

- The *Database per service* design pattern, which dictates that each microservice should have its own database.
- The *Saga* design pattern, which states that business transactions should be implemented as a series of local transactions, with individual microservices communicating with each other by publishing a message to a messaging system.
- The *Event Sourcing* design pattern states that changes to state or application data should be captured as a series of events, events are then stored in an event store or message broker, then interested subscribers receive the event and react accordingly.

Database Per Service

Microservices need to be loosely coupled so that they can be independently developed and deployed. Because of this, it isn't a good idea to share persistent data between microservices, as changes in a database schema would impact multiple code bases, slowing down development significantly. Having microservices share a common database has been described as an anti-pattern, known as

the “Distributed Monolith”. In order to have a true microservice architecture, each service must have its own, independent database.

Independent databases do not necessarily have to be separate instances of an RDBMS or NoSQL database, it could simply be having separate schemas for each service or having private database tables for each service in the same schema.

Saga

Since a best practice of design patterns is to have one database per service, transactions to each database are independent of each other. Because of this, a local transaction updating all relevant database tables is not possible. Additionally, since each microservice is independent of each other, a two-phase commit is not possible either. Not being able to easily roll back a single transaction and not being able to use two-phase commit presents a challenge if we have to roll back one or more related transactions. The Saga design pattern allows us to deal with situations like these.

In a nutshell, the Saga design pattern states that distributed transactions are implemented as local transactions for each microservice. If one of the transactions needs to roll back because of a system or business rule error, then a series of compensating transactions to undo the initial transactions are initiated, after all the compensating transactions are completed, the system is left in its initial state.

The Saga pattern generates an event stream in which some services generate events, and other services subscribe to those events, and react accordingly when receiving those events from the event stream.

Event Sourcing

The Event Sourcing design pattern dictates that changes to our application state are stored as a sequence of events. When using the Saga design pattern described in the previous section, each microservice stores an event into an event or message store, then other microservices listen to the event and react accordingly, typically committing a local transaction of their own to their own local database.

Java Message Service

In Java EE applications, messaging is typically implemented via the standard Java Message Service (JMS) Java EE API. The JMS API allows Java EE applications to interact with messaging systems such as message brokers or Message Oriented Middleware (MOM). Adhering to the JMS standard allows application developers to develop applications that communicate via messaging systems without tying the implementation to any product, neither a specific message broker nor a particular Java EE application server.

Message Brokers

Frequently, microservices are implemented as a series of RESTful web services, however, when implementing an event-driven architecture, it is common to use a message broker. Microservices generating events would post a message to the message broker, and other microservices that need to react to the event would retrieve said messages. Traditional message brokers include products such as Apache ActiveMQ and IBM WebSphere MQ, these kinds of products are sometimes referred to as Message Oriented Middleware (MOM). Most message brokers include a connector developed using the Java Connector Architecture (JCA) to allow Java EE applications developed using standards compliant APIs to interact with their products.

Java Connector Architecture (JCA)

Most message broker vendors include a resource archive (RAR) file developed using the Java Connector Architecture (JCA). This RAR file is then used by Java EE application developers and architects to interact with the products via standard Java EE APIs such as JMS and Message Driven Beans. Using RAR files allow Java EE applications utilizing these products to be developed in a portable way, allowing application developers to easily switch message brokers without having to alter their Java EE code.

Cloud and Non-Traditional Message Brokers

These days, deploying applications to the cloud has become very popular. Most cloud providers supply their own message broker, for example, Amazon Web Services provides Amazon Simple Queue Service (SQS), and Microsoft's Azure cloud provides the Azure Service Bus.

Additionally, there are non-traditional message brokers such as MQ Telemetry Transport (MQTT), designed for low bandwidth or unreliable networks, and Apache Kafka, which is optimized for processing a high volume of messages.

While most cloud and non-traditional message broker products provide a Java API that can be used to interact with them, these APIs are specific to each message broker product, coding against message broker-specific APIs ties the codebase to a specific message broker, making it difficult to migrate to other products should the need arise.

Payara Cloud Connectors

Since many popular cloud and non-traditional message brokers don't provide JCA adapters, Payara created a series of cloud connectors implemented as JCA RAR files. These RAR files can be used to interact with many popular cloud and non-traditional message brokers, including Apache Kafka, Amazon SQS, MQTT, and Azure Service Bus. These adapters are available in the Maven Central repository and to use them, they simply need to be declared as a dependency in our projects.

Payara Cloud Connectors allow Java EE application developers to write standards compliant, portable code interacting with these popular message brokers. Additionally, the RAR files provided by Payara Cloud Connectors do not tie application developers to any particular Java EE application server, while they can certainly be used with Payara Server, they can also be used with any Java EE compliant application server.

Example Using Payara Micro

So far, we've covered some common design patterns used when implementing an event-driven architecture with microservices, as well as explained how Java EE applications can leverage existing message brokers via the JMS API. Let's now develop an example application following an event driven architecture, our example will use Apache Kafka as our message broker.

Our example will be a simple simulation of a banking system, we will implement logic to successfully transfer 100 currency units (dollars, pounds, euros, etc.) from a checking account to a savings account. We will both generate a successful transaction and simulate an error condition so that we can see compensating transactions in action, illustrating how to implement the Saga design pattern both when things work as expected, and when there is an error condition.

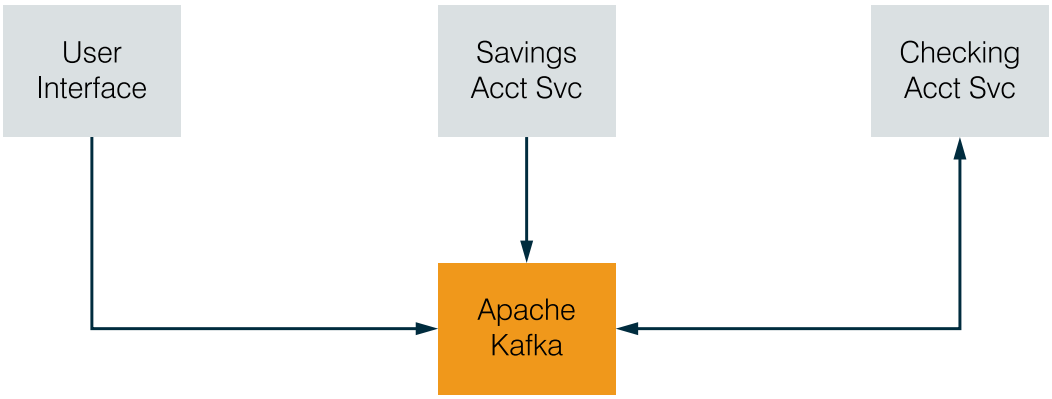


Diagram 1

Since we will be following an event-driven architecture in our application, different modules/services in our application will not communicate directly, instead, message producers will send messages to an Apache Kafka topic, message consumers will subscribe to these topics and retrieve and react to these messages. Diagram 1 provides a high-level view of our system architecture.

> ***We will only cover the most relevant sections of our example code, the complete codebase for the example can be obtained from GitHub on the Payara Examples Repository at https://github.com/payara/Payara-Examples***
>
> ***Code for our particular example can be found under https://github.com/payara/Payara-Examples/tree/master/payara-micro/payara-micro-event-sourcing-example***

Initial Setup

Before we can develop and deploy our application, we will need to install Apache Kafka and, if necessary, Payara Micro. Java 8 and Maven 3 are also required, we are going to assume those are already installed.

Obtaining Payara Micro

Our application will use Payara Micro 5 to take advantage of the new Java EE 8 APIs. At the time of publication, the latest version is 5.183.

Payara Micro 5.183, can be downloaded from https://www.payara.fish/downloads

Payara Micro 5.183 is distributed as an executable JAR file. It can be run directly from the command line by issuing the following command: `java -jar payara-micro-5.183.jar`

We will provide specific instructions on how to deploy the different modules in our application as we discuss them.

Obtaining Apache Kafka

Before we can deploy our application, we need to download Apache Kafka 2.0 from https://kafka.apache.org/downloads

Apache Kafka is distributed as a compressed .tgz file, extract to a directory of your choice.

Starting ZooKeeper

Apache Kafka depends on Apache ZooKeeper for much of its functionality, recent versions of Apache Kafka include ZooKeeper as part of the download bundle. Before we can start Kafka, we need to start ZooKeeper. Kafka comes with a handy shell script and configuration file that we can use to readily start ZooKeeper.

```
bin/zookeeper-server-start.sh config/zookeeper.properties
```

After a few seconds, ZooKeeper should start, after successfully starting, the last few lines of its console output should look similar to the following:

```
[2018-09-04 14:11:23,111] INFO tickTime set to 3000 (org.apache.zookeeper.
server.ZooKeeperServer)
[2018-09-04 14:11:23,111] INFO minSessionTimeout set to -1 (org.apache.
zookeeper.server.ZooKeeperServer)
[2018-09-04 14:11:23,111] INFO maxSessionTimeout set to -1 (org.apache.
zookeeper.server.ZooKeeperServer)
[2018-09-04 14:11:23,130] INFO Using org.apache.zookeeper.server.
NIOServerCnxnFactory as server connection factory (org.apache.zookeeper.server.
ServerCnxnFactory)
[2018-09-04 14:11:23,136] INFO binding to port 0.0.0.0/0.0.0.0:2181 (org.
apache.zookeeper.server.NIOServerCnxnFactory)
[2018-09-04 14:11:31,104] INFO Expiring session 0x100004a5f510000, timeout of
6000ms exceeded (org.apache.zookeeper.server.ZooKeeperServer)
[2018-09-04 14:11:31,105] INFO Processed session termination for sessionid:
0x100004a5f510000 (org.apache.zookeeper.server.PrepRequestProcessor)
```

Starting Apache Kafka

Once ZooKeeper is up and running, we can start Apache Kafka with the provided script and property file.

```
bin/kafka-server-start.sh config/server.properties
```

After a successful startup, the last few lines of Apache Kafka's console output should look similar to the following:

```
[2018-09-04 14:13:28,665] INFO [TransactionCoordinator id=0] Starting up.
(kafka.coordinator.transaction.TransactionCoordinator)
[2018-09-04 14:13:28,669] INFO [TransactionCoordinator id=0] Startup complete.
(kafka.coordinator.transaction.TransactionCoordinator)
[2018-09-04 14:13:28,684] INFO [Transaction Marker Channel Manager 0]: Starting
(kafka.coordinator.transaction.TransactionMarkerChannelManager)
[2018-09-04 14:13:28,799] INFO [/config/changes-event-process-thread]: Starting
(kafka.common.ZkNodeChangeNotificationListener$ChangeEventProcessThread)
[2018-09-04 14:13:28,809] INFO [SocketServer brokerId=0] Started processors for
1 acceptors (kafka.network.SocketServer)
[2018-09-04 14:13:28,819] INFO Kafka version : 2.0.0 (org.apache.kafka.common.
utils.AppInfoParser)
[2018-09-04 14:13:28,819] INFO Kafka commitId : 3402a8361b734732 (org.apache.
kafka.common.utils.AppInfoParser)
[2018-09-04 14:13:28,821] INFO [KafkaServer id=0] started (kafka.server.
KafkaServer)
[2018-09-04 14:13:28,665] INFO [TransactionCoordinator id=0] Starting up.
(kafka.coordinator.transaction.TransactionCoordinator)
```

Creating Topics

Our application will use the JMS Publish/Subscribe (pub/sub) messaging domain, therefore we need to create a couple of Kafka topics before we can successfully execute our code.

The first topic is called `checkingacct-topic`, our checking account service will subscribe to this topic to receive any messages sent to it. To create the topic, execute the following command in the command line:

```
bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1
    --partitions 1 --topic checkingacct-topic
```

The second topic is called `savingsacct-topic`, our savings account topic will subscribe to it so that it can process any messages sent to it. The topic can be created by entering the following command in the command line:

```
bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1
    --partitions 1 --topic savingsacct-topic
```

Setting up Our Dependencies

We will be using Apache Maven to build our application. Pom files for all modules in our project will need the Apache Kafka JCA adapter as a dependency, fortunately, the Payara Apache Kafka JCA adapter artifacts are in Maven Central, therefore no custom Maven repositories need to be added to our project.

The following snippet contains the Maven `pom.xml` for our user interface module, pom files for other modules are very similar.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.
w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/
POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <artifactId>payara-micro-event-sourcing-example</artifactId>
        <groupId>fish.payara.eventsourcing</groupId>
        <version>1.1</version>
    </parent>

    <groupId>fish.payara.eventsourcing</groupId>
    <artifactId>fundtransferui</artifactId>
    <version>1.1</version>
```

```
    <packaging>war</packaging>

    <name>fundtransferui</name>

    <dependencies>
        <dependency>
            <groupId>fish.payara.cloud.connectors.kafka</groupId>
            <artifactId>kafka-rar</artifactId>
            <version>0.3.0</version>
            <type>rar</type>
        </dependency>
        <dependency>
            <groupId>fish.payara.cloud.connectors.kafka</groupId>
            <artifactId>kafka-jca-api</artifactId>
            <version>0.3.0</version>
            <scope>provided</scope>
        </dependency>
        <!-- Other dependencies omitted for brevity -->
    </dependencies>

    <build>
        <plugins>
            <!-- Other plugins omitted for brevity -->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-dependency-plugin</artifactId>
                <version>3.1.1</version>
                <executions>
                    <execution>
                        <id>copy-dependencies</id>
                        <phase>package</phase>
                        <goals>
                            <goal>copy-dependencies</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

As we can see from the above `pom.xml` file, we need to add two dependencies to our project so that we can successfully interact with Apache Kafka in our code. Artifact ID's for the dependencies are `kafka-rar` and `kafka-jca-api`, both in the `fish.payara.cloud.connectors.kafka` group id.

> ***It is very important to make sure the the `kafka-jca-api` dependency has a scope of "provided", otherwise we won't be able to successfully deploy our application to Payara Micro***

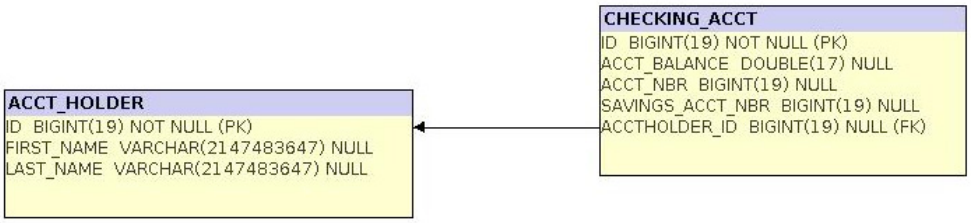
Another important thing to note about our `pom.xml` file is that we must use the Maven dependency plugin and configure it to copy our dependencies, they will end up in a directory called `dependency` under the `target` directory that Maven creates when building our code. The Kafka resource archive (RAR) file will end up in this directory, we will need it when deploying to Payara Micro, more on that later.

Database Per Service Implementation

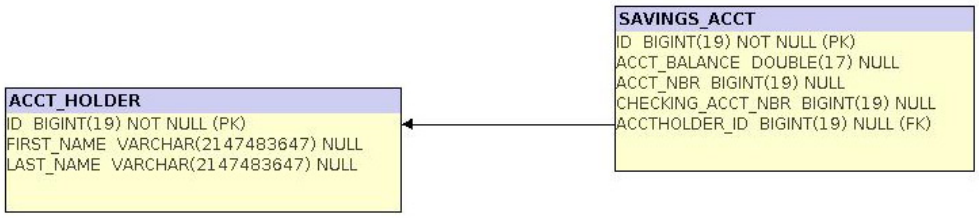
Our example will implement the Database Per Service design pattern. For simplicity, we will be using the H2 Database Engine, since it can be easily embedded into Java applications. We will develop a Savings Account Service and a Checking Account Service, both of which will have their own instance of an embedded H2 Database.

Both the Checking Account Service and the Savings Account Services have an eagerly loaded application scoped CDI bean that will automatically create and populate the database on deployment.

Both schemas are very simple, consisting of two tables each, the schema for the Checking Account Service Database looks as follows:



And the schema for the Savings Account Service Database looks as follows:



We will query the database before and after transferring funds, both successfully and when simulating an error condition, to verify that all account balances match what we are expecting.

Developing the User Interface

Our (overly simplistic) user interface will have two buttons, one to transfer funds successfully from a checking to a savings account, another one to simulate an error so that we can send compensating transactions to undo previously committed transactions (since we are following the "Database Per Service" design pattern and neither rolling back transactions nor two phase commit is possible).

We will be using Java Server Faces (JSF) 2.3, introduced in Java EE 8, to develop our user interface, it will consist of a Facelets page and a CDI bean acting as a controller.

Our Facelets page looks as follows:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.
w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core">
    <h:head>
        <title>Fund Transfer</title>
    </h:head>
    <h:body>
        <h:form>
            <h:messages id="msgs" showSummary="true" showDetail="false"/>
            <h:panelGrid columns="1">
                <h:commandButton value="Transfer funds from checkings to
savings"
```

```
                    actionListener="#{fundTransferController.transferFunds()}">
                      <f:ajax render="msgs"/>
                  </h:commandButton>
                  <h:commandButton value="Simulate transaction error"
                    actionListener=
                        "#{fundTransferController.simulateTransactionError()}">
                      <f:ajax render="msgs"/>
                  </h:commandButton>
              </h:panelGrid>
          </h:form>
      </h:body>
</html>
```

The above Facelets page has two buttons, the first one will send an Ajax request to our CDI bean to successfully transfer funds from a checking account to a savings account. The second one will simulate an error condition so that an already committed transaction can be undone by creating a compensating transaction.

Our controller bean receives requests from the Facelets page and processes them.

```
package fish.payara.eventsourcing.fundtransferui.controller;

//imports omitted for brevity

@Named
@ViewScoped
public class FundTransferController implements Serializable {

    @Resource(lookup = "java:module/env/KafkaConnectionFactory")
    private KafkaConnectionFactory kafkaConnectionFactory;

    @Inject
    private FacesContext facesContext;

    private static final Logger LOGGER =
        Logger.getLogger(FundTransferController.class.getName());

    public void transferFunds() {
        LOGGER.log(Level.INFO, String.format("{0}.transferFunds() invoked",
            this.getClass().getClass()));
        FundTransferDTO fundTransferDTO = new FundTransferDTO();
```

```
        fundTransferDTO.setSourceAcctType(AccountType.CHECKING);
        fundTransferDTO.setSourceAcctNbr(1234L);
        fundTransferDTO.setDestAcctType(AccountType.SAVINGS);
        fundTransferDTO.setDestAcctNbr(1123L);
        fundTransferDTO.setAmt(100.00);
        sendCheckingMessage(fundTransferDTO);
        facesContext.addMessage("msgs",
            new FacesMessage(FacesMessage.SEVERITY_INFO,
            "Transfer funds message sent successfully",
            "Transfer funds message sent successfully"));
    }

    public void simulateTransactionError() {
        LOGGER.log(Level.INFO,
            String.format("{0}.simulateTransactionError() invoked",
            this.getClass().getClass()));
        FundTransferDTO fundTransferDTO = new FundTransferDTO();

        fundTransferDTO.setSourceAcctType(AccountType.CHECKING);
        fundTransferDTO.setSourceAcctNbr(1234L); //checking account number
        fundTransferDTO.setDestAcctType(AccountType.SAVINGS);
        //invalid savings account number, will trigger sending
        //a compensating transaction to "rollback" the checking transaction
        fundTransferDTO.setDestAcctNbr(2222L);
        fundTransferDTO.setAmt(100.00); //amount to transfer
        sendCheckingMessage(fundTransferDTO);
        facesContext.addMessage("msgs",
            new FacesMessage(FacesMessage.SEVERITY_INFO,
            "Message simulating transaction error sent successfully.",
                "Message simulating transaction error sent successfully."));
    }

    private void sendCheckingMessage(FundTransferDTO fundTransferDTO) {
        String fundTransferDTOJson;

        fundTransferDTOJson =
            FundTransferDTOUtil.fundTransferDTOToJson(fundTransferDTO);

        try (KafkaConnection kafkaConnection =
            kafkaConnectionFactory.createConnection()) {
            kafkaConnection.send(new ProducerRecord("checkingacct-topic",
```

```
                    fundTransferDTOJson));
        } catch (Exception ex) {
            Logger.getLogger(getClass().getName()).log(Level.SEVERE, null, ex);
        }
    }
}
```

The `transferFunds()` method on our controller populates a Data Transfer Object (DTO) with relevant data such as the source and destination account number and type, plus the amount to transfer. It then invokes the private `sendCheckingMessage()` method, which converts the DTO to its JSON representation via the `fundTransferDTOToJson()` static method in the `FundTransferDTOUtil` utility class (not shown).

> *`FundTransferDTOUtil.fundTransferDTOToJson()`* ***uses the new Java API for JSON Binding (JSON-B) to convert the DTO to its equivalent JSON representation.***

The `sendCheckingMessage()` method then puts a message in the checking-cct-topic topic, to do this, it creates an instance of `fish.payara.cloud.connectors.kafka.api.KafkaConnection` by invoking the `createConnection()` method of the injected instance of `fish.payara.cloud.connectors.kafka.api.KafkaConnectionFactory`, then creating an instance of `org.apache.kafka.clients.producer.ProducerRecord`, passing the topic name and JSON representation of the DTO as parameters, and finally invoking the send() method of the created instance of `KafkaConnection`, passing the new `ProducerRecord` instance as a parameter.

Our checking service will subscribe to `checkingacct-topic`, therefore it will receive the message and process it accordingly.

The `simulateTransactionError()` in our controller is very similar to the transferFunds() method we just discussed, the only difference is that it sets the destination account number to a non-existing account number, which will eventually result in a compensating transaction being generated so that the funds that were withdrawn from the savings account can be recovered.

We can deploy our user interface module via the following command in the command line (substitute paths as necessary):

```
java -jar payara-micro-5.183.jar --deploy
    fundtransferui/target/dependency/kafka-rar-0.3.0.rar
    --deploy fundtransferui/target/fundtransferui-1.1.war --port 10080
    --noCluster
```

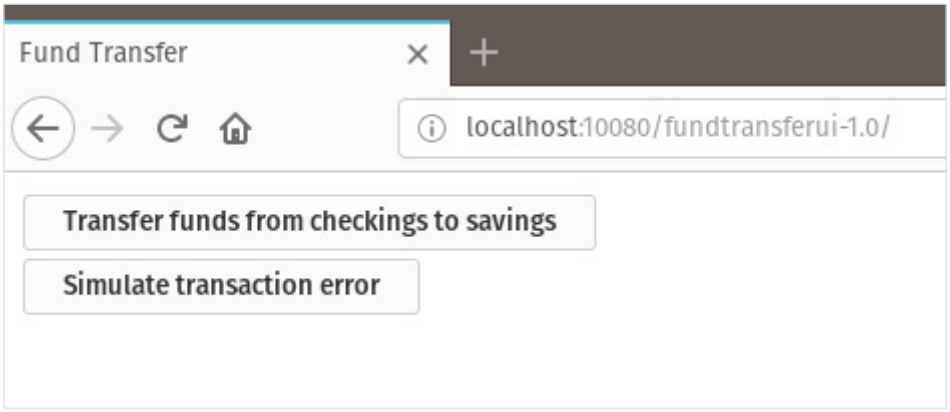
Notice that we need to deploy not only the WAR file containing our code, but also the RAR file containing the Apache Kafka JCA adapter. Deployment order is important, we need to deploy the RAR file first in order for our code to work properly.

If the module deployed successfully, we should see a few lines similar to the following in the Payara Micro output:

```
[2018-09-04T14:53:43.329+0100] [] [INFO] [] [PayaraMicro] [tid: _ThreadID=1 _
ThreadName=main] [timeMillis: 1536069223329] [levelValue: 800] [[

Payara Micro URLs:
http://localhost:10080/fundtransferui-1.1
```

Once we have successfully deployed our UI module to Payara Micro, we can point our browser to http://localhost:10080/fundtransferui-1.0/ to see our simple UI in action.



At this point we have successfully deployed our UI module, it will send messages to the Apache Kafka topic named `checkingacct-topic` we created earlier, however, there is nothing listening on that topic yet, so let's develop the Checking Account Service which will receive and process messages sent to that topic.

Developing the Checking Account Service

The entry point for our Checking Account Service is a message driven bean listening for messages on checkingacct-topic.

```
package fish.payara.eventsourcing.checking.messagelistener;

//imports omitted for brevity

@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName = "clientId", propertyValue =
        "checkingAcctFundTransferListener"),
    @ActivationConfigProperty(propertyName = "groupIdConfig", propertyValue =
        "fundTransfer"),
    @ActivationConfigProperty(propertyName = "topics", propertyValue =
        "checkingacct-topic"),
    @ActivationConfigProperty(propertyName = "bootstrapServersConfig",
        propertyValue = "localhost:9092"),
    @ActivationConfigProperty(propertyName = "autoCommitInterval",
        propertyValue = "100"),
    @ActivationConfigProperty(propertyName = "retryBackoff",
        propertyValue = "1000"),
    @ActivationConfigProperty(propertyName = "keyDeserializer",
        propertyValue =
        "org.apache.kafka.common.serialization.StringDeserializer"),
    @ActivationConfigProperty(propertyName = "valueDeserializer",
        propertyValue =
        "org.apache.kafka.common.serialization.StringDeserializer"),
    @ActivationConfigProperty(propertyName = "pollInterval", propertyValue =
        "1000"),})
public class CheckingFundTransferListener implements KafkaListener {

    private static final Logger LOGGER =
        Logger.getLogger(CheckingFundTransferListener.class.getName());

    @Resource(lookup = "java:comp/env/KafkaConnectionFactory")
    private KafkaConnectionFactory kafkaConnectionFactory;

    @Inject
    private CheckingAcctMgr checkingAcctMgr;
```

```
    @OnRecord(topics = {"checkingacct-topic"})
    public void transferFunds(ConsumerRecord consumerRecord) {
        String fundTransferDTOJson = (String) consumerRecord.value();
        FundTransferDTO fundTransferDTO =
            FundTransferDTOUtil.jsonToFundTransferDTO(fundTransferDTOJson);

        if (fundTransferDTO.getSourceAcctType().equals(AccountType.CHECKING)) {
            LOGGER.log(Level.INFO, String.format(
            "Withdrawing %.2f currency units from checking",
            fundTransferDTO.getAmt()));
            if (checkingAcctMgr.withdrawFunds(fundTransferDTO)) {
                try (KafkaConnection kafkaConnection =
                    kafkaConnectionFactory.createConnection()) {
                    kafkaConnection.send(new ProducerRecord("savingsacct-
topic",
                    fundTransferDTOJson));
                } catch (Exception ex) {
                    Logger.getLogger(getClass().getName()).log(Level.SEVERE,
                        null, ex);
                }
            } else {
                LOGGER.log(Level.WARNING,
                  "There was a problem withdrawing funds, aborting transfer.");
            }
        } else if (fundTransferDTO.getDestAcctType().equals(
                    AccountType.CHECKING)) {
            LOGGER.log(Level.INFO, String.format(
                "Depositing %.2f currency units to checking",
                fundTransferDTO.getAmt()));
            checkingAcctMgr.depositFunds(fundTransferDTO);
        }
    }
}
```

Since this class is a Message Driven Bean, it needs to be decorated with the standard EJB `@MessageDriven` annotation, in order for it to interact successfully with our Kafka instance, our Message driven bean needs to be configured via a number of `@ActivationConfigProperty` annotations.

- The `clientId` property is used to track the source of requests.
- The `groupIdConfigProperty` uniquely identifies a set of consumers within the same consumer group.

- The `topics` property is used to identify the topic our Message Driven Bean subscribes to.
- The `bootstrapServersConfig` property indicates the Kafka hostname or IP address and port.
- The `autoCommitInterval` property indicates how often (in milliseconds) consumer offsets are auto-committed to Kafka Kafka consumer offsets are a kind of unique identifier.
- The `retryBackoff` property indicates how long to wait before retrying to retrieve a message from a topic.

The `keyDeserializer` property indicates the deserializer class to use to deserialize message keys into Java objects, its value must be a class implementing `org.apache.kafka.common.serialization.Deserializer`.

- The `valueDeserializer` property indicates the deserializer class to use to deserialize message values into Java objects, its value must be a class implementing `org.apache.kafka.common.serialization.Deserializer`.
- The `pollInterval` property indicates how long to wait (in milliseconds) between polls to the topic to retrieve additional messages.

In order to successfully retrieve messages from Apache Kafka, our Message Driven Bean must implement `fish.payara.cloud.connectors.kafka.api.KafkaListener`, this interface contains no methods, it is a marker interface that indicates that this Message Driven Bean retrieves messages from Apache Kafka.

Finally, our Message Driven Bean must have a public method decorated with the `@OnRecord` annotation, this annotation has a `topics` attribute that accepts an array of strings indicating the topics that our Message Driven Bean listens to. The decorated method must accept a parameter of type `org.apache.kafka.clients.consumer.ConsumerRecord`, used to retrieve the message received from the queue. In our example, the decorated method is named `transferFunds()`, it verifies that the source account type in FundTransferDTO is of type “checking”, then updates the database, subtracting the amount indicated in the DTO from the account number indicated on the DTO. It then places a new message in `savingsacct-topic`, which is meant to be consumed by the Savings Account Service we will be developing. Our Message Driven Bean also checks if the destination account type is “checking”, if this is the case it updates the database, adding the amount indicated in the DTO to the checking account number specified in the DTO.

As we can see, our Message Driven Bean delegates the business logic to a class called `CheckingAcctMgr`, this class contains two simple methods that do simple math on the DAO properties and update the database via a class called `CheckingAcctFacade`, which acts as a Data Access Object (DAO) and interacts with the database via the Java Persistence API (JPA).

```
package fish.payara.eventsourcing.checking.business;

//imports omitted for brevity

@RequestScoped
@Transactional
public class CheckingAcctMgr implements Serializable {

    private static final Logger LOGGER =
        Logger.getLogger(CheckingAcctMgr.class.getName());

    @Inject
    private CheckingAcctFacade checkingAcctFacade;

    public boolean withdrawFunds(FundTransferDTO fundTransferDTO) {
        boolean success;
        CheckingAcct checkingAcct =
            checkingAcctFacade.findByAcctNbr(fundTransferDTO.
getSourceAcctNbr());

        if (fundTransferDTO.getAmt() <= checkingAcct.getAcctBalance()) {
            checkingAcct.setAcctBalance(checkingAcct.getAcctBalance() -
                fundTransferDTO.getAmt());
            checkingAcctFacade.edit(checkingAcct);
            success = true;
        } else {
            LOGGER.log(Level.WARNING, "Insufficient funds in checking
account");
            success = false;
        }

        return success;
    }

    public void depositFunds(FundTransferDTO fundTransferDTO) {
        CheckingAcct checkingAcct =
            checkingAcctFacade.findByAcctNbr(fundTransferDTO.getDestAcctNbr());

        checkingAcct.setAcctBalance(checkingAcct.getAcctBalance() +
            fundTransferDTO.getAmt());
        checkingAcctFacade.edit(checkingAcct);
    }
}
```

The logic for our `CheckingAcctMgr` class is very straightforward, therefore we will not discuss it in detail.

We can deploy our code by issuing the following command in the command line (substitute file paths as appropriate):

```
java -jar payara-micro-5.183.jar
    --deploy checking-acct-service/target/dependency/kafka-rar-0.3.0.rar
    --deploy checking-acct-service/target/checking-acct-service-1.1.war
    -noCluster
```

If the application deploys successfully, we should see a few lines similar to the following in the Payara Micro output:

```
[2018-09-04T15:01:38.687+0100] [] [INFO] [] [PayaraMicro] [tid: _ThreadID=1 _
ThreadName=main] [timeMillis: 1536069698687] [levelValue: 800] [[

{
    "Instance Configuration": {
        "Host": "localhost",
        "Http Port(s)": "8080",
        "Https Port(s)": "",
        "Instance Name": "payara-micro",
        "Instance Group": "no-cluster",
        "Deployed": [
            {
                "Name": "checking-acct-service-1.1",
                "Type": "war",
                "Context Root": "/checking-acct-service-1.1"
            },
            {
                "Name": "kafka-rar-0.3.0",
                "Type": "rar",
                "Context Root": "N/A"
            }
        ]
    }
Payara Micro URLs:
http://localhost:8080/checking-acct-service-1.1

]]
```

Now that we have successfully developed and deployed our Checking Account Service, it is time to start working on the Savings Account Service.

Developing the Savings Account Service

Just like the Checking Account Service, our Savings Account Service has a Message Driven Bean that listens for messages in a Kafka topic, in this case the name of the topic is `savingsacct-topic`.

```
package fish.payara.eventsourcing.checking.messagelistener;

//imports omitted for brevity

@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName = "clientId",
        propertyValue = "savingsAcctFundTransferListener"),
    @ActivationConfigProperty(propertyName = "groupIdConfig",
        propertyValue = "fundTransfer"),
    @ActivationConfigProperty(propertyName = "topics",
        propertyValue = "savingsacct-topic"),
    @ActivationConfigProperty(propertyName = "bootstrapServersConfig",
        propertyValue = "localhost:9092"),
    @ActivationConfigProperty(propertyName = "autoCommitInterval",
        propertyValue = "100"),
    @ActivationConfigProperty(propertyName = "retryBackoff",
        propertyValue = "1000"),
    @ActivationConfigProperty(propertyName = "keyDeserializer",
        propertyValue =
        "org.apache.kafka.common.serialization.StringDeserializer"),
    @ActivationConfigProperty(propertyName = "valueDeserializer",
        propertyValue =
        "org.apache.kafka.common.serialization.StringDeserializer"),
    @ActivationConfigProperty(propertyName = "pollInterval",
        propertyValue = "1000"),})
public class SavingsFundTransferListener implements KafkaListener {

    private static final Logger LOGGER =
        Logger.getLogger(SavingsFundTransferListener.class.getName());

    @Inject
    private SavingsAcctMgr savingsAcctMgr;
```

```
    @OnRecord(topics = {"savingsacct-topic"})
    public void transferFunds(ConsumerRecord consumerRecord) {
        String fundTransferDTOJson = (String) consumerRecord.value();
        FundTransferDTO fundTransferDTO =
          FundTransferDTOUtil.jsonToFundTransferDTO(fundTransferDTOJson);

        if (fundTransferDTO.getDestAcctType().equals(AccountType.SAVINGS)) {
            LOGGER.log(Level.INFO, String.format(
            "Depositing %.2f currency units to savings",
            fundTransferDTO.getAmt()));
            savingsAcctMgr.depositFunds(fundTransferDTO);
        } else if
           (fundTransferDTO.getSourceAcctType().equals(AccountType.SAVINGS)) {
                LOGGER.log(Level.INFO, String.format(
                "Withdrawing %.2f currency units from savings",
                fundTransferDTO.getAmt()));
            savingsAcctMgr.withdrawFunds(fundTransferDTO);
        }
    }
}
```

We already covered the properties specified on the array of `@ActivationConfigProperty` annotations when we discussed the Checking Account Service, it is worth mentioning that in this case, the value for the topics property is `savingsacct-topic`, since this is the topic that our Message Driven Bean will be listening to, unsurprisingly, we use this value as well as the value of the topics attribute of the `@OnRecord` annotation.

The logic in our Message Driven Bean is very similar to the logic in the Message Driven Bean for the Checking Account Service, our bean checks to see if the source or destination account type is of type “savings” and acts accordingly. Just like before, business logic is delegated to another class, in this case the class taking care of the business logic is called `SavingsAcctMgr`.

```
package fish.payara.eventsourcing.savings.business;

//imports omitted for brevity

@RequestScoped
@Transactional
public class SavingsAcctMgr implements Serializable {
```

```
private static final Logger LOGGER =
  Logger.getLogger(SavingsAcctMgr.class.getName());

@Resource(lookup = "java:module/env/KafkaConnectionFactory")
private KafkaConnectionFactory kafkaConnectionFactory;

@Inject
private SavingsAcctFacade savingsAcctFacade;

public void withdrawFunds(FundTransferDTO fundTransferDTO) {
    SavingsAcct savingsAcct =
        savingsAcctFacade.findByAcctNbr(fundTransferDTO.getDestAcctNbr());

    savingsAcct.setAcctBalance(savingsAcct.getAcctBalance() -
        fundTransferDTO.getAmt());
    savingsAcctFacade.edit(savingsAcct);
}

public void depositFunds(FundTransferDTO fundTransferDTO) {
    SavingsAcct savingsAcct =
        savingsAcctFacade.findByAcctNbr(fundTransferDTO.getDestAcctNbr());

    if (savingsAcct == null) {
        //invalid savings account, can't deposit,
        //need to send a compensating transaction to undo the withdrawal in
        //the savings account.
        LOGGER.log(Level.WARNING, String.format(
            "Received invalid savings account number: %d",
            fundTransferDTO.getDestAcctNbr()));

        //switch the source and destination accounts
        fundTransferDTO.setDestAcctType(AccountType.CHECKING);
        fundTransferDTO.setDestAcctNbr(fundTransferDTO.getSourceAcctNbr());
        fundTransferDTO.setSourceAcctType(AccountType.SAVINGS);
        fundTransferDTO.setSourceAcctNbr(fundTransferDTO.getDestAcctNbr());

        //generate a compensating transaction to re-deposit the funds into
        //the checking account
        try (KafkaConnection kafkaConnection =
            kafkaConnectionFactory.createConnection()) {
            kafkaConnection.send(new ProducerRecord("checkingacct-topic",
            FundTransferDTOUtil.fundTransferDTOToJson(fundTransferDTO)));
```

```
            } catch (Exception ex) {
                Logger.getLogger(getClass().getName()).log(Level.SEVERE, null,
                    ex);
            }
        } else {
            savingsAcct.setAcctBalance(savingsAcct.getAcctBalance() +
                fundTransferDTO.getAmt());
            savingsAcctFacade.edit(savingsAcct);
        }
    }
}
```

Our `SavingsAcctMgr` class has methods for depositing or withdrawing funds to or from a savings account. The `depositFunds()` method has some error checking in place, the first thing it does is retrieve savings account information from the database, if the savings account is not found, it modifies the DTO, switching the source and destination accounts, then places a message into `checkingacct-topic`, so that funds that were previously withdrawn from the checking account can be deposited back, effectively initiating a compensating transaction. If there are no error conditions our `SavingsAcctMgr` class simply deposits or withdraws from a savings account as appropriate.

At this point we are ready to deploy our application to Payara Micro, it can be deployed via the typical command in the command line (as usual, substitute file paths as appropriate):

```
java -jar payara-micro-5.183.jar
    --deploy savings-acct-service/target/dependency/kafka-rar-0.3.0.rar
    --deploy savings-acct-service/target/savings-acct-service-1.0.war
    --port 9080 --noCluster
```

Once the module has been deployed successfully, we should see a few lines similar to the following in the Payara Micro output:

```
[2017-12-21T12:48:43.800-0500] [] [INFO] [] [PayaraMicro] [tid: _ThreadID=1 _
ThreadName=main] [timeMillis: 1513878523800] [levelValue: 800] [[

Payara Micro URLs:
http://localhost:9080/savings-acct-service-1.1
```

Testing Our Application

Now that we have developed and deployed all modules in our application, it's time to take it for a spin. Before testing our functionality, let's take a look at the contents of the `CHECKING_ACCT` table in the Checking Account Service Database schema.

ID	ACCT_BALANCE	ACCT_NBR	SAVINGS_ACCT_NBR	ACCTHOLDER_ID
4	300.0	1234	1123	1
5	5000.0	2234	2123	2
6	100000.0	3234	3123	3

Of interest are the account numbers and amounts.

Similarly, the contents of the `SAVINGS_ACCT` schema are as follows:

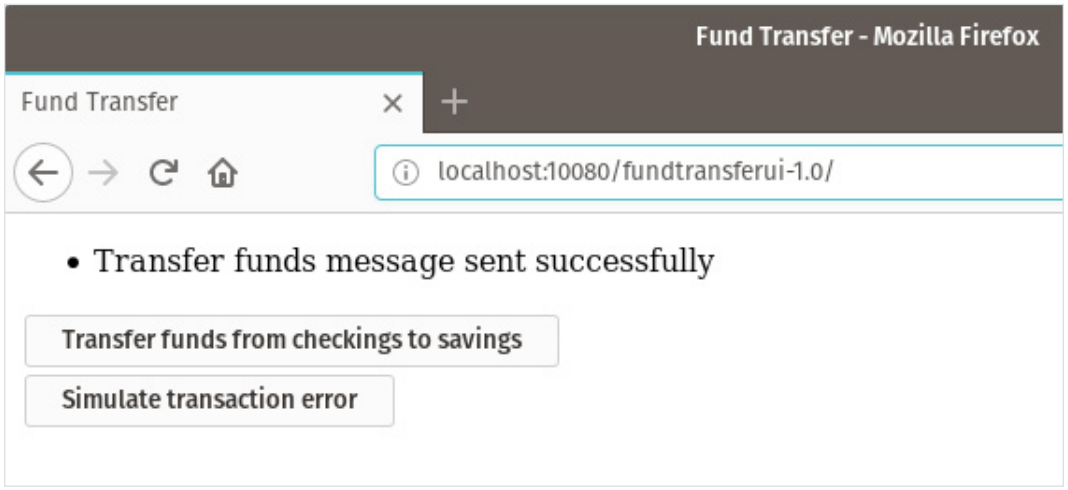
ID	ACCT_BALANCE	ACCT_NBR	SAVINGS_ACCT_NBR	ACCTHOLDER_ID
4	250.0	1123	1234	1
5	1000000.0	2123	2234	2
6	100000.0	3123	3234	3

Again, we're primarily interested in the `ACCT_BALANCE` and `ACCT_NBR` columns.

Testing a Successful Fund Transfer

We will first test a successful transfer from a checking to a savings account, do so, we point our browser to http://localhost:10080/fundtransferui-1.1/ and click on the button labelled "Transfer funds from checkings to savings".

If everything went well, we should see a success message at the top of the page.



If we look at the output of the Payara Micro instance where the Checking Account Service is deployed, we see the following entry:

```
[2018-09-04T15:07:29.488+0100] [] [INFO] [] [fish.payara.eventsourcing.
checking.messagelistener.CheckingFundTransferListener] [tid: _ThreadID=86 _
ThreadName=orb-thread-pool-1 (pool #1): worker-1] [timeMillis: 1536070049488]
[levelValue: 800] Withdrawing 100.00 currency units from checking
```

So far everything looks correct, now let's take another look at the CHECKING_ACCT table in the database schema for the Checking Account Service to make sure the database was updated correctly.

ID	ACCT_BALANCE	ACCT_NBR	SAVINGS_ACCT_NBR	ACCTHOLDER_ID
4	200.0	1234	1123	1
5	5000.0	2234	2123	2
6	100000.0	3234	3123	3

As we can see, checking account number 1234 now has a balance of 200 currency units, exactly 100 currently units less than before we started our test.

Let's now focus our attention to the Savings Account Service, in the output of the corresponding Payara Micro instance, we see the following entry:

```
[2018-09-04T15:07:29.821+0100] [] [INFO] [] [fish.payara.eventsourcing.
checking.messagelistener.SavingsFundTransferListener] [tid: _ThreadID=88 _
ThreadName=orb-thread-pool-1 (pool #1): worker-1] [timeMillis: 1536070049821]
[levelValue: 800] Depositing 100.00 currency units to savings
```

So far everything looks correct, let's now take a look at the SAVINGS_ACCT table in the database schema for our Savings Account Service to verify that its data was updated successfully:

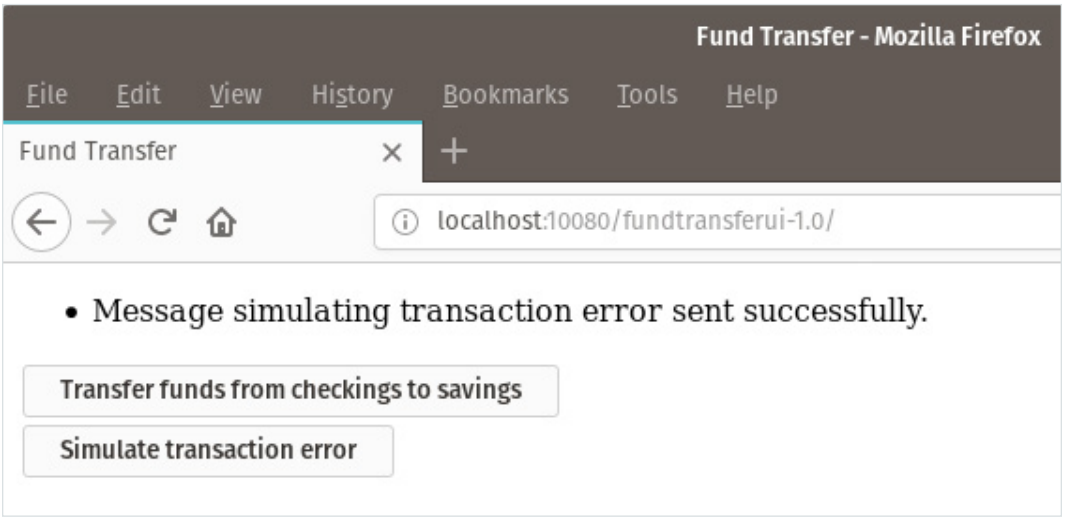
ID	ACCT_BALANCE	ACCT_NBR	SAVINGS_ACCT_NBR	ACCTHOLDER_ID
4	350.0	1123	1234	1
5	1000000.0	2123	2234	2
6	100000.0	3123	3234	3

Notice that the account balance for savings account number 1123 increased by exactly 100 currency units, the exact amount we just transferred from a checking account. We have now confirmed that we successfully transferred 100 currency units from a checking account to a savings account.

Testing Error Handling

Let's now turn our attention to generating compensating transactions to handle error conditions. As previously mentioned, our application simulates an error condition by attempting to transfer funds from a valid checking account to a non-existing savings account. Our Savings Account Service will detect the savings account number is invalid, then generate a compensating transaction to re-deposit the funds that were withdrawn from the savings account.

We simulate an error condition by clicking on the button labelled "Simulate transaction error" in our web-based user interface.



If we look at the output of the Savings Account Service, we see the following two entries:

```
[2018-09-04T15:09:42.067+0100] [] [INFO] [] [fish.payara.eventsourcing.
checking.messagelistener.CheckingFundTransferListener] [tid: _ThreadID=88 _
ThreadName=orb-thread-pool-1 (pool #1): worker-2] [timeMillis: 1536070182067]
[levelValue: 800] Withdrawing 100.00 currency units from checking

[2018-09-04T15:09:42.906+0100] [] [INFO] [] [fish.payara.eventsourcing.
checking.messagelistener.CheckingFundTransferListener] [tid: _ThreadID=89 _
ThreadName=orb-thread-pool-1 (pool #1): worker-3] [timeMillis: 1536070182906]
[levelValue: 800] Depositing 100.00 currency units to checking
```

Based on what we see in the output, our Checking Account Service received a message that prompted it to withdraw 100 currency units from a checking account, followed by another message that prompted it to deposit 100 currency units back into the same checking account. So far everything looks correct.

Turning our attention to the Savings Account Service, we see the following output:

```
[2018-09-04T15:09:42.148+0100] [] [SEVERE] [] [fish.payara.eventsourcing.jpa.
dao.SavingsAcctFacade] [tid: _ThreadID=90 _ThreadName=orb-thread-pool-1 (pool
#1): worker-2] [timeMillis: 1536070182148] [levelValue: 1000] Savings account
2222 not found
```

```
[2018-09-04T15:09:42.152+0100] [] [WARNING] [] [fish.payara.eventsourcing.
savings.business.SavingsAcctMgr] [tid: _ThreadID=90 _ThreadName=orb-thread-
pool-1 (pool #1): worker-2] [timeMillis: 1536070182152] [levelValue: 900]
Received invalid savings account number: 2222
```

As expected, the Savings Account Service received an invalid savings account number and handled the error condition properly, sending a message to `checkingacct-topic` that triggered a compensating transaction so that funds that were previously withdrawn from the checking account would be refunded.

If we query the `CHECKING_ACCT` we see that the account balance didn't change.

ID	ACCT_BALANCE	ACCT_NBR	SAVINGS_ACCT_NBR	ACCTHOLDER_ID
4	200.0	1234	1123	1
5	5000.0	2234	2123	2
6	10000.0	3234	3123	3

In this particular test case, we used an invalid savings account, therefore it would be very surprising the account balance on any savings account changed, but let's query the `SAVINGS_ACCT` table for good measure.

ID	ACCT_BALANCE	ACCT_NBR	SAVINGS_ACCT_NBR	ACCTHOLDER_ID
4	350.0	1123	1234	1
5	1000000.0	2123	2234	2
6	100000.0	3123	3234	3

Unsurprisingly, just like we expected, none of the balances on the **SAVINGS_ACCT** table changed.

We have now verified that all of our services communicate successfully by using Apache Kafka as a message broker. We have successfully implemented our application following an Event-Driven architecture as a series of microservices implementing the Database Per Service, Saga and Event Sourcing design pattern, made possible by the Kafka JCA adapter provided as part of Payara Cloud Connectors.

Conclusion

Following an event-driven architecture allows us to develop resilient applications as a series of microservices. Having our microservices communicate via a message broker provides benefits over direct RESTful calls, for example, if one of our microservices happens to be down when a message was sent to it, the message won't be lost and will be processed successfully when the server comes back up.

In this article, we covered some microservice design patterns that facilitate development of an Event-Driven architecture, such as Database Per Service, Saga, and Event Sourcing.

We then went through the Payara Cloud Connectors, which allow implementing Event-Driven architectures via standard Java EE APIs with non-traditional message brokers such as Apache Kafka, Amazon SQS, MQTT and Azure Service Bus.

Finally, we provided an example illustrating how to develop microservices in Java EE, leveraging Payara Cloud Connectors to implement an Event-Driven architecture.

Payara Micro – Small, Simple, Serious

Payara Micro is perfect for microservice and cloud environments. It enables you to run war files from the command line without any application server installation. It is small, less than 70 MB in size and incredibly simple to use. With its automatic and elastic clustering, Payara Micro is designed for running Java EE applications in a modern virtualized infrastructure.

That's not all! Using the Hazelcast integration each Payara Micro process will automatically cluster with other Payara Micro processes on the network, giving web session resilience and a fully distributed data cache using Payara's JCache support.

Payara Micro is fully compatible with Eclipse MicroProfile. It also comes with a Java API so it can be embedded and launched from your own Java applications.

Find out more & download Payara Micro from: www.payara.fish/payara_micro

About the Author

David Heffelfinger *is a Java Champion and Apache NetBeans committer, as well as an independent consultant focusing on Java EE. He is a frequent speaker at Java conferences and is the author of several books on Java and related technologies, such as "Java EE 8 Application Development", "Java EE 7 with GlassFish 4 Application Server" and others.*

David was named by TechBeacon as one of 39 Java leaders and experts to follow on Twitter. You can follow David on Twitter at @ensode.

sales@payara.fish **+44 207 754 0481** **www.payara.fish**