Apache Wayang meets Apache Kafka

# Intro

This article is the first of a three part series with focus on the summary of my Apache Kafka client implementation for Apache Wayang. We started with the Java Platform. An Apache Spark implementation follows (W.I.P.) in part two. The use case behind this work is an imaginary data collaboration scenario. One which we expect to emerge in the near future, as soon businesses stop moving data around the world but rather care about data locality while they expose and share specific information to others by using data federation. For this purpose, we illustrate a cross organizational data sharing scenario we may see in the finance sector soon.

# Background

Our goal is the implementation of a cross organization decentralized data processing scenario, in which protected local data should be processed in combination with public data from public sources in a collaborative manner. Instead of copying all data into a central data lake or a central data platform we decided to use federated analytics. Apache Wayang is the tool we work with. In our case, the public data is hosted on publicly available websites or data pods. A client can use the HTTP(S) protocol to read the data which is given in a well defined format. For simplicity we decided to use CSV format. When we look into the data of each participant we have a different perspective.  
Our processing procedure should calculate a particular metric on the “local data” of each participant. An example of such a metric is the average spending of all users on a particular product category per month. This can vary from partner to partner, hence, we want to be able to calculate a peer-group comparison so that each partner can see its own metric compared with a global average calculated from contributions by all partners. Such a process requires global averaging and local averaging. And due to governance constraints, we can’t bring all raw data together in one place.

Instead, we want to use Apache Wayang for this purpose. We simplify the procedure and split it into two phases. Phase one is the process, which allows each participant to calculate the local metrics. This requires only local data. The second phase requires data from all collaborating partners. The monthly sum and counter values per partner and category are needed in one place by all other parties. Hence, the algorithm of the first phase stores the local results locally, and the contributions to the global results in an externally accessible Kafka topic. We assume this is done by each of the partners. Now we have a scenario, in which an Apache Wayang plan must be able to read data from multiple Apache Kafka topics from multiple Apache Kafka clusters but finally writes into a single Kafka topic, which then can be accessed by all the participating clients.

A diagram of a company

Description automatically generated

The illustration shows the data flows in such a scenario. Jobs with red border are executed by the participants in isolation within their own data processing environments. But they share some of the data, using publicly accessible Kafka topics, marked by A. Job 4 is the Apache Wayang job in our focus: here we intent to read data from 3 different source systems, and write results into a fourth system (marked as B), which can be accesses by all participants again.

With this in mind we want to implement an Apache Wayang application which implements the illustrated “Job 4”. Since as of today, there is now KafkaSource and KafkaSink available in Apache Wayang, this will be our first step. Our assumption is, that in the beginning, there won’t be much data. Apache Spark is not required to cope with the load, but we expect, that in the future, a single Java application would not be able to handle our workload. Hence, we want to utilize the Apache Wayang abstraction over multiple processing platforms, starting with Java. Later, we want to switch to Apache Spark.

In this first part of the article series we describe the implementation of the Kafka Source and Kafka Sink component for Apache Wayang. We look into the “Read- and Write-Path” for our data items, called DataQuanta.

# Apache Wayang’s Read & Write Path for Kafka topics

To describe the read and write paths for data in the context of the created Apache Wayang code snippet, the primary classes and interfaces we need to understand are as follows:

**WayangContext**: This class is essential for initializing the Wayang processing environment. It allows you to configure the execution environment and register plugins that define which platforms Wayang can use for data processing tasks, such as Java.basicPlugin() for local Java execution.

**JavaPlanBuilder**: This class is used to build and define the data processing pipeline (or plan) in Wayang. It provides a fluent API to specify the operations to be performed on the data, from reading the input to processing it and writing the output.

## Read Path

The read path describes how data is ingested from a source into the Wayang processing pipeline:

**Reading from Kafka Topic:** The method readKafkaTopic(topicName) is used to ingest data from a specified Kafka topic. This is the starting point of the data processing pipeline, where topicName represents the name of the Kafka topic from which data is read.

**Data Tokenization and Preparation:** Once the data is read from Kafka, it undergoes several transformations such as **Splitting,** Filtering, and Mapping. What follows are the procedures known as **Reducing** and **Counting**.

## Write Path

Writing to Kafka Topic: The final step in the pipeline involves writing the processed data back to a Kafka topic using .writeKafkaTopic(...). This method takes parameters that specify the target Kafka topic, a serialization function to format the data as strings, and additional configuration for load profile estimation, which optimizes the writing process.

This read-write path provides a comprehensive flow of data from ingestion from Kafka, through various processing steps, and finally back to Kafka, showcasing a full cycle of data processing within Apache Wayang's abstracted environment and is implemented in our example program shown in listing 1.

<<< LISTING 1>>>

# Implementation of Input- and Output Operators

The next section shows how a new pair of operators can be implemented to extend Apache Wayang’s capabilities on the input and output side. We created the Kafka Source and Kafka Sink components so that our cross organizational data collaboration scenario can be implemented using data streaming infrastructure.

## Level 1 – Wayang execution plan with abstract operators

The implementation of our Kafka Source and Kafka Sink components for Apache Wayang requires new methods and classes on three layers. First of all in the API package. Here we use the **JavaPlanBuilder** to expose the function for selecting a Kafka topic as the source to be used by client. The **JavaPlanBuilder.scala** in package **org.apache.wayang.api** in the project **wayang-api/wayang-api-scala-java** exposes our new functionality to our external client.   
An instance of the JavaPlanBuilder is used to define the data processing pipeline. We use its **readKafkaTopic()** which specifies the source Kafka topic to read from, and for the write path we use the **writeKafkaTopic()** method. Both Methods do only trigger activities in the background.

For the output side, we use the **DataQuantaBuilder** class, which offers an implementation of the **writeKafkaTopic** function. This function is designed to send processed data, referred to as **DataQuanta**, to a specified Kafka topic. Essentially, it marks the final step in a data processing sequence constructed using the Apache Wayang framework.

In the **DataQuanta** class we implemented the methods **writeKafkaTopic** and **writeKafkaTopicJava** which use the **KafkaTopicSink** class. In this API layer we use the Scala programming language, but we utilize the Java classes, implemented in the layer below.

## Level 2 – Wiring between Platform Abstraction and Implementation

The second layer builds the bridge between the **WayangContext** and **PlanBuilders** which work together with **DataQuanta** and the **DataQuantaBuilder**.

Also, the mapping between the abstract components and the specific implementations are defined in this layer.

Therefore, the mappings package has a class **Mappings.java** in which all relevant input and output components are listed. We use it to register the **KafkaSourceMapping** and a **KafkaSinkMapping** for the particular platform, Java in our case. These classes allow the Wayang framework to use the Java implementation of the **KafkaTopicSource** component (and **KafkaTopicSink** respectively). While the Wayang execution plan uses the higher abstractions, here on the “platform level” we have to link the specific implementation for the target platform. In our case this leads to a Java program running on a JVM which is set up by the Wayang framework using the logical components of the execution plan.

Such mappings link the real implementation of our operators used in an execution plan.

The **JavaKafkaTopicSource** and the **JavaKafkaTopicSink** extend the **KafkaTopicSource** and **KafkaTopicSink** so that the lower level implementation of those classes become available within Wayang’s Java Platform context.

In this layer, the **KafkaConsumer** class and the **KafkaProducer** class are used, but both are configured and instantiated in the next layer underneath. All this is done in the project **wayang-plarforms/wayang-java**.

## Layer 3 – Input/Output Connector Layer

The **KafkaTopicSource** and **KafkaTopicSink** classes build the third layer of our implementation. Both are implemented in Java programming language. In this layer, the real Kafka-Client logic is defined. Details about consumer and producers, client configuration, and schema handling have to be handled here.

## Summary

Both classes in the third layer implement the Kafka client logic which is needed by the Wayang-execution plan when external data flows should be established. The layer above handles the mapping of the components at startup time. All this wiring is needed to keep Wayang open and flexible so that multiple external systems can be used in a variety of combinations and using multiple target platforms in combinations.

# Outlook

The next part of the article series will cover the creation of an Kafka Source and Sink component for the Apache Spark platform, which allows our use case to scale. Finally, in part three we bring all puzzles together, and show the full implementation of the multi organizational data collaboration use case.