LIGADATA

Fatafat 1.0 -
Implementing Custom Kafka Consumers and Producers

This document describes how to write custom input and output adapters, referred to here as "consumers" and "producers" that FATAFAT can use to process messages. Kafka adapters are provided with the code base, and therefore most references in this document are made to Apache Kafka. However the origin of messages themselves is not limited to any specific source, as long as the Trait/Interface described here is implemented correctly.

Implementing Custom Kafka Consumers

To implement a custom Kafka consumer/producer for Fatafat, these are the rules that need to be followed.

A custom Adapter should implement the "wrapper" for a Kafka SimpleConsumer as described by the Kafka documentation found here: http://kafka.apache.org. The reason for this is that Fatafat relies on the Kafka adapters to be able to handle specific Kafka partition offsets.

An example of a basic Kafka SimpleConsumer java implementation can be found here: (https://cwiki.apache.org/confluence/display/KAFKA/0.8.0+SimpleConsumer+Example)

To be able to use the Custom Adapter with the Fatafat Engine, the new adapter should have the following scala trait.

trait `com.ligadata.OnLEPBase.InputAdapter`

This will enforce the presence of the following required methods in the implementation class.

Message Signatures

The message signatures are as follows.

```
// Shutdown the adapter now.
def Shutdown: Unit

// Finish processing current work and stop.
def StopProcessing: Unit

// Start reading the queue with the following parameters.
def StartProcessing(
     maxParts: Int,
     partitionInfo: Array[
       (PartitionUniqueRecordKey,
        PartitionUniqueRecordValue,
        Long,
        (PartitionUniqueRecordValue, Int, Int)
```

```
        )
       ],
      ignoreFirstMsg: Boolean): Unit

// Gets an array of PartitionUniqueRecordKey, each describing a
//partition of a Kafka topic being read.
def GetAllPartitionUniqueRecordKey: Array[PartitionUniqueRecordKey]

// Convert the KEY in a string format into the
KafkaPartitionUniqueRecordKey
def DeserializeKey(k: String): PartitionUniqueRecordValue

// Convert the Value in a string format into the
KafkaPartitionUniqueRecordValue
def DeserializeValue(v: String): PartitionUniqueRecordValue

def getAllPartitionBeginValues: Array[
(PartitionUniqueRecordKey,  PartitionUniqueRecordValue)]

def getAllPartitionEndValues: Array[
(PartitionUniqueRecordKey,  PartitionUniqueRecordValue)]
```

Method Definitions

Here are the details about each method defined in the scala trait.

Shutdown/StopProcessing – these methods stop the execution of all threads that are involved in reading from a Kafka Topic.

GetAllPartitionUniqueRecordKey – This method is called by the Fatafat engine on an instance of a `KafkaInputAdapter`. The output of this method needs to be an array of `KafkaPartionUniqueRecordKey` objects. Each object will need to have a `PartitionId` that Kafka has stored for this given Topic. Each object also needs to have the topic name. This information will be used to call the `GetAllPartitionBeginValues/GetAllpartionEndValue`s method to find out the offsets within a Kafka partition from where to start processing.

GetAllPartitionBeginValues/GetAllpartionEndValues – this will return an array of `KafkaPartionUniqueRecordKey`, `KafkaPartionUniqueRecordValue` pairs. The `KafkaPartionUniqueRecordKey` will have information about a specific partition ID, while the `KafkaPartionUniqueRecordKey` will have information about the offset for the partition in question. After the Fatafat engine gets this array, it will issue the method below, `StartProcessing`.

StartProcessing - This call will start processing of the specified partitions.

- maxParts - This should be the maximum Partitions that the Fatafat engine wants to monitor.

- TBD
- ignoreFirst - if the value is TRUE, then the adapter should not call back the Fatafat engine with the first message that it retrieves from each of the partitions. This is used for

The custom adapter should have the following constructor signature.

```
(inputConfig: com.ligadata.OnLEPBase.AdapterConfiguration,
 output: Array[com.ligadata.OnLEPBase.OutputAdapter],
 envCtxt: com.ligadata.OnLEPBase.EnvContext,
 mkExecCtxt: com.ligadata.OnLEPBase.MakeExecContext,
 cntrAdapter: com.ligadata.OnLEPBase.CountersAdapter)
```

inputConfig is used to create and maintain the connection to a specific Kafka Broker for a specific topic. The custom adapter must create a connection to a Broker/Topic using this info.

Output is an array of `OutputAdapters` - *This explanation will be provided shortly*.

envCtnx - *This explanation will be provided shortly*.

MakeExecContext is used to create an Execution Context in the Adapter, which will be used to call back the Fatafat engine as messages are processed.

cntrAdapter is used by the adapter to keep track of relevant statistics.

Implementing Custom Kafka Producers

This is a relatively easy task, at least compared to implementing a Kafka consumer. The custom code needs to implement the `com.ligadata.OnLEPBase.OutputAdapter` scala trait (see below). The methods in red must be overwritten by the customer implementation.

```
trait OutputAdapter {
  val inputConfig: AdapterConfiguration // Configuration

  def send(message: String, partKey: String): Unit
  def send(message: Array[Byte], partKey: Array[Byte]): Unit
  def Shutdown: Unit
  def Category = "Output"
}
```

The send method needs to insert `KeyedMessage` into a specified Kafka Producer object (see Kafka documentation for instructions on how to instantiate and use Kafka Producer objects)

The constructor for this producer must have two arguments:

```
AdapterConfiguration
CountersAdapter
```

The `AdapterConfiguration` object is described in the Kafka Consumer section
The `CountersAdapter` is described in the Kafka Consumer section.