LIGADATA

Fatafat 1.0 -
Implementing Custom 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, including concepts like `partitions` and `offsets`. However the origin of messages themselves is not limited to any specific source, as long as the Trait/Interface described here is implemented correctly.

For example, the two methods, `GetAllPartitionBeginValues` and `GetAllPartitionEndValues`, make reference to Kafka Partitions/Offsets, but if you are using simple files, you need to provide the values required by your specific adapter implementations. For instance, for simple files it could be the FileName/Location and the byte offsets within the file.

Implementing Custom Consumers

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

A custom input Adapter (consumer) should be able to process messages via their `offsets` in their respective data sources. For example, if the Fatafat engine asks for a message at `offset` X, then that message is returned back to the engine.

Custom adapters 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.

Method Signatures

The method 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 Message Topic.

GetAllPartitionUniqueRecordKey – This method is called by the Fatafat engine on an instance of a `InputAdapter`. The output of this method needs to be an array of `PartionUniqueRecordKey` objects. Each object will need to have a `PartitionId` that The message source 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 the offsets within a message source partition from where to start processing.

GetAllPartitionBeginValues/GetAllpartionEndValues – this will return an array of `PartionUniqueRecordKey`, `PartionUniqueRecordValue` pairs. The

`PartionUniqueRecordKey` will have information about a specific partition ID, while the `PartionUniqueRecordKey` 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.
- partitionInfo - Information about the sources of the messages with each array element representing a message source, for example a kafka partition or a file.
  - First element in the structure should represent the partition identifier. For Kafka, this is a partition ID as represented by the `PartionUniqueRecordKey` object.
  - Second element in the structure should represent the position in the source at which reading starts. For Kafka, this is the partition offset, as represented by the `PartionUniqueRecordValue` object.
  - Third element in the structure should represent the beginning transaction ID. Increment this value by one for each message processed (after sending that message to the Fatafat engine).
  - Fourth element in the structure is a Triplet. This is used by the Fatafat server to ensure only once processing in case of a failure.
    - The first element in the triplet is a "marker", which tells the adapter that for any message that is read from the source with the "offset" lower or equal to this "marker", pass back the second and third parameters to the server, otherwise, they pass back zeroes. In the below example these values are `processingXformMsg, totalXformMsg.`

  ```
  Array  [
         (PartitionUniqueRecordKey, PartitionUniqueRecordValue, Long,
         (PartitionUniqueRecordValue, Int, Int)
         ]
  ```
- 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.

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 message Broker (or Kafka Broker) for a specific topic. The custom adapter must create a connection to a Message Source/Topic using this info.

Output is an array of `OutputAdapters` - This array of adapters emits the output – passing it from input adapter to engine, which evaluates the input data and sends the output to these output adapters.

envCtxt - The input adapter passes the environment context, envCtxt, to the engine as one of the arguments in MakeExecContext, and it is eventually used by the Engine.

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.

To call back to the server from a custom adapter.

To call back to the server create an object by calling a `CreateExecContext` method on the `com.ligadata.OnLEPBase.MakeExecContext` passed as the fourth parameter in the constructor.

```
execThread = mkExecCtxt.CreateExecContext(input, partitionId,
output, envCtxt)
```

Then call the execute method with these parameters.

`execThread.execute(`
`transactionId` – as described above.
`message` – UTF8 string
`format` - CSV/JSON etc
`uniqueKey` - This is the `PartitionUniqueRecordKey` representation of the partition: `(partitionID, topic name).`
`uniqueVal` - Offset of the message within the source.
`readTmNs` – `System.nanoTime` when the message was retrieved for the source.
`readTmMs` -`System.currentTimeMillis` when the message was retrieved for the source
`dontSendOutputToOutputAdap` – A boolean flag... true if the 1st Element of the (`PartitionUniqueRecordValue, Int, Int`) structure is equals or greater then Offset of this message.
`processingXformMsg` -described .above
`totalXformMsg` – described above.
`)`

Implementing Custom Producers

This is a relatively easy task, at least compared to implementing a custom 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 Message Destination 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.