import java.io.{BufferedWriter, File, FileWriter, PrintWriter}  
  
import org.apache.spark.SparkConf  
import org.apache.spark.streaming.StreamingContext  
import org.apache.spark.streaming.Seconds  
import org.apache.kafka.common.serialization.StringDeserializer  
  
import scala.io.\_  
import org.apache.spark.streaming.kafka010.{ConsumerStrategies, KafkaUtils, LocationStrategies}  
object KafkaStream {  
def main (args:Array[String]): Unit = {  
 val outfile=new File("D:\\study\_material\\scalapractise\\csvfiles\\outwordcount\\output.txt")  
 // val pw=new PrintWriter(outfile)  
 val bw=new BufferedWriter( new FileWriter(outfile))  
  
 val topics=*Array*("test")  
 val sparkConf=new SparkConf().setMaster("local[\*]").setAppName("goli")  
 val ssc=new StreamingContext(sparkConf,*Seconds*(30))  
 val sc=ssc.sparkContext  
 sc.setLogLevel("OFF")  
  
 val kafkaParams=*Map*[String,Object](  
 "bootstrap.servers"-> "localhost:9092",  
 "key.deserializer"-> *classOf*[StringDeserializer],  
 "value.deserializer"-> *classOf*[StringDeserializer],  
 "group.id"-> "GMP",  
 "auto.offset.reset" -> "latest"  
 )  
 val messages=KafkaUtils.*createDirectStream*[String,String](  
 ssc,LocationStrategies.*PreferConsistent*,ConsumerStrategies.*Subscribe*[String,String](topics,kafkaParams)  
 )  
 val line=messages.map(\_.value())  
 //val words=line.flatMap(\_.split(" "))  
 //val wordscount=words.map( x => (x,1)).reduceByKey(\_+\_)  
 line.print()  
 line.foreachRDD( a => { a.coalesce(1).saveAsTextFile("D:\\study\_material\\scalapractise\\csvfiles\\outwordcount") } )  
 // wordscount.foreachRDD( a => { a.toDF("value").coalesce(1).write.mode(SaveMode.Append).save("D:\\study\_material\\scalapractise\\csvfiles\\outwordcount")})  
  
 ssc.start()  
 ssc.awaitTermination()  
}  
  
}

Kafka Utils:

package com.ericsson.mediafirst.data.providers.kafka  
  
import java.io.{PrintWriter, StringWriter}  
import java.util  
  
import com.ericsson.mediafirst.utils.tools.RetryUtils  
import com.typesafe.config.{Config, ConfigFactory, ConfigValueFactory}  
import org.apache.kafka.clients.consumer.{ConsumerConfig, KafkaConsumer, OffsetAndMetadata, OffsetCommitCallback}  
import org.apache.kafka.common.PartitionInfo  
import org.apache.kafka.clients.consumer.internals.ConsumerCoordinator  
import org.apache.kafka.clients.producer.ProducerConfig  
import org.apache.kafka.common.TopicPartition  
  
import scala.collection.JavaConversions.\_  
import scala.util.Try  
  
  
*/\*\*  
 \* Utility object to get information about a Kafka Cluster  
 \*  
 \** ***@author*** *eforjul,epenwei  
 \*  
 \*/*object KafkaClusterUtils {  
  
 */\*\*  
 \* Default number of retries attempts for each request sent to Zookeeper  
 \*/* final val *DEFAULT\_RETRIES\_ATTEMPTS* = 10  
 */\*\*  
 \* Default delay in seconds between each request attempt sent to Zookeeper  
 \*/* final val *DEFAULT\_RETRIES\_DELAY* = 10  
 */\*\*  
 \* Default logger function to report failed request attempts sent to Zookeeper  
 \*/* final val *DEFAULT\_LOGGER* = (s: String, error: Boolean) => if (error) System.*err*.println(s) else *println*(s)  
 */\*\*  
 \* Default group Id constant  
 \*/* final val *DEFAULT\_GROUP\_ID* = "defaultGroupId"  
 */\*\*  
 \* Get list of all available topics on the Kafka cluster  
 \*  
 \** ***@param kafkaConfig*** *brokersConnectionString: Kafka broker addresses  
 \** ***@param logger*** *logger function used to report failed Zookeeper connection and requests attempts  
 \** ***@return*** *a set of available topics  
 \*/* def getAvailableTopics(kafkaConfig: Config, logger: (String, Boolean) => Unit = *DEFAULT\_LOGGER*): Set[String] = {  
 val brokersConnectionString = kafkaConfig.getString("brokersConnectionString")  
 // Give default group Id as the group Id is not mandatory for getting all topics  
 val groupId = if(kafkaConfig.hasPath("groupId")) kafkaConfig.getString("groupId") else *DEFAULT\_GROUP\_ID* val retriesAttempts = *Try*(kafkaConfig.getInt("retries.attempts")).getOrElse(*DEFAULT\_RETRIES\_ATTEMPTS*)  
 val retryDelay = *Try*(kafkaConfig.getInt("retries.delay")).getOrElse(*DEFAULT\_RETRIES\_DELAY*)  
 RetryUtils.*retryOrDie*(retriesAttempts, retryDelay = retryDelay \* 1000,  
 loopFn = {  
 logger("Could not get available topic list from Kafka brokers, retrying...", true)  
 },  
 failureFn = {  
 logger("Could not get available topic list from Kafka brokers, giving up...", true)  
 },  
 throwableParserFn = *getThrowableLoggerFn*(logger)  
 )(*getAvailableTopicsOnce*(brokersConnectionString, groupId))  
 }  
  
 */\*\*  
 \* Get list of all available topics on the Kafka cluster without retry  
 \** ***@param brokersConnectionString*** *brokersConnectionString: Kafka broker addresses  
 \** ***@return*** *a set of available topics  
 \*/* def getAvailableTopicsOnce(brokersConnectionString: String, groupId: String):Set[String]= {  
 val offsetManager = *getOffsetManager*(brokersConnectionString, groupId)  
 val topics = offsetManager.listTopics().keySet()  
 offsetManager.close()  
 topics.toSet  
 }  
  
 */\*\*  
 \* Get the default Kafka consumer parameters for creating Kafka consumer.  
 \** ***@param brokersConnectionString*** *the broker connection string  
 \** ***@param groupId*** *the group Id  
 \** ***@return*** *the default Kafka parameters  
 \*/* def getKafkaConsumerParameters(brokersConnectionString: String, groupId: String): util.Map[String, Object]={  
 val kafkaParameters = new util.HashMap[String, Object]()  
 kafkaParameters.put(ConsumerConfig.*ENABLE\_AUTO\_COMMIT\_CONFIG*, "false")  
 kafkaParameters.put(ConsumerConfig.*AUTO\_OFFSET\_RESET\_CONFIG*, "none")  
 kafkaParameters.put(ConsumerConfig.*BOOTSTRAP\_SERVERS\_CONFIG*, brokersConnectionString)  
 kafkaParameters.put(ConsumerConfig.*GROUP\_ID\_CONFIG*, groupId)  
 kafkaParameters.put(ConsumerConfig.*FETCH\_MIN\_BYTES\_CONFIG*, "100000")  
 kafkaParameters.put(ConsumerConfig.*KEY\_DESERIALIZER\_CLASS\_CONFIG*, "org.apache.kafka.common.serialization.StringDeserializer")  
 kafkaParameters.put(ConsumerConfig.*VALUE\_DESERIALIZER\_CLASS\_CONFIG*, "org.apache.kafka.common.serialization.StringDeserializer")  
 kafkaParameters  
 }  
  
 */\*\*  
 \* Get the default Kafka producer parameters for creating Kafka producer.  
 \** ***@param brokersConnectionString*** *the broker connection string  
 \** ***@return*** *the default Kafka parameters  
 \*/* def getKafkaProducerParameters(brokersConnectionString: String): util.Map[String, Object]={  
 val kafkaParameters = new util.HashMap[String, Object]()  
 kafkaParameters.put(ProducerConfig.*BOOTSTRAP\_SERVERS\_CONFIG*, brokersConnectionString)  
 kafkaParameters.put(ProducerConfig.*BATCH\_SIZE\_CONFIG*, "256000")  
 kafkaParameters.put(ProducerConfig.*LINGER\_MS\_CONFIG*, "200")  
 // Comment this to disable the message compression  
 //kafkaParameters.put(ProducerConfig.COMPRESSION\_TYPE\_CONFIG, "lz4")  
 //kafkaParameters.put(ProducerConfig.BUFFER\_MEMORY\_CONFIG, "100000000")  
 kafkaParameters.put(ProducerConfig.*VALUE\_SERIALIZER\_CLASS\_CONFIG*, "org.apache.kafka.common.serialization.StringSerializer")  
 kafkaParameters.put(ProducerConfig.*KEY\_SERIALIZER\_CLASS\_CONFIG*, "org.apache.kafka.common.serialization.StringSerializer")  
 kafkaParameters  
 }  
  
 */\*\*  
 \* Check if an existing topics on Kafka is empty where empty means that no data is available on Kafka (not that the offsets are equal to 0)  
 \*  
 \** ***@param config*** *configuration for the Kafka connection, such as brokersConnectionString  
 \** ***@param topic*** *the name of the topics to be checked  
 \** ***@param logger*** *optional custom logger function to be used  
 \** ***@return*** *true if the topics is empty (i.e. if for each partition, the minimum offset is equal to the maximum offset) false data is available or if the  
 \* topics does not exist at all  
 \*/* def topicIsEmpty(config: Config, topic: String, logger: (String, Boolean) => Unit = *DEFAULT\_LOGGER*): Boolean = {  
 val offsetRanges = *getTopicOffsetRanges*(config, topic, logger)  
 if (offsetRanges.isEmpty) false else offsetRanges.forall { case (partitionId, (minOffset, maxOffset)) => minOffset == maxOffset}  
 }  
  
 */\*\*  
 \* Get offset ranges for given Kafka topics  
 \*  
 \** ***@param config*** *configuration for the Kafka connection, such as brokersConnectionString  
 \** ***@param topics*** *the name of the topics whose offset ranges are to be returned  
 \** ***@param logger*** *optional custom logger function to be used  
 \** ***@return*** *An Either error string or a Map containing Kafka partition -> (earliestoffset, latestoffset) mapping  
 \*/* def getTopicsOffsetRanges(config: Config, topics: List[String], logger: (String, Boolean) => Unit = *DEFAULT\_LOGGER*): Map[TopicPartition, (Long, Long)] = {  
 val brokersConnectionString = config.getString("brokersConnectionString")  
 val groupId = config.getString("groupId")  
 logger(s"Prepare to get offset range for topics: **$**topics, groupId: **$**groupId", false)  
 val retriesAttempts = *Try*(config.getInt("retries.attempts")).getOrElse(*DEFAULT\_RETRIES\_ATTEMPTS*)  
 val retryDelay = *Try*(config.getInt("retries.delay")).getOrElse(*DEFAULT\_RETRIES\_DELAY*)  
 val offsetRanges = RetryUtils.*retryOrDie*(retriesAttempts, retryDelay = retryDelay \* 1000,  
 loopFn = {  
 logger(s"Could not get offset range for topics **$**{topics}, retrying...", true)  
 },  
 failureFn = {  
 logger(s"Could not get offset range for topics **$**{topics}, giving up...", true)  
 },  
 throwableParserFn = *getThrowableLoggerFn*(logger)  
 )(*getTopicsOffsetRangesOnce*(brokersConnectionString, groupId, topics, logger))  
 logger(s"The Topic **$**topics offset ranges are **$**{offsetRanges.toList}", false)  
 offsetRanges  
 }  
  
 */\*\*  
 \* Get topic offset ranges withough retry  
 \** ***@param brokersConnectionString*** *the Kafka broker address  
 \** ***@param topics*** *the topic list  
 \** ***@return*** *the topic partition and corresponding offsets  
 \*/* def getTopicsOffsetRangesOnce(brokersConnectionString: String, groupId: String, topics: List[String], logger: (String, Boolean) => Unit) : Map[TopicPartition, (Long, Long)] ={  
 val offsetManager = *getOffsetManager*(brokersConnectionString, groupId)  
 val partitions = *getPartitions*(offsetManager, topics, logger).map {  
 partitionInfo: PartitionInfo => new TopicPartition(partitionInfo.topic(), partitionInfo.partition())  
 }  
 val beginOffsets = offsetManager.beginningOffsets(partitions)  
 val endOffsets = offsetManager.endOffsets(partitions)  
 offsetManager.close()  
 val offsetRanges = partitions.map(topicPartition => topicPartition -> (beginOffsets.get(topicPartition).toLong, endOffsets.get(topicPartition).toLong)).toMap  
 if (offsetRanges.size != partitions.size) throw new RuntimeException(s"Fetched offsets ranges are not complete, offset ranges: **$**{offsetRanges}, partitions: **$**{partitions}")  
 offsetRanges  
 }  
  
 */\*\*  
 \* Get all the partitions for a topic  
 \** ***@param kafkaConsumer*** *\** ***@param topics*** *\** ***@return*** *\*/* def getPartitions(kafkaConsumer: KafkaConsumer[String, String], topics: List[String], logger: (String, Boolean) => Unit):List[PartitionInfo] = {  
 topics.map { topic =>  
 val partitions = kafkaConsumer.partitionsFor(topic).toList  
 if (partitions.map(p => p.partition()).max + 1 != partitions.size) {  
 val existingPartitons = partitions.map(p => p.partition())  
 val errorMsg = s"Fail to get partitions for topic: **$**topic, some partition missing, total partition number: **$**{partitions.size}, existing partitions: **$**existingPartitons"  
 logger(errorMsg, true)  
 throw new RuntimeException(errorMsg)  
 } else partitions  
 }.flatten  
 }  
  
 */\*\*  
 \* Get offset ranges for a single Kafka topic  
 \** ***@param config*** *configuration for the Kafka connection, such as brokersConnectionString  
 \** ***@param topic*** *the name of the topics whose offset ranges are to be returned  
 \** ***@param logger*** *optional custom logger function to be used  
 \** ***@return*** *an option for a map containing the offset ranges. If no brokers could be found for the topics partitions, None is returned, otherwise, a Map whose  
 \* keys are the partition ids and values are (minOffset, maxOffset) tuples is returned  
 \*/* def getTopicOffsetRanges(config: Config, topic: String, logger: (String, Boolean) => Unit = *DEFAULT\_LOGGER*): Map[Int, (Long, Long)] = {  
 val partitionOffsets = *getTopicsOffsetRanges*(config, *List*(topic), logger)  
 for ((k, v) <- partitionOffsets) yield (k.partition() -> v)  
 }  
  
 */\*\*  
 \* shows if all the available topics exist  
 \** ***@param kafkaTopics*** *all topics names  
 \** ***@param brokerConnectionString*** *zookeeper connection  
 \** ***@return*** *a map from topics name to kafka existence flag  
 \*/* def kafkaTopicsExist(kafkaTopics: List[String], brokerConnectionString: String, logger: (String, Boolean) => Unit = *DEFAULT\_LOGGER*): Map[String,Boolean] = {  
 val kafkaConfig = ConfigFactory.*empty*().withValue("brokersConnectionString", ConfigValueFactory.*fromAnyRef*(brokerConnectionString))  
 val availableTopics = KafkaClusterUtils.*getAvailableTopics*(kafkaConfig, logger)  
 kafkaTopics.map(topic => (topic,availableTopics.contains(topic))).toMap  
 }  
  
 */\*\*  
 \* Fetch the stored offset from Kafka  
 \** ***@param config*** *configuration for the Kafka connection, such as brokersConnectionString, debug etc  
 \** ***@param topic*** *the name of the topics whose offset ranges are to be returned  
 \** ***@param logger*** *optional custom logger function to be used  
 \** ***@return*** *the stored offsets in Kafka  
 \*/* def fetchOffsets(config: Config, topic: String, logger: (String, Boolean) => Unit = *DEFAULT\_LOGGER*):Map[Int, Long] = {  
 val topicOffsets = *fetchOffsets*(config, *List*(topic), logger)  
 for ((k, v) <- topicOffsets) yield (k.partition() -> v)  
 }  
  
 */\*\*  
 \* Fetch the stored offset from Kafka  
 \** ***@param config*** *configuration for the Kafka connection, such as brokersConnectionString, debug etc  
 \** ***@param topics*** *the name of the topics whose offset ranges are to be returned  
 \** ***@return*** *the stored offsets in Kafka  
 \*/* def fetchOffsets(config: Config, topics: List[String], logger: (String, Boolean) => Unit): Map[TopicPartition, Long] = {  
 val brokersConnectionString = config.getString("brokersConnectionString")  
 val groupId = config.getString("groupId")  
 logger(s"Prepare to fetch stored offsets for topic: **$**topics, groupId: **$**groupId", false)  
 if (topics.isEmpty) throw new IllegalArgumentException("Topic is needed to fetch the topic offsets")  
 val retriesAttempts = *Try*(config.getInt("retries.attempts")).getOrElse(*DEFAULT\_RETRIES\_ATTEMPTS*)  
 val retryDelay = *Try*(config.getInt("retries.delay")).getOrElse(*DEFAULT\_RETRIES\_DELAY*)  
 val offsets = RetryUtils.*retryOrDie*(retriesAttempts, retryDelay = retryDelay \* 1000,  
 loopFn = {  
 logger(s"Could not fetech stored offsets for topics: **$**topics, retrying...", true)  
 },  
 failureFn = {  
 logger(s"Could not fetech stored offsets for topics: **$**topics, giving up...", true)  
 },  
 throwableParserFn = *getThrowableLoggerFn*(logger)  
 )(*fetchOffsetsOnce*(brokersConnectionString, groupId, topics, logger))  
 logger(s"Fetched stored offsets for topics **$**topics are **$**offsets", false)  
 offsets  
 }  
  
 */\*\*  
 \* Fetch stored offsets from Kafka without retry  
 \** ***@param brokersConnectionString*** *the Kafka connection string  
 \** ***@param topics*** *the topic list  
 \** ***@return*** *the topic partitions and the offsets  
 \*/* def fetchOffsetsOnce(brokersConnectionString: String, groupId: String, topics: List[String], logger: (String, Boolean) => Unit):Map[TopicPartition, Long]= {  
 val offsetManager = *getOffsetManager*(brokersConnectionString, groupId)  
 val partitions = *getPartitions*(offsetManager, topics, logger)  
 val topicPartitions = partitions.map(partitionInfo =>new TopicPartition(partitionInfo.topic(), partitionInfo.partition()))  
 logger(s"Constructed topic partitions.**$**topics, **$**topicPartitions", false)  
 val offsets = *batchFetchOffset*(offsetManager, topicPartitions.toSet).toMap  
 offsetManager.close()  
 if (offsets.size != partitions.size) throw new RuntimeException(s"Fetched offsets are not complete, offsets: **$**{offsets}, partitions: **$**{partitions}")  
 offsets  
 }  
  
 */\*\*  
 \* Internal codes to fetch the offsets, this is to resolve the Kafka limitation to do batch fetching  
 \** ***@param kafkaConsumer*** *the KafkaConsumer which to do the batch fetching  
 \** ***@param topicsPartitions*** *the topic partitions to fetch offsets  
 \** ***@return*** *the offsets according to partitions  
 \*/* private def batchFetchOffset(kafkaConsumer: KafkaConsumer[String, String], topicsPartitions: Set[TopicPartition]) = {  
 val f = kafkaConsumer.getClass.getDeclaredField("coordinator") //NoSuchFieldException  
 f.setAccessible(true)  
 val coordinator = f.get(kafkaConsumer).asInstanceOf[ConsumerCoordinator]  
 val offsets = coordinator.fetchCommittedOffsets(topicsPartitions)  
 val partitionEarliestOffsets = if (offsets.size() != topicsPartitions.size) {  
 (for((k,v)<-kafkaConsumer.beginningOffsets(topicsPartitions)) yield k->v.toLong).toMap  
 } else *Map*[TopicPartition, Long]()  
 for(topicPartition <-topicsPartitions)  
 yield if (offsets.containsKey(topicPartition)) {  
 (topicPartition -> offsets(topicPartition).offset())  
 } else {  
 // If there is no stored offsets, use the earliest offset by default  
 // TBD choose the offset based on the options  
 (topicPartition -> partitionEarliestOffsets(topicPartition))  
 }  
  
 }  
  
 */\*\*  
 \* Commit stored offset for single topic to Kafka  
 \** ***@param config*** *configuration for the Kafka connection, such as brokersConnectionString, debug etc  
 \** ***@param offsets*** *the offsets to commit to Kafka  
 \** ***@param committedTopic*** *the topic which the offsets related to  
 \** ***@param logger*** *the logger function  
 \** ***@return*** *the commit error or Unit if succeed  
 \*/* def commitOffsets(config: Config, offsets: Map[Int, Long], committedTopic: String = "", logger: (String, Boolean) => Unit = *DEFAULT\_LOGGER*): Unit = {  
 val topic = if (committedTopic.equals("")) config.getString("topic") else committedTopic  
 val topicPartitionOffsets = for ((partition, offset) <- offsets) yield new TopicPartition(committedTopic, partition) -> offset  
 *commitOffsets*(config, topicPartitionOffsets, logger)  
 }  
  
 */\*\*  
 \* Commit the stored offset for multiple topics to Kafka  
 \** ***@param config*** *configuration for the Kafka connection, such as brokersConnectionString, debug etc  
 \** ***@param offsets*** *the offsets to commit to Kafka  
 \** ***@return*** *the commit error or Unit if succeed  
 \*/* def commitOffsets(config: Config, offsets: Map[TopicPartition, Long], logger: (String, Boolean) => Unit): Unit = {  
 val brokersConnectionString = config.getString("brokersConnectionString")  
 val groupId = config.getString("groupId")  
 logger(s"Prepare to commit offsets: **$**offsets, groupId: **$**groupId", false)  
 val retriesAttempts = *Try*(config.getInt("retries.attempts")).getOrElse(*DEFAULT\_RETRIES\_ATTEMPTS*)  
 val retryDelay = *Try*(config.getInt("retries.delay")).getOrElse(*DEFAULT\_RETRIES\_DELAY*)  
 RetryUtils.*retryOrDie*(retriesAttempts, retryDelay = retryDelay \* 1000,  
 loopFn = {  
 logger("Could not commit offsets to Kafka brokers, retrying...", true)  
 },  
 failureFn = {  
 logger("Could not commit offsets to Kafka brokers, giving up...", true)  
 },  
 throwableParserFn = *getThrowableLoggerFn*(logger)  
 )(*commitOffsetsOnce*(brokersConnectionString, groupId, offsets))  
  
 logger("Offsets have been committed.", false)  
  
 }  
  
 */\*\*  
 \* Commit the topic offset once without retry  
 \** ***@param brokersConnectionString*** *the broker connection string  
 \** ***@param offsets*** *the offsets  
 \*/* def commitOffsetsOnce(brokersConnectionString: String, groupId: String, offsets: Map[TopicPartition, Long]): Unit = {  
 val offsetManager = *getOffsetManager*(brokersConnectionString, groupId)  
 val offsetsMap = offsets.map { case (topicPartiton, offset) => topicPartiton -> new OffsetAndMetadata(offset, "") }  
 offsetManager.commitSync(offsetsMap)  
 offsetManager.close()  
 }  
  
 */\*\*  
 \* Reset the stored offset in Kafka  
 \** ***@param config*** *configuration for the Kafka connection, such as brokersConnectionString, debug etc  
 \** ***@param resetTopic*** *the topic which the offsets related to  
 \** ***@return*** *the error or Unit if succeed  
 \*/* def resetOffsets(config: Config, resetTopic: String = "", logger: (String, Boolean) => Unit = *DEFAULT\_LOGGER*): Unit = {  
 val brokersConnectionString = config.getString("brokersConnectionString")  
 val topic = if (resetTopic.equals("")) config.getString("topic") else resetTopic  
 logger(s"Start to reset offsets for topic: **$**topic", false)  
 val partitionCount = ProducerUtils.*getTopicPartitionCount*(config, topic)  
 logger(s"Total partition count is: **$**partitionCount", false)  
 if (partitionCount > 0) {  
 val newOffsets = ((0 to partitionCount - 1).map(i => i -> 0l)).toMap  
 *commitOffsets*(config, newOffsets, topic, logger)  
 } else {  
 throw new RuntimeException(s"Can not get the partition count for topic **$**topic to reset")  
 }  
 }  
  
 */\*\*  
 \* Internal function to return the OffsetManager(Consumer)  
 \** ***@param brokerConnectionString*** *the brokers connection string  
 \** ***@return*** *the error or the OffsetManager(KafkaConsumer)  
 \*/* private def getOffsetManager(brokerConnectionString: String, groupId: String): KafkaConsumer[String, String] = {  
 if (brokerConnectionString.isEmpty) throw new IllegalArgumentException("Brokers connection string is a mandatory input")  
 val props = *getKafkaConsumerParameters*(brokerConnectionString, groupId)  
 new KafkaConsumer[String, String](props)  
 }  
  
 */\*\*  
 \* Get the throwable printer based on existing logger  
 \** ***@param logger*** *the current logger  
 \** ***@return*** *the throwable printer  
 \*/* def getThrowableLoggerFn(logger: (String, Boolean) => Unit = *DEFAULT\_LOGGER*) = {  
 def throwableLogger(e: Throwable) =  
 {  
 val sw = new StringWriter  
 // Below is for futher trouble shooting to see the error detail.  
 if (e.getCause != null) {  
 sw.write("[ERROR] The error type is:" + e.getClass.toString)  
 sw.write("[ERROR] The error root cause type is :" + e.getCause.getClass.toString)  
 } else {  
 sw.write("[ERROR] The error type is:" + e.getClass.toString)  
 }  
 e.printStackTrace(new PrintWriter(sw))  
 logger(sw.toString, true)  
 }  
 throwableLogger(\_)  
 }  
  
 */\*\*  
 \* Get the condition check function  
 \** ***@return*** *the function of condition check  
 \*/* def isKafkaOffsetOutOfRange(e: Throwable) = {  
 e.getCause != null && e.getCause.isInstanceOf[org.apache.kafka.clients.consumer.OffsetOutOfRangeException]  
 }  
  
}

Spark Streaming Utils:

package com.ericsson.mediafirst.data.providers.kafka.spark  
  
import com.ericsson.mediafirst.data.providers.kafka.KafkaClusterUtils  
import com.ericsson.mediafirst.data.providers.kafka.KafkaClusterUtils.*kafkaTopicsExist*import com.ericsson.mediafirst.data.transformations.spark.TransformRDD  
import com.ericsson.mediafirst.utils.logging.SparkLogger  
import com.ericsson.mediafirst.utils.serialization.DeserializerUtils  
import com.ericsson.mediafirst.utils.tools.DataDef.DataMap  
import com.ericsson.mediafirst.utils.tools.TimeExtractor  
import com.typesafe.config.Config  
import org.apache.kafka.clients.consumer.ConsumerConfig  
import org.apache.kafka.common.TopicPartition  
import org.apache.spark.streaming.StreamingContext  
import org.apache.spark.streaming.dstream.DStream  
import org.apache.spark.streaming.kafka010.ConsumerStrategies.Subscribe  
import org.apache.spark.streaming.kafka010.LocationStrategies.*PreferConsistent*import org.apache.spark.streaming.kafka010.{HasOffsetRanges, KafkaUtils}  
  
import scala.collection.JavaConversions.\_  
import scala.util.Try  
  
*/\*\*  
 \* Created by khalighi on 4/6/16.  
 \*/  
/\*\*  
 \* utilities for creating KafkaTopics  
 \*/*object SparkStreamingUtils {  
  
 def getDirectDataStream(streamName: String, kafkaConfig: Config, ssc: StreamingContext): (DataStream, SparkStreamingOffsetManager) = {  
  
 val debug = if (kafkaConfig hasPath "debug") kafkaConfig.getBoolean("debug") else false  
 val timeExtractor = if (kafkaConfig hasPath "timeField") {  
 val timeField = kafkaConfig.getString("timeField")  
 if (kafkaConfig hasPath "timeFormat") *Some*(*TimeExtractor*(timeField, kafkaConfig.getString("timeFormat"))) else *Some*(*TimeExtractor*(timeField))  
 } else {  
 None  
 }  
  
 val (rawStream, offsetManager) = *getDirectStream*(kafkaConfig, ssc)  
  
 val stream = if (kafkaConfig hasPath "transformations") {  
 val transformations = kafkaConfig.getStringList("transformations")  
 val deserializerSchemaList = if (kafkaConfig hasPath "deserialize") *Some*(DeserializerUtils.*getDeserializerSchemaListFromConfig*(kafkaConfig)) else None  
 transformations.foldLeft(rawStream)(op = (thisStream, transformation) => {  
 val params: Option[DataMap] = if (transformation == "MapParseJsonAtField") {  
 *Some*(*Map*(  
 "dataField" -> *Try* {  
 kafkaConfig.getString("transformationDataField")  
 }.getOrElse(None)))  
 } else if (transformation == "MapParseGrokAtField") {  
 *Some*(*Map*(  
 "dataField" -> *Try* {  
 kafkaConfig.getString("transformationDataField")  
 }.getOrElse(None),  
 "grokPattern" -> *Try* {  
 kafkaConfig.getString("transformationGrok")  
 }.getOrElse(None)))  
 } else None  
 TransformRDD.*applyTransformationDStream*(transformation, thisStream, deserializerSchemaList, timeExtractor, params)  
 })  
 } else {  
 rawStream  
 }  
  
 val destination = if (kafkaConfig hasPath "destination") *Some*(kafkaConfig.getString("destination")) else None  
  
 (*DataStream*(streamName, stream, timeExtractor, destination, debug), offsetManager)  
 }  
  
 def getDirectDataStreamList(config: Config, ssc: StreamingContext): List[(DataStream, SparkStreamingOffsetManager)] = {  
 val streamNameList = config.getObject("stream").keySet().toList.filter(streamName => config.getBoolean(s"stream.**$**streamName.enable"))  
 streamNameList.map { streamName =>  
 val kafkaConfig = config.getConfig(s"stream.**$**streamName")  
 *getDirectDataStream*(streamName, kafkaConfig, ssc)  
 }  
 }  
  
 */\*\*  
 \* shows if all the available topics exist  
 \** ***@param kafkaTopics*** *all topic names  
 \** ***@param connectionString*** *Kafka broker connection string  
 \** ***@return*** *\*/* def exitIfKafkaTopicIsMissing(kafkaTopics: List[String], connectionString: String): Unit = {  
 val topicsDoNotExistMap = *kafkaTopicsExist*(kafkaTopics,connectionString).filter(\_.\_2 == false)  
  
 if (topicsDoNotExistMap.nonEmpty) {  
 val missingTopicList = topicsDoNotExistMap.keys.toList  
 SparkLogger.*tsg*(s"Required kafka topics **$**{missingTopicList.mkString(",")} are missing, exiting. Make sure kafka topics are created by the offline worker", "NA")  
 SparkLogger.*error*(s"Required kafka topics **$**{missingTopicList.mkString(",")} are missing, exiting.")  
 }  
 }  
  
 */\*\*  
 \* shows if all the available topics exist  
 \** ***@param kafkaTopic*** *topic name  
 \** ***@param connectionString*** *Kafka broker connection string  
 \*/* def exitIfKafkaTopicIsMissing(kafkaTopic: String, connectionString: String): Unit = {  
 *exitIfKafkaTopicIsMissing*(*List*(kafkaTopic), connectionString)  
 }  
  
 */\*\*  
 \* Get the Spark direct stream with the offsets maintained by Stream offset manager  
 \** ***@param kafkaConfig*** *\** ***@param ssc*** *\** ***@return*** *\*/* def getDirectStream(kafkaConfig: Config, ssc: StreamingContext): (DStream[DataMap], SparkStreamingOffsetManager) = {  
 //Get topic partition information  
 val topicList = *Try*(kafkaConfig.getStringList("topics").toList).getOrElse(*List*(kafkaConfig.getString("topic"))).asInstanceOf[List[String]]  
  
 val connectionString = kafkaConfig.getString("brokersConnectionString")  
 val groupId = kafkaConfig.getString("groupId")  
 // The reason to have this consumer group Id different from offset handle group id is because the Spark stream will break when using the same group id to commit offsets.  
 val consumerGroupId = groupId + "-consumer"  
 SparkLogger.*log*(s"Connect to Kafka brokers: **$**connectionString, topics: **$**topicList, groupId: **$**groupId")  
 val kafkaParams = KafkaClusterUtils.*getKafkaConsumerParameters*(connectionString, consumerGroupId)  
  
 //Retrieve the start offset for each partition  
 val offsetManager = new SparkStreamingOffsetManager(kafkaConfig)  
  
 val fromOffsets: Map[TopicPartition, Long] = kafkaConfig.getString("offsetStrategy") match {  
 case "largestOnAbsent" => offsetManager.getFromOffsets(topicList, "largest")  
 case "smallestOnAbsent" => offsetManager.getFromOffsets(topicList, "smallest")  
 case "largestForced" => offsetManager.getFromOffsets(topicList, "largest", true)  
 case "smallestForced" => offsetManager.getFromOffsets(topicList, "smallest", true)  
 case other => offsetManager.getFromOffsets(topicList, "largest")  
 }  
  
 val deserializer = kafkaConfig.getString("deserializer")  
 var i = 0  
 val stream = deserializer match {  
 case "PayloadJsonDeserializer" =>  
 val topicList = fromOffsets.keys.map(topicPartition=> topicPartition.topic()).toList  
 kafkaParams.put(ConsumerConfig.*VALUE\_DESERIALIZER\_CLASS\_CONFIG*, "com.ericsson.mediafirst.data.providers.kafka.decoders.PayloadJsonDeserializer")  
 KafkaUtils.*createDirectStream*[String, DataMap](ssc, *PreferConsistent*, *Subscribe*[String, DataMap](topicList, kafkaParams, fromOffsets)).transform { rdd =>  
 val ranges = rdd.asInstanceOf[HasOffsetRanges].offsetRanges  
 var count = 0L  
 ranges.foreach(range => {  
 count += range.untilOffset - range.fromOffset  
 })  
 SparkLogger.*debug*("Got an KafkaRDD Range --------------------------- " + i + "(" + count + ")")  
 i += 1  
  
 offsetManager.append(ranges)  
 rdd.map(item => item.value())  
 }  
 case "JsonStringDeserializer" =>  
 val topicList = fromOffsets.keys.map(topicPartition=> topicPartition.topic()).toList  
 kafkaParams.put(ConsumerConfig.*VALUE\_DESERIALIZER\_CLASS\_CONFIG*, "com.ericsson.mediafirst.data.providers.kafka.decoders.JsonStringDeserializer")  
 KafkaUtils.*createDirectStream*[String, DataMap](ssc, *PreferConsistent*, *Subscribe*[String, DataMap](topicList, kafkaParams, fromOffsets)).transform { rdd =>  
 val ranges = rdd.asInstanceOf[HasOffsetRanges].offsetRanges  
 var count = 0L  
 ranges.foreach(range => {  
 count += range.untilOffset - range.fromOffset  
 })  
 SparkLogger.*debug*("Got an KafkaRDD Range --------------------------- " + i + "(" + count + ")")  
 i += 1  
  
 offsetManager.append(ranges)  
 rdd.map(item => item.value())  
 }  
 case other => throw new UnsupportedOperationException(s"invalid deserializer: **$**deserializer")  
 }  
  
 (stream, offsetManager)  
 }  
  
  
}