Amazon Kinesis Data Stream Development

Step 1. To create a stream

1. Open Kinesis console.
2. Choose **Go to Streams**.
3. In the navigation bar, expand the region selector and choose a region.
4. Choose **Create Stream**.
5. Type a name for your stream (for example, **StockTradeStream**).
6. Type **1** for the number of shards, but leave **Estimate the number of shards you'll need**  collapsed.
7. Choose **Create**.

2. Create an Appropriate IAM policy and user with secret keys **(credentials)**

3. Download and build the implementation code

1. Download the source code on to your computer.
2. Create a project in your favorite IDE with the source code, and the AWS SDK and Kinesis Client  Library added to the project as libraries.
3. Depending on your IDE, the project might be built automatically. If not, build the project using the appropriate steps for your IDE.

If you complete these steps successfully, you are now ready to move to the next section. If your build generates errors at any stage, you need to investigate and fix them before proceeding.

Step 4: Implement the Producer

This application uses the real-world scenario of stock market trade monitoring. The following principles briefly explain how this scenario maps to the producer and supporting code structure.

Refer to the source code and review the following information.

**StockTrade class**

An individual stock trade is represented by an instance of the StockTrade class, which contains attributes such as the ticker symbol, price, number of shares, the type of the trade (buy or sell), and an ID uniquely identifying the trade. This class is implemented for you.

**Stream record**

A stream is a sequence of records. A record is a serialization of a StockTrade instance in JSON format. For example:

**StockTradeGenerator class**

StockTradeGenerator has a method called getRandomTrade() that returns a new randomly generated stock trade every time it is invoked. This class is implemented for you.

**StockTradesWriter class**

The main class of the producer, StockTradesWriter continuously retrieves a random trade and then sends it to Streams by performing the following tasks:

1. Read stream name and region name as input
2. Read credentials from ~/.aws/credentials
3. Create an AmazonKinesisClient using those credentials.
4. Check that the stream exists and is active (if not, it will exit with an error).
5. In a continuous loop, calls StockTradeGenerator.getRandomTrade() and calls the sendStockTrade method to send the trade to the stream every 100 milliseconds.

The sendStockTrade method has not been implemented.

**To implement the producer**

• Add the following code to the sendStockTrade method of the StockTradesWriter class

private static void sendStockTrade(StockTrade trade, AmazonKinesis kinesisClient, String streamName) {

byte[] bytes = trade.toJsonAsBytes();

// The bytes could be null if there is an issue with the JSON serialization by the Jackson JSON library.

if (bytes == null) {

LOG.warn("Could not get JSON bytes for stock trade");

return;

}

LOG.info("Putting trade: " + trade.toString());

PutRecordRequest putRecord = new PutRecordRequest();

putRecord.setStreamName(streamName);

// We use the ticker symbol as the partition key, explained in the Supplemental Information section below.

putRecord.setPartitionKey(trade.getTickerSymbol());

putRecord.setData(ByteBuffer.wrap(bytes));

try {

kinesisClient.putRecord(putRecord);

} catch (AmazonClientException ex) {

LOG.warn("Error sending record to Amazon Kinesis.", ex);

}

}

Refer to the following code breakdown:

* The PutRecord API expects a byte array, and you need to convert trade to JSON format. This single line of code performs that operation:

byte[] bytes = trade.toJsonAsBytes();

* Before you can send the trade, you create a new PutRecordRequest instance (called putRecord in this case):

PutRecordRequest putRecord = new PutRecordRequest();

Each PutRecord call requires the stream name, partition key, and data blob. The following code populates these fields in the putRecord object using its setXxxx() methods:

putRecord.setStreamName(streamName); putRecord.setPartitionKey(trade.getTickerSymbol()); putRecord.setData(ByteBuffer.wrap(bytes));

The example uses a stock ticket as a partition key, which maps the record to a specific shard. In practice, you should have hundreds or thousands of partition keys per shard such that records are evenly dispersed across your stream.

Now putRecord is ready to send to the client (the put operation):

kinesisClient.putRecord(putRecord);

* Error checking and logging are always useful additions. This code logs error conditions:

if (bytes == null) {

LOG.warn("Could not get JSON bytes for stock trade");

return; }

Add the try/catch block around the put operation:

try {

kinesisClient.putRecord(putRecord);

}

catch (AmazonClientException ex) {

LOG.warn("Error sending record to Amazon Kinesis.", ex);

}

This is because a Kinesis Data Streams put operation can fail because of a network error, or due to the stream reaching its throughput limits and getting throttled. We recommend carefully considering your retry policy for put operations to avoid data loss, such using as a simple retry.

* Status logging is helpful but optional:

LOG.info("Putting trade: " + trade.toString());

The producer shown here uses the Kinesis Data Streams API single record functionality, PutRecord. In practice, if an individual producer is generating a lot of records, it is often more efficient to use the multiple records functionality of PutRecords and send batches of records at a time.

**To run the producer**

1. Verify that the access key and secret key pair retrieved earlier (when creating the IAM user) are saved in the file ~/.aws/credentials.
2. Run the StockTradeWriter class with the following arguments:

StockTradeStream us-west-2

If you created your stream in a Region other than us-west-2, you have to specify that Region here instead.

**Step 5: Implement the Consumer**

The consumer application you are developing continuously processes the stock trades stream that you created in previous step, and outputs the most popular stocks being bought and sold every minute. The application is built on top of the KCL, which does a lot of the heavy lifting common to consumer apps.

Refer to the source code and review the following information.

**StockTradesProcessor class**

Main class of the consumer, provided for you, which performs the following tasks:

* Read the application, stream, and Region names, passed in as arguments.
* Read credentials from ~/.aws/credentials.
* Create a RecordProcessorFactory instance that serves instances of RecordProcessor, implemented by a StockTradeRecordProcessor instance.
* Create a KCL worker with the RecordProcessorFactory instance and a standard configuration including the stream name, credentials, and application name.
* The worker creates a new thread for each shard (assigned to this consumer instance), which continuously loops to read records from Kinesis Data Streams and then invokes the RecordProcessor instance to process each batch of records received.

**StockTradeRecordProcessor class**

Implementation of the RecordProcessor instance, which in turn implements three required methods: initialize, processRecords, and shutdown.

As the names suggest, initialize and shutdown are used by KCL to let the record processor know when it should be ready to start receiving records and when it should expect to stop receiving records, respectively, so it can do any application-specific setup and termination tasks. The code for these is provided for you. The main processing happens in the processRecords method, which in turn uses processRecord for each record. This latter method is provided as mostly empty skeleton code for you to implement in the next step, where it is explained further.

Also of note is the implementation of support methods for processRecord: reportStats, andresetStats, which are empty in the original source code.

The processsRecords method is implemented for you, and performs the following steps:

* For each record passed in, call processRecord on it.
* If at least 1 minute has elapsed since the last report, call reportStats() which prints out the latest stats, and then resetStats() which clears the stats so that the next interval includes only new records.
* Set the next reporting time.
* If at least 1 minute has elapsed since the last checkpoint, call checkpoint().
* Set the next checkpointing time.

This method uses 60-second intervals for the reporting and checkpointing rate.

**StockStats class**

This class provides data retention and statistics tracking for the most popular stocks over time. This code is provided for you and contains the following methods:

* addStockTrade(StockTrade): Injects the given StockTrade into the running statistics.
* toString(): Returns the statistics in a formatted string.

The way this class keeps track of the most popular stock is that it keeps a running count of the total number of trades for each stock and the maximum count. It keeps these counts updated whenever a stock trade arrives.

Add code to the methods of the StockTradeRecordProcessor class, as shown in the following steps.

**To implement the consumer**

1. Implement the processRecord method by instantiating a correctly sized StockTrade object and adding the record data to it, logging a warning if there's a problem.

StockTrade trade = StockTrade.fromJsonAsBytes(record.getData().array());

if (trade == null) {

LOG.warn("Skipping record. Unable to parse record into StockTrade. Partition Key: " + record.getPartitionKey());

return;

}

stockStats.addStockTrade(trade);

1. Implement a simple reportStats method. Feel free to modify the output format to your preferences.

System.out.println("\*\*\*\*\*\* Shard " + kinesisShardId + " stats for last 1 minute \*\*\*\*\*\*\n" +

stockStats + "\n" +

"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

1. Finally, implement the resetStats method, which creates a new stockStats instance.

stockStats = new StockStats();

**To run the consumer**

1. Run the producer you wrote in the previous module to inject simulated stock trade records into your stream.
2. Verify that the access key and secret key pair retrieved earlier (when creating the IAM user) are saved in the file ~/.aws/credentials .
3. Run the StockTradesProcessor class with the following arguments:

StockTradesProcessor StockTradeStream us-west-2

Note that if you created your stream in a Region other than us-west-2, you have to specify that Region here instead.

**Step 6: (Optional) Extending the Consumer**

The application shown here may already be sufficient for your purposes. This optional section shows how you might want to extend the consumer code for a slightly more elaborate scenario.

If you want to know about the biggest sell orders each minute, this is a matter of modifying the StockStats class in three places to accommodate this new priority.

**To extend the consumer**

1. Add new instance variables:

// Ticker symbol of the stock that had the largest quantity of shares sold

private String largestSellOrderStock;

// Quantity of shares for the largest sell order trade

private long largestSellOrderQuantity;

1. Add the following code to addStockTrade:

if (type == TradeType.SELL) {

if (largestSellOrderStock == null || trade.getQuantity() > largestSellOrderQuantity) {

largestSellOrderStock = trade.getTickerSymbol();

largestSellOrderQuantity = trade.getQuantity();

}

}

1. Modify the toString method to print the additional information:

public String toString() {

return String.format(

"Most popular stock being bought: %s, %d buys.%n" +

"Most popular stock being sold: %s, %d sells.%n" +

"Largest sell order: %d shares of %s.",

getMostPopularStock(TradeType.BUY), getMostPopularStockCount(TradeType.BUY),

getMostPopularStock(TradeType.SELL), getMostPopularStockCount(TradeType.SELL),

largestSellOrderQuantity, largestSellOrderStock);

}

If you run the consumer now (remember to run the producer also), you should see output similar to this:

\*\*\*\*\*\* Shard shardId-000000000001 stats for last 1 minute \*\*\*\*\*\*

Most popular stock being bought: WMT, 27 buys.

Most popular stock being sold: PTR, 14 sells.

Largest sell order: 996 shares of BUD.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Step 7: Finishing Up

**To delete the stream and table**

1. Shut down any producers and consumers that you may still have running.
2. Open the Kinesis console.
3. Choose the stream that you created for this application (StockTradeStream).
4. Choose **Delete Stream**.
5. Open the DynamoDB console.
6. Delete the StockTradesProcessor table.