Amazon Kinesis

Developer Guide

API Version 2013-12-02

Amazon Kinesis: Developer Guide

Table of Contents

What Is Amazon Kinesis?

Amazon Kinesis is a managed service that scales elastically for real-time processing of streaming big data. This section introduces the Amazon Kinesis service.

Topics

Amazon Kinesis takes in large streams of data records that can then be consumed in real time by multiple data-processing applications that can be run on Amazon Elastic Compute Cloud (Amazon EC2) instances. The data-processing applications use the Amazon Kinesis Client Library and are called "Amazon Kinesis applications." For more information about Amazon EC2, see the Amazon Elastic Compute Cloud Getting Started Guide.

Amazon Kinesis applications read data from the Amazon Kinesis stream and process the read data in real time. The processed records can be emitted to dashboards, used to generate alerts, dynamically change pricing and advertising strategies, or emit data to a variety of other Amazon big data services such as Amazon Simple Storage Service (Amazon S3), Amazon Elastic MapReduce (Amazon EMR), or Amazon Redshift. Amazon Kinesis applications can also emit data into another Amazon Kinesis stream, enabling more complex data processing. For information about Amazon Kinesis service highlights and pricing, go to Amazon Kinesis.

What Can I Do with Amazon Kinesis?

Amazon Kinesis takes in large streams of data for processing in real time. The most common Amazon Kinesis use case scenario is rapid and continuous data intake and aggregation. The type of data used in an Amazon Kinesis use case includes IT infrastructure log data, application logs, social media, market data feeds, web clickstream data, and more. Because the response time for the data intake and processing is in real time, the processing is typically lightweight.

Amazon Kinesis enables sophisticated streaming data processing, because one Amazon Kinesis application may emit Amazon Kinesis stream data into another Amazon Kinesis stream. Near-real-time aggregation of data enables processing logic that can extract complex key performance indicators and metrics from that data. For example, complex data-processing graphs can be generated by emitting data from multiple Amazon Kinesis applications to another Amazon Kinesis stream for downstream processing by a different Amazon Kinesis application.

The following are typical scenarios for using Amazon Kinesis.

- Accelerated log and data feed intake and processing: Using Amazon Kinesis you can have producers push data directly into an Amazon Kinesis stream. For example, system and application logs can be submitted to Amazon Kinesis and be available for processing in seconds. This prevents the log data from being lost if the front end or application server fails. Amazon Kinesis provides accelerated data feed intake because you are not batching up the data on the servers before you submit them for intake.
- Real-time metrics and reporting: You can use data ingested into Amazon Kinesis for simple data analysis and reporting in real time. For example, metrics and reporting for system and application logs ingested into the Amazon Kinesis stream are available in real time. This enables data-processing application logic to work on data as it is streaming in, rather than wait for data batches to be sent to the data-processing applications.
- Real-time data analytics: Amazon Kinesis enables real-time analytics of streaming big data, combining the power of parallel processing with the value of real-time data. For example, website clickstreams can be ingested in real time, and then site usability engagement can be analyzed by many different Amazon Kinesis client applications running in parallel.
- Complex stream processing: Lastly, Amazon Kinesis enables you to create Directed Acyclic Graphs (DAGs) of Amazon Kinesis applications and data streams. This scenario typically involves emitting data from multiple Amazon Kinesis applications to another Amazon Kinesis stream for downstream processing by a different Amazon Kinesis application.

Benefits of Using Amazon Kinesis

While Amazon Kinesis can be used to solve a variety of streaming data problems, a common use is the real-time aggregation of data followed by loading the aggregate data into a data warehouse or map-reduce cluster.

Data can be taken into Amazon Kinesis streams, which will ensure durability and elasticity. The delay between the time a record is added to the stream and the time it can be retrieved (put-to-get delay) is less than 10 seconds — in other words, Amazon Kinesis applications can start consuming the data from the stream less than 10 seconds after the data is added. The managed service aspect of Amazon Kinesis relieves customers of the operational burden of creating and running a data intake pipeline. Customers can create streaming map-reduce type applications, and the elasticity of the Amazon Kinesis service enables customers to scale the stream up or down, ensuring they never lose data records prior to their expiration.

Multiple Amazon Kinesis applications can consume data from an Amazon Kinesis stream, so that multiple actions, like archiving and processing, can take place concurrently and independently. For example, two Amazon Kinesis applications can read data from the same Amazon Kinesis stream. The first Amazon Kinesis application calculates running aggregates and updates an Amazon DynamoDB table, and the second Amazon Kinesis application compresses and archives data to a data store like Amazon S3. The Amazon DynamoDB table with running aggregates is then read by a dashboard for up-to-the-minute reports.

The Amazon Kinesis Client Library enables fault-tolerant consumption of data from the Amazon Kinesis stream and provides scaling support for Amazon Kinesis applications.

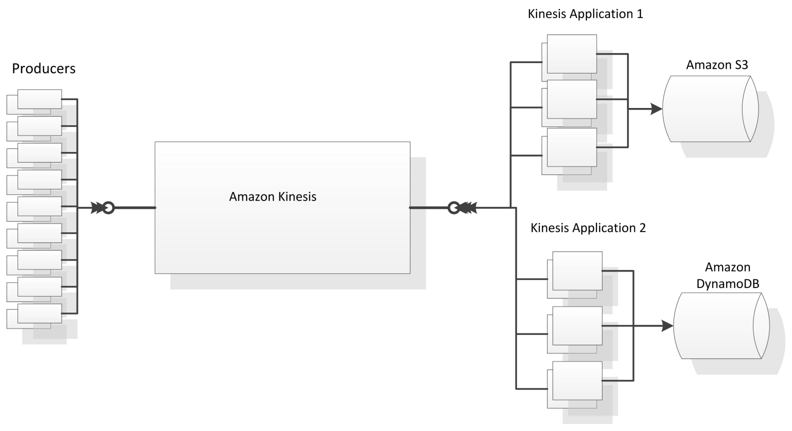
Key Concepts

This section describes the concepts and terminology you need to understand to use Amazon Kinesis effectively in the following topics.

Topics

Amazon Kinesis High-level Architecture

The following diagram illustrates the high-level architecture of Amazon Kinesis.



Amazon Kinesis Terminology

The following terms describe concepts you need to understand in order to use Amazon Kinesis effectively.

Data Record

Data records are the units of data that are stored in an Amazon Kinesis stream. Data records are composed of a sequence number, a partition key, and a data blob, which is an un-interpreted, immutable sequence of bytes. The Amazon Kinesis service does not inspect, interpret, or change the data in the blob in any way. The maximum size of a data blob (the data payload after Base64-decoding) is 50 kilobytes (KB).

Stream

A stream is an ordered sequence of data records. Each record in the stream has a sequence number (p. 4) that is assigned by the service. The data records in the stream are distributed into shards (p. 3).

Shard

A shard is a uniquely identified group of data records in an Amazon Kinesis stream. A stream is composed of multiple shards, each of which provides a fixed unit of capacity. Each open shard can support up to 5 read transactions per second, up to a maximum total of 2 MB of data read per second. Each shard can support up to 1000 write transactions per second, up to a maximum total of 1 MB data written per second. The data capacity of your stream is a function of the number of shards that you specify for the stream. The total capacity of the stream is the sum of the capacities of its shards.

If your data rate increases, then you just add more shards to increase the size of your stream. Similarly, you can remove shards if the data rate decreases.

Partition Key

The partition key is used to group data by shard within the stream. Amazon Kinesis segregates the data records belonging to a data stream into multiple shards, using the partition key associated with each data record to determine which shard a given data record belongs to. Partition keys are Unicode strings with a maximum length limit of 256 bytes. An MD5 hash function is used to map partition keys to 128-bit integer values and to map associated data records to shards. A partition key is specified by the applications putting the data into a stream.

Sequence Number

Each data record has a unique sequence number. Amazon Kinesis assigns the sequence number when a producer calls `client.putRecord()` to add a data record to the stream. Sequence numbers for the same partition key generally increase over time; the longer the time period between `PutRecord` requests, the larger the sequence numbers become. For more details on sequence numbers and partition keys, see Sequence Number (p. 20) in Using the Amazon Kinesis Service API (p. 17).

Amazon Kinesis Application

An Amazon Kinesis application is a consumer of an Amazon Kinesis stream that commonly runs on a fleet of Amazon Elastic Compute Cloud (Amazon EC2) instances. The Amazon Kinesis Client Library instantiates a record processor for each shard of the stream that it is processing. A Amazon Kinesis application uses the Amazon Kinesis Client Library to read from the Amazon Kinesis stream.

Alternatively, a Amazon Kinesis application could read from the Amazon Kinesis stream directly using the GetRecords operation for the Amazon Kinesis Service. A GetRecords operation request can retrieve up to 10 MB of data. For more information about `GetRecords`, see the Amazon Kinesis API Reference.

Output of a Amazon Kinesis application may be input for another Amazon Kinesis stream, allowing creation of complex topologies that process data in real time. A Amazon Kinesis application may also emit data to a variety of other data services such as Amazon Simple Storage Service (Amazon S3), Amazon Elastic MapReduce (Amazon EMR), or Amazon Redshift. There can be multiple Amazon Kinesis applications for one Amazon Kinesis stream, and each application can consume data from the Amazon Kinesis stream independently and concurrently.

Amazon Kinesis Client Library

The Amazon Kinesis Client Library is compiled into your application to enable fault-tolerant consumption of data from the Amazon Kinesis stream. The Amazon Kinesis Client Library ensures that for every shard there is a record processor running and processing that shard. The library also simplifies reading data from the Amazon Kinesis stream. The Amazon Kinesis Client Library uses an Amazon DynamoDB table to store control data. It creates one table per application that is processing data.

Application Name

The name for a Amazon Kinesis application that identifies the application. Each of your Amazon Kinesis applications must have a unique name that is scoped to the AWS account and region used by the application. This name is used as a name for the control table in Amazon DynamoDB and the namespace for Amazon CloudWatch metrics.

Producers

Producers are the entities that submit records to the Amazon Kinesis stream. For example, a web server sending log data to an Amazon Kinesis stream would be a producer. Producers add data to the stream, and consumers, Amazon Kinesis applications, process data from the stream.

Amazon Kinesis Sizes and Limits

Make sure you're aware of the following Amazon Kinesis sizes and limits. Many of the limits are directly related to API operations. For more information about these operations, see the Amazon Kinesis API Reference.

- The default shard limit is 10 (10 shards per account per region). For information about requesting an increase, see AWS Service Limits in the AWS General Reference.
- Data records are accessible for only 24 hours from the time they are added to an Amazon Kinesis stream.
- The maximum size of a data blob (the data payload after Base64-decoding) is 50 kilobytes (KB).
- ListStreams operation — 5 transactions per second per account.
- CreateStream, DeleteStream, MergeShards, SplitShard operations — 5 transactions per second per account.
- DescribeStream — 10 transactions per second per account.
- GetShardIterator — 5 transactions per second per account per open shard.
- A GetRecords operation request can retrieve up to 10 MB of data.
- Each open shard can support up to 5 read transactions per second, up to a maximum total of 2 MB of data read per second.
- Each shard can support up to 1000 write transactions per second, up to a maximum total of 1 MB data written per second.
- A shard iterator returned by the `getShardIterator()` method will time out after 5 minutes if you do not use it.

Read limits are based on the number of *open* shards. For more information about shard states, see Data Routing, Data Persistence, and Shard State after a Reshard (p. 29).

Related Resources

- Amazon Kinesis API Reference

Getting Started

This section describes how to get started using the Amazon Kinesis service.

Topics

Before You Begin

Before you can use Amazon Kinesis, you must sign up for an AWS account (if you don't already have one).

If you already have an AWS account, skip to the next section. Otherwise, follow these steps.

To sign up for an AWS account

1. Go to http://aws.amazon.com, and then click **Sign Up**.
2. Follow the on-screen instructions.

   Part of the sign-up procedure involves receiving a phone call and entering a PIN using the phone keypad.

For information about using AWS Identity and Access Management (IAM) with Amazon Kinesis, see Controlling Access to Amazon Kinesis Resources with IAM (p. 42).

Libraries and Tools

The following libraries and tools will help you work with Amazon Kinesis:

- The Amazon Kinesis Service API is the basic set of operations that the Amazon Kinesis service supports. Using the Amazon Kinesis Service API shows you how to perform basic operations using Java code.

- The AWS SDKs for Java, JavaScript, .NET, Node.js, PHP, Python, and Ruby include Amazon Kinesis support and samples.

  Note
  If your version of the AWS SDK for Java does not include Kinesis samples, you can also download them from GitHub.
- The Amazon Kinesis Client Library (KCL) provides an easy-to-use programming model for processing data. The Amazon Kinesis Client Library is used throughout this guide and will help you get started quickly with Amazon Kinesis. The KCL can be used in conjunction with the Amazon Kinesis Connector Library to reliably move data from Amazon Kinesis to Amazon DynamoDB, Amazon Redshift, and Amazon S3.
- The AWS Command Line Interface supports Amazon Kinesis. The AWS CLI allows you to control multiple AWS services from the command line and automate them through scripts.
- (Optional) The Amazon Kinesis Connector Library helps you integrate Amazon Kinesis with other AWS services.

Requirements

In order to use the Amazon Kinesis Client Library, you'll need to ensure that your Java development environment meets the following requirements:

- Java 1.7 (Java SE 7 JDK) or later. You can download the latest Java software from http://developers.sun.com/downloads.
- Apache Commons package (Code, HTTP Client, and Logging)
- Jackson JSON processor

Note
The AWS SDK for Java includes Apache Commons and Jackson in the third-party folder. Note, however, that the SDK for Java will work with Java 1.6, while the Amazon Kinesis Client Library requires Java 1.7.

Overview of Using Amazon Kinesis

Before creating an Amazon Kinesis stream, you must determine the size of your stream. For more information, see How Do I Size an Amazon Kinesis Stream? (p. 8)

The following steps provide an overview of using the Amazon Kinesis service.

To get started using the Amazon Kinesis service

1. Create the Amazon Kinesis stream with an appropriate number of shards.
2. Configure the data sources to continually push data into the Amazon Kinesis stream.
3. Build a Amazon Kinesis application to consume data from the Amazon Kinesis stream and connect the application to the Amazon Kinesis stream.
4. Operate the Amazon Kinesis application. The Amazon Kinesis Client Library will help with scale-out and fault-tolerant processing.

These steps are described in detail in the getting started exercise, which begins with Step 1: Create an Amazon Kinesis Stream (p. 8).

How Do I Size an Amazon Kinesis Stream?

Data is stored in an Amazon Kinesis stream and a stream is composed of multiple shards. You must determine the size of the stream you need before you create the Amazon Kinesis stream. To determine the size of an Amazon Kinesis stream, you must determine how many shards you will need for the stream.

Determining How Many Shards You Need

An Amazon Kinesis stream is composed of multiple shards. Each open shard can support up to 5 read transactions per second, up to a maximum total of 2 MB of data read per second. Each shard can support up to 1000 write transactions per second, up to a maximum total of 1 MB data written per second. Multiple different Amazon Kinesis applications can read from a shard.

Before you create an Amazon Kinesis stream, you need to determine the initial size of the stream. You can dynamically resize your Amazon Kinesis stream or add and remove shards after the stream has been created and while it has a Amazon Kinesis application running that is consuming data from the stream.

To determine the initial size of an Amazon Kinesis stream, you need the following input values.

1. The average size of the data record written to the stream in kilobytes (KB), rounded up to the nearest 1 KB, the data size (`average_data_size_in_KB`).
2. The number of data records written to and read from the stream per second, that is, transactions per second (`number_of_transactions_per_second`).
3. The number of Amazon Kinesis applications that consume data concurrently and independently from the Amazon Kinesis stream, that is, the consumers (`number_of_consumers`).
4. The incoming write bandwidth in KB (`incoming_write_bandwidth_in_KB`),

   which is equal to the `average_data_size_in_KB` multiplied by the `number_of_transactions_per_seconds`.
5. The outgoing read bandwidth in KB (`outgoing_read_bandwidth_in_KB`),

   which is equal to the `incoming_write_bandwidth_in_KB` multiplied by the `number_of_consumers`.

You can calculate the initial number of shards (`number_of_shards`) that your Amazon Kinesis stream will need by using the input values you determine in the following formula:

```
number_of_shards = max(incoming_write_bandwidth_in_KB/1000,
outgoing_read_bandwidth_in_KB/2000)
```

Step 1: Create an Amazon Kinesis Stream

This getting started exercise shows you how to create and configure an Amazon Kinesis stream and how to build a Amazon Kinesis application to interact with the stream.

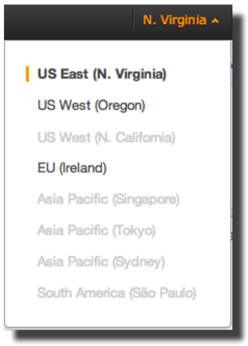
Before you create an Amazon Kinesis stream, you must determine the size that you need the stream to be. For information about determining stream size, see How Do I Size an Amazon Kinesis Stream? (p. 8).

To create a stream

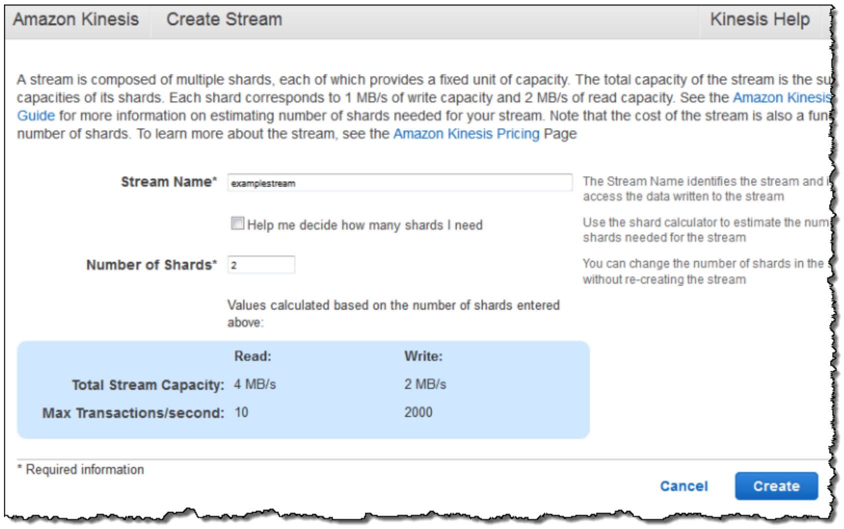
1. Sign in to the AWS Management Console and go to the Amazon Kinesis console at https://console.aws.amazon.com/kinesis/.

If you haven't yet signed up for the Amazon Kinesis service, you'll be prompted to sign up when you go to the console.

2. In the navigation bar, expand the region selector and select a region.



3. Click **Create Stream**.
4. On the **Create Stream** page, enter a name for your stream and the number of shards you need, and then click **Create**.



A stream is composed of multiple shards, each of which provides a fixed unit of capacity. The total capacity of the stream is the su
capacities of its shards. Each shard corresponds to 1 MB/s of write capacity and 2 MB/s of read capacity. See the Amazon Kinesis
Guide for more information on estimating number of shards needed for your stream. Note that the cost of the stream is also a fun
number of shards. To learn more about the stream, see the Amazon Kinesis Pricing Page

Stream Name* examplestream — The Stream Name identifies the stream and i access the data written to the stream

Help me decide how many shards I need — Use the shard calculator to estimate the num shards needed for the stream

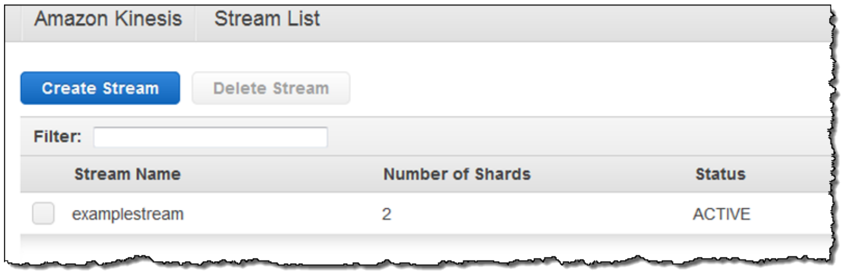
Number of Shards* 2 — You can change the number of shards in the without re-creating the stream

Values calculated based on the number of shards entered above:

	Read:	Write:
Total Stream Capacity:	4 MB/s	2 MB/s
Max Transactions/second:	10	2000

* Required information

On the **Stream List** page, your stream's **Status** is CREATING while the stream is being created. When the stream is ready to use, the **Status** changes to ACTIVE.



Stream Name	Number of Shards	Status
examplestream	2	ACTIVE

5. Click the name of your stream. The **Stream Details** page displays a summary of your stream configuration, along with monitoring information.

For information on how to create an Amazon Kinesis stream programmatically, see Using the Amazon Kinesis Service API (p. 17).

For information about using AWS Identity and Access Management (IAM) with Amazon Kinesis, see Controlling Access to Amazon Kinesis Resources with IAM (p. 42).

Step 2: Configure the Producers to Continually Push Data into the Amazon Kinesis Stream

Producers submit data records to the Amazon Kinesis stream. After you create the Amazon Kinesis stream, configure the producers to put data into the stream.

To put data into the stream, call the PutRecord operation for the Amazon Kinesis service on your Amazon Kinesis stream. Each of the `PutRecord` calls will need the name of the Amazon Kinesis stream, a partition key, and the data blob to be added to the Amazon Kinesis stream. The partition key is used to determine which shard in the stream the data record will be added to. For more information about `PutRecord`, see the Amazon Kinesis API Reference.

All the data in the shard is sent to the same Amazon Kinesis worker that is processing the shard. Which partition key you will use depends on your application logic. For example, if the application is calculating a bill for a customer using data on the customer's usage, a suitable partition key would be `customer ID`.

The samples in the Using the Amazon Kinesis Service API (p. 17) section show you how to add data to an Amazon Kinesis stream.

Important
Data records are accessible for only 24 hours from the time they are added to an Amazon Kinesis stream.

Step 3: Build a Amazon Kinesis Application to Consume Data from the Amazon Kinesis Stream and Connect It to the Amazon Kinesis Stream

A Amazon Kinesis application can use the Amazon Kinesis Client Library to simplify parallel processing of the stream by a fleet of Amazon Kinesis workers running on a fleet of Amazon Elastic Compute Cloud (Amazon EC2) instances. The client library simplifies writing code to read from the shards in the stream and ensures that there is a worker allocated to every shard in the stream. The Amazon Kinesis Client Library also provides help with fault tolerance by providing checkpointing capabilities. The best way to get started is to work from the samples in the Developing Record Consumer Applications with the Amazon Kinesis Client Library (p. 32).

To create a Amazon Kinesis application, you implement the Amazon Kinesis Client Library's `IRecordProcessor` Java class. This class implements the business logic for processing the data retrieved from the Amazon Kinesis stream. You also need to specify the configuration for the Amazon Kinesis application. This configuration information includes the name of the Amazon Kinesis stream that the data will be retrieved from and a unique name for the Amazon Kinesis application.

Each of your Amazon Kinesis applications must have a unique name that is scoped to the AWS account and region used by the application. This name is used as a name for the control table in Amazon DynamoDB and the namespace for Amazon CloudWatch metrics. When your Amazon Kinesis application starts up, it creates an Amazon DynamoDB table to store the application state, connects to the specified Amazon Kinesis stream, and then starts consuming data from the stream. You can view the Amazon Kinesis application metrics in the AWS CloudWatch console.

The Developing Record Consumer Applications with the Amazon Kinesis Client Library (p. 32) section walks you through writing a Amazon Kinesis application.

Step 4: Operate the Amazon Kinesis Application

You can follow your own best practices for deploying code to an Amazon EC2 instance when you deploy a Amazon Kinesis application. For example, you can add your Amazon Kinesis application to one of your Amazon EC2 AMIs.

You can elastically scale the entire Amazon Kinesis application by running it on multiple Amazon EC2 instances under an Auto Scaling group. Using an Auto Scaling group can help automatically start new instances in the event of an Amazon EC2 instance failure and can also elastically scale the number of instances as the load on the Amazon Kinesis application changes over time. Auto Scaling groups ensure that a certain number of Amazon EC2 instances are always running for the Amazon Kinesis application.

To trigger scaling events in the Auto Scaling group, you can specify metrics such as CPU and memory utilization to scale up or down the number of Amazon EC2 instances processing data from the Amazon Kinesis stream. For more information on Auto Scaling, see the Auto Scaling Getting Started Guide.

Where Do I Go From Here?

Now that you have read through the getting started section, you should read the following sections to learn more about Amazon Kinesis.

- Using the Amazon Kinesis Service API (p. 17) — This section walks through some short Java code samples that show how to perform basic operations with the Amazon Kinesis service.
- Developing Record Consumer Applications with the Amazon Kinesis Client Library (p. 32) — This section shows you how to develop an Amazon Kinesis application using the Amazon Kinesis Client Library.
- Analyze Amazon Kinesis Data in the *Amazon Elastic MapReduce Developer Guide* — Two examples of how to use Amazon EMR clusters to read and process Amazon Kinesis streams directly.

Visualizing Web Traffic with the Amazon Kinesis Sample Application

This section helps you to get started using the Amazon Kinesis service by walking you through a sample application that demonstrates how to use Amazon Kinesis to process and analyze streaming data in real time. You can set up and run the application, then see the analysis results displayed in real time.

This section covers the following tasks:

Amazon Kinesis Data Visualization Sample Application

The Amazon Kinesis data visualization sample application demonstrates how to use Kinesis for real-time data ingestion and analysis. The sample application generates simulated visitor counts from various URLs. It sends the counts to Kinesis, then calculates how many visitors originated from a particular URL.

A typical Kinesis application takes data from data generators called *producers* and puts it in the form of *data records* into the Amazon Kinesis stream. For example, the sample application's stream writer component puts data into the Kinesis stream. The records are randomly generated data that are meant to represent visitor counts (HTTPS requests) on various websites.

When an application creates the data stream, it specifies the number of *shards* the stream contains. Amazon Kinesis uses a value called a *partition key* to determine to which shard a particular record belongs. The sample application creates a data stream containing 2 shards, which together have a write capacity of 2 MB/sec and a read capacity of 4 MB/sec.

You use the Amazon Kinesis Client Library (KCL) to create a Kinesis client application, also called a *consumer* application. A consumer application retrieves data from the shards in the stream and processes them; the KCL ensures that there is a record processor to process data from each shard. The KCL uses an Amazon DynamoDB table to track the progress of data retrieval from the shards.

In the sample application's consumer component, the record processor counts visitor requests as they are read from the shards. It calculates the count of visitor requests over a sliding window, and persists the counts to an Amazon DynamoDB database for subsequent display. The sample application employs Top-N analysis over a sliding window. These are common processing patterns that are demonstrative of real-world data analyses that are implemented using Amazon Kinesis.

Before You Begin

This documentation helps you set up, run, and view the results of the Kinesis data visualization sample application. To get started with the sample application, you first need to do the following:

- Set up a computer with a working Internet connection.
- Sign up for an AWS account.

Running the Sample Application

This procedure creates a sample application using an AWS CloudFormation template. The application has a stream writer that randomly generates records and sends them to a Kinesis stream, a counter that counts the number of HTTPS requests to a resource, and a web server that displays the outputs of the stream processing data as a continuously updated graph.

Note
When you create this application, it will immediately deploy and start running, so there will be a nominal cost for AWS service usage. AWS has made efforts to keep the sample application resources in the free tier, except for the Kinesis service usage. Make sure to delete your AWS CloudFormation stack and Amazon DynamoDB instances as described in Deleting Sample Application Resources (p. 16).

Note
The Amazon Kinesis data visualization sample application is configured to work only with AWS resources in the US East (Northern Virginia) Region (us-east-1).

Note
You might experience a page load issue with Internet Explorer 11. For information on how to fix this, see "Troubleshooting" in Getting Started with the AWS Management Console.

To set up and run the sample application

1. Download the sample application template (kinesis-data-vis-sample-app.template) from GitHub. Save the template file to your computer's local drive.
2. Log in to the AWS CloudFormation console at https://console.aws.amazon.com/cloudformation/home.
3. The console displays a list of AWS CloudFormation stacks (if any currently exist). Click **Create New Stack** and provide the following settings:

   a. **Name**: Give the stack an identifiable name, such as 'KinesisDataVisSampleApp'.
   b. **Template**: In **Source**, select **Upload a template to Amazon S3**.
   c. Click **Choose File**, and in the **Open** dialog box, browse and select the template file. Click **Open**.
   d. Click **Next**.

4. In **Specify Parameters**, the following parameter values will be provided in the fields:

   a. **ApplicationArchive**:
      https://github.com/awslabs/amazon-kinesis-data-visualization-sample/releases/download/v1.0.0/amazon-kine
      This is the URL of the sample application archive file. It allows you to package and store modifications you make to the application. To create an archive, run `mvn package`, upload the target .zip file, and store it in S3 with global public read permissions. Use the link to that archive as the **ApplicationArchive** property for the template.
      InstanceType: t1.micro

   b. **KeyName**: KeyName is optional; leave it blank. This is the name of the key pair you use to connect to the EC2 instance in which the application runs. For more information, see Amazon EC2 Key Pairs.
   c. **SSHLocation**: 0.0.0.0/0
   d. Click **Next**.

5. In **Options**:

   a. **Tags**
      None. Tags are optional.

   b. **Advanced**
      Rollback on failure: Yes
      This cleans up any resources in case of failure so that you are not inadvertently billed. However, if you want to debug the cause of the stack failure, you need to set this option to **No**.

6. In **Review**, you see a summary of all the settings you configured for the stack. Under **Capabilities**, the page will notify you that IAM resources will be created. These policies grant access to an account's resources. The sample application grants read/write access for the EC2 instance to the resources that it creates.

   a. Check the option **I acknowledge that this template might cause AWS CloudFormation to create IAM resources**.
   b. Click **Create**.
   c. The AWS CloudFormation service requires several minutes to create the stack. When the status changes to **CREATE_COMPLETE**, the application is deployed and running.

   Note The sample application failed to complete its initiation if you see that the stack has a status of **ROLLBACK_IN_PROGRESS** in the AWS CloudFormation console. You can retry the sample application by going through the above steps again.

The AWS CloudFormation template creates a Kinesis stream with two shards, an EC2 instance, and two Amazon DynamoDB tables. The template also creates some auxiliary resources, such as an IAM Policy and EC2 security groups, which can be viewed on the Resources tab in the AWS CloudFormation console.

Viewing the Results of the Sample Application

When you set up and run the sample application according to the previous procedure, AWS CloudFormation creates an application stack for the Kinesis application. You can now use the AWS Management Console at https://console.aws.amazon.com/console/home to view the generated data analysis graph and the stream details.

The Kinesis application contains a stream with two shards for data processing. You can view the stream details by navigating to the Amazon Kinesis console at https://console.aws.amazon.com/kinesis/home, which displays all the streams that you have created.

To view the real-time data analysis graph

1. When the stack has been created, open (or refresh if necessary) the AWS CloudFormation console. Click the **KinesisDataVisSample** stack to select it. This causes stack details to appear in the lower portion of the console window.
2. Select the **Outputs** tab, then click the link in **URL**. The form of the URL should be similar to "http://ec2-xx-xx-xx-xx.compute-1.amazonaws.com."
3. The real-time data analysis graph appears on a separate page, titled **Amazon Kinesis Data Visualization Sample**. It displays the number of requests sent by the referring URL over a 10 second span, and the chart is updated every 1 second. The span of the graph is the last 2 minutes.

To view the stream details

1. In the AWS Management Console, open the **Services** menu and click **Kinesis** to navigate to the Amazon Kinesis console.
2. In the Amazon Kinesis console, on the **Stream List** page, you should see a stream with 2 shards and a status of **ACTIVE**. The stream name begins with 'KinesisDataVisSampleApp-KinesisStream', appended with a randomly generated unique identifier.
3. Click on the link to the stream to open the **Stream Details** page.
4. The **Stream Details** page displays statistics on the stream in the following graphs:

   a. Write Capacity and Read Capacity (in bytes per second)
   b. Put Latency and Get Latency (in milliseconds)
   c. Put Requests and Get Requests (direct count)

The stream contains 2 open shards, which can receive data from producers and make data available to consumers. Each shard has a write capacity of 2 MB/sec and a read capacity of 4 MB/sec. The Put and Get Requests graphs display actual put and get activity as blue lines. The graphs display actual operation data, because the sample application stack is already putting records into the stream, and the consumer component is getting and processing data from the stream.

To view instance data in the EC2 Console

1. In the AWS Management Console, open the **Services** menu and click **EC2** to navigate to the EC2 console at https://console.aws.amazon.com/ec2/v2/home.
2. In the EC2 console, in the menu at left, click **Instances**. All your instances are listed (active or terminated), including one for the instance you recently created.
3. Click the instance for the Kinesis sample application; a number of tabs will appear below the list. Select the **Description** tab.
4. Copy the Public DNS; it will have a form similar to "ec2-xx-xx-xxx-xxx.compute-1.amazonaws.com."
5. Paste the Public DNS value in your browser. You should see a continuously updated graph generated by the sample application.

To view the Amazon DynamoDB tables

1. In the AWS Management Console, open the **Services** menu and click **DynamoDB** to navigate to the Amazon DynamoDB console.

2. The Amazon DynamoDB console lists all the tables in your AWS account. The list includes the two tables that the sample application creates. Each table name begins with the following names, appended with a randomly generated unique identifier:
   a. KinesisDataVisSampleApp-CountsDynamoDBTable
   b. KinesisDataVisSampleApp-KCLDynamoDBTable
3. Click each table in the list to see details on it in the tabs below. The status should be **ACTIVE** and each should have a read capacity of 10 units and a write capacity of 5 units. (These units are not real and are only meant to represent units of capacity.)

Deleting Sample Application Resources

AWS has made efforts to keep the sample application resources in the free tier, except for the Kinesis service usage. The application creates two shards and will generate charges for shard usage while it runs. To ensure that your AWS account does not continue to be billed, make sure to delete your AWS CloudFormation stack and Amazon DynamoDB instances.

To delete application resources

1. Open the AWS Management Console (https://console.aws.amazon.com/) and select the AWS CloudFormation service; a list of your stacks appears.
2. Select the stack that you created and click **Delete Stack**. The status changes to **DELETE_IN_PROGRESS** while AWS CloudFormation cleans up the resources associated with the sample application. When AWS CloudFormation is finished removing all resources, the stack is removed from the list.

Using the Amazon Kinesis Service API

This section walks through some short Java code samples that show you how to perform basic operations with Amazon Kinesis. The samples are intended to illustrate the semantics of the Amazon Kinesis service; they do not represent production-ready code in that they do not check for all possible exceptions and do not account for all possible security or performance considerations.

Note
Although the samples in this section are in Java, all AWS SDK platforms support Amazon Kinesis.

This section covers the following operations:

- Create the Amazon Kinesis Client (p. 17)
- Create the Stream (p. 18)
- List All Streams in an AWS Account (p. 19)
- Add Data to a Stream (p. 20)
- Retrieve the Shards from a Stream (p. 21)
- Get Data from the Shards in a Stream (p. 22)
- Resharding a Stream (p. 26)
- Delete the Stream (p. 30)

Although this section describes the Amazon Kinesis service operations for reading data from an Amazon Kinesis stream, we recommend using the `IRecordProcessor` interface for applications that consume data from Amazon Kinesis streams. For more information, see the IRecordProcessor sample and Developing Record Consumer Applications with the Amazon Kinesis Client Library (p. 32).

Create the Amazon Kinesis Client

The first step in using the Amazon Kinesis service API is to instantiate a client object for the service. The following code creates the Amazon Kinesis service client and sets the endpoint information for the client. This overload of `setEndpoint()` also includes the service name and region.

```
client = new AmazonKinesisClient();
client.setEndpoint(endpoint, serviceName, regionId);
```

Set `serviceName` to `kinesis`. For more information about Amazon Kinesis endpoints and regions, go to Regions and Endpoints in the AWS General Reference.

Create the Stream

To create a stream, instantiate a `CreateStreamRequest` object and specify a name for the stream and the number of shards that the stream will use.

```
CreateStreamRequest createStreamRequest = new CreateStreamRequest();
createStreamRequest.setStreamName( myStreamName );
createStreamRequest.setShardCount( myStreamSize );
```

The stream name identifies the stream. The name is scoped to the AWS account used by the application. It is also scoped by region. That is, two streams in two different AWS accounts can have the same name, and two streams in the same AWS account but in two different regions can have the same name.

The throughput of the stream is a function of the number of shards; more shards are required for greater provisioned throughput. More shards also increase the cost that AWS charges for the stream. See How Do I Size an Amazon Kinesis Stream? (p. 8) for information about how to calculate the appropriate number of shards for your application.

After the `createStreamRequest` object is configured, create the stream by calling the `createStream()` method on the Amazon Kinesis client. After calling `createStream()`, you need to wait for the stream to reach an "ACTIVE" state before performing any further operations on the stream. To check the state of the stream, you need to call the `describeStream()` method. However, `describeStream()` throws an exception if the stream does not exist. Therefore, enclose the `describeStream()` call in a try/catch block.

```
client.createStream( createStreamRequest );
DescribeStreamRequest describeStreamRequest = new DescribeStreamRequest();
describeStreamRequest.setStreamName( myStreamName );

long startTime = System.currentTimeMillis();
long endTime = startTime + ( 10 * 60 * 1000 );
while ( System.currentTimeMillis() < endTime ) {
  try {
    Thread.sleep(20 * 1000);
  }
  catch ( Exception e ) {}

  try {
    DescribeStreamResult describeStreamResponse = client.describeStream( de
scribeStreamRequest );
    String streamStatus = describeStreamResponse.getStreamDescription().get
StreamStatus();
    if ( streamStatus.equals( "ACTIVE" ) ) {
      break;
    }
```

```
   //
    // sleep for one second
    //
    try {
      Thread.sleep( 1000 );
    }
    catch ( Exception e ) {}
  }
  catch ( ResourceNotFoundException e ) {}
}
if ( System.currentTimeMillis() >= endTime ) {
  throw new RuntimeException( "Stream " + myStreamName + " never went active"
);
}
```

List All Streams in an AWS Account

Streams are scoped to the AWS account associated with the AWS credentials used to instantiate the Amazon Kinesis client and also to the region specified for the Amazon Kinesis client. A given AWS account could have many streams active at one time. The following code shows how to list all the streams in the AWS account.

The code first creates a new instance of `ListStreamsRequest` and calls its `setLimit()` method to specify that a maximum of 20 streams should be returned for each call to `listStreams()`. Note that the service might return fewer streams than specified by this limit even if there are more streams than that in the account and region. If you do not specify a value for `setLimit()`, the service returns a number of streams less than or equal to the number in the account. To ensure that you retrieve all the streams, use the `getHasMoreStreams()` method described further below.

The code passes the `listStreamsRequest` to the `listStreams()` method of the Amazon Kinesis service client. The return value of calling `listStreams()` is stored in a `ListStreamsResult` object. The code calls the `getStreamNames()` method on this object and stores the returned stream names in the `streamNames` list.

```
ListStreamsRequest listStreamsRequest = new ListStreamsRequest();
listStreamsRequest.setLimit(20);
ListStreamsResult listStreamsResult = client.listStreams(listStreamsRequest);
List<String> streamNames = listStreamsResult.getStreamNames();
```

The code then calls the `getHasMoreStreams()` method on `listStreamsRequest` to check if there are additional streams available beyond the ones returned in the initial call to `listStreams()`. If so, the code calls the `setExclusiveStartStreamName()` method with the name of the last stream that was returned in the previous call to `listStreams()`. The `setExclusiveStartStreamName()` method causes the next call to `listStreams()` to start after that stream. The group of stream names returned by that call is then added to the `streamNames` list. This process continues until all the stream names have been collected in the list.

```
  while (listStreamsResult.getHasMoreStreams()) {
    if (streamNames.size() > 0) {
      listStreamsRequest.setExclusiveStartStreamName(streamNames
       .get(streamNames.size() - 1));
    }
    listStreamsResult = client.listStreams(listStreamsRequest);
    streamNames.addAll(listStreamsResult.getStreamNames());
  }
```

Streams returned by `listStreams()` can be in any of the following states:

- CREATING
- ACTIVE
- UPDATING
- DELETING

You can check the state of a given stream using the `describeStream()` method shown in the section on how to create a stream.

Add Data to a Stream

You add data to the stream in the form of records. A record is a data structure that contains the data to be processed. After you store the data in the record, the Amazon Kinesis service does not inspect, interpret, or change the data in any way. Also associated with each record is a sequence number and a partition key.

Note that as your source application is adding data to the stream using the Amazon Kinesis service API, consumer applications are, at the same time, processing data off the stream.

> **Important**
> Data records are accessible for only 24 hours from the time they are added to an Amazon Kinesis stream.

Sequence Number

Each data record has a unique sequence number. The sequence number is assigned by Amazon Kinesis after you call `client.putRecord()`, which adds the data record to the stream. Sequence numbers for the same partition key generally increase over time; the longer the time period between `PutRecord` requests, the larger the sequence numbers become.

When puts occur in quick succession, the returned sequence numbers are not guaranteed to increase because the put operations appear essentially as simultaneous to the service. To guarantee strictly increasing sequence numbers for the same partition key, use the `SequenceNumberForOrdering` parameter, as shown in the Partition Key (p. 20) code sample below.

Whether or not you use `SequenceNumberForOrdering`, records that Amazon Kinesis receives through a `GetRecords` call are strictly ordered by sequence number.

> **Note**
> Sequence numbers cannot be used as indexes to sets of data within the same stream. To logically separate sets of data, use partition keys or create a separate stream for each data set.

Partition Key

You specify the partition key for the record. The partition key is used to group data within the stream. A data record is assigned to a shard within the stream based on its partition key. Specifically, the service uses the partition key as input to a hash function that maps the partition key (and associated data) to a specific shard.

As a result of this hashing mechanism, all data records with the same partition key map to the same shard within the stream. However, if the number of partition keys exceeds the number of shards, some shards necessarily contain records with different partition keys. From a design standpoint, to ensure that all your shards are well utilized, the number of shards (specified by the `setShardCount()` method of `CreateStreamRequest`) should be substantially less than the number of unique partition keys, and the amount of data flowing to a single partition key should be substantially less than the capacity of a shard.

The following code creates ten data records, distributed across two partition keys, and puts them in a stream called `myStreamName`.

```
for (int j = 0; j < 10; j++) {
  PutRecordRequest putRecordRequest = new PutRecordRequest();
  putRecordRequest.setStreamName( myStreamName );
  putRecordRequest.setData(ByteBuffer.wrap( String.format( "testData-%d", j
).getBytes() ));
  putRecordRequest.setPartitionKey( String.format( "partitionKey-%d", j%5 ));

  putRecordRequest.setSequenceNumberForOrdering( sequenceNumberOfPreviousRecord
 );
  PutRecordResult putRecordResult = client.putRecord( putRecordRequest );
  sequenceNumberOfPreviousRecord = putRecordResult.getSequenceNumber();
}
```

The preceding code sample uses `setSequenceNumberForOrdering` to guarantee strictly increasing ordering within each partition key. To use this parameter effectively, set the `SequenceNumberForOrdering` of the current record (record *n*) to the sequence number of the preceding record (record *n-1*). To get the sequence number of a record that has been added to the stream, call `getSequenceNumber()` on the result of `putRecord()`.

The `SequenceNumberForOrdering` parameter ensures strictly increasing sequence numbers for the same partition key, when the same client calls `PutRecord`. `SequenceNumberForOrdering` does not provide ordering guarantees across records that are added from multiple concurrent applications, or across multiple partition keys.

Retrieve the Shards from a Stream

The response object returned by the `describeStream()` method enables you to retrieve information about the shards that comprise the stream. To retrieve the shards, call the `getShards()` method on this object. This method might not return all the shards from the stream in a single call. In the following code, we check the `getHasMoreShards()` method on `getStreamDescription` to see if there are additional shards that were not returned. If so, that is, if this method returns true, we continue to call `getShards()` in a loop, adding each new batch of returned shards to our list of shards. The loop exits when `getHasMoreShards()` returns false, that is, all shards have been returned. Note that `getShards()` does not return shards that are in the EXPIRED state. For more information about shard states, including the EXPIRED state, see Data Routing, Data Persistence, and Shard State after a Reshard (p. 29).

```
DescribeStreamRequest describeStreamRequest = new DescribeStreamRequest();
describeStreamRequest.setStreamName( myStreamName );
List<Shard> shards = new ArrayList<>();
String exclusiveStartShardId = null;
do {
    describeStreamRequest.setExclusiveStartShardId( exclusiveStartShardId );
    DescribeStreamResult describeStreamResult = client.describeStream( describe
StreamRequest );
    shards.addAll( describeStreamResult.getStreamDescription().getShards() );
    if (describeStreamResult.getStreamDescription().getHasMoreShards() &&
shards.size() > 0) {
        exclusiveStartShardId = shards.get(shards.size() - 1).getShardId();
    } else {
        exclusiveStartShardId = null;
    }
} while ( exclusiveStartShardId != null );
```

Get Data from the Shards in a Stream

The Amazon Kinesis service API provides the `getShardIterator()` and `getRecords()` methods to retrieve data from an Amazon Kinesis stream. These methods represent a "pull" model in which your code draws data directly from specified shards in the stream.

In general, however, we recommend that you use the `IRecordProcessor` interface supported in the Amazon Kinesis Client Library (KCL) to retrieve stream data. The `IRecordProcessor` interface represents a "push" model in which all you need to do is implement the code that processes the data. The Amazon Kinesis Client Library does the work of retrieving data records from the stream and delivering them to your application code. In addition, the Amazon Kinesis Client Library provides failover, recovery, and load balancing functionality.

However, in some cases you might prefer to use the Amazon Kinesis service API. For example, you might use the API to implement custom tools for monitoring or debugging your Amazon Kinesis streams. This section focuses on the Amazon Kinesis service API.

Important
Data records are accessible for only 24 hours from the time they are added to an Amazon Kinesis stream.

Using Shard Iterators

When using the Amazon Kinesis service API, you retrieve records from the stream on a per-shard basis. For each shard, and for each batch of records that you retrieve from that shard, you need to obtain a *shard iterator*. The shard iterator is used in the `getRecordsRequest` object to specify the shard from which records should be retrieved. The type associated with the shard iterator determines the point in the shard from which the records should be retrieved (see below for more details).

The initial shard iterator is obtained using the `getShardIterator()` method. Shard iterators for additional batches of records are obtained using the `getNextShardIterator()` method of the `getRecordsResult` object returned by the `getRecords()` method.

Important
A shard iterator returned by the `getShardIterator()` method will time out after five minutes if you do not use it.

To obtain the initial shard iterator, instantiate a `GetShardIteratorRequest` and pass it to the `getShardIterator()` method. To configure the request, you need to specify the stream and the shard ID. For information about how to obtain the streams in your AWS account, see List All Streams in an AWS Account (p. 19). For information about how to obtain the shards in a stream, see Retrieve the Shards from a Stream (p. 21).

```
  String shardIterator;
  GetShardIteratorRequest getShardIteratorRequest = new GetShardIteratorRe
quest();
  getShardIteratorRequest.setStreamName(myStreamName);
  getShardIteratorRequest.setShardId(shard.getShardId());
  getShardIteratorRequest.setShardIteratorType("TRIM_HORIZON");

  GetShardIteratorResult getShardIteratorResult = client.getShardIterator(get
ShardIteratorRequest);
  shardIterator = getShardIteratorResult.getShardIterator();
```

The sample code above specifies TRIM_HORIZON as the iterator type when obtaining the initial shard iterator. This iterator type means that records should be returned beginning with the first record added to the shard — rather than beginning with the most recently added record, also known as the *tip*. The possible iterator types are:

- AT_SEQUENCE_NUMBER
- AFTER_SEQUENCE_NUMBER
- TRIM_HORIZON
- LATEST

For more details on iterator types, go to the Amazon Kinesis API Reference.

Some iterator types require that you specify a sequence number in addition to the type. For example:

```
getShardIteratorRequest.setShardIteratorType("AT_SEQUENCE_NUMBER");
getShardIteratorRequest.setStartingSequenceNumber(specialSequenceNumber);
```

After you've obtained a record via `getRecords()`, you can get the sequence number for the record by calling the record's `getSequenceNumber()` method.

```
record.getSequenceNumber()
```

In addition, the code that adds records to the data stream can get the sequence number for an added record by calling `getSequenceNumber()` on the result of `putRecord()`.

```
lastSequenceNumber = putRecordResult.getSequenceNumber();
```

You can use sequence numbers to guarantee strictly increasing ordering of records. For more information, see the code sample in the Partition Key (p. 20) section above.

Using GetRecords()

After you've obtained the shard iterator, instantiate a `GetRecordsRequest` object. Specify the iterator for the request using the `setShardIterator()` method.

Optionally, you can also set the number of records to retrieve using the `setLimit()` method. The number of records returned by `GetRecords()` is always equal to or less than this limit. If you do not specify this limit, `GetRecords()` returns up to 1 MB of retrieved records. The sample code below sets this limit to 1000.

If no records are returned, that means no data records are *currently* available from this shard at the sequence number referenced by the shard iterator. When this situation occurs, your application should wait for an appropriate amount of time given the nature of the data sources to the stream — in all cases, at least one second — and then retry getting data from the shard using the shard iterator returned by the preceding call to GetRecords. Note that there is about a three-second latency from the time that a record is added to the stream to the time that it is available from `GetRecords()`.

Pass the `getRecordsRequest` to the `getRecords()` method and capture the returned value as a `getRecordsResult` object. To get the data records, call the `getRecords()` method on the `getRecordsResult` object.

```
GetRecordsRequest getRecordsRequest = new GetRecordsRequest();
getRecordsRequest.setShardIterator(shardIterator);
getRecordsRequest.setLimit(1000);

GetRecordsResult getRecordsResult = client.getRecords(getRecordsRequest);
List<Record> records = getRecordsResult.getRecords();
```

To prepare for another call to `getRecords()`, obtain the next shard iterator from `getRecordsResult`.

```
shardIterator = getRecordsResult.getNextShardIterator();
```

For best results, sleep for at least one second (1000 milliseconds) between calls to `getRecords()` to avoid exceeding the limit on `getRecords()` frequency.

```
try {
  Thread.sleep(1000);
}
catch (InterruptedException e) {}
```

Note
Typically, calls to `GetRecords()` should be made in a loop, even when you're retrieving a single record in a test scenario. A single call to `GetRecords()` may return an empty record list, even when the shard contains more records at later sequence numbers. When this occurs, the `NextShardIterator` returned along with the empty record list references a later sequence number in the shard, and successive `GetRecords()` calls will eventually return the records. The sample below illustrates the use of a loop.

GetRecords() Sample

The following code sample reflects the `GetRecords()` tips in this section, including making calls in a loop.

```
//Continuously read data records from shard.
```

```
List<Record> records;

while (true) {

  //Create new GetRecordsRequest with existing shardIterator.
  //Set maximum records to return to 1000.
  GetRecordsRequest getRecordsRequest = new GetRecordsRequest();
  getRecordsRequest.setShardIterator(shardIterator);
  getRecordsRequest.setLimit(1000);

  GetRecordsResult result = client.getRecords(getRecordsRequest);

  //Put result into record list. Result may be empty.
  records = result.getRecords();

  try {
    Thread.sleep(1000);
  }
  catch (InterruptedException exception) {
    throw new RuntimeException(exception);
  }

  shardIterator = result.getNextShardIterator();
}
```

If you are using the Amazon Kinesis Client Library, you may notice that the KCL makes multiple calls before returning data. This behavior is by design and does not indicate a problem with the KCL or your data.

Adapting to a Reshard

If `getRecordsResult.getNextShardIterator()` returns null, it indicates the following: a shard split or merge has occurred that involved this shard, this shard is now in a CLOSED state, and you have read all available data records from this shard.

In this scenario, you should re-enumerate the shards in the stream to pick up the new shards that were created by the split or merge.

In the case of a split, the two new shards both have `parentShardId` equal to the shard ID of the shard that you were processing previously. The value of `adjacentParentShardId` for both of these shards is null.

In the case of a merge, the single new shard created by the merge has `parentShardId` equal to shard ID of one of the parent shards and `adjacentParentShardId` equal to the shard ID of the other parent shard. Your application will have already read all the data from one of these shards; this is the shard for which `getRecordsResult.getNextShardIterator()` returned null. If the order of the data is important to your application, you should ensure that it reads all the data from the other parent shard as well, before reading any new data from the child shard created by the merge.

If you are using multiple processors to retrieve data from the stream, say one processor per shard, and a shard split or merge occurs, you should adjust the number of processors up or down to adapt to the change in the number of shards.

For more information about resharding, including a discussion of shards states—such as CLOSED—see Resharding a Stream (p. 26).

Resharding a Stream

The Amazon Kinesis service supports *resharding*, which enables you to adjust the number of shards in your stream in order to adapt to changes in the rate of data flow through the stream. There are two types of resharding operations: shard split and shard merge. In a shard split, you divide a single shard into two shards. In a shard merge, you combine two shards into a single shard. Resharding is always "pairwise" in the sense that you cannot split into more than two shards in a single operation, and you cannot merge more than two shards in a single operation. The shard or pair of shards that the resharding operation acts on are referred to as *parent* shards. The shard or pair of shards that result from the resharding operation are referred to as *child* shards.

Splitting increases the number of shards in your stream and therefore increases the data capacity of the stream. Because you are charged on a per-shard basis, splitting increases the cost of your stream. Similarly, merging reduces the number of shards in your stream and therefore decreases the data capacity—and cost—of the stream.

Resharding is typically performed by an administrative application which is distinct from the producer applications, which add data to the stream, and the consumer applications, which get data from the stream. Such an administrative application monitors the overall performance of the stream based on metrics provided by CloudWatch or based on metrics collected from the producers and consumers. The administrative application would also need a broader set of IAM permissions than the consumers or producers; the consumers and producers usually should not need access to the APIs used for resharding. For more information about IAM permissions for the Amazon Kinesis service, see Controlling Access to Amazon Kinesis Resources with IAM (p. 42).

Split a Shard

In order to split a shard, you need to specify how hash key values from the parent shard should be redistributed to the child shards. When you add a data record to a stream, it is assigned to a shard based on a hash key value. The hash key value is the MD5 hash of the partition key that you specify for the data record at the time that you add the data record to the stream; data records that have the same partition key will also have the same hash key value.

The possible hash key values for a given shard constitute a set of ordered contiguous non-negative integers. This range of possible hash key values is given by:

```
shard.getHashKeyRange().getStartingHashKey();
shard.getHashKeyRange().getEndingHashKey();
```

When you split the shard, you specify a value in this range. That hash key value and all higher hash key values are distributed to one of the child shards. All the lower hash key values are distributed to the other child shard.

The following code demonstrates a shard split operation that redistributes the hash keys evenly between each of the child shards, essentially splitting the parent shard in half. This is just one possible way of dividing the parent shard. You could, for example, split the shard so that the lower one-third of the keys from the parent go to one child shard and the upper two-thirds of the keys go to other child shard. However, for many applications, splitting shards in half is an effective approach.

The code assumes that `myStreamName` holds the name of your Amazon Kinesis stream and the object variable `shard` holds the shard that you will split. Begin by instantiating a new `splitShardRequest` object and setting the stream name and shard ID

```
SplitShardRequest splitShardRequest = new SplitShardRequest();
splitShardRequest.setStreamName(myStreamName);
splitShardRequest.setShardToSplit(shard.getShardId());
```

Determine the hash key value which is half-way between the lowest and highest values in the shard. This is the starting hash key value for the child shard that will contain the upper half of the hash keys from the parent shard. Specify this value in the `setNewStartingHashKey()` method. You need specify only this value; the Amazon Kinesis service automatically distributes the hash keys below this value to the other child shard that is created by the split. The last step is to call the `splitShard()` method on the Amazon Kinesis service client.

```
BigInteger startingHashKey = new BigInteger(shard.getHashKeyRange().getStarting
HashKey());
BigInteger endingHashKey   = new BigInteger(shard.getHashKeyRange().getEnding
HashKey());
String newStartingHashKey  = startingHashKey.add(endingHashKey).divide(new Bi
gInteger("2")).toString();

splitShardRequest.setNewStartingHashKey(newStartingHashKey);
client.splitShard(splitShardRequest);
```

Merge Two Shards

A shard merge operation takes two specified shards and combines them into a single shard. After the merge, the single child shard receives data for all hash key values covered by the two parent shards.

Shard Adjacency

In order to merge two shards, the shards must be *adjacent*. Two shards are considered adjacent if the union of the hash key ranges for the two shards form a contiguous set with no gaps. For example, if you have two shards, one with a hash key range of 276...381 and the other with a hash key range of 382...454, then you could merge these two shards into a single shard that would have a hash key range of 276...454.

To take another example, if you have two shards, one with a hash key range of 276..381 and the other with a hash key range of 455...560, then you could not merge these two shards because there would be one or more shards between these two that cover the range 382..454.

The set of all OPEN shards in a stream—as a group—always spans the entire range of MD5 hash key values. For more information about shard states—such as CLOSED—see Data Routing, Data Persistence, and Shard State after a Reshard (p. 29).

To identify shards that are candidates for merging, you should filter out all shards that are in a CLOSED state. Shards that are OPEN—that is, not CLOSED—have an ending sequence number of null. You can test the ending sequence number for a shard using:

```
if( null == shard.getSequenceNumberRange().getEndingSequenceNumber() ) {
  // Shard is OPEN, so it is a possible candidate to be merged.
}
```

After filtering out the closed shards, sort the remaining shards by the highest hash key value supported by each shard. You can retrieve this value using:

```
shard.getHashKeyRange().getEndingHashKey();
```

If two shards are adjacent in this filtered, sorted list, they can be merged.

Code for the Merge Operation

The following code merges two shards. The code assumes that `myStreamName` holds the name of your Amazon Kinesis stream and the object variables `shard1` and `shard2` hold the two adjacent shards that you will merge.

For the merge operation, begin by instantiating a new `mergeShardsRequest` object. Specify the stream name with the `setStreamName()` method. Then specify the two shards to merge using the `setShardToMerge()` and `setAdjacentShardToMerge()` methods. Finally, call the `mergeShards()` method on the Amazon Kinesis service client to carry out the operation.

```
MergeShardsRequest mergeShardsRequest = new MergeShardsRequest();
mergeShardsRequest.setStreamName(myStreamName);
mergeShardsRequest.setShardToMerge(shard1.getShardId());
mergeShardsRequest.setAdjacentShardToMerge(shard2.getShardId());
client.mergeShards(mergeShardsRequest);
```

Wait after Resharding for Stream to Become Active

After you call a resharding operation, either `splitShard()` or `mergeShards()`, you need to wait for the stream to become active again. The code to use is the same as when you wait for a stream to become active after creating a stream (p. 18). That code is reproduced below.

```
DescribeStreamRequest describeStreamRequest = new DescribeStreamRequest();
describeStreamRequest.setStreamName( myStreamName );

long startTime = System.currentTimeMillis();
long endTime = startTime + ( 10 * 60 * 1000 );
while ( System.currentTimeMillis() < endTime ) {
  try {
    Thread.sleep(20 * 1000);
  }
  catch ( Exception e ) {}

  try {
    DescribeStreamResult describeStreamResponse = client.describeStream( de
scribeStreamRequest );
    String streamStatus = describeStreamResponse.getStreamDescription().get
StreamStatus();
    if ( streamStatus.equals( "ACTIVE" ) ) {
      break;
    }
   //
    // sleep for one second
    //
    try {
      Thread.sleep( 1000 );
    }
    catch ( Exception e ) {}
  }
  catch ( ResourceNotFoundException e ) {}
}
if ( System.currentTimeMillis() >= endTime ) {
  throw new RuntimeException( "Stream " + myStreamName + " never went active"
);
}
```

Data Routing, Data Persistence, and Shard State after a Reshard

Amazon Kinesis is a real-time data streaming service, which is to say that your applications should assume that data is flowing continuously through the shards in your stream. When you reshard, data records that were flowing to the parent shards are re-routed to flow to the child shards based on the hash key values that the data-record partition keys map to. However, any data records that were in the parent shards before the reshard remain in those shards. In other words, the parent shards do not disappear when the reshard occurs; they persist along with the data they contained prior to the reshard. The data records in the parent shards are accessible using the `getShardIterator` and `getRecords()` (p. 22) operations in the Amazon Kinesis service API or using the Amazon Kinesis Client Library.

> **Note**
> These data records are accessible for only 24 hours from the time that they were added to the stream. (This 24-hour retention limit applies to all data records in Amazon Kinesis shards, not just parent shards that remain after a reshard.)

In the process of resharding, a parent shard transitions from an OPEN state to a CLOSED state to an EXPIRED state.

- **OPEN**: Before a reshard operation, a parent shard is in the OPEN state, which means that data records can be both added to the shard and retrieved from the shard.
- **CLOSED**: After a reshard operation, the parent shard transitions to a CLOSED state. This means that data records are no longer added to the shard. Data records that would have been added to this shard are now added to a child shard instead. However, data records can still be retrieved from the shard for a limited time.
- **EXPIRED**: After 24 hours, all the data records in the parent shard have expired and are no longer accessible. At this point, the shard itself transitions to an EXPIRED state. Calls to `getStreamDescription().getShards()` to enumerate the shards in the stream do not include EXPIRED shards in the list shards returned.

Exhaust Parent Shard Data before Reading Child Shard Data

After the reshard has occurred and the stream is once again in an ACTIVE state, you *could* immediately begin to read data from the child shards. However, the parent shards that remain after the reshard could still contain data that you haven't read yet that was added to the stream prior to the reshard. If you read data from the child shards before having read all data from the parent shards, you could read data for a particular hash key out of the order given by the data records' sequence numbers. Therefore, assuming that the order of the data is important, you should, after a reshard, always continue to read data from the parent shards until it is exhausted, and only then begin reading data from the child shards. When `getRecordsResult.getNextShardIterator()` returns null, it indicates that you have read all the data in the parent shard. If you are reading data using the Amazon Kinesis Client Library, the library ensures that you receive the data in order even if a reshard occurs.

Strategies for Resharding

The purpose of resharding is to enable your stream to adapt to changes in the rate of data flow. You split shards to increase the capacity (and cost) of your stream. You merge shards to reduce the cost (and capacity) of your stream.

One approach to resharding could be to simply split every shard in the stream—which would double the stream's capacity. However, this might provide more additional capacity than you actually need and therefore create unnecessary cost.

You can also use metrics to determine which are your "hot" or "cold" shards, that is, shards that are receiving much more data, or much less data, than expected. You could then selectively split the hot shards to increase capacity for the hash keys that target those shards. Similarly, you could merge cold shards to make better use of their unused capacity.

You can obtain some performance data for your stream from the CloudWatch metrics that the Amazon Kinesis service publishes. However, you can also collect some of your own metrics for your streams. One approach would be to log the hash key values generated by the partition keys for your data records. Recall that you specify the partition key at the time that you add the record to the stream.

```
putRecordRequest.setPartitionKey( String.format( "myPartitionKey" ) );
```

The Amazon Kinesis service uses MD5 to compute the hash key from the partition key. Because you specify the partition key for the record, you could use MD5 to compute the hash key value for that record and log it.

You could also log the IDs of the shards that your data records are assigned to. The shard ID is available by using the `getShardId()` method of the `putRecordResult` object returned by the `putRecord()` method.

```
String shardId = putRecordResult.getShardId();
```

With the shard IDs and the hash key values, you can determine which shards and hash keys are receiving the most or least traffic. You can then use resharding to provide more or less capacity, as appropriate for these keys.

Delete the Stream

The following code deletes the stream.

```
DeleteStreamRequest deleteStreamRequest = new DeleteStreamRequest();
deleteStreamRequest.setStreamName(myStreamName);
client.deleteStream(deleteStreamRequest);
```

You should shut down any applications that are operating on the stream before you delete it. If an application attempts to operate on a deleted stream, it will receive ResourceNotFound exceptions. Also, if you subsequently create a new stream that has the same name as your previous stream, and applications that were operating on the previous stream are still running, these applications might try to interact with the new stream as though it was the previous stream—which would result in unpredictable behavior.

Related Resources

- Amazon Kinesis API Reference

Developing Record Consumer Applications with the Amazon Kinesis Client Library

This section explains how to develop an Amazon Kinesis record consumer application using the Amazon Kinesis Client Library (KCL). Although you can use the Amazon Kinesis service API to get data from an Amazon Kinesis stream, we recommend using the design patterns and code for Kinesis applications provided in this section.

Topics

Overview

This section introduces the Amazon Kinesis Client Library, Kinesis applications, and the relationship between the two. Each is described in more detail in following sections.

Amazon Kinesis Client Library (KCL)

The Amazon Kinesis Client Library for Java (KCL) helps you consume and process data from an Amazon Kinesis stream. The KCL takes care of many of the complex tasks associated with distributed computing, such as load-balancing across multiple instances, responding to instance failures, checkpointing processed records, and reacting to resharding. The KCL allows you to focus on writing record processing logic.

Note that the KCL is different from the Amazon Kinesis service API that is available in the AWS SDKs. The Amazon Kinesis API helps you manage many aspects of the service (including creating streams, resharding, and putting and getting records), while the KCL provides a layer of abstraction specifically for processing data. For information about the Amazon Kinesis API, see the Amazon Kinesis API Reference.

Before you work through this section, download the KCL from Maven or GitHub.

Kinesis Applications

A "Kinesis application" is simply a Java application built with the KCL. A Kinesis application instantiates a worker with configuration information, and then uses the `IRecordProcessor` interface to process data received from an Amazon Kinesis stream.

A Kinesis application can be run on any number of instances. Multiple instances of the same application coordinate on failures and load-balance dynamically. You can also have multiple Amazon Kinesis applications working on the same stream, subject to throughput limits.

Building a Kinesis Application

This section introduces the components and methods of Kinesis applications. For illustration purposes, this section refers to the Kinesis application sample code that is available on GitHub. The sample application uses Apache Commons Logging. You can change the logging configuration in the static `configure()` method defined in the file *SampleAmazon KinesisApplication.java*.

For more information about how to use Apache Commons Logging with Log4j and AWS Java applications, go to Logging with Log4j in the *AWS Java Developer Guide*.

Role of the Amazon Kinesis Client Library

The Amazon Kinesis Client Library acts as an intermediary between your record processing logic and the Amazon Kinesis service itself.

When a Kinesis application starts, it calls the KCL to instantiate a worker. This call provides the KCL with configuration information for the application, such as the stream name and AWS credentials.

For more information, see Initializing the Application (p. 36).

In a Kinesis application, the KCL performs the following tasks:

- Connects to the stream
- Enumerates the shards
- Coordinates shard associations with other workers (if any)
- Instantiates a record processor for every shard it will manage, using a factory for the class that implements `IRecordProcessor`
- Pulls data records from the Amazon Kinesis stream
- Pushes the records to the corresponding record processor, calling the `processRecords()` method of the `IRecordProcessor` interface, and passing an instance of `IRecordProcessorCheckpointer` along with the list of records
- Checkpoints processed records
- Balances shard-worker associations when the worker instance count changes
- Balances shard-worker associations when shards are split or merged

Amazon Kinesis Application Components

Every Kinesis application must include these three components:

- The `IRecordProcessor` interface
- A factory for the class that implements the `IRecordProcessor` interface

- Code that initializes the application and creates the worker

Implementing the IRecordProcessor Interface

The `IRecordProcessor` interface exposes the following methods. Your Kinesis application must implement all of these methods.

```
public void initialize(String shardId)

public void processRecords(List<Record> records, IRecordProcessorCheckpointer
checkpointer)

public void shutdown(IRecordProcessorCheckpointer checkpointer, ShutdownReason
 reason)
```

The sample provides implementations that you can use as a starting point for your own application. See the class `SampleRecordProcessor` in the file *SampleRecordProcessor.java*.

initialize

```
  public void initialize(String shardId)
```

The KCL calls the `initialize()` method when the record processor is instantiated, passing a specific shard ID as a parameter. This record processor processes only this shard, and typically, the reverse is also true (this shard is processed only by this record processor). However, your application should account for the possibility that a data record might be processed more than once. Amazon Kinesis has "at least once" semantics, meaning that every data record from a shard will be processed at least once by a worker in your Amazon Kinesis application.

For more information about cases in which a particular shard may be processed by more than one worker, see Resharding, Scaling, and Parallel Processing (p. 38).

processRecords

```
public void processRecords(List<Record> records, IRecordProcessorCheckpointer
checkpointer)
```

The KCL calls the `processRecords()` method, passing a list of data records from the shard specified by `initialize(shardId)`. The record processor processes the data in these records according to the semantics of the application. For example, the worker might perform a transformation on the data and then store the result in an Amazon S3 bucket.

In addition to the data itself, the record also contains a sequence number and partition key. The worker can use these values when processing the data. For example, the worker could choose the Amazon S3 bucket in which to store the data based on the value of the partition key. The Record class exposes the following methods that provide access to the record's data, sequence number, and partition key.

```
record.getData()
record.getSequenceNumber()

record.getPartitionKey()
```

In the sample, the private method `processRecordsWithRetries()` has code that shows how a worker can access the record's data, sequence number, and partition key.

checkpoint

The Amazon Kinesis service requires the record processor to keep track of the records that have already been processed in a given shard. The KCL takes care of this tracking by passing an `IRecordProcessorCheckpointer` interface to `processRecords()`. The record processor calls the `checkpoint()` method on this interface to inform the KCL of how far it has progressed in processing the records in the shard. In the event that the worker fails, the KCL uses this information to restart the processing of the shard at the last known processed record.

In the case of a split or merge operation, the KCL won't start processing the new shard(s) until the processors for the original shards have called `checkpoint()` to signal that all processing on the original shards is complete.

The KCL assumes that the call to `checkpoint()` means that all records have been processed, up to the last record that was passed to the record processor. Thus the record processor should call `checkpoint()` only after it has processed all the records in the list that was passed to it. Record processors do not need to call `checkpoint()` on each call to `processRecords()`. A processor could, for example, call `checkpoint()` on every third call to `processRecords()`.

In the sample, the private method `checkpoint()` shows how to call the `IRecordProcessorCheckpointer.checkpoint()` method using appropriate exception handling and retry logic.

The KCL relies on `processRecords()` to handle any exceptions that arise from processing the data records. If an exception is thrown out of `processRecords()`, the KCL skips over the data records that were passed to `processRecords()` prior to the exception; that is, these records are not re-sent to the record processor that threw the exception or to any other record processor in the application.

shutdown

```
public void shutdown(IRecordProcessorCheckpointer checkpointer, ShutdownReason
 reason)
```

The KCL calls the `shutdown()` method if either of the following is true:

- **Processing ends:** This record processor will not receive any further records from the shard, for one of these reasons:
  - A shard split or shard merge operation has occurred and has affected this shard.
  - The stream itself is being deleted, and all data in this shard of the stream has been processed.

  In these cases, the `ShutdownReason` parameter of the `shutdown()` method has a value of TERMINATE.
- **Unresponsive worker:** It appears that the worker is no longer responding to the KCL. In this case, the `ShutdownReason` parameter has a value of ZOMBIE.

The KCL also passes an `IRecordProcessorCheckpointer` interface to `shutdown()`. If the `ShutdownReason` parameter is TERMINATE, the record processor should finish processing any data records, and then call the `checkpoint()` method on this interface. For more information, see checkpoint (p. 35).

Implementing a Class Factory for IRecordProcessor

You'll also need to implement a factory for the class that implements the `IRecordProcessor` interface methods. When your application instantiates the worker, it passes a reference to this factory.

The sample implements the factory class `SampleRecordProcessorFactory` in the file *SampleRecordProcessorFactory.java*.

```
public class SampleRecordProcessorFactory implements IRecordProcessorFactory {

    /**
     * Constructor.
     */
    public SampleRecordProcessorFactory() {
        super();
    }
    /**
     * {@inheritDoc}
     */
    @Override
    public IRecordProcessor createProcessor() {
        return new SampleRecordProcessor();
    }
}
```

Initializing the Application

A Kinesis application must be initialized with the following configuration values:

- The application name
- The name of the stream that the application will process
- The HTTPS endpoint for the Amazon Kinesis service

The sample initialization code provides default configuration values in the file *SampleAmazon KinesisApplication.java*. You can override any of these with your own values, using a Java properties file. If you're using a properties file, specify it on the command line when you run the sample application.

Application Name

The KCL uses the application name configuration value in the following ways:

- All workers associated with this application name are assumed to be working together on the same stream. These workers may be distributed on multiple instances. If you run an additional instance of the same application code, but with a different application name, the KCL treats the second instance as an entirely separate application that is also operating on the same stream.
- The KCL creates an Amazon DynamoDB table with the application name and uses the table to maintain state information (such as checkpoints and worker-shard mapping) for the application. Each application has its own DynamoDB table.

  For more information, see Kinesis Application State and Amazon DynamoDB (p. 38).

For the KCL to function as expected, the application name must be unique among your Kinesis applications, and it must also be unique among DynamoDB tables in the same account and region. Choose your application name carefully.

Worker ID

The sample initialization code creates an ID for the worker using the name of the local computer and appending a globally unique identifier:

```
String workerId = InetAddress.getLocalHost().getCanonicalHostName() + ":" +
UUID.randomUUID();
```

This approach supports the scenario of having multiple instances of the application running on a single computer.

Credentials

If you are running your Kinesis application on an Amazon EC2 instance, we recommend that you configure the instance with an AWS Identity and Access Management role. AWS credentials that reflect the permissions associated with this IAM role are made available to applications on the instance through the instance metadata service (IMDS). This is the most secure way to manage credentials for a Kinesis application running on an Amazon EC2 instance.

The sample application includes code that first attempts to retrieve IAM credentials from the IMDS:

```
credentialsProvider = new InstanceProfileCredentialsProvider();
```

If the sample application cannot obtain credentials from the IMDS, it attempts to retrieve credentials from a properties file:

```
credentialsProvider = new ClasspathPropertiesFileCredentialsProvider();
```

For more information about IMDS, see Using IAM.

The sample consolidates the configuration data for the worker in a `KinesisClientLibConfiguration` object. This object and a reference to the class factory for `IRecordProcessor` are passed in the call that instantiates the worker.

Advanced Topics and Techniques

This section helps you optimize your Kinesis application and understand advanced Amazon Kinesis concepts.

Topics

Kinesis Application State and Amazon DynamoDB

For each Kinesis application, the KCL uses a unique Amazon DynamoDB table to keep track of the application's state. Because the KCL uses the Kinesis application name to create the name of the table, each application name must be unique.

You can view the table using the Amazon DynamoDB console while the application is running.

If the Amazon DynamoDB table for your Kinesis application does not exist when the application starts up, one of the workers creates the table and calls the `describeStream()` method to populate the table. See Application State Data (p. 38) for details.

Important
Your account is charged for the costs associated with the DynamoDB table, in addition to the costs associated with Amazon Kinesis itself.

Throughput

If your Kinesis application receives provisioned-throughput exceptions, you should increase the provisioned throughput for the DynamoDB table. The KCL creates the table with a provisioned throughput of 10 reads per second and 10 writes per second, but this might not be sufficient for your application. For example, if your Kinesis application does frequent checkpointing or operates on a stream that is composed of many shards, you might need more throughput.

For information about provisioned throughput in DynamoDB, see Provisioned Throughput in Amazon DynamoDB and Working with Tables in the Amazon DynamoDB Developer Guide.

Application State Data

Each row in the DynamoDB table represents a shard that is being processed by your application. The hash key for the table is the shard ID.

In addition to the shard ID, each row also includes the following data:

- **Checkpoint:** The most recent checkpoint sequence number for the shard. This value is unique across all shards in the stream.
- **Worker ID:** The ID of the worker that is processing the shard. A single worker ID can be associated with multiple record processors and therefore with multiple shards. As a result, the same worker ID might appear in multiple rows in the table.
- **Heartbeat:** A count that is periodically incremented by the record processor to indicate that it is still actively processing records from the shard. The workers associated with the application periodically scan these values to ensure that all shards continue to be processed. If the heartbeat count does not increase within a configurable timeout period, other workers take over processing of that shard.

Resharding, Scaling, and Parallel Processing

Resharding

Resharding allows you to increase or decrease the number of shards in a stream in order to adapt to changes in the rate of data flowing through the stream. Resharding is typically performed by an administrative application that monitors shard data-handling metrics. Although the KCL itself doesn't initiate resharding operations, it is designed to adapt to changes in the number of shards that result from resharding.

As noted in Kinesis Application State and Amazon DynamoDB (p. 38), the KCL tracks the shards in the stream using an Amazon DynamoDB table. When new shards are created as a result of resharding, the KCL discovers the new shards and populates new rows in the table. The workers automatically discover the new shards and create processors to handle the data from them. The KCL also distributes the shards in the stream across all the available workers and record processors.

The KCL ensures that any data that existed in shards prior to the resharding is processed first. After that data has been processed, data from the new shards is sent to record processors. In this way, the KCL preserves the order in which data records were added to the stream for a particular partition key.

Example: Resharding, Scaling, and Parallel Processing

The following example illustrates how the KCL helps you handle scaling and resharding:

- Let's say your application is running on one Amazon EC2 instance, and is processing one Amazon Kinesis stream that has four shards. This one instance has one KCL worker and four record processors (one record processor for every shard). These four record processors run in parallel within the same process.
- Next, let’s say you scale the application to use another instance, so now you have two instances processing one stream that has four shards. When the KCL worker starts up on the second instance, it load-balances with the first instance, so that each instance now processes two shards.
- Suppose you then decide to split the four shards into five shards. The KCL again coordinates the processing across instances: one instance processes three shards, and the other processes two shards. A similar coordination occurs when you merge shards.

Typically, when you use the KCL, you should ensure that the number of instances does not exceed the number of shards (except for failure standby purposes). Each shard is processed by exactly one KCL worker and has exactly one corresponding IRecordProcessor, so you never need multiple instances to process one shard. However, one worker can process any number of shards, so it's fine if the number of shards exceeds the number of instances.

To scale up processing in your application, you should test a combination of these approaches:

- Increasing the instance size (because all record processors run in parallel within a process)
- Increasing the number of instances up to the maximum number of open shards (because shards can be independently processed)
- Increasing the number of shards (which increases the level of parallelism)

Note that you can use Auto Scaling to automatically scale your instances based on appropriate metrics.

When resharding increases the number of shards in the stream, the corresponding increase in the number of record processors increases the load on the Amazon EC2 instances that are hosting them. If the Amazon EC2 instances are part of an autoscaling group, and the load increases sufficiently, the autoscaling group adds more instances to handle the increased load. You should configure your Amazon EC2 instances to launch your Kinesis application at startup, so that additional workers and record processors become active on the new instance right away.

For more details about resharding, see Resharding a Stream (p. 26).

Failover and Recovery

Failure can occur at several levels when you use a Kinesis application to process data from an Amazon Kinesis stream.

- A record processor could fail
- A worker could fail, or the instance of the Kinesis application that instantiated the worker could fail

- An Amazon EC2 instance that is hosting one or more instances of the application could fail

Record Processor Failure

The worker invokes RecordProcessor APIs, such as `processRecords`, using Java ExecutorService tasks. If a task fails, the worker retains control of the shard that the record processor was processing. The worker starts a new record processor task to process that shard. See also Read Throttling (p. 41).

Worker or Application Failure

If a worker — or an instance of the Kinesis application itself — fails, you should detect and handle the situation. For example, if the `Worker.run()` method throws an exception, you should catch and handle it.

If the application itself fails, you should detect this and restart it. When the application starts up, it instantiates a new worker, which in turn instantiates new record processors that are automatically assigned shards to process. These could be the same shards that these record processors were processing before the failure, or shards that are new to these processors.

If the worker or application fails but you do not detect the failure, and there are other instances of the application running on other Amazon EC2 instances, the workers on these instances will handle the failure: they will create additional record processors to process the shards that are no longer being processed by the failed worker. The load on these other Amazon EC2 instances will increase accordingly.

The scenario described here assumes that although the worker or application has failed, the hosting Amazon EC2 instance is still running and is therefore not restarted by an Auto Scaling group.

Amazon EC2 Instance Failure

We recommend that you run the Amazon EC2 instances for your application in an Auto Scaling group. This way, if one of the Amazon EC2 instances fails, the Auto Scaling group automatically launches a new instance to replace it. You should configure the Amazon EC2 instances to launch your Kinesis application at startup.

Startup, Shutdown, and Throttling

Here are some additional considerations to incorporate into the design of your Kinesis application.

Starting Up Data Producers and Kinesis Applications

By default, the KCL begins reading records from the *tip* of the stream; i.e., the most recently added record. In this configuration, if a data-producing application adds records to the stream before any receiving record processors are running, the records will not be read by the record processors after they start up.

To change the behavior of the record processors to always read data from the *beginning* of the stream instead, set the following value in the properties file for your Kinesis application:

```
initialPositionInStream = TRIM_HORIZON
```

Important
The Amazon Kinesis service keeps records for no more than 24 hours. Even with the TRIM_HORIZON setting, if a record processor were to start more than 24 hours after a data-producing application began adding records to the stream, some of the records would not be available to the record processor.

In some scenarios, it may be fine for record processors to miss the first few records in the stream. For example, you might run some initial records through the stream to test that the stream is working end-to-end as expected. After doing this initial verification, you would then start your workers and begin to put production data into the stream.

For more information about the TRIM_HORIZON setting, see Using Shard Iterators.

Shutting Down a Kinesis Application

When your Kinesis application has completed its intended task, you should shut it down by terminating the Amazon EC2 instances on which it is running. You can terminate the instances using the AWS Management Console or the AWS Command Line Interface.

After shutting down a Kinesis application, you should delete the Amazon DynamoDB table that the KCL used to track the application's state.

Read Throttling

The throughput of a stream is provisioned at the shard level. Each shard has a read throughput of 2 MB/sec, at 5 transactions per second. If an application (or a group of applications operating on the same stream) attempts to get data from a shard at a faster rate, the service throttles the corresponding Get operations.

In a Java application, throttling typically appears as an exception. In a Amazon Kinesis application, if a record processor is processing data faster than the 2 MB/sec limit — such as in the case of a failover — throttling will occur. Because the Amazon Kinesis Client Library (KCL) (p. 32) manages the interactions between the application and the Amazon Kinesis service, throttling exceptions occur in the KCL code rather than in the application code. However, because the KCL logs these exceptions, you will see them in the logs.

If you find that your application is throttled consistently, you should consider provisioning a higher number of shards for the stream.

Controlling Access to Amazon Kinesis Resources with IAM

Amazon Kinesis integrates with AWS Identity and Access Management (IAM), a service that enables you to do the following:

- Create users and groups under your AWS account
- Easily share your AWS resources between the users in your AWS account
- Assign unique security credentials to each user
- Control each user's access to services and resources
- Get a single bill for all users in your AWS account

Each IAM user or group must be granted access to Amazon Kinesis. For examples of policies that allow access to Amazon Kinesis operations and resources, see Example Policies for Amazon Kinesis (p. 43) below. As you create and manage IAM policies, you might want to use the AWS Policy Generator and the IAM Policy Simulator.

If you are developing an application using the Amazon Kinesis Client Library, your policy also needs to include permissions for Amazon DynamoDB; the Amazon Kinesis Client Library uses Amazon DynamoDB to track state information for the application. For more information about Amazon Kinesis Client Library applications, see Developing Record Consumer Applications with the Amazon Kinesis Client Library (p. 32). For more information about IAM and Amazon DynamoDB, go to Using IAM to Control Access to Amazon DynamoDB Resources in the Amazon DynamoDB Developer Guide.

Amazon Kinesis Actions

In the `Action` element of an IAM policy, you can specify any set of Amazon Kinesis operations. `Action` supports all Kinesis operations. You must prefix each action name with the string `kinesis`. The following examples specify several Kinesis operations. An astersik (*) specifies all Amazon Kinesis operations.

```
kinesis:CreateStream
kinesis:DescribeStream
```

```
kinesis:ListStreams
kinesis:*
```

For a list of all the Amazon Kinesis operations, go to the Amazon Kinesis API Reference.

Amazon Resource Names (ARNs) for Amazon Kinesis

Amazon Kinesis supports IAM resource-level permissions, which use ARNs to represent streams. Use the following ARN resource format:

```
"Resource": arn:aws:kinesis:<region>:<accountID>:stream/<streamname>
```

For example:

```
"Resource": arn:aws:kinesis:*:111122223333:stream/mystream
```

Example Policies for Amazon Kinesis

This section shows example policies for controlling user access to Amazon Kinesis streams.

Example 1: Allow users to get data from an Amazon Kinesis stream

This policy allows a user or group to perform the `DescribeStream`, `ListStreams`, `GetShardIterator`, and `GetRecords` operations on the specified stream. This policy could be applied to users who should be able to get data from a specific stream.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "kinesis: Get*"
            ],
            "Resource": [
                "arn:aws:kinesis:us-east-1:111122223333:stream/stream1"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "kinesis:DescribeStream"
            ],
            "Resource": [
                "arn:aws:kinesis:us-east-1:111122223333:stream/stream1"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "kinesis:ListStreams"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

Example 2: Allow users to add data to any Amazon Kinesis stream in the account

This policy allows a user or group to use the `PutRecord` operation with any of the account's streams. This policy could be applied to users that should be able to add data records to all streams in an account.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "kinesis:PutRecord"
            ],
            "Resource": [
                "arn:aws:kinesis:us-east-1:111122223333:stream/*"
            ]
        }
    ]
}
```

Example 3: Allow any Amazon Kinesis action on a specific stream

This policy allows a user or group to use any Amazon Kinesis operation on the specified stream. This policy could be applied to users that should have administrative control over a specific stream.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "kinesis:*",
            "Resource": [
                "arn:aws:kinesis:us-east-1:111122223333:stream/stream1"
            ]
        }
    ]
}
```

Example 4: Allow any Amazon Kinesis action on any stream

This policy allows a user or group to use any Amazon Kinesis operation on any stream in an account. Because this policy grants full access to all your Amazon Kinesis streams, you should restrict it to administrators only.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "kinesis:*",
            "Resource": [
                "arn:aws:kinesis:*:111122223333:stream/*"
            ]
        }
    ]
}
```

For more information about IAM, see the following:

- Identity and Access Management (IAM)
- IAM Getting Started Guide
- Using IAM

Logging Amazon Kinesis API Calls Using AWS CloudTrail

Amazon Kinesis is integrated with CloudTrail, a service that captures API calls made by or on behalf of Amazon Kinesis and delivers the log files to an Amazon S3 bucket that you specify. The API calls can be made indirectly by using the Amazon Kinesis console or directly by using the Amazon Kinesis API. Using the information collected by CloudTrail, you can determine what request was made to Amazon Kinesis, the source IP address from which the request was made, who made the request, when it was made, and so on. To learn more about CloudTrail, including how to configure and enable it, see the *AWS CloudTrail User Guide*.

Amazon Kinesis Information in CloudTrail

When CloudTrail logging is enabled, calls made to Amazon Kinesis actions are tracked in log files. Kinesis records are written together with any other AWS service records in a log file. CloudTrail determines when to create and write to a new file based on a specified time period and file size.

The following actions are supported:

- **CreateStream**
- **DeleteStream**
- **DescribeStream**
- **ListStreams**
- **MergeShards**
- **SplitShard**

For more information about these actions, see Amazon Kinesis Actions.

Every log entry contains information about who generated the request. For example, if a request is made to create a Amazon Kinesis stream (**CreateStream**), the user identity of the person or service that made the request is logged. The user identity information helps you determine whether the request was made with root or IAM user credentials, with temporary security credentials for a role or federated user, or by another AWS service. For more information, see the **userIdentity** field in the CloudTrail Event Reference.

You can store your log files in your bucket for as long as you want, but you can also define Amazon S3 lifecycle rules to archive or delete log files automatically. By default, your log files are encrypted by using Amazon S3 server-side encryption (SSE).

You can choose to have CloudTrail publish SNS notifications when new log files are delivered if you want to take quick action upon log file delivery. For information, see Configuring Amazon SNS Notifications.

You can also aggregate Amazon Kinesis log files from multiple AWS regions and multiple AWS accounts into a single Amazon S3 bucket. For information, see Aggregating CloudTrail Log Files to a Single Amazon S3 Bucket.

Understanding Amazon Kinesis Log File Entries

CloudTrail log files can contain one or more log entries where each entry is made up of multiple JSON-formatted events. A log entry represents a single request from any source and includes information about the requested action, any parameters, the date and time of the action, and so on. The log entries are not guaranteed to be in any particular order. That is, they are not an ordered stack trace of the public API calls.

The following example shows a CloudTrail log entry:

```
{
    "Records": [
        {
            "eventVersion": "1.01",
            "userIdentity": {
                "type": "IAMUser",
                "principalId": "EX_PRINCIPAL_ID",
                "arn": "arn:aws:iam::012345678910:user/Alice",
                "accountId": "012345678910",
                "accessKeyId": "EXAMPLE_KEY_ID",
                "userName": "Alice"
            },
            "eventTime": "2014-04-19T00:16:31Z",
            "eventSource": "kinesis.amazonaws.com",
            "eventName": "CreateStream",
            "awsRegion": "us-east-1",
            "sourceIPAddress": "127.0.0.1",
            "userAgent": "aws-sdk-java/unknown-version Linux/x.xx",
            "requestParameters": {
                "shardCount": 1,
                "streamName": "GoodStream"
            },
            "responseElements": null,
            "requestID": "db6c59f8-c757-11e3-bc3b-57923b443c1c",
            "eventID": "b7acfcd0-6ca9-4ee1-a3d7-c4e8d420d99b"
        },
        {
            "eventVersion": "1.01",
            "userIdentity": {
                "type": "IAMUser",
                "principalId": "EX_PRINCIPAL_ID",
                "arn": "arn:aws:iam::012345678910:user/Alice",
                "accountId": "012345678910",
                "accessKeyId": "EXAMPLE_KEY_ID",
                "userName": "Alice"
            },
            "eventTime": "2014-04-19T00:17:06Z",
            "eventSource": "kinesis.amazonaws.com",
            "eventName": "DescribeStream",
            "awsRegion": "us-east-1",
            "sourceIPAddress": "127.0.0.1",
            "userAgent": "aws-sdk-java/unknown-version Linux/x.xx",
            "requestParameters": {
                "streamName": "GoodStream"
            },
            "responseElements": null,
            "requestID": "f0944d86-c757-11e3-b4ae-25654b1d3136",
            "eventID": "0b2f1396-88af-4561-b16f-398f8eaea596"
        },
        {
            "eventVersion": "1.01",
            "userIdentity": {
                "type": "IAMUser",
                "principalId": "EX_PRINCIPAL_ID",
                "arn": "arn:aws:iam::012345678910:user/Alice",
                "accountId": "012345678910",
                "accessKeyId": "EXAMPLE_KEY_ID",
                "userName": "Alice"
            },
            "eventTime": "2014-04-19T00:15:02Z",
            "eventSource": "kinesis.amazonaws.com",
            "eventName": "ListStreams",
            "awsRegion": "us-east-1",
            "sourceIPAddress": "127.0.0.1",
            "userAgent": "aws-sdk-java/unknown-version Linux/x.xx",
            "requestParameters": {
                "limit": 10
            },
            "responseElements": null,
            "requestID": "a68541ca-c757-11e3-901b-cbcfe5b3677a",
            "eventID": "22a5fb8f-4e61-4bee-a8ad-3b72046b4c4d"
        },
        {
            "eventVersion": "1.01",
            "userIdentity": {
                "type": "IAMUser",
                "principalId": "EX_PRINCIPAL_ID",
                "arn": "arn:aws:iam::012345678910:user/Alice",
                "accountId": "012345678910",
                "accessKeyId": "EXAMPLE_KEY_ID",
                "userName": "Alice"
            },
            "eventTime": "2014-04-19T00:17:07Z",
            "eventSource": "kinesis.amazonaws.com",
            "eventName": "DeleteStream",
            "awsRegion": "us-east-1",
            "sourceIPAddress": "127.0.0.1",
            "userAgent": "aws-sdk-java/unknown-version Linux/x.xx",
            "requestParameters": {
                "streamName": "GoodStream"
            },
            "responseElements": null,
            "requestID": "f10cd97c-c757-11e3-901b-cbcfe5b3677a",
            "eventID": "607e7217-311a-4a08-a904-ec02944596dd"
        },
        {
            "eventVersion": "1.01",
            "userIdentity": {
                "type": "IAMUser",
                "principalId": "EX_PRINCIPAL_ID",
                "arn": "arn:aws:iam::012345678910:user/Alice",
                "accountId": "012345678910",
                "accessKeyId": "EXAMPLE_KEY_ID",
                "userName": "Alice"
            },
            "eventTime": "2014-04-19T00:15:03Z",
            "eventSource": "kinesis.amazonaws.com",
            "eventName": "SplitShard",
            "awsRegion": "us-east-1",
            "sourceIPAddress": "127.0.0.1",
            "userAgent": "aws-sdk-java/unknown-version Linux/x.xx",
            "requestParameters": {
                "shardToSplit": "shardId-000000000000",
                "streamName": "GoodStream",
                "newStartingHashKey": "11111111"
            },
            "responseElements": null,
            "requestID": "a6e6e9cd-c757-11e3-901b-cbcfe5b3677a",
            "eventID": "dcd2126f-c8d2-4186-b32a-192dd48d7e33"
        },
        {
            "eventVersion": "1.01",
            "userIdentity": {
                "type": "IAMUser",
                "principalId": "EX_PRINCIPAL_ID",
                "arn": "arn:aws:iam::012345678910:user/Alice",
                "accountId": "012345678910",
                "accessKeyId": "EXAMPLE_KEY_ID",
                "userName": "Alice"
            },
            "eventTime": "2014-04-19T00:16:56Z",
            "eventSource": "kinesis.amazonaws.com",
            "eventName": "MergeShards",
            "awsRegion": "us-east-1",
            "sourceIPAddress": "127.0.0.1",
            "userAgent": "aws-sdk-java/unknown-version Linux/x.xx",
            "requestParameters": {
                "streamName": "GoodStream",
                "adjacentShardToMerge": "shardId-000000000002",
                "shardToMerge": "shardId-000000000001"
            },
            "responseElements": null,
            "requestID": "e9f9c8eb-c757-11e3-bf1d-6948db3cd570",
            "eventID": "77cf0d06-ce90-42da-9576-71986fec411f"
        }
    ]
}
```

History

The following table describes the important changes to the documentation in this release of Amazon Kinesis.

- **API version:** 2013-12-02
- **Latest documentation update:** June 30, 2014

Change	Description	Date Changed
New sample application section	Added Visualizing Web Traffic with the Amazon Kinesis Sample Application (p. 12).	June 27, 2014
Amazon Kinesis Client Library revisions	Expanded and revised Developing Record Consumer Applications with the Amazon Kinesis Client Library (p. 32).	April 30, 2014
Default shard limit	Updated the Amazon Kinesis Sizes and Limits (p. 5): the default shard limit has been raised from 5 to 10.	February 25, 2014
Default shard limit	Updated the Amazon Kinesis Sizes and Limits (p. 5): the default shard limit has been raised from 2 to 5.	January 28, 2014
Amazon Kinesis Client Library links	Links to the Amazon Kinesis Client Library have been added throughout the guide.	December 20, 2013
API version updates	Updates for version 2013-12-02 of the Amazon Kinesis API.	December 12, 2013
Initial release	Initial release of the Amazon Kinesis Developer Guide.	November 14, 2013