O'REILLY®

Second Edition

Kafka The Definitive Guide

Real-Time Data and Stream Processing at Scale

Gwen Shapira, Todd Palino, Rajini Sivaram & Krit Petty

CHAPTER 5

Managing Apache Kafka Programmatically

There are many CLI and GUI tools for managing Kafka (we'll discuss them in Chapter 9), but there are also times when you want to execute some administrative commands from within your client application. Creating new topics on demand based on user input or data is an especially common use case: Internet of Things (IoT) apps often receive events from user devices, and write events to topics based on the device type. If the manufacturer produces a new type of device, you either have to remember, via some process, to also create a topic, or the application can dynamically create a new topic if it receives events with an unrecognized device type. The second alternative has downsides, but avoiding the dependency on an additional process to generate topics is an attractive feature in the right scenarios.

Apache Kafka added the AdminClient in version 0.11 to provide a programmatic API for administrative functionality that was previously done in the command line: listing, creating, and deleting topics; describing the cluster; managing ACLs; and modifying configuration.

Here's one example. Your application is going to produce events to a specific topic. This means that before producing the first event, the topic has to exist. Before Apache Kafka added the AdminClient, there were few options, none of them particularly user-friendly: you could capture an `UNKNOWN_TOPIC_OR_PARTITION` exception from the `producer.send()` method and let your user know that they needed to create the topic, or you could hope that the Kafka cluster you were writing to enabled automatic topic creation, or you could try to rely on internal APIs and deal with the consequences of no compatibility guarantees. Now that Apache Kafka provides AdminClient, there is a much better solution: use AdminClient to check whether the topic exists, and if it does not, create it on the spot.

In this chapter we'll give an overview of the AdminClient before we drill down into the details of how to use it in your applications. We'll focus on the most commonly used functionality: management of topics, consumer groups, and entity configuration.

AdminClient Overview

As you start using Kafka AdminClient, it helps to be aware of its core design principles. When you understand how the AdminClient was designed and how it should be used, the specifics of each method will be much more intuitive.

Asynchronous and Eventually Consistent API

Perhaps the most important thing to understand about Kafka's AdminClient is that it is asynchronous. Each method returns immediately after delivering a request to the cluster controller, and each method returns one or more `Future` objects. `Future` objects are the result of asynchronous operations, and they have methods for checking the status of the asynchronous operation, canceling it, waiting for it to complete, and executing functions after its completion. Kafka's AdminClient wraps the `Future` objects into `Result` objects, which provide methods to wait for the operation to complete and helper methods for common follow-up operations. For example, `Kafka AdminClient.createTopics` returns the `CreateTopicsResult` object, which lets you wait until all topics are created, check each topic status individually, and retrieve the configuration of a specific topic after it was created.

Because Kafka's propagation of metadata from the controller to the brokers is asynchronous, the `Futures` that AdminClient APIs return are considered complete when the controller state has been fully updated. At that point, not every broker might be aware of the new state, so a `listTopics` request may end up handled by a broker that is not up-to-date and will not contain a topic that was very recently created. This property is also called *eventual consistency*: eventually every broker will know about every topic, but we can't guarantee exactly when this will happen.

Options

Every method in AdminClient takes as an argument an `Options` object that is specific to that method. For example, the `listTopics` method takes the `ListTopicsOptions` object as an argument, and `describeCluster` takes `DescribeClusterOptions` as an argument. Those objects contain different settings for how the request will be handled by the broker. The one setting that all AdminClient methods have is `timeoutMs`: this controls how long the client will wait for a response from the cluster before throwing a `TimeoutException`. This limits the time in which your application may be blocked by AdminClient operation. Other options include whether `listTopics` should also return internal topics and whether `describeCluster` should also return which operations the client is authorized to perform on the cluster.

Flat Hierarchy

All admin operations supported by the Apache Kafka protocol are implemented in `KafkaAdminClient` directly. There is no object hierarchy or namespaces. This is a bit controversial as the interface can be quite large and perhaps a bit overwhelming, but the main benefit is that if you want to know how to programmatically perform any admin operation on Kafka, you have exactly one JavaDoc to search, and your IDE autocomplete will be quite handy. You don't have to wonder whether you are just missing the right place to look. If it isn't in AdminClient, it was not implemented yet (but contributions are welcome!).

If you are interested in contributing to Apache Kafka, take a look at our "How to Contribute" guide. Start with smaller, noncontroversial bug fixes and improvements before tackling a more significant change to the architecture or the protocol. Noncode contributions such as bug reports, documentation improvements, responses to questions, and blog posts are also encouraged.

Additional Notes

All the operations that modify the cluster state—create, delete, and alter—are handled by the controller. Operations that read the cluster state—list and describe—can be handled by any broker and are directed to the least-loaded broker (based on what the client knows). This shouldn't impact you as an API user, but it can be good to know in case you are seeing unexpected behavior, you notice that some operations succeed while others fail, or if you are trying to figure out why an operation is taking too long.

At the time we are writing this chapter (Apache Kafka 2.5 is about to be released), most admin operations can be performed either through AdminClient or directly by modifying the cluster metadata in ZooKeeper. We highly encourage you to never use ZooKeeper directly, and if you absolutely have to, report this as a bug to Apache Kafka. The reason is that in the near future, the Apache Kafka community will remove the ZooKeeper dependency, and every application that uses ZooKeeper directly for admin operations will have to be modified. On the other hand, the AdminClient API will remain exactly the same, just with a different implementation inside the Kafka cluster.

AdminClient Lifecycle: Creating, Configuring, and Closing

To use Kafka's AdminClient, the first thing you have to do is construct an instance of the AdminClient class. This is quite straightforward:

```
Properties props = new Properties();
props.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
AdminClient admin = AdminClient.create(props);
// TODO: Do something useful with AdminClient
admin.close(Duration.ofSeconds(30));
```

The static `create` method takes as an argument a `Properties` object with configuration. The only mandatory configuration is the URI for your cluster: a comma-separated list of brokers to connect to. As usual, in production environments, you want to specify at least three brokers just in case one is currently unavailable. We'll discuss how to configure a secure and authenticated connection separately in Chapter 11.

If you start an AdminClient, eventually you want to close it. It is important to remember that when you call `close`, there could still be some AdminClient operations in progress. Therefore, the `close` method accepts a timeout parameter. Once you call `close`, you can't call any other methods and send any more requests, but the client will wait for responses until the timeout expires. After the timeout expires, the client will abort all ongoing operations with timeout exception and release all resources. Calling `close` without a timeout implies that the client will wait as long as it takes for all ongoing operations to complete.

You probably recall from Chapters 3 and 4 that the `KafkaProducer` and `KafkaConsumer` have quite a few important configuration parameters. The good news is that AdminClient is much simpler, and there is not much to configure. You can read about all the configuration parameters in the Kafka documentation. In our opinion, the important configuration parameters are described in the following sections.

client.dns.lookup

This configuration was introduced in the Apache Kafka 2.1.0 release.

By default, Kafka validates, resolves, and creates connections based on the hostname provided in the bootstrap server configuration (and later in the names returned by the brokers as specified in the `advertised.listeners` configuration). This simple model works most of the time but fails to cover two important use cases: the use of DNS aliases, especially in a bootstrap configuration, and the use of a single DNS that maps to multiple IP addresses. These sound similar but are slightly different. Let's look at each of these mutually exclusive scenarios in a bit more detail.

Use of a DNS alias

Suppose you have multiple brokers with the following naming convention: `broker1.hostname.com`, `broker2.hostname.com`, etc. Rather than specifying all of them in a bootstrap server configuration, which can easily become challenging to maintain, you may want to create a single DNS alias that will map to all of them.

You'll use `all-brokers.hostname.com` for bootstrapping, since you don't actually care which broker gets the initial connection from clients. This is all very convenient, except if you use SASL to authenticate. If you use SASL, the client will try to authenticate `all-brokers.hostname.com`, but the server principal will be `broker2.hostname.com`. If the names don't match, SASL will refuse to authenticate (the broker certificate could be a man-in-the-middle attack), and the connection will fail.

In this scenario, you'll want to use `client.dns.lookup=resolve_canonical_bootstrap_servers_only`. With this configuration, the client will "expend" the DNS alias, and the result will be the same as if you included all the broker names the DNS alias connects to as brokers in the original bootstrap list.

DNS name with multiple IP addresses

With modern network architectures, it is common to put all the brokers behind a proxy or a load balancer. This is especially common if you use Kubernetes, where load balancers are necessary to allow connections from outside the Kubernetes cluster. In these cases, you don't want the load balancers to become a single point of failure. It is therefore very common to have `broker1.hostname.com` point at a list of IPs, all of which resolve to load balancers, and all of which route traffic to the same broker. These IPs are also likely to change over time. By default, the Kafka client will just try to connect to the first IP that the hostname resolves. This means that if that IP becomes unavailable, the client will fail to connect, even though the broker is fully available. It is therefore highly recommended to use `client.dns.lookup=use_all_dns_ips` to make sure the client doesn't miss out on the benefits of a highly available load balancing layer.

request.timeout.ms

This configuration limits the time that your application can spend waiting for AdminClient to respond. This includes the time spent on retrying if the client receives a retriable error.

The default value is 120 seconds, which is quite long, but some AdminClient operations, especially consumer group management commands, can take a while to respond. As we mentioned in "AdminClient Overview" on page 114, each AdminClient method accepts an `Options` object, which can contain a timeout value that applies specifically to that call. If an AdminClient operation is on the critical path for your application, you may want to use a lower timeout value and handle a lack of timely response from Kafka in a different way. A common example is that services try to validate the existence of specific topics when they first start, but if Kafka takes longer than 30 seconds to respond, you may want to continue starting the server and validate the existence of topics later (or skip this validation entirely).

Essential Topic Management

Now that we created and configured an AdminClient, it's time to see what we can do with it. The most common use case for Kafka's AdminClient is topic management. This includes listing topics, describing them, creating topics, and deleting them.

Let's start by listing all topics in the cluster:

```
ListTopicsResult topics = admin.listTopics();
topics.names().get().forEach(System.out::println);
```

Note that `admin.listTopics()` returns the `ListTopicsResult` object, which is a thin wrapper over a collection of `Futures`. Note also that `topics.name()` returns a `Future` set of `name`. When we call `get()` on this `Future`, the executing thread will wait until the server responds with a set of topic names, or we get a timeout exception. Once we get the list, we iterate over it to print all the topic names.

Now let's try something a bit more ambitious: check if a topic exists, and create it if it doesn't. One way to check if a specific topic exists is to get a list of all topics and check if the topic you need is in the list. On a large cluster, this can be inefficient. In addition, sometimes you want to check for more than just whether the topic exists—you want to make sure the topic has the right number of partitions and replicas. For example, Kafka Connect and Confluent Schema Registry use a Kafka topic to store configuration. When they start up, they check if the configuration topic exists, that it has only one partition to guarantee that configuration changes will arrive in strict order, that it has three replicas to guarantee availability, and that the topic is compacted so the old configuration will be retained indefinitely:

```
DescribeTopicsResult demoTopic = admin.describeTopics(TOPIC_LIST); ❶

try {
    topicDescription = demoTopic.values().get(TOPIC_NAME).get(); ❷
    System.out.println("Description of demo topic:" + topicDescription);

    if (topicDescription.partitions().size() != NUM_PARTITIONS) { ❸
      System.out.println("Topic has wrong number of partitions. Exiting.");
      System.exit(-1);
    }
} catch (ExecutionException e) { ❹
    // exit early for almost all exceptions
    if (! (e.getCause() instanceof UnknownTopicOrPartitionException)) {
        e.printStackTrace();
        throw e;
    }

    // if we are here, topic doesn't exist
    System.out.println("Topic " + TOPIC_NAME +
        " does not exist. Going to create it now");
    // Note that number of partitions and replicas is optional. If they are
```

```
        // not specified, the defaults configured on the Kafka brokers will be used
        CreateTopicsResult newTopic = admin.createTopics(Collections.singletonList(
                new NewTopic(TOPIC_NAME, NUM_PARTITIONS, REP_FACTOR))); ❺

        // Check that the topic was created correctly:
        if (newTopic.numPartitions(TOPIC_NAME).get() != NUM_PARTITIONS) { ❻
            System.out.println("Topic has wrong number of partitions.");
            System.exit(-1);
        }
    }
```

❶ To check that the topic exists with the correct configuration, we call `describeTopics()` with a list of topic names we want to validate. This returns `DescribeTopicResult` object, which wraps a map of topic names to `Future` descriptions.

❷ We've already seen that if we wait for the `Future` to complete, using `get()` we can get the result we wanted, in this case, a `TopicDescription`. But there is also a possibility that the server can't complete the request correctly—if the topic does not exist, the server can't respond with its description. In this case, the server will send back an error, and the `Future` will complete by throwing an `ExecutionException`. The actual error sent by the server will be the `cause` of the exception. Since we want to handle the case where the topic doesn't exist, we handle these exceptions.

❸ If the topic does exist, the `Future` completes by returning a `TopicDescription`, which contains a list of all the partitions of the topic, and for each partition in which a broker is the leader, a list of replicas and a list of in-sync replicas. Note that this does not include the configuration of the topic. We'll discuss configuration later in this chapter.

❹ Note that all AdminClient result objects throw `ExecutionException` when Kafka responds with an error. This is because AdminClient results are wrapped `Future` objects, and those wrap exceptions. You always need to examine the cause of `ExecutionException` to get the error that Kafka returned.

❺ If the topic does not exist, we create a new topic. When creating a topic, you can specify just the name and use default values for all the details. You can also specify the number of partitions, number of replicas, and the configuration.

❻ Finally, you want to wait for topic creation to return, and perhaps validate the result. In this example, we are checking the number of partitions. Since we specified the number of partitions when we created the topic, we are fairly certain it is correct. Checking the result is more common if you relied on broker defaults when creating the topic. Note that since we are again calling `get()` to check the results of CreateTopic, this method could throw an exception. TopicExists Exception is common in this scenario, and you'll want to handle it (perhaps by describing the topic to check for the correct configuration).

Now that we have a topic, let's delete it:

```
admin.deleteTopics(TOPIC_LIST).all().get();

// Check that it is gone. Note that due to the async nature of deletes,
// it is possible that at this point the topic still exists
try {
    topicDescription = demoTopic.values().get(TOPIC_NAME).get();
    System.out.println("Topic " + TOPIC_NAME + " is still around");
} catch (ExecutionException e) {
    System.out.println("Topic " + TOPIC_NAME + " is gone");
}
```

At this point the code should be quite familiar. We call the method deleteTopics with a list of topic names to delete, and we use get() to wait for this to complete.

> Although the code is simple, please remember that in Kafka, deletion of topics is final—there is no recycle bin or trash can to help you rescue the deleted topic, and no checks to validate that the topic is empty and that you really meant to delete it. Deleting the wrong topic could mean unrecoverable loss of data, so handle this method with extra care.

All the examples so far have used the blocking get() call on the Future returned by the different AdminClient methods. Most of the time, this is all you need—admin operations are rare, and waiting until the operation succeeds or times out is usually acceptable. There is one exception: if you are writing to a server that is expected to process a large number of admin requests. In this case, you don't want to block the server threads while waiting for Kafka to respond. You want to continue accepting requests from your users and sending them to Kafka, and when Kafka responds, send the response to the client. In these scenarios, the versatility of KafkaFuture becomes quite useful. Here's a simple example.

```
vertx.createHttpServer().requestHandler(request -> { ❶
    String topic = request.getParam("topic"); ❷
    String timeout = request.getParam("timeout");
    int timeoutMs = NumberUtils.toInt(timeout, 1000);

    DescribeTopicsResult demoTopic = admin.describeTopics( ❸
            Collections.singletonList(topic),
            new DescribeTopicsOptions().timeoutMs(timeoutMs));

    demoTopic.values().get(topic).whenComplete( ❹
            new KafkaFuture.BiConsumer<TopicDescription, Throwable>() {
                @Override
                public void accept(final TopicDescription topicDescription,
                                   final Throwable throwable) {
                    if (throwable != null) {
                      request.response().end("Error trying to describe topic "
                              + topic + " due to " + throwable.getMessage()); ❺
                    } else {
                        request.response().end(topicDescription.toString()); ❻
                    }
                }
            });
}).listen(8080);
```

❶ We are using Vert.x to create a simple HTTP server. Whenever this server receives a request, it calls the `requestHandler` that we are defining here.

❷ The request includes a topic name as a parameter, and we'll respond with a description of this topic.

❸ We call `AdminClient.describeTopics` as usual and get a wrapped `Future` in response.

❹ Instead of using the blocking `get()` call, we construct a function that will be called when the `Future` completes.

❺ If the `Future` completes with an exception, we send the error to the HTTP client.

❻ If the `Future` completes successfully, we respond to the client with the topic description.

The key here is that we are not waiting for a response from Kafka. `DescribeTopicResult` will send the response to the HTTP client when a response arrives from Kafka. Meanwhile, the HTTP server can continue processing other requests. You can check this behavior by using `SIGSTOP` to pause Kafka (don't try this in production!) and send two HTTP requests to Vert.x: one with a long timeout value and one with a short value. Even though you sent the second request after the first, it will respond earlier thanks to the lower timeout value, and not block behind the first request.

Configuration Management

Configuration management is done by describing and updating collections of `ConfigResource`. Config resources can be brokers, broker loggers, and topics. Checking and modifying broker and broker logging configuration is typically done using tools like `kafka-config.sh` or other Kafka management tools, but checking and updating topic configuration from the applications that use them is quite common.

For example, many applications rely on compacted topics for correct operation. It makes sense that periodically (more frequently than the default retention period, just to be safe), those applications will check that the topic is indeed compacted and take action to correct the topic configuration if it is not.

Here's an example of how this is done:

```
ConfigResource configResource =
        new ConfigResource(ConfigResource.Type.TOPIC, TOPIC_NAME); ❶
DescribeConfigsResult configsResult =
        admin.describeConfigs(Collections.singleton(configResource));
Config configs = configsResult.all().get().get(configResource);

// print nondefault configs
configs.entries().stream().filter(
        entry -> !entry.isDefault()).forEach(System.out::println); ❷


// Check if topic is compacted
ConfigEntry compaction = new ConfigEntry(TopicConfig.CLEANUP_POLICY_CONFIG,
        TopicConfig.CLEANUP_POLICY_COMPACT);
if (!configs.entries().contains(compaction)) {
    // if topic is not compacted, compact it
    Collection<AlterConfigOp> configOp = new ArrayList<AlterConfigOp>();
    configOp.add(new AlterConfigOp(compaction, AlterConfigOp.OpType.SET)); ❸
    Map<ConfigResource, Collection<AlterConfigOp>> alterConf = new HashMap<>();
    alterConf.put(configResource, configOp);
    admin.incrementalAlterConfigs(alterConf).all().get();
} else {
    System.out.println("Topic " + TOPIC_NAME + " is compacted topic");
}
```

❶ As mentioned above, there are several types of `ConfigResource`; here we are checking the configuration for a specific topic. You can specify multiple different resources from different types in the same request.

❷ The result of `describeConfigs` is a map from each `ConfigResource` to a collection of configurations. Each configuration entry has an `isDefault()` method that lets us know which configs were modified. A topic configuration is considered nondefault if a user configured the topic to have a nondefault value, or if a broker-level configuration was modified and the topic that was created inherited this nondefault value from the broker.

❸ To modify a configuration, specify a map of the `ConfigResource` you want to modify and a collection of operations. Each configuration modifying operation consists of a configuration entry (the name and value of the configuration; in this case, `cleanup.policy` is the configuration name and `compacted` is the value) and the operation type. Four types of operations modify configuration in Kafka: `SET`, which sets the configuration value; `DELETE`, which removes the value and resets to the default; `APPEND`; and `SUBSTRACT`. The last two apply only to configurations with a `List` type and allow adding and removing values from the list without having to send the entire list to Kafka every time.

Describing the configuration can be surprisingly handy in an emergency. We remember a time when during an upgrade, the configuration file for the brokers was accidentally replaced with a broken copy. This was discovered after restarting the first broker and noticing that it failed to start. The team did not have a way to recover the original, and we prepared for significant trial and error as we attempted to reconstruct the correct configuration and bring the broker back to life. A site reliability engineer (SRE) saved the day by connecting to one of the remaining brokers and dumping its configuration using the AdminClient.

Consumer Group Management

We've mentioned before that unlike most message queues, Kafka allows you to reprocess data in the exact order in which it was consumed and processed earlier. In Chapter 4, where we discussed consumer groups, we explained how to use the Consumer APIs to go back and reread older messages from a topic. But using these APIs means that you programmed the ability to reprocess data in advance into your application. Your application itself must expose the "reprocess" functionality.

There are several scenarios in which you'll want to cause an application to reprocess messages, even if this capability was not built into the application in advance. Troubleshooting a malfunctioning application during an incident is one such scenario. Another is when preparing an application to start running on a new cluster during a disaster recovery failover scenario (we'll discuss this in more detail in Chapter 9, when we discuss disaster recovery techniques).

In this section, we'll look at how you can use the AdminClient to programmatically explore and modify consumer groups and the offsets that were committed by those groups. In Chapter 10 we'll look at external tools available to perform the same operations.

Exploring Consumer Groups

If you want to explore and modify consumer groups, the first step is to list them:

```
admin.listConsumerGroups().valid().get().forEach(System.out::println);
```

Note that by using `valid()` method, the collection that `get()` will return will only contain the consumer groups that the cluster returned without errors, if any. Any errors will be completely ignored, rather than thrown as exceptions. The `errors()` method can be used to get all the exceptions. If you use `all()` as we did in other examples, only the first error the cluster returned will be thrown as an exception. Likely causes of such errors are authorization, where you don't have permission to view the group, or cases when the coordinator for some of the consumer groups is not available.

If we want more information about some of the groups, we can describe them:

```
ConsumerGroupDescription groupDescription = admin
        .describeConsumerGroups(CONSUMER_GRP_LIST)
        .describedGroups().get(CONSUMER_GROUP).get();
        System.out.println("Description of group " + CONSUMER_GROUP
                + ":" + groupDescription);
```

The description contains a wealth of information about the group. This includes the group members, their identifiers and hosts, the partitions assigned to them, the algorithm used for the assignment, and the host of the group coordinator. This description is very useful when troubleshooting consumer groups. One of the most important pieces of information about a consumer group is missing from this description—inevitably, we'll want to know what was the last offset committed by the group for each partition that it is consuming and how much it is lagging behind the latest messages in the log.

In the past, the only way to get this information was to parse the commit messages that the consumer groups wrote to an internal Kafka topic. While this method accomplished its intent, Kafka does not guarantee compatibility of the internal message formats, and therefore the old method is not recommended. We'll take a look at how Kafka's AdminClient allows us to retrieve this information:

```
Map<TopicPartition, OffsetAndMetadata> offsets =
        admin.listConsumerGroupOffsets(CONSUMER_GROUP)
                .partitionsToOffsetAndMetadata().get(); ❶

Map<TopicPartition, OffsetSpec> requestLatestOffsets = new HashMap<>();

for(TopicPartition tp: offsets.keySet()) {
    requestLatestOffsets.put(tp, OffsetSpec.latest()); ❷
}

Map<TopicPartition, ListOffsetsResult.ListOffsetsResultInfo> latestOffsets =
        admin.listOffsets(requestLatestOffsets).all().get();

for (Map.Entry<TopicPartition, OffsetAndMetadata> e: offsets.entrySet()) { ❸
    String topic = e.getKey().topic();
    int partition =  e.getKey().partition();
    long committedOffset = e.getValue().offset();
    long latestOffset = latestOffsets.get(e.getKey()).offset();

    System.out.println("Consumer group " + CONSUMER_GROUP
            + " has committed offset " + committedOffset
            + " to topic " + topic + " partition " + partition
```

```
                + ". The latest offset in the partition is "
                +  latestOffset + " so consumer group is "
                + (latestOffset - committedOffset) + " records behind");
    }
```

❶ We retrieve a map of all topics and partitions that the consumer group handles, and the latest committed offset for each. Note that unlike `describeConsumerGroups`, `listConsumerGroupOffsets` only accepts a single consumer group and not a collection.

❷ For each topic and partition in the results, we want to get the offset of the last message in the partition. `OffsetSpec` has three very convenient implementations: `earliest()`, `latest()`, and `forTimestamp()`, which allow us to get the earlier and latest offsets in the partition, as well as the offset of the record written on or immediately after the time specified.

❸ Finally, we iterate over all the partitions, and for each partition print the last committed offset, the latest offset in the partition, and the lag between them.

Modifying Consumer Groups

Until now, we just explored available information. AdminClient also has methods for modifying consumer groups: deleting groups, removing members, deleting committed offsets, and modifying offsets. These are commonly used by SREs to build ad hoc tooling to recover from an emergency.

From all those, modifying offsets is the most useful. Deleting offsets might seem like a simple way to get a consumer to "start from scratch," but this really depends on the configuration of the consumer—if the consumer starts and no offsets are found, will it start from the beginning? Or jump to the latest message? Unless we have the value of `auto.offset.reset`, we can't know. Explicitly modifying the committed offsets to the earliest available offsets will force the consumer to start processing from the beginning of the topic, and essentially cause the consumer to "reset."

Do keep in mind that consumer groups don't receive updates when offsets change in the offset topic. They only read offsets when a consumer is assigned a new partition or on startup. To prevent you from making changes to offsets that the consumers will not know about (and will therefore override), Kafka will prevent you from modifying offsets while the consumer group is active.

Also keep in mind that if the consumer application maintains state (and most stream processing applications maintain state), resetting the offsets and causing the consumer group to start processing from the beginning of the topic can have a strange impact on the stored state. For example, suppose you have a stream application that is continuously counting shoes sold in your store, and suppose that at 8:00 a.m. you discover that there was an error in inputs and you want to completely recalculate the count since 3:00 a.m. If you reset the offsets to 3:00 a.m. without appropriately modifying the stored aggregate, you will count every shoe that was sold today twice (you will also process all the data between 3:00 a.m. and 8:00 a.m., but let's assume that this is necessary to correct the error). You need to take care to update the stored state accordingly. In a development environment, we usually delete the state store completely before resetting the offsets to the start of the input topic.

With all these warnings in mind, let's look at an example:

```
Map<TopicPartition, ListOffsetsResult.ListOffsetsResultInfo> earliestOffsets =
    admin.listOffsets(requestEarliestOffsets).all().get(); ❶

Map<TopicPartition, OffsetAndMetadata> resetOffsets = new HashMap<>();
for (Map.Entry<TopicPartition, ListOffsetsResult.ListOffsetsResultInfo> e:
        earliestOffsets.entrySet()) {
  resetOffsets.put(e.getKey(), new OffsetAndMetadata(e.getValue().offset())); ❷
}

try {
  admin.alterConsumerGroupOffsets(CONSUMER_GROUP, resetOffsets).all().get(); ❸
} catch (ExecutionException e) {
  System.out.println("Failed to update the offsets committed by group "
            + CONSUMER_GROUP + " with error " + e.getMessage());
  if (e.getCause() instanceof UnknownMemberIdException)
      System.out.println("Check if consumer group is still active."); ❹
}
```

❶ To reset the consumer group so it will start processing from the earliest offset, we need to get the earliest offsets first. Getting the earliest offsets is similar to getting the latest, shown in the previous example.

❷ In this loop we convert the map with `ListOffsetsResultInfo` values that were returned by `listOffsets` into a map with `OffsetAndMetadata` values that are required by `alterConsumerGroupOffsets`.

❸ After calling `alterConsumerGroupOffsets`, we are waiting on the `Future` to complete so we can see if it completed successfully.

❹ One of the most common reasons that `alterConsumerGroupOffsets` fails is that we didn't stop the consumer group first (this has to be done by shutting down the consuming application directly; there is no admin command for shutting down a consumer group). If the group is still active, our attempt to modify the offsets will appear to the consumer coordinator as if a client that is not a member of the group is committing an offset for that group. In this case, we'll get `UnknownMemberIdException`.

Cluster Metadata

It is rare that an application has to explicitly discover anything at all about the cluster to which it connected. You can produce and consume messages without ever learning how many brokers exist and which one is the controller. Kafka clients abstract away this information—clients only need to be concerned with topics and partitions.

But just in case you are curious, this little snippet will satisfy your curiosity:

```
DescribeClusterResult cluster = admin.describeCluster();

System.out.println("Connected to cluster " + cluster.clusterId().get()); (1)
System.out.println("The brokers in the cluster are:");
cluster.nodes().get().forEach(node -> System.out.println("    * " + node));
System.out.println("The controller is: " + cluster.controller().get());
```

(1) Cluster identifier is a GUID and therefore is not human readable. It is still useful to check whether your client connected to the correct cluster.

Advanced Admin Operations

In this section, we'll discuss a few methods that are rarely used, and can be risky to use, but are incredibly useful when needed. Those are mostly important for SREs during incidents—but don't wait until you are in an incident to learn how to use them. Read and practice before it is too late. Note that the methods here have little to do with one another, except that they all fit into this category.

Adding Partitions to a Topic

Usually the number of partitions in a topic is set when a topic is created. And since each partition can have very high throughput, bumping against the capacity limits of a topic is rare. In addition, if messages in the topic have keys, then consumers can assume that all messages with the same key will always go to the same partition and will be processed in the same order by the same consumer.

For these reasons, adding partitions to a topic is rarely needed and can be risky. You'll need to check that the operation will not break any application that consumes from the topic. At times, however, you will really hit the ceiling of how much throughput you can process with the existing partitions and have no choice but to add some.

You can add partitions to a collection of topics using the `createPartitions` method. Note that if you try to expand multiple topics at once, it is possible that some of the topics will be successfully expanded, while others will fail.

```
Map<String, NewPartitions> newPartitions = new HashMap<>();
newPartitions.put(TOPIC_NAME, NewPartitions.increaseTo(NUM_PARTITIONS+2)); (1)
admin.createPartitions(newPartitions).all().get();
```

❶ When expanding topics, you need to specify the total number of partitions the topic will have after the partitions are added, not the number of new partitions.

Since the `createPartition` method takes as a parameter the total number of partitions in the topic after new partitions are added, you may need to describe the topic and find out how many partitions exist prior to expanding it.

Deleting Records from a Topic

Current privacy laws mandate specific retention policies for data. Unfortunately, while Kafka has retention policies for topics, they were not implemented in a way that guarantees legal compliance. A topic with a retention policy of 30 days can store older data if all the data fits into a single segment in each partition.

The `deleteRecords` method will mark as deleted all the records with offsets older than those specified when calling the method and make them inaccessible by Kafka consumers. The method returns the highest deleted offsets, so we can check if the deletion indeed happened as expected. Full cleanup from disk will happen asynchronously. Remember that the `listOffsets` method can be used to get offsets for records that were written on or immediately after a specific time. Together, these methods can be used to delete records older than any specific point in time:

```
Map<TopicPartition, ListOffsetsResult.ListOffsetsResultInfo> olderOffsets =
        admin.listOffsets(requestOlderOffsets).all().get();
Map<TopicPartition, RecordsToDelete> recordsToDelete = new HashMap<>();
for (Map.Entry<TopicPartition, ListOffsetsResult.ListOffsetsResultInfo>  e:
        olderOffsets.entrySet())
    recordsToDelete.put(e.getKey(),
            RecordsToDelete.beforeOffset(e.getValue().offset()));
 admin.deleteRecords(recordsToDelete).all().get();
```

Leader Election

This method allows you to trigger two different types of leader election:

Preferred leader election
: Each partition has a replica that is designated as the *preferred leader*. It is preferred because if all partitions use their preferred leader replica as the leader, the number of leaders on each broker should be balanced. By default, Kafka will check every five minutes if the preferred leader replica is indeed the leader, and if it isn't but it is eligible to become the leader, it will elect the preferred leader replica as leader. If `auto.leader.rebalance.enable` is `false`, or if you want this to happen faster, the `electLeader()` method can trigger this process.

Unclean leader election

If the leader replica of a partition becomes unavailable, and the other replicas are not eligible to become leaders (usually because they are missing data), the partition will be without a leader and therefore unavailable. One way to resolve this is to trigger *unclean leader* election, which means electing a replica that is otherwise ineligible to become a leader as the leader anyway. This will cause data loss—all the events that were written to the old leader and were not replicated to the new leader will be lost. The `electLeader()` method can also be used to trigger unclean leader elections.

The method is asynchronous, which means that even after it returns successfully, it takes a while until all brokers become aware of the new state, and calls to `describeTopics()` can return inconsistent results. If you trigger leader election for multiple partitions, it is possible that the operation will be successful for some partitions and fail for others:

```
Set<TopicPartition> electableTopics = new HashSet<>();
electableTopics.add(new TopicPartition(TOPIC_NAME, 0));
try {
    admin.electLeaders(ElectionType.PREFERRED, electableTopics).all().get(); ❶
} catch (ExecutionException e) {
    if (e.getCause() instanceof ElectionNotNeededException) {
        System.out.println("All leaders are preferred already"); ❷
    }
}
```

❶ We are electing the preferred leader on a single partition of a specific topic. We can specify any number of partitions and topics. If you call the command with `null` instead of a collection of partitions, it will trigger the election type you chose for all partitions.

❷ If the cluster is in a healthy state, the command will do nothing. Preferred leader election and unclean leader election only take effect when a replica other than the preferred leader is the current leader.

Reassigning Replicas

Sometimes, you don't like the current location of some of the replicas. Maybe a broker is overloaded and you want to move some replicas. Maybe you want to add more replicas. Maybe you want to move all replicas from a broker so you can remove the machine. Or maybe a few topics are so noisy that you need to isolate them from the rest of the workload. In all these scenarios, `alterPartitionReassignments` gives you fine-grain control over the placement of every single replica for a partition. Keep in mind that reassigning replicas from one broker to another may involve copying large amounts of data from one broker to another. Be mindful of the available network bandwidth, and throttle replication using quotas if needed; quotas are a broker configuration, so you can describe them and update them with `AdminClient`.

For this example, assume that we have a single broker with ID 0. Our topic has several partitions, all with one replica on this broker. After adding a new broker, we want to use it to store some of the replicas of the topic. We are going to assign each partition in the topic in a slightly different way:

```
Map<TopicPartition, Optional<NewPartitionReassignment>> reassignment = new Hash-
Map<>();
reassignment.put(new TopicPartition(TOPIC_NAME, 0),
        Optional.of(new NewPartitionReassignment(Arrays.asList(0,1)))); ❶
reassignment.put(new TopicPartition(TOPIC_NAME, 1),
        Optional.of(new NewPartitionReassignment(Arrays.asList(1)))); ❷
reassignment.put(new TopicPartition(TOPIC_NAME, 2),
        Optional.of(new NewPartitionReassignment(Arrays.asList(1,0)))); ❸
reassignment.put(new TopicPartition(TOPIC_NAME, 3), Optional.empty()); ❹

admin.alterPartitionReassignments(reassignment).all().get();

System.out.println("currently reassigning: " +
        admin.listPartitionReassignments().reassignments().get()); ❺
demoTopic = admin.describeTopics(TOPIC_LIST);
topicDescription = demoTopic.values().get(TOPIC_NAME).get();
System.out.println("Description of demo topic:" + topicDescription); ❻
```

❶ We've added another replica to partition 0, placed the new replica on the new broker, which has ID 1, but left the leader unchanged.

❷ We didn't add any replicas to partition 1; we simply moved the one existing replica to the new broker. Since we have only one replica, it is also the leader.

❸ We've added another replica to partition 2 and made it the preferred leader. The next preferred leader election will switch leadership to the new replica on the new broker. The existing replica will then become a follower.

❹ There is no ongoing reassignment for partition 3, but if there was, this would have canceled it and returned the state to what it was before the reassignment operation started.

❺ We can list the ongoing reassignments.

❻ We can also print the new state, but remember that it can take awhile until it shows consistent results.

Testing

Apache Kafka provides a test class, `MockAdminClient`, which you can initialize with any number of brokers and use to test that your applications behave correctly without having to run an actual Kafka cluster and really perform the admin operations on it. While `MockAdminClient` is not part of the Kafka API and therefore subject to change without warning, it mocks methods that are public, and therefore the method signatures will remain compatible. There is a bit of a trade-off on whether the convenience of this class is worth the risk that it will change and break your tests, so keep this in mind.

What makes this test class especially compelling is that some of the common methods have very comprehensive mocking: you can create topics with `MockAdminClient`, and a subsequent call to `listTopics()` will list the topics you "created."

However, not all methods are mocked. If you use `AdminClient` with version 2.5 or earlier and call `incrementalAlterConfigs()` of the `MockAdminClient`, you will get an `UnsupportedOperationException`, but you can handle this by injecting your own implementation.

To demonstrate how to test using `MockAdminClient`, let's start by implementing a class that is instantiated with an admin client and uses it to create topics:

```
public TopicCreator(AdminClient admin) {
    this.admin = admin;
}

// Example of a method that will create a topic if its name starts with "test"
public void maybeCreateTopic(String topicName)
        throws ExecutionException, InterruptedException {
    Collection<NewTopic> topics = new ArrayList<>();
    topics.add(new NewTopic(topicName, 1, (short) 1));
    if (topicName.toLowerCase().startsWith("test")) {
        admin.createTopics(topics);

        // alter configs just to demonstrate a point
        ConfigResource configResource =
                  new ConfigResource(ConfigResource.Type.TOPIC, topicName);
        ConfigEntry compaction =
                  new ConfigEntry(TopicConfig.CLEANUP_POLICY_CONFIG,
                          TopicConfig.CLEANUP_POLICY_COMPACT);
        Collection<AlterConfigOp> configOp = new ArrayList<AlterConfigOp>();
        configOp.add(new AlterConfigOp(compaction, AlterConfigOp.OpType.SET));
        Map<ConfigResource, Collection<AlterConfigOp>> alterConf =
            new HashMap<>();
        alterConf.put(configResource, configOp);
        admin.incrementalAlterConfigs(alterConf).all().get();
    }
}
```

The logic here isn't sophisticated: `maybeCreateTopic` will create the topic if the topic name starts with "test." We are also modifying the topic configuration, so we can show how to handle a case where the method we use isn't implemented in the mock client.

> We are using the Mockito testing framework to verify that the `MockAdminClient` methods are called as expected and to fill in for the unimplemented methods. Mockito is a fairly simple mocking framework with nice APIs, which makes it a good fit for a small example of a unit test.

We'll start testing by instantiating our mock client:

```
@Before
public void setUp() {
    Node broker = new Node(0,"localhost",9092);
    this.admin = spy(new MockAdminClient(Collections.singletonList(broker),
        broker)); ❶

    // without this, the tests will throw
    // `java.lang.UnsupportedOperationException: Not implemented yet`
    AlterConfigsResult emptyResult = mock(AlterConfigsResult.class);
    doReturn(KafkaFuture.completedFuture(null)).when(emptyResult).all();
    doReturn(emptyResult).when(admin).incrementalAlterConfigs(any()); ❷
}
```

❶ `MockAdminClient` is instantiated with a list of brokers (here we're using just one), and one broker that will be our controller. The brokers are just the broker ID, hostname, and port—all fake, of course. No brokers will run while executing these tests. We'll use Mockito's `spy` injection, so we can later check that `TopicCreator` executed correctly.

❷ Here we use Mockito's `doReturn` methods to make sure the mock admin client doesn't throw exceptions. The method we are testing expects the `AlterConfigResult` object with an `all()` method that returns a `KafkaFuture`. We made sure that the fake `incrementalAlterConfigs` returns exactly that.

Now that we have a properly fake AdminClient, we can use it to test whether the `maybeCreateTopic()` method works properly:

```
@Test
public void testCreateTestTopic()
        throws ExecutionException, InterruptedException {
    TopicCreator tc = new TopicCreator(admin);
    tc.maybeCreateTopic("test.is.a.test.topic");
    verify(admin, times(1)).createTopics(any()); ❶
}
```

```
@Test
public void testNotTopic() throws ExecutionException, InterruptedException {
    TopicCreator tc = new TopicCreator(admin);
    tc.maybeCreateTopic("not.a.test");
    verify(admin, never()).createTopics(any()); ❷
}
```

❶ The topic name starts with "test," so we expect `maybeCreateTopic()` to create a topic. We check that `createTopics()` was called once.

❷ When the topic name doesn't start with "test," we verify that `createTopics()` was not called at all.

One last note: Apache Kafka published `MockAdminClient` in a test jar, so make sure your *pom.xml* includes a test dependency:

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>2.5.0</version>
    <classifier>test</classifier>
    <scope>test</scope>
</dependency>
```

Summary

AdminClient is a useful tool to have in your Kafka development kit. It is useful for application developers who want to create topics on the fly and validate that the topics they are using are configured correctly for their application. It is also useful for operators and SREs who want to create tooling and automation around Kafka or need to recover from an incident. AdminClient has so many useful methods that SREs can think of it as a Swiss Army knife for Kafka operations.

In this chapter we covered all the basics of using Kafka's AdminClient: topic management, configuration management, and consumer group management, plus a few other useful methods that are good to have in your back pocket—you never know when you'll need them.