Chapter 5: Getting Started

With the theoretical foundations nailed, and a fresh installation of Kafka standing by, it is time to roll up our sleeves for a more practical approach to learning Kafka.

This chapter will focus on the two fundamental operations: publishing records to Kafka topics and subsequently consuming them. We are going to explore the various mechanisms for interacting with the broker and also for exploring the contents of topics and partitions.

Publishing and consuming using the CLI

When discussing producers and consumers, the first thing that might spring to mind is a set of bespoke applications that *someone* — an individual, or more likely, a team of developers — will build and maintain as part of operating a broader event-streaming system. But one does not need a fully-fledged application to publish to or consume from a Kafka topic — this task can be accomplished using the set of CLI (command-line interface) tools that are shipped with Kafka, located in the `$KAFKA_HOME/bin` directory.

Creating a topic

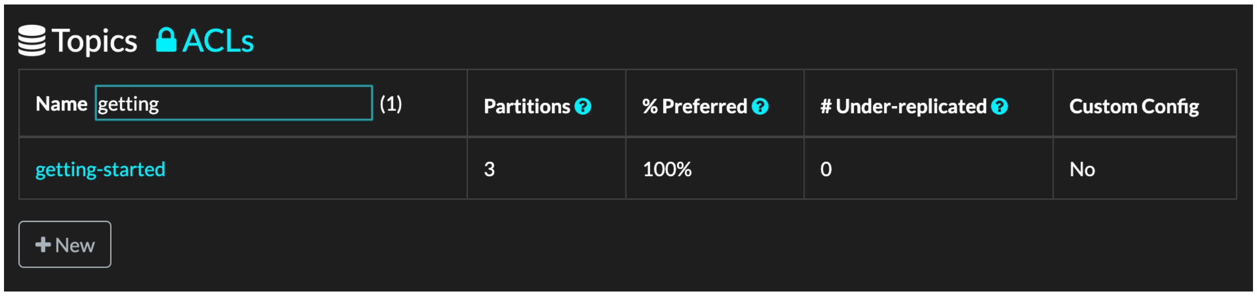
Let's get started then. The first thing is to create a topic, which can be accomplished using the `kafka-topics.sh` tool:

```
$KAFKA_HOME/bin/kafka-topics.sh --bootstrap-server localhost:9092 \
    --create --partitions 3 --replication-factor 1 \
    --topic getting-started
```

Observe, although the parameter `--bootstrap-server` is named in singular form, the `kafka-topics.sh` tool will, rather unexpectedly, accept a comma-separated list of brokers. We have specified `localhost:9092` as the bootstrap server, because that is where our test cluster is currently running. If you are using a remote Kafka broker or a managed Kafka service, you will have been provided with an alternate list of broker addresses.

The packaged CLI utilities are not the most intuitive of tools that one can use with Kafka; in fact, they are widely regarded as being awkward to use and barely adequate in functionality. Most Kafka practitioners have long abandoned the out-of-the-box utilities in favour of other open-source and commercial tools; Kafdrop is one such tool, but there are several others. This book covers the packaged CLI tools because that is what you are sure to get with every Kafka installation. Having basic awareness of the built-in tooling is about as essential as knowing the basic `vi` commands when working in Linux — you can berate the archaic tooling and laud the alternatives, but that will only get you so far as your first production incident. (In saying that, comparing Kafka's built-in tooling to Vim is a travesty.)

Switching to Kafdrop, we can see the `getting-started` topic appear in the 'Topics' section. If there are lots of topics, you can use the filter text box in the table area to refine the displayed topics to just those that match a chosen substring.

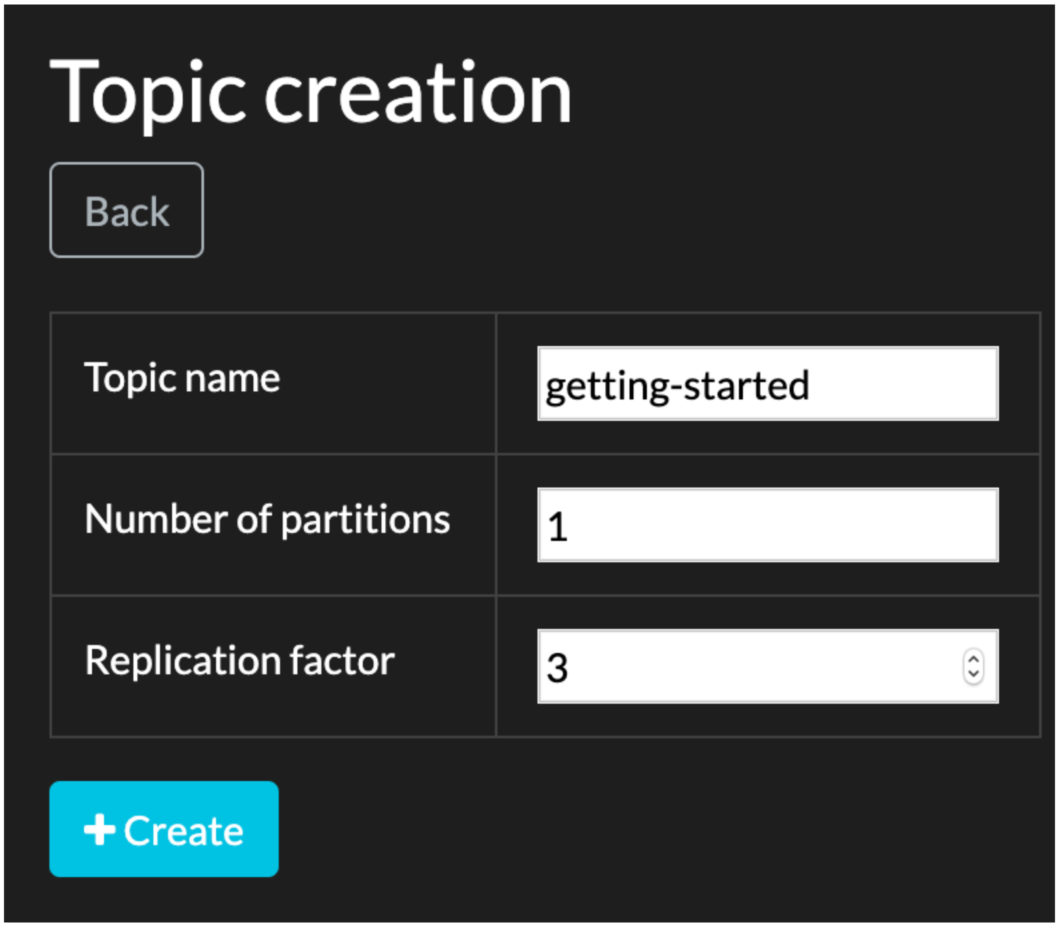

Topics ACLs

Name getting (1)	Partitions	% Preferred	# Under-replicated	Custom Config
getting-started	3	100%	0	No

Kafdrop — showing topics

We can tell at a glance that the topic has three partitions, the topic has no replication issues, and that no custom configuration has been specified for this topic. The mechanics for specifying per-topic configuration will be explained in Chapter 9: Broker Configuration.

Now, we could have just as easily created a topic by clicking the 'New' button under the topics list:



Topic name	getting-started
Number of partitions	1
Replication factor	3

Kafdrop — creating a new topic

However, the point of the exercise is to demonstrate the CLI tool, rather than to explore all the possible ways one can create a topic in Kafka.

This might be a good segue to discuss the importance of explicit topic creation. Kafka does not require clients to create topics by default. When the `auto.create.topics.enable` broker configuration property is set to `true`, Kafka will automatically create the topic when clients attempt to produce, consume, or fetch metadata for a non-existent topic. This might sound like a nifty idea at first, but the drawbacks significantly outweigh the minor convenience one gets from not having to create the topic explicitly.

Firstly, Kafka's defaults to back as far as 2011, and are generally optimised for the sorts of use cases that Kafka was originally designed for — high volume log shipping. Many things have changed since; as it happens, Kafka is no longer a one-trick pony. As such, one would typically want to configure the topic at the point of creation, or immediately thereafter — certainly before it gets a real workout.

Secondly, the partition count: Kafka allows you to specify the default number of partitions for all newly created topics using the `num.partitions` broker setting, but this is largely a meaningless number. Topics should be sized individually on the basis of expected parallelism, and a number of other factors, which are discussed in Chapter 6: Design Considerations. Specifying the partition count requires explicit topic creation. A similar statement might be made regarding the replication factor, but it is arguably easier to agree on a sensible default for the replication factor than it is for the partition count.

Finally, having Kafka auto-create topics when a client subscribes to a topic or simply fetches the topic metadata is careless, to put it mildly. A misbehaving client may initiate arbitrary metadata queries that could inadvertently create a copious number of stray topics.

Publishing records

With the topic creation out of the way, let's publish a few records. We are going to use the `kafka-console-producer.sh` tool:

```
$KAFKA_HOME/bin/kafka-console-producer.sh \
    --broker-list localhost:9092 \
    --topic getting-started --property "parse.key=true" \
    --property "key.separator=:"
```

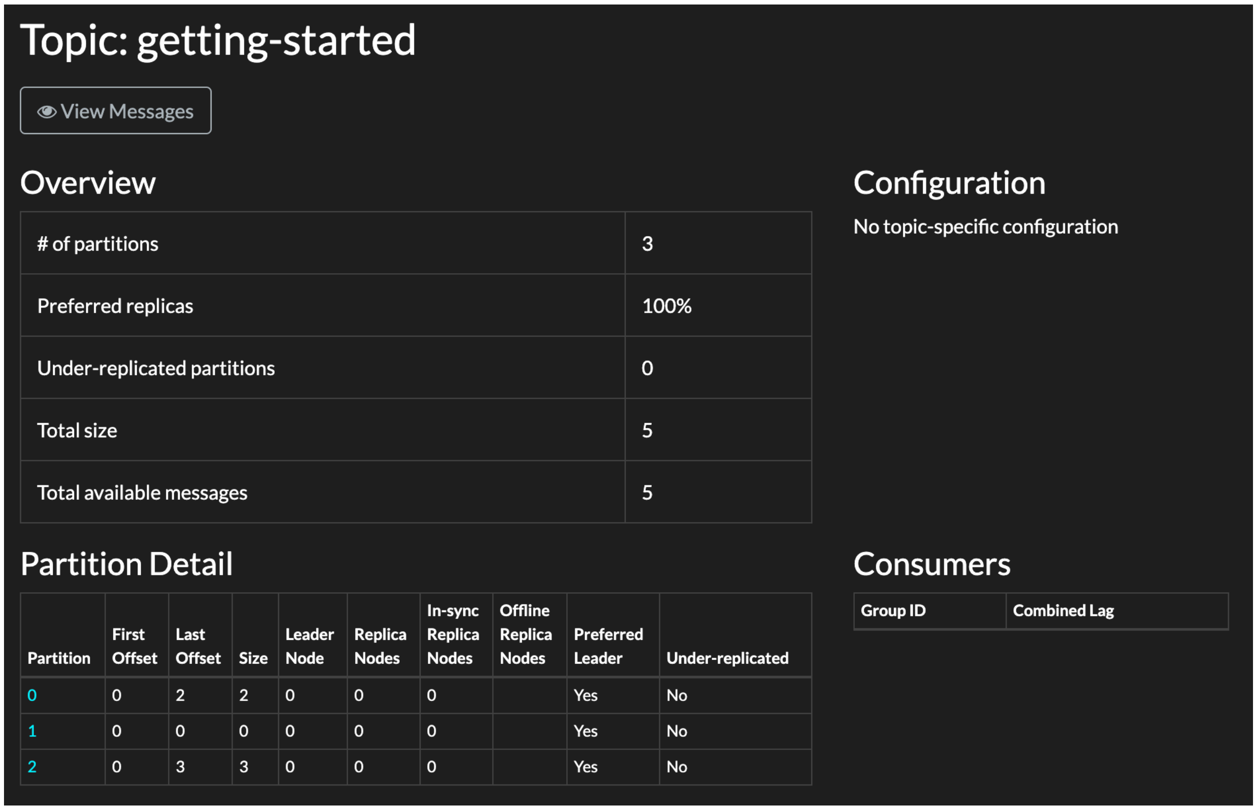
Records are separated by newlines. The key and the value parts are delimited by colons, as indicated by the `key.separator` property. For the sake of an example, type in the following (a copy-paste will do):

```
foo:first message
foo:second message
bar:first message
foo:third message
bar:second message
```

Press `CTRL+D` when done. The terminal echoes a right angle bracket (>) for every record published.

Note, the `kafka-topics.sh` tool uses the `--bootstrap-server` parameter to configure the Kafka broker list, while `kafka-console-producer.sh` uses the `--broker-list` parameter for an identical purpose. Also, `--property` arguments are largely undocumented — be prepared to Google your way around.

At this point we can switch back to Kafdrop and view the contents of the `getting-started` topic. We are presented with an overview of the topic, along with a detailed breakdown of the underlying partitions:



Topic: getting-started

Overview

# of partitions	3
Preferred replicas	100%
Under-replicated partitions	0
Total size	5
Total available messages	5

Configuration

No topic-specific configuration

Partition Detail

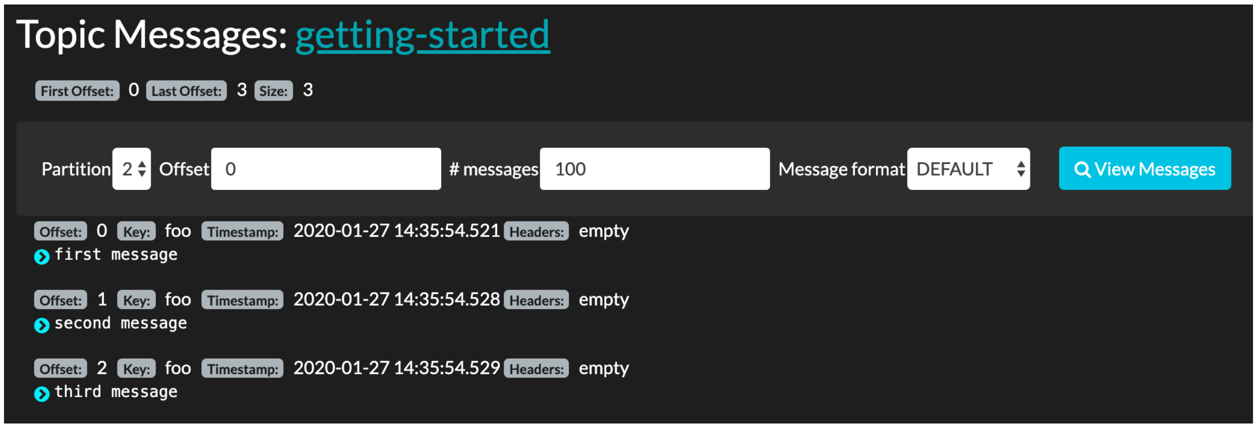
Partition	First Offset	Last Offset	Size	Leader Node	Replica Nodes	In-sync Replica Nodes	Offline Replica Nodes	Preferred Leader	Under-replicated
0	0	2	2	0	0	0		Yes	No
1	0	0	0	0	0	0		Yes	No
2	0	3	3	0	0	0		Yes	No

Consumers

Group ID	Combined Lag

Kafdrop — view topic contents

Focusing on the partition detail, we can tell at a glance that of the three partitions, two have data and one is empty. The 'first offset' and 'last offset' columns correspond to the low-water and high-water marks, respectively. As the reader might recall from Chapter 3: Architecture and Core Concepts, subtracting the two yields the maximum number of records persisted in the partition. Let's click on partition #2. Kafdrop will show the individual records, arranged in chronological order.



Topic Messages: getting-started

First Offset: 0 Last Offset: 3 Size: 3

Partition 2 Offset 0 # messages 100 Message format DEFAULT

```
Offset: 0 Key: foo Timestamp: 2020-01-27 14:35:54.521 Headers: empty
first message

Offset: 1 Key: foo Timestamp: 2020-01-27 14:35:54.528 Headers: empty
second message

Offset: 2 Key: foo Timestamp: 2020-01-27 14:35:54.529 Headers: empty
third message
```

Kafdrop — view partition contents

In case you were wondering, the arrow to the left of the record lets you expand and pretty-print JSON-encoded records. As our examples didn't use JSON, there's nothing to pretty-print.

Consuming records

```
$KAFKA_HOME/bin/kafka-console-consumer.sh \
    --bootstrap-server localhost:9092 \
    --topic getting-started --group cli-consumer --from-beginning \
    --property "print.key=true" --property "key.separator=:"
```

The terminal will echo the following:

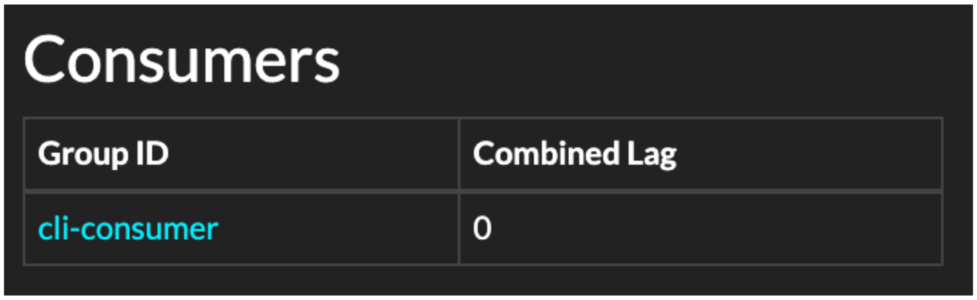
```
bar:first message
bar:second message
foo:first message
foo:second message
foo:third message
```

Because the consumer is running as a subscription, with a provided consumer group, the output will stall on the last record. The consumer will effectively tail the topic — continuously polling for new records and printing them as they arrive on the topic. To terminate the consumer, press `CTRL+D`.

Note that we specified the `--from-beginning` flag when invoking the command above. By default, a first-time consumer (for a previously non-existent group) will have its offsets reset to the topic's high-water mark. In order to read the previously published records, we override the default offset reset strategy to tail from the topic's low-water mark. If we run the same command again, we will see no records — the consumer will halt, waiting for the arrival of new records.

There is no `--from-end` flag. To tail from the end of the topic, simply delete the consumer offsets and start the CLI consumer. Deleting offsets and other offset manipulation commands are described in the section that follows.

Having consumed the backlog of records with the new `cli-consumer` consumer group, we can now switch back to Kafdrop to observe the addition of the new group. The new group appears in the topic overview screen, under the section 'Consumers', in the bottom-right.

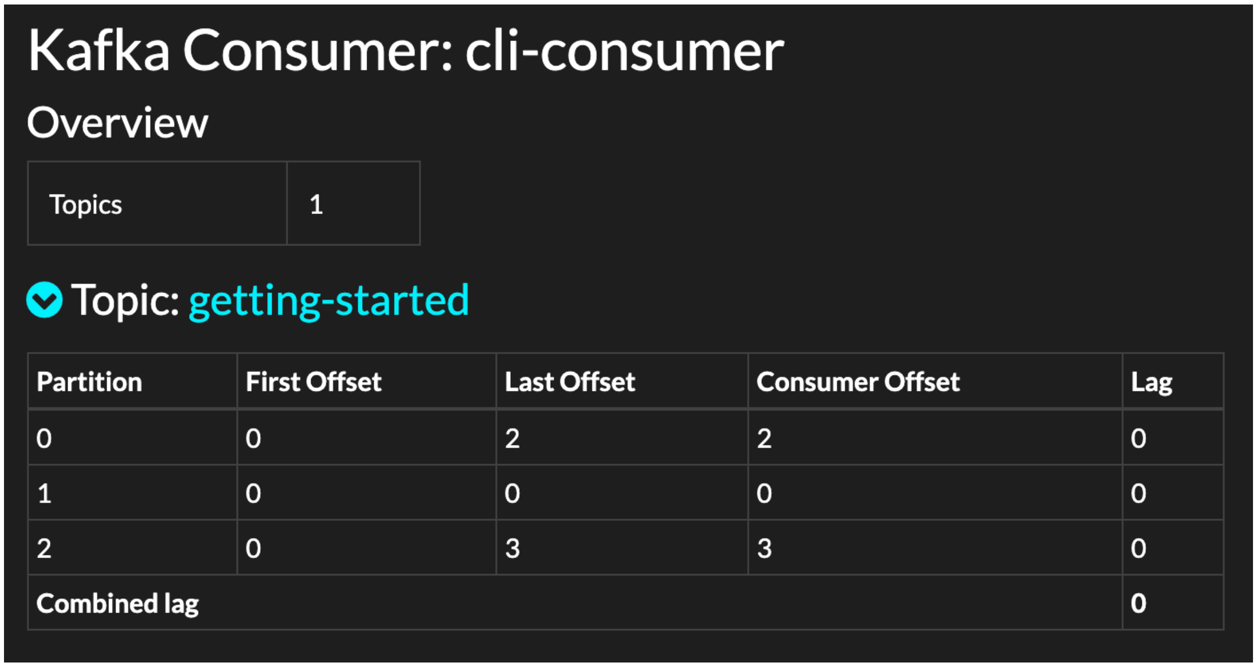


Consumers

Group ID	Combined Lag
cli-consumer	0

Kafdrop — consumer groups for a topic

Clicking through the consumer link takes us to the consumer overview. This screen enumerates over all topics within the consumer's subscription, as well as the per-partition offsets for each topic.



Kafka Consumer: cli-consumer

Overview

Topics	1

Topic: getting-started

Partition	First Offset	Last Offset	Consumer Offset	Lag
0	0	2	2	0
1	0	0	0	0
2	0	3	3	0
Combined lag				0

Kafdrop — consumer overview

In our example, the consumer offset recorded for each partition is the same as the respective high-water mark. The consumer lag is zero for each column. This is the difference between the committed offset and the high-water mark. When the lag is zero, it means that the consumer has worked through the entire backlog of records for the partition; in other words, the consumer has caught up to the producer. Lag may vary between partitions — the busier the partition, in terms of record throughput, the more likely it will accumulate lag. The aggregate lag (also known as the combined lag) is the sum of all individual per-partition lags.

Among the useful characteristics of tools such as Kafdrop and the Kafka CLI is the ability to enumerate and monitor individual consumer groups — inspect the per-partition lags and spot leading indicators of degraded consumer performance or, in the worst-case scenario, a stalled consumer. Much like any other middleware, a solid comprehension of the available tooling — be it the built-in suite or the external tools — is essential to effective operation. This is particularly crucial for overseeing mission-critical systems in production environments, where minutes of downtime and the fruitless head-scratching of the engineering and support personnel can result in significant incurred losses.

So there you have it. We have published and consumed records from a Kafka topic using the built-in CLI tools. It isn't much, but it's a start.

Useful CLI commands

To close off the section on the CLI, we will take a brief look at the other useful actions that can be performed using the built-in tools.

Listing topics

The `kafka-topics.sh` tool can be used to list topics, as per the example below.

```
$KAFKA_HOME/bin/kafka-topics.sh \
    --bootstrap-server localhost:9092 \
    --list --exclude-internal
```

The `--exclude-internal` flag, as the name suggests, eliminates the internal topics (e.g. `__consumer_-offsets`) from the query results.

Describing a topic

By passing the `--describe` flag and a topic name to `kafka-topics.sh`, we can get more detailed information about a specific topic, including the partition leaders, follower replicas, and the in-sync replica set:

```
$KAFKA_HOME/bin/kafka-topics.sh \
    --bootstrap-server localhost:9092 \
    --describe --topic getting-started
```

Produces:

```
Topic: getting-started  PartitionCount: 3 ReplicationFactor: 1 
    Configs: segment.bytes=1073741824
  Topic: getting-started  Partition: 0  Leader: 0 Replicas: 0 Isr: 0
  Topic: getting-started  Partition: 1  Leader: 0 Replicas: 0 Isr: 0
  Topic: getting-started  Partition: 2  Leader: 0 Replicas: 0 Isr: 0
```

Deleting a topic

To delete an existing topic, use the `kafka-topics.sh` tool. The example below deletes the `getting-started` topic from our test cluster.

```
$KAFKA_HOME/bin/kafka-topics.sh \
    --bootstrap-server localhost:9092 \
    --topic getting-started --delete
```

Topic deletion is an asynchronous operation — a topic is initially marked for deletion, to be subsequently cleaned up by a background process at an indeterminate time in the future. In-between the marking and the final deletion, a topic might appear to linger around — only to disappear moments later.

The asynchronous behaviour of topic deletion should be taken into account when dealing with short-lived topics — for example, when conducting an integration test. The latter typically requires a state reset between successive runs, wiping associated database tables and event streams. Because there is no equivalent of a blocking `DELETE TABLE` DDL operation in Kafka, one must think outside the box. The options are:

1. Forcibly reset consumer offsets to the high-water mark prior to each test, delete the offsets, or delete the consumer group (all three will achieve equivalent results);
2. Truncate the underlying partitions by shifting the low-water mark (truncation will be described shortly); or
3. Use unique, disposable topic names for each test, deleting any ephemeral topics when the test ends.

The latter is the recommended option, as it creates due isolation between tests and allows multiple tests to operate concurrently with no mutually-observable side-effects.

Truncating partitions

Although a partition is backed by an immutable log, Kafka offers a mechanism to truncate all records in the log up to a user-specified low-water mark. This can be achieved by passing a JSON document to the `kafka-delete-records.sh` tool, specifying the topics and partitions for truncation, with the new low-water mark in the `offset` attribute. Several topic-partition-offset triples can be specified as a batch. In the example below, we are truncating the first record from `getting-started:2`, leaving records at offset 1 and newer intact.

```
cat << EOF > /tmp/offsets.json
{
  "partitions": [
    {"topic": "getting-started", "partition": 2, "offset": 1}
  ],
  "version": 1
}
EOF
$KAFKA_HOME/bin/kafka-delete-records.sh \
    --bootstrap-server localhost:9092 \
    --offset-json-file /tmp/offsets.json
```

In an analogous manner, we can truncate the entire partition by specifying the current high-water mark in the `offset` attribute.

Listing consumer groups

The `kafka-consumer-groups.sh` tool can be used to query Kafka for a list of consumer groups.

```
$KAFKA_HOME/bin/kafka-consumer-groups.sh \
    --bootstrap-server localhost:9092 --list
```

The result is a newline-separated list of consumer group names. This output format conveniently allows us to iterate over groups, enacting repetitive group-related administrative operations from a shell script.

```
#!/bin/bash

list_groups_cmd="$KAFKA_HOME/bin/kafka-consumer-groups.sh \
    --bootstrap-server localhost:9092 --list"
for group in $(bash -c $list_groups_cmd); do
  # do something with the $group variable
done
```

Describing a consumer group

The same tool can be used to display detailed state information about each consumer group — namely, its partition offsets for the set of subscribed topics. A sample invocation and the resulting output is shown below.

```
$KAFKA_HOME/bin/kafka-consumer-groups.sh \
    --bootstrap-server localhost:9092 \
    --group cli-consumer --describe --all-topics
```

Produces the following when no consumers are connected:

```
Consumer group 'cli-consumer' has no active members.

GROUP         TOPIC           PARTITION  CURRENT-OFFSET
cli-consumer  getting-started 1          0
cli-consumer  getting-started 0          2
cli-consumer  getting-started 2          3
 □
LOG-END-OFFSET  LAG  CONSUMER-ID  HOST  CLIENT-ID
0               0    -            -     -
2               0    -            -     -
3               0    -            -     -
```

If, on the other hand, we attach a consumer (from an earlier example, using the `kafka-console-consumer.sh` tool), the output resembles the following:

```
GROUP           TOPIC           PARTITION  CURRENT-OFFSET
cli-consumer    getting-started 0          2
cli-consumer    getting-started 1          0
cli-consumer    getting-started 2          3
 □
LOG-END-OFFSET  LAG
2               0
0               0
3               0
 □
CONSUMER-ID
consumer-cli-consumer-1-077c1bf1-df64-4d3e-a479-350e962119cc
consumer-cli-consumer-1-077c1bf1-df64-4d3e-a479-350e962119cc
consumer-cli-consumer-1-077c1bf1-df64-4d3e-a479-350e962119cc
 □
HOST            CLIENT-ID
/127.0.0.1      consumer-cli-consumer-1
/127.0.0.1      consumer-cli-consumer-1
/127.0.0.1      consumer-cli-consumer-1
```

In addition to describing a specific consumer group, this tool can be used to describe all groups:

```
$KAFKA_HOME/bin/kafka-consumer-groups.sh \
    --bootstrap-server localhost:9092 \
    --describe --all-groups --all-topics
```

The `--describe` flag has an complementary flag — `--state` — that drills into the present state of the consumer group. This includes the ID of the coordinator node, the assignment strategy, the number of active members, and the state of the group. These attributes are explained in greater detail in Chapter 15: Group Membership and Partition Assignment. The example below illustrates this command and its sample output.

```
$KAFKA_HOME/bin/kafka-consumer-groups.sh \
    --bootstrap-server localhost:9092 \
    --describe --all-groups --state
```

```
GROUP                     COORDINATOR (ID)
cli-consumer              localhost:9092 (0)
 
ASSIGNMENT-STRATEGY  STATE            #MEMBERS
range                Stable           1
```

Resetting offsets

In the course of working with Kafka, we occasionally come across a situation where the committed offsets of a consumer group require minor adjustment; in the more extreme case, that adjustment might entail a complete reset of the offsets. An adjustment might be necessary if, for example, the consumer has to skip over some records — perhaps due to the records containing erroneous data. (These are sometimes referred to as 'poisoned' records.) Alternatively, the consumer may be required to reprocess earlier records — possibly due to a bug in the application which was subsequently resolved. Whichever the reason, the `kafka-consumer-groups.sh` tool can be used with the `--reset-offsets` flag to affect fine-grained control over the consumer group's committed offsets.

The example below rewinds the offsets for the consumer group `cli-consumer` to the low-water mark, using the `--to-earliest` flag — resulting in the forced reprocessing of all records when the consumer group reconnects. Alternatively, the `--to-latest` flag can be used to fast-forward the offsets to the high-water mark extremity, skipping all backlogged records. Resetting offsets is an offline operation; the operation will not proceed in the presence of a connected consumer.

```
$KAFKA_HOME/bin/kafka-consumer-groups.sh \
    --bootstrap-server localhost:9092 \
    --topic getting-started --group cli-consumer \
    --reset-offsets --to-earliest --execute
```

By default, passing the `--reset-offsets` flag will result in a *dry run*, whereby the tool will list the partitions that will be subject to a reset, the existing offsets, as well as the candidate offsets that will be assigned upon completion. This is equivalent of running the tool with the `--dry-run` flag, and is designed to protect the user from accidentally corrupting the consumer group's state. To enact the change, run the command with the `--execute` flag, as shown in the example above.

In addition to resetting offsets for the entire topic, the reset operation can be performed selectively on a subset of the topic's partitions. This can be accomplished by passing in a list of partition numbers following the topic name, in the form `<topic-name>:<first-partition>,<second-partition>,...,<N-th-partitic` An example of this syntax is featured below. Also, rather than resetting the offset to a partition extremity, this example uses the `--to-offset` parameter to specify a numeric offset.

```
$KAFKA_HOME/bin/kafka-consumer-groups.sh \
    --bootstrap-server localhost:9092 \
    --topic getting-started:0,1 --group cli-consumer \
    --reset-offsets --to-offset 2 --execute
```

The next example uses Kafka's record time-stamping to locate an offset based on the given date-time value, quoted in ISO 8601 form. Specifically, the offsets will be reset to the earliest point in time that occurs at the specified timestamp or after it. This feature is convenient when one needs to wind the offsets back to a known point in time. When using the `--to-datetime` parameter, ensure that the offset is passed using the correct timezone; if unspecified, the timezone defaults to the Coordinated Universal Time (UTC), also known as Zulu time. In the example below, the timezone had to be adjusted to Australian Eastern Daylight Time (AEDT), eleven hours east of Zulu, as this book was written in Sydney.

```
$KAFKA_HOME/bin/kafka-consumer-groups.sh \
    --bootstrap-server localhost:9092 \
    --topic getting-started:2 --group cli-consumer \
    --reset-offsets --to-datetime 2020-01-27T14:35:54.528+11:00 \
    --execute
```

The final option offered by this tool is to shift the offsets by a fixed quantity *n*, using the `--shift-by` parameter. The magnitude of the shift may be a positive number — for a forward movement, or a negative number — to rewind the offsets. The extent of the shift is bounded by the partition extremities; the result of 'current offset' + *n* will be capped by the low-water and high-water marks.

Deleting offsets

Another method of resetting the offsets is to delete the offsets altogether, shown in the example below. This is, in effect, a lazy form of reset — the assignment of new offsets does not occur until a consumer connects to the cluster. When this happens, the `auto.offset.reset` client property will stipulate which extremity the offset should be reset to — either the earliest offset or the latest.

```
$KAFKA_HOME/bin/kafka-consumer-groups.sh \
    --bootstrap-server localhost:9092 \
    --topic getting-started --group cli-consumer --delete-offsets
```

Deleting a consumer group

Deleting a consumer group erases all persistent state associated with it. This is accomplished by passing the `--delete` flag to the `kafka-consumer-groups.sh` CLI, as shown below.

```
$KAFKA_HOME/bin/kafka-consumer-groups.sh \
    --bootstrap-server localhost:9092 \
    --group cli-consumer --delete
```

Deleting the consumer group is equivalent to deleting offsets for all topics and all partitions.

A basic Java producer and consumer

A CLI is a great place to start, and can be driven programmatically from a shell script. But this is more of a convenience; an automation of repetitive functionality, if you will. Any serious event streaming application will employ a high-level language such as Java, C, or Python to implement the business logic required to publish records and to react to events emitted by other applications.

Client libraries

Unlike the built-in CLI, which relies on the presence of binaries and a pre-installed Java runtime, applications rely solely on distributable client libraries. These are available for just about every programming language under the sun, from the mainstream to the esoteric.

In this book, we are going to focus solely on the Java ecosystem — being among the most popular mainstream software development environments and the ‘home turf’ of Kafka and many related event streaming technologies. The Java client implementation is the most mature of the available client libraries, being developed alongside and at the same cadence as the Kafka broker. Other languages will have similar clients; they are maintained independently of Kafka and feature varying level of feature support and stability. Bear in mind, these libraries will slightly lag the mainstream Kafka releases in terms of feature sets; if you are after ‘bleeding edge’ capabilities, you will be best served by the Java client library and, to a marginally lesser extent, the C library — `librdkafka`, maintained by Magnus Edenhill.

Using the Java library

To add a Kafka client library to your project, add the following to your `build.gradle` (if using Gradle):

```
dependencies {
    implementation "org.apache.kafka:kafka-clients:2.4.0"
}
```

Alternatively, if using Maven, add the following to your `pom.xml`:

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>2.4.0</version>
</dependency>
```

The examples above assume Kafka version 2.4.0 — the latest at the time of writing. Replace this with a more up-to-date version if appropriate.

The complete source code for the upcoming examples is available at github.com/ekoutanov/effectivekafka[6] in the `src/main/java/effectivekafka/basic` directory. Code listings will have their package declaration removed for brevity, and often will strip out `import` statements and outer class declarations.

Interfacing with the Kafka client libraries is done primarily using the following classes:

- `Producer`: The public interface of the producer client, containing the necessary method signatures for publishing records and using transactions. This interface is surprisingly light on documentation; method comments simply delegate the documentation to the concrete implementation.
- `KafkaProducer`: The implementation of `Producer`. In addition, a `KafkaProducer` contains detailed Javadoc comments for each method.
- `ProducerRecord`: A data structure encompassing the attributes of a record, as perceived by a producer. To be precise, this is the representation of a record *before* it has been published to a partition; as such, it contains only the basic set of attributes: topic name, partition number, optional headers, key, value, and a timestamp.
- `Consumer`: The definition of a consumer entity, containing message signatures for controlling subscriptions and topic/partition assignment, fetching records from the cluster, committing offsets, and obtaining information about the available topics and partitions.

[6]https://github.com/ekoutanov/effectivekafka/tree/master/src/main/java/effectivekafka/basic

- `KafkaConsumer`: The implementation of `Consumer`. Like its producer counterpart, this implementation contains the complete set of Javadocs.
- `ConsumerRecord`: A consumer-centric structure for housing record attributes. A `ConsumerRecord` is effectively a superset of the `ProducerRecord`, containing additional metadata such as the record offset, the checksum, and some other internal attributes.

There are other classes that are used, from time to time, to interface with the client library. However, the bulk of record publishing and consumption can be achieved using little more than just the six classes above.

Publishing records

A simple, yet complete example illustrating the publishing of Kafka records is presented below.

```java
import static java.lang.System.*;

import java.util.*;

import org.apache.kafka.clients.producer.*;
import org.apache.kafka.common.serialization.*;

public final class BasicProducerSample {
  public static void main(String[] args)
      throws InterruptedException {
    final var topic = "getting-started";

    final Map<String, Object> config =
        Map.of(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
               "localhost:9092",
               ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
               StringSerializer.class.getName(),
               ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
               StringSerializer.class.getName(),
               ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG,
               true);

    try (var producer = new KafkaProducer<String, String>(config)) {
      while (true) {
        final var key = "myKey";
        final var value = new Date().toString();
        out.format("Publishing record with value %s%n",
                   value);
        final Callback callback = (metadata, exception) -> {
          out.format("Published with metadata: %s, error: %s%n",
                     metadata, exception);
        };

        // publish the record, handling the metadata in the callback
        producer.send(new ProducerRecord<>(topic, key, value),
                      callback);

        // wait a second before publishing another
        Thread.sleep(1000);
      }
    }
  }
}
```

The first item on our to-do list is to configure the client. This is done by building a mapping of property names to configured values. More detailed information on configuration is presented in Chapter 10: Client Configuration; for the time being, we will limit ourselves to the most basic configuration options — just enough to get us going with a functioning producer.

The configuration keys are strings — being among the permissible property names defined in the official Kafka documentation, available online at kafka.apache.org/documentation[7]. Rather than quoting strings directly, our example employs the static constants defined in the `ProducerConfig` class, thereby avoiding a mistype.

Of the four configuration mappings supplied, the first specifies a list of so-called *bootstrap servers*. In our example, this is a singleton list comprising the endpoint `localhost:9092` — the address of our test broker. Bootstrapping is a moderately involved topic, described in Chapter 8: Bootstrapping and Advertised Listeners.

The next two mappings specify the serializers that the producer should use for the records' keys and values. Kafka offers lots of options around how keys and values are marshalled — using either built-in or custom serializers. For the sake of expediency, we will go with the simplest option at our disposal — writing records as plain strings. More elaborate forms of marshalling will be explored in Chapter 7: Serialization.

Whereas the first three items represent mandatory configuration, the fourth is entirely optional. By default, in the absence of idempotence, a producer may inadvertently publish a record in duplicate or out-of-order — if one of the queued records experiences a timeout during publishing and is reattempted after one or more of its successors have gone through. With the `enable.idempotence` option set to `true`, the broker will maintain an internal sequence number for each producer and

[7]https://kafka.apache.org/documentation/#producerconfigs

partition pair, ensuring that records are not processed in duplicate or out-of-order. So it's good practice to enable idempotence.

Prior to publishing a record, we need to instantiate a `KafkaProducer`, giving it the assembled config map in the constructor. A producer cannot be reconfigured following instantiation. One instantiated, we will keep a reference to the `KafkaProducer` instance, as it must be closed when the application no longer needs it. This is important because a `KafkaProducer` maintains TCP connections to multiple brokers and also operates a background I/O thread to ferry the records across. Failure to close the producer instance may result in resource starvation on the client, as well as on the brokers. As `Producer` extends the `Closeable` interface, the best way to ensure that the producer instance is properly disposed of is to use a *try-with-resources* block, as shown in the listing above.

Sometimes we need a producer to hang around indefinitely — for example, when an application publishes events in response to some external stimuli, such as responding to an API request. In this scenario, the use of a *try-with-resources* is inappropriate, as the lifecycle of a `KafkaProducer` instance is obviously aligned with that of the API controller or the associated business logic layer (depending on how the application is architected). Instead, we would let the owner of the producer, whichever component that may be, deal with lifecycle concerns.

In order to actually publish a record, one must use the `Producer.send()` API. There are two overloaded variations of `send()` method:

1. `Future<RecordMetadata> send(ProducerRecord<K, V> record)`: asynchronously sends the record, returning a `Future` containing the record metadata.
2. `Future<RecordMetadata> send(ProducerRecord<K, V> record, Callback callback)`: asynchronously sends the record, invoking the given `Callback` implementation when either the record has been successfully persisted on the broker or an error has occurred. Our example uses this variant.

The `send()` methods are asynchronous, returning as soon as the record is serialized and staged in the accumulator buffer. The actual sending of the record will be performed in the background, by a dedicated I/O thread. To block on the result, the application can invoke the `get()` method of the provided `Future`.

The publishing of records takes place in a loop, with a one second sleep between each successive `send()` call. For simplicity, we are publishing the current date, keyed to a constant `"myKey"`. This means that all records will appear on the same partition.

Running the example above results in the following output (until terminated):

```
13:14:09/0  INFO  [main]: [Producer clientId=basic-producer-sample] ▯
    Instantiated an idempotent producer.
13:14:09/66 INFO  [main]: [Producer clientId=basic-producer-sample] ▯
    Overriding the default retries config to the recommended ▯
    value of 2147483647 since the idempotent producer is ▯
    enabled.
13:14:09/66 INFO  [main]: [Producer clientId=basic-producer-sample] ▯
    Overriding the default acks to all since idempotence is enabled.
13:14:09/82 INFO  [main]: Kafka version: 2.4.0
13:14:09/82 INFO  [main]: Kafka commitId: 77a89fcf8d7fa018
13:14:09/82 INFO  [main]: Kafka startTimeMs: 1570264049533
Publishing record with value Wed Jan 02 13:14:09 AEDT 2020
13:14:09/495 INFO  [kafka-producer-network-thread | basic- ▯
    producer-sample]: [Producer clientId=basic-producer-sample] ▯
    Cluster ID: efkResGcSUWMV6zqj9D8vw
13:14:09/497 INFO  [kafka-producer-network-thread | basic- ▯
    producer-sample]: [Producer clientId=basic-producer-sample] ▯
    ProducerId set to 12000 with epoch 0
Published with metadata: getting-started-0@0, error: null
Publishing record with value Wed Jan 02 13:14:10 AEDT 2020
Published with metadata: getting-started-0@1, error: null
Publishing record with value Wed Jan 02 13:14:11 AEDT 2020
Published with metadata: getting-started-0@2, error: null
Publishing record with value Wed Jan 02 13:14:12 AEDT 2020
Published with metadata: getting-started-0@3, error: null
```

Consuming records

The following listing demonstrates how records are consumed.

```java
import static java.lang.System.*;

import java.time.*;
import java.util.*;

import org.apache.kafka.clients.consumer.*;
import org.apache.kafka.common.serialization.*;

public final class BasicConsumerSample {
  public static void main(String[] args) {
    final var topic = "getting-started";
    final Map<String, Object> config =
        Map.of(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
               "localhost:9092",
               ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
               StringDeserializer.class.getName(),
               ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
               StringDeserializer.class.getName(),
               ConsumerConfig.GROUP_ID_CONFIG,
               "basic-consumer-sample",
               ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
               "earliest",
               ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG,
               false);

    try (var consumer = new KafkaConsumer<String, String>(config)) {
      consumer.subscribe(Set.of(topic));

      while (true) {
        final var records = consumer.poll(Duration.ofMillis(100));
        for (var record : records) {
          out.format("Got record with value %s%n", record.value());
        }
        consumer.commitAsync();
      }
    }
  }
}
```

This example is strikingly similar to the producer — same building of a config map, instantiation of a client, and the use of a *try-with-resources* block to ensure the client is closed after it leaves scope.

The first difference is in the configuration. Consumer have some elements common with producers — such as the `bootstrap.servers` list, and several others — but by and large they are different. The deserializer configuration is symmetric to the producer's serializer properties; there are key and value equivalents.

The group ID configuration is optional — it specifies the ID of the consumer group. This example uses a consumer group named `basic-consumer-sample`. The auto offset reset configuration stipulates what happens when the consumer subscribes to the topic for the first time. In this case, we would like the consumer's offset to be reset to the low-water mark for every affected partition, meaning that the consumer will get any backlogged records that existed prior to the creation of the group in Kafka. The default setting is `latest`, meaning the consumer will not read any prior records.

Finally, the auto-commit setting is disabled, meaning that the application will commit offsets at its discretion. The default setting is to enable auto-commit with a minimum interval of five seconds.

Prior to polling for records, the application must subscribe to one or more topics using the `Consumer.subscribe()` method.

Once subscribed, the application will repeatedly invoke `Consumer.poll()` in a loop, blocking up to a maximum specified duration or until a batch of records is received. For each received records, this example simply prints the record's value. Once all records have been printed, the offsets are committed asynchronously using the `Consumer.commitAsync()` method. The latter returns as soon the offsets are enqueued internally; the actual sending of the commit message to the group coordinator will take place on the background I/O thread. (The group coordinator is responsible for arbitrating the state of the consumer group.) The reader might also recall from Chapter 3: Architecture and Core Concepts, that the repeated polling and handling of records is called the *poll-process* loop.

Running the example above results in the following output (until terminated):

```
10:47:16/0     INFO  [main]: Kafka version: 2.4.0
10:47:16/0     INFO  [main]: Kafka commitId: 77a89fcf8d7fa018
10:47:16/0     INFO  [main]: Kafka startTimeMs: 1580341636585
10:47:16/2     INFO  [main]: [Consumer clientId=consumer-basic- ▯
    consumer-sample-1, groupId=basic-consumer-sample] Subscribed ▯
    to topic(s): getting-started
10:47:17/431   INFO  [main]: [Consumer clientId=consumer-basic- ▯
    consumer-sample-1, groupId=basic-consumer-sample] Cluster ID: ▯
    efkResGcSUWMV6zqj9D8vw
10:47:18/1740  INFO  [main]: [Consumer clientId=consumer-basic- ▯
    consumer-sample-1, groupId=basic-consumer-sample] Discovered ▯
    group coordinator 172.20.40.148:9092 (id: 2147483647 rack: null)
10:47:18/1744  INFO  [main]: [Consumer clientId=consumer-basic- ▯
    consumer-sample-1, groupId=basic-consumer-sample] ▯
    (Re-)joining group
10:47:18/1795  INFO  [main]: [Consumer clientId=consumer-basic- ▯
    consumer-sample-1, groupId=basic-consumer-sample] ▯
    (Re-)joining group
10:47:18/1828  INFO  [main]: [Consumer clientId=consumer-basic- ▯
    consumer-sample-1, groupId=basic-consumer-sample] Finished ▯
    assignment for group at generation 1: {consumer-basic-consumer- ▯
    sample-1-26dce919-7f7d-4e04-98d9-99b091c73b3d=org.apache.kafka. ▯
    clients.consumer.ConsumerPartitionAssignor$Assignment@66ea810}
10:47:18/1881  INFO  [main]: [Consumer clientId=consumer-basic- ▯
    consumer-sample-1, groupId=basic-consumer-sample] Successfully ▯
    joined group with generation 1
10:47:18/1884  INFO  [main]: [Consumer clientId=consumer-basic- ▯
    consumer-sample-1, groupId=basic-consumer-sample] Adding newly ▯
    assigned partitions: getting-started-1, getting-started-0, ▯
    getting-started-2
10:47:18/1900  INFO  [main]: [Consumer clientId=consumer-basic-
    consumer-sample-1, groupId=basic-consumer-sample] Found no
    committed offset for partition getting-started-1
10:47:18/1900  INFO  [main]: [Consumer clientId=consumer-basic-
    consumer-sample-1, groupId=basic-consumer-sample] Found no
    committed offset for partition getting-started-0
10:47:18/1900  INFO  [main]: [Consumer clientId=consumer-basic-
    consumer-sample-1, groupId=basic-consumer-sample] Found no
    committed offset for partition getting-started-2
10:47:18/1921  INFO  [main]: [Consumer clientId=consumer-basic-
    consumer-sample-1, groupId=basic-consumer-sample] Resetting
    offset for partition getting-started-1 to offset 0.
10:47:18/1921  INFO  [main]: [Consumer clientId=consumer-basic-
    consumer-sample-1, groupId=basic-consumer-sample] Resetting
    offset for partition getting-started-0 to offset 0.
10:47:18/1921  INFO  [main]: [Consumer clientId=consumer-basic-
    consumer-sample-1, groupId=basic-consumer-sample] Resetting
    offset for partition getting-started-2 to offset 0.
Got record with value Wed Jan 02 13:14:09 AEDT 2020
Got record with value Wed Jan 02 13:14:10 AEDT 2020
Got record with value Wed Jan 02 13:14:11 AEDT 2020
Got record with value Wed Jan 02 13:14:12 AEDT 2020
Got record with value Wed Jan 02 13:14:13 AEDT 2020
Got record with value Wed Jan 02 13:14:14 AEDT 2020
Got record with value Wed Jan 02 13:14:15 AEDT 2020
Got record with value Wed Jan 02 13:14:16 AEDT 2020
Got record with value Wed Jan 02 13:14:17 AEDT 2020
```

This chapter has hopefully served as a practical reflection on the theoretical concepts that were outlined in Chapter 3: Architecture and Core Concepts. Specifically, we learned how to interact with a Kafka cluster using two distinct, yet commonly used approaches.

The part explored the use of the built-in CLI tools. These are basic utilities that allow a user to publish and consume records, administer topics and consumer groups, make various configuration changes, and query various aspects of the cluster state. As we have come to realise, the built-in tooling is far from perfect, but it is sufficient to carry out basic administrative operations, and at times it may be the only toolset at our disposal.

The second part looked at the programmatic interaction with Kafka, using the Java client library. We looked at simple examples for publishing and consuming records and learned the basics of the

Java client API. Real applications will undoubtedly be more complex than the provided examples, but they will invariably utilise the exact same building blocks.