Chapter 4: Installation

Previous chapters have given us a reasonably grounded understanding of what Kafka is and isn't, where it is used, and how it dovetails into the rest of the software landscape. So far we have just been circling the shore; time to dive in for a deeper look. Before we can get much further, we need a running Kafka setup.

This will require us to install several things:

1. **Kafka and ZooKeeper**. Recall from our earlier coverage of the Kafka architecture, a functioning setup requires *both* Kafka and ZooKeeper nodes. These are plain Java applications with no other requirements or dependencies, and can run on any operating system and any hardware supported by Java.
2. **Kafdrop**. While this isn't strictly required to operate Kafka, Kafdrop is the most widely-used web-based tool for working with Kafka, and would be widely considered as an essential item of your Kafka toolkit.
3. **A Java Development Kit (JDK)**. Kafka is part-written in Scala and part in Java, and requires Java version 8 or newer to run. Kafdrop is somewhat more modern, requiring Java 11.

If you haven't done so already, install a copy of the JDK. Version 11 or newer will do. The rest of the chapter assumes that you have a JDK installed.

The requirement for JDK 11 is to accommodate Kafdrop. If you are installing Kafka on its own, JDK 8 will suffice. However, Java 11 is still recommended over Java 8 as it is the current *Long-Term Support* (LTS) version. The last free public updates of JDK 8 will have ended in January 2019. Free updates for JDK 11 will continue through to September 2021, by which point the world would have transitioned to JDK 17 LTS. (LTS releases are publicly supported for three years.)

Installing Kafka and ZooKeeper

There are at least four avenues at one's disposal for installing Kafka and ZooKeeper:

1. Run Kafka and ZooKeeper using Docker.
2. Install Kafka and ZooKeeper using a package manager, such as DNF (formerly known as YUM) for RedHat, CentOS and Fedora Linux distributions, APT for Debian and Ubuntu, and Homebrew for macOS. There are many others.

3. Clone the Kafka and ZooKeeper repositories and build from source code.
4. Download and unpack the official Kafka distribution from kafka.apache.org/downloads[1], which comes bundled with ZooKeeper.

Let's briefly touch upon each of these options. It might sound like overkill at first — we could just pick the easiest and get cracking — but we're in it for the long haul. And sometimes the easiest approach isn't the right one.

Docker is an excellent all-round approach for getting started with Kafka, developing against a local Kafka broker, and even running Kafka in a production configuration. And best of all, a Docker image will come with an appropriate version of the JDK.

There is one drawback, however. Kafka in Docker is notoriously difficult to configure, as everything is baked in and not designed for change. (That's not to say it can't be done.) We won't go down the Docker path for now, just because we will be making lots of changes to the broker configuration at various points along our journey, and ideally, we would want this process to be as simple and painless as possible.

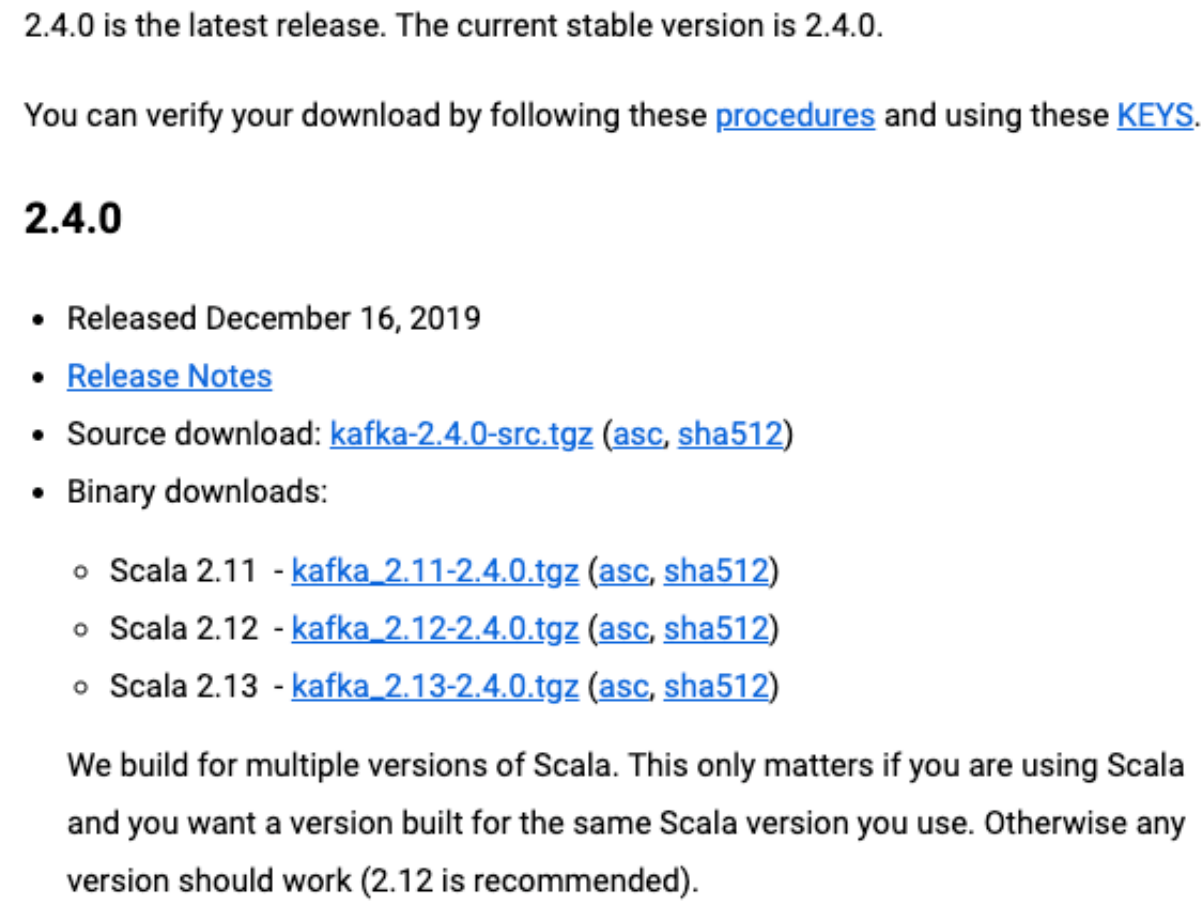
Installing Kafka and ZooKeeper from a package is convenient too. And while it doesn't come bundled with a JDK, a package will typically declare the JDK as a dependency, which the package manager will attempt to resolve at the point of installation (or when updating the package). Still, this approach isn't without its drawbacks: there may be other applications installed on the target machine that might require a different version of the JDK, which would warrant further configuration of Kafka and ZooKeeper to wire it up to the correct JDK.

Another drawback to the 'packaged Kafka' approach is that the installation path and the layout of the files will vary depending on the chosen package manager. For example, on macOS, Homebrew installs Kafka into `/usr/local/etc`, `/usr/local/bin`, and `/usr/local/var/lib/`. On the other hand, YUM will install it under `/bin`, `/opt`, and `/var/lib`. This makes it tremendously difficult to write about and include worked examples that work consistently for all readers. Rather than focusing on the subject matter, this book would have been polluted with excerpts and call-outs targeting different operating systems and package managers.

The third option is building from source code. It might sound a bit extreme for someone who has just opened a book on Kafka. Nonetheless, it is a valid option and the *only* option if you happen to be a contributor. Understandably, we won't delve into it much deeper, and blissfully pretend it doesn't exist — at least in the universe bound by this book.

The final option, and the one we will inevitably proceed with, is to download the latest version of the official Kafka tarball from kafka.apache.org/downloads[2]. There might be several options — pick the one in 'Binary downloads' that targets the latest version of Scala, as shown in the screenshot below.

[1] https://kafka.apache.org/downloads
[2] https://kafka.apache.org/downloads



2.4.0 is the latest release. The current stable version is 2.4.0.

You can verify your download by following these procedures and using these KEYS.

2.4.0

- Released December 16, 2019
- Release Notes
- Source download: kafka-2.4.0-src.tgz (asc, sha512)
- Binary downloads:
  - Scala 2.11 - kafka_2.11-2.4.0.tgz (asc, sha512)
  - Scala 2.12 - kafka_2.12-2.4.0.tgz (asc, sha512)
  - Scala 2.13 - kafka_2.13-2.4.0.tgz (asc, sha512)

We build for multiple versions of Scala. This only matters if you are using Scala and you want a version built for the same Scala version you use. Otherwise any version should work (2.12 is recommended).

Download options

Copy the downloaded `.tgz` file into a directory of your choice, and unpack with `tar zxf kafka_2.13-2.4.0.tgz`, replacing the filename as appropriate. (In this example, the downloaded version is 2.4.0, but your version will likely be newer.) The files will be unpacked to a subdirectory named `kafka_2.13-2.4.0`. We will refer to this directory as the **Kafka home directory**.

When referring to the home directory from a command-line example, we'll use the constant `$KAFKA_HOME`. You have the choice of either manually substituting `$KAFKA_HOME` for the installation directory, or assigning the installation path to the `KAFKA_HOME` environment variable, as shown in the example below.

```
export KAFKA_HOME=/Users/me/opt/kafka_2.13-2.4.0
```

You would need to run `export KAFKA_HOME...` at the beginning of every terminal session. Alternatively, you can append the `export ...` command to your shell's startup file. If you are using Bash, this is typically `~/.bashrc` or `~/.bash_profile`.

Take a moment to look around the home directory. You'll see several subdirectories, chief among them being `bin`, `libs`, and `config`.

```
.
├── LICENSE
├── NOTICE
├── bin
├── config
├── libs
└── site-docs
```

The `bin` directory contains the scripts to start and stop Kafka and ZooKeeper, as well as various CLI (command-line interface) utilities for working with Kafka. Also, `bin` contains a `windows` subdirectory, which (you've guessed it) contains the equivalent scripts for Microsoft Windows.

```
bin
├── connect-distributed.sh
├── connect-standalone.sh
├── kafka-acls.sh
├── kafka-broker-api-versions.sh
├── kafka-configs.sh
├── kafka-console-consumer.sh
├── kafka-console-producer.sh
├── kafka-consumer-groups.sh
├── kafka-consumer-perf-test.sh
├── kafka-delegation-tokens.sh
├── kafka-delete-records.sh
├── kafka-dump-log.sh
├── kafka-log-dirs.sh
├── kafka-mirror-maker.sh
├── kafka-preferred-replica-election.sh
├── kafka-producer-perf-test.sh
├── kafka-reassign-partitions.sh
├── kafka-replica-verification.sh
├── kafka-run-class.sh
├── kafka-server-start.sh
├── kafka-server-stop.sh
├── kafka-streams-application-reset.sh
├── kafka-topics.sh
├── kafka-verifiable-consumer.sh
├── kafka-verifiable-producer.sh
├── trogdor.sh
├── windows
```

```
├── zookeeper-security-migration.sh
├── zookeeper-server-start.sh
├── zookeeper-server-stop.sh
└── zookeeper-shell.sh
```

The `config` directory is another important one. It contains `.properties` files that are used to configure the various components that make up the Kafka ecosystem.

```
config
├── connect-console-sink.properties
├── connect-console-source.properties
├── connect-distributed.properties
├── connect-file-sink.properties
├── connect-file-source.properties
├── connect-log4j.properties
├── connect-standalone.properties
├── consumer.properties
├── log4j.properties
├── producer.properties
├── server.properties
├── tools-log4j.properties
├── trogdor.conf
└── zookeeper.properties
```

The `lib` directory contains the Kafka binary distribution, as well as its direct and transitive dependencies. You'll never need to modify the contents of this directory.

Launching Kafka and ZooKeeper

Now that the applications have been installed, we can start them. The first cab off the rank will be ZooKeeper, as it is a runtime requirement for Kafka. Run the following command in a terminal:

```
$KAFKA_HOME/bin/zookeeper-server-start.sh \
    $KAFKA_HOME/config/zookeeper.properties
```

This will launch ZooKeeper in foreground mode. You should see a bunch of messages logged to the console, signifying the starting of ZooKeeper. Among them, you might spot one warning message:

```
[2019-12-25 13:30:36,951] WARN Either no config or no quorum defined ␣
    in config, running in standalone mode (org.apache.zookeeper. ␣
    server.quorum.QuorumPeerMain)
```

All this is saying is that we started ZooKeeper in standalone mode, without configuring a quorum. When running ZooKeeper locally, availability is rarely a concern, and a standalone (ensemble of one member node) configuration is sufficient.

Recall from a prior discussion on the Kafka architecture, it was stated that ZooKeeper acts as an arbiter — electing a sole controller among the available Kafka broker nodes. Internally, ZooKeeper employs an atomic broadcast protocol to agree on and subsequently maintain a consistent view of the cluster state throughout the ZooKeepr *ensemble*. This protocol operates on the concept of a *majority vote*, also known as *quorum*, which in turn, requires an odd number of participating ZooKeeper nodes. When running in a production environment, ensure that at least three nodes are deployed in a manner that no pair of nodes may be impacted by the same contingency. Ideally, ZooKeeper nodes should be deployed in geographically separate data centres.

Now that ZooKeeper is running, we can start Kafka. Run the following in a new terminal window:

```
$KAFKA_HOME/bin/kafka-server-start.sh \
    $KAFKA_HOME/config/server.properties
```

Kafka's logs are a bit more verbose than ZooKeeper's. The first really useful part of the log relates to the ZooKeeper connection. Specifically, which ZooKeeper instance(s) Kafka is trying to connect to, and the status of the connection:

```
[2019-12-25 14:02:21,380] INFO Initiating client connection, ␣
    connectString=localhost:2181 sessionTimeout=6000 watcher= ␣
    kafka.zookeeper.ZooKeeperClient$ZooKeeperClientWatcher$@ ␣
    624ea235 (org.apache.zookeeper.ZooKeeper)
[2019-12-25 14:02:21,399] INFO [ZooKeeperClient Kafka server] ␣
    Waiting until connected. (kafka.zookeeper.ZooKeeperClient)
[2019-12-25 14:02:21,401] INFO Opening socket connection to server ␣
    localhost/0:0:0:0:0:0:0:1:2181. Will not attempt to ␣
    authenticate using SASL (unknown error) (org.apache.zookeeper. ␣
    ClientCnxn)
[2019-12-25 14:02:21,416] INFO Socket connection established to ␣
    localhost/0:0:0:0:0:0:0:1:2181, initiating session ␣
    (org.apache.zookeeper.ClientCnxn)
[2019-12-25 14:02:21,458] INFO Session establishment complete on ␣
    server localhost/0:0:0:0:0:0:0:1:2181, sessionid = ␣
```

```
    0x100433a40960000, negotiated timeout = 6000 □
    (org.apache.zookeeper.ClientCnxn)
[2019-12-25 14:02:21,461] INFO [ZooKeeperClient Kafka server] □
    Connected. (kafka.zookeeper.ZooKeeperClient)
```

Among the logs we can also find the complete Kafka broker configuration:

```
[2019-12-25 14:02:22,118] INFO KafkaConfig values:
  advertised.host.name = null
  advertised.listeners = null
  advertised.port = null
  alter.config.policy.class.name = null
  alter.log.dirs.replication.quota.window.num = 11
  alter.log.dirs.replication.quota.window.size.seconds = 1
  authorizer.class.name =
  auto.create.topics.enable = true
  auto.leader.rebalance.enable = true
  background.threads = 10
  broker.id = 0
  broker.id.generation.enable = true
  broker.rack = null
  client.quota.callback.class = null
  compression.type = producer
  connection.failed.authentication.delay.ms = 100
  ...
  (rest of the log omitted for brevity)
```

This is actually more useful than one might initially imagine. The Kafka broker configuration is defined in `$KAFKA_HOME/config/server.properties`, but the file is relatively small and initially contains mostly commented-out entries. This means that most settings are assigned their default values. Rather than consulting the official documentation to determine what the defaults might be and whether or not they are actually overridden in your configuration, you need only look at the broker logs. This is particularly useful when you need to debug the configuration. Suppose a particular configuration value isn't being applied correctly — perhaps due to a simple typo, or maybe because there are two entries for the same configuration key. The configuration printout in Kafka's logs provides its vantage point — as seen from the eyes of the broker.

The next useful bit of information is emitted by the socket listener:

```
[2019-12-25 14:02:22,587] INFO Awaiting socket connections on □
    0.0.0.0:9092. (kafka.network.Acceptor)
```

This tells us that Kafka is listening for inbound connections on port `9092`, and is bound to all network interfaces (indicated by the IP meta-address `0.0.0.0`). This is corroborated by the deprecated property `port`, which defaults to `9092`. There is a much more sophisticated mechanism for configuring listeners, which we will examine in one of the following chapters. For now, a `0.0.0.0:9092` listener will suffice.

Believe it or not, the most useful information one can get out of Kafka's logs is actually the version number. Admittedly, it sounds somewhat banal, but how many times have you stared helplessly at the screen wondering why a piece of software that was just upgraded to the latest version still has the same bug that the authors have sworn they had fixed? Invariably, it is always some simple mistake — a symlink to the wrong binary, a typo in the path, a wrong value in an environment variable, or some other moth-eaten stuff-up along those lines. Printing the application version number in the logs is a simple way of eradicating these classes of errors.

Running in the background

When launching ZooKeeper or Kafka, you have the option of running it as a daemon by passing it the `-daemon` flag. In simple terms, this means launching ZooKeeper in the background, without holding up the terminal. Kill the existing ZooKeeper process by pressing `CTRL+C`, and try the following:

```
$KAFKA_HOME/bin/zookeeper-server-start.sh -daemon \
    $KAFKA_HOME/config/zookeeper.properties
```

That's all well and good, but where have the logs gone? When launched as a daemon, the standard output of the ZooKeeper process is piped to `$KAFKA_HOME/logs/zookeeper.out`. We can tail the logs by running:

```
tail -f $KAFKA_HOME/logs/zookeeper.out
```

To stop a daemon ZooKeeper process, run `$KAFKA_HOME/bin/zookeeper-server-stop.sh`. This will stop the background process if there is one running. If not, it will respond with `No zookeeper server to stop`.

ZooKeeper and Kafka shell scripts are essentially mirror images of each other. To launch Kafka as a daemon, run:

```
$KAFKA_HOME/bin/kafka-server-start.sh -daemon \
    $KAFKA_HOME/config/server.properties
```

Kafka standard output logs are written to `$KAFKA_HOME/logs/kafkaServer.out`. To stop a daemon Kafka process, run `$KAFKA_HOME/bin/kafka-server-stop.sh`.

Installing Kafdrop

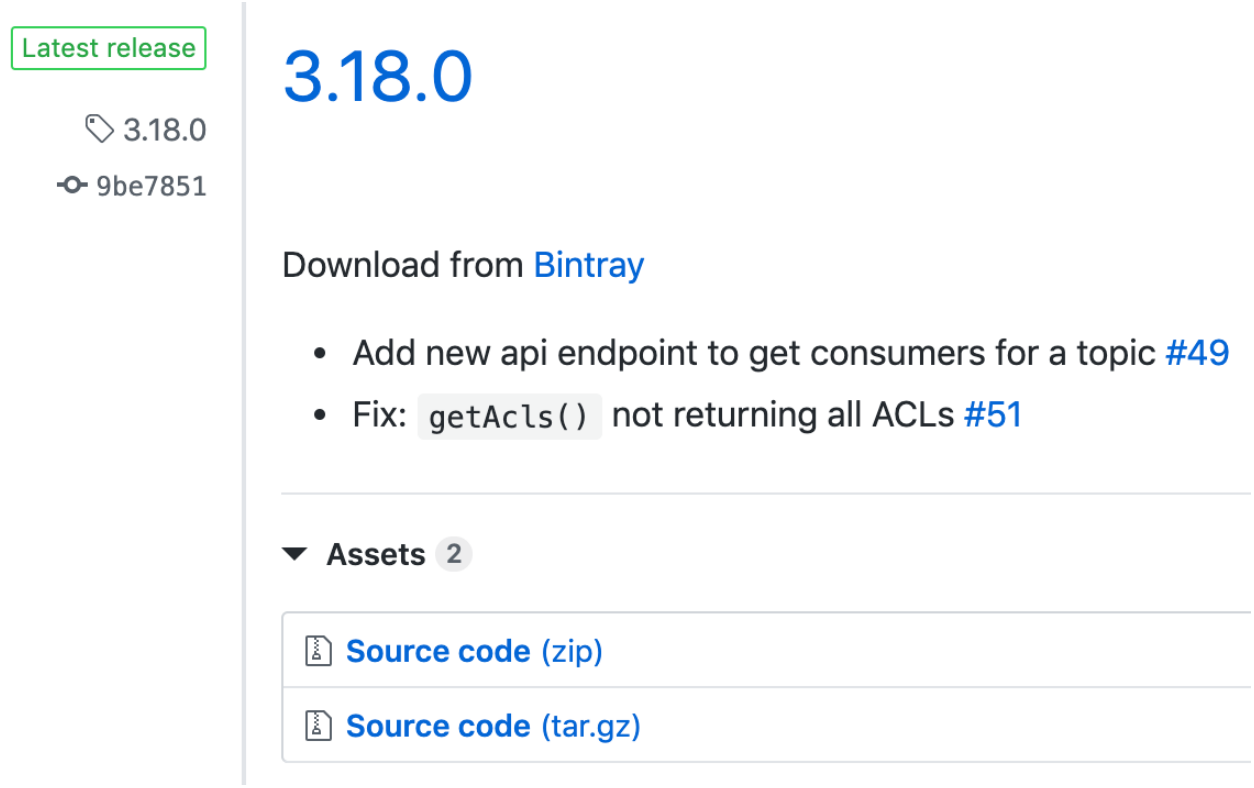
Next on our list is Kafdrop. It's a Java application with no dependencies, and the avenues for installing it are mostly similar to Kafka, except it does not offer package-based installation. In practice, Docker largely obviates the need for packages, and the sheer number of Kafdrop Docker pulls (over a million at the time of writing) is a testament to that.

As practical as a Docker image may be, we are going to ditch this option for now. Because we are running Kafka on `localhost`, Docker will struggle to connect to our broker, as Docker containers are normally unaware of processes running on the host machine. There is a way to change this but it is not portable across Linux and macOS, and will also require changes to the Kafka broker configuration — something we are not yet prepared to do. We will revisit Docker later. For now, we will go with the official Kafdrop binary distribution.

Kafdrop binaries are hosted on Bintray, with a download link embedded in each release on GitHub. Open the releases page: github.com/obsidiandynamics/kafdrop/releases[3] and pick the latest from the list. Alternatively, you can navigate straight to the latest Kafdrop release by following this shortcut: github.com/obsidiandynamics/kafdrop/releases/latest[4].

The release will typically include an outline of changes, and will contain a 'Download from Bintray' link, as shown in the example below:

[3] https://github.com/obsidiandynamics/kafdrop/releases
[4] https://github.com/obsidiandynamics/kafdrop/releases/latest



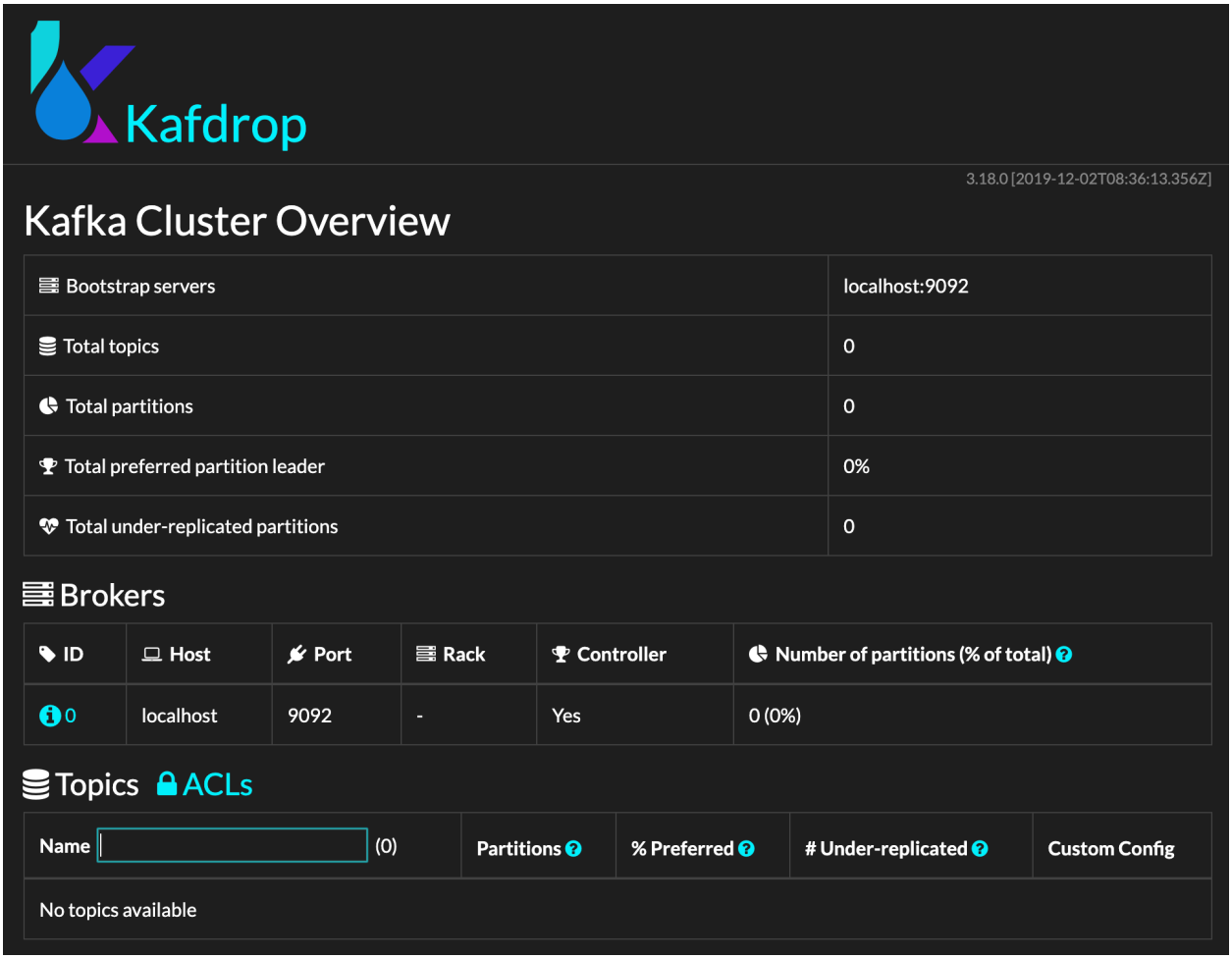
Download Kafdrop release

Clicking on the link will download a `.jar` file. Save it in a directory of your choice and run it as shown in the example below, replacing the filename as appropriate.

```
java -jar kafdrop-3.18.0.jar --kafka.brokerConnect=localhost:9092
```

Once started, you can open Kafdrop in your browser by navigating to localhost:9000[5]. You'll be presented with a Kafdrop cluster overview screen, showing our fresh, single-node Kafka cluster.

[5] http://localhost:9000



Kafdrop cluster overview

There are a few things of interest here. On the top-right corner, you should see the Kafdrop release version and buildstamp. This can be very useful if you are connecting to a remote Kafdrop instance, and don't have the logs that disclose which version of Kafdrop is running.

The next section provides a summary of the cluster. Note the 'Bootstrap servers' field: it mirrors the `--kafka.brokerConnect` command-line argument, telling us how Kafdrop has been configured to discover the Kafka nodes.

Bootstrapping and broker discovery is a whole topic on its own, which we are going to gloss over for now. For the time being, and unless stated otherwise, assume that the 'bootstrap servers' list is `localhost:9092`. We will revisit this topic in Chapter 8: Bootstrapping and Advertised Listeners.

The 'Brokers' section enumerates over the individual brokers in the cluster. We are running a single-broker setup just now, so seeing a one-line table entry should come as no surprise. Naturally, being the only broker in the cluster, it will have been assigned the controller role.

The 'Topics' section is empty, as we haven't created any topics yet. Nor do we have any ACLs defined. This is all yet to come.

Switch back to the shell running Kafdrop. Looking over the standard output logs we can spot the version number. Keep looking and you'll notice the port that Kafdrop is listening on and the context path. This is all configuration, and it's something that may need to change between environments.

```
2019-12-25 19:08:49.465 INFO 82515 [main] k.s.BuildInfo: Kafdrop 
    version: 3.18.0, build time: 2019-12-02T08:36:13.356Z
...
(some logs omitted)
...
2019-12-25 19:08:50.752 INFO 82515 [main] o.s.b.w.e.u. 
    UndertowServletWebServer: Undertow started on port(s) 
    9000 (http) with context path ''
```

We have learned about the various ways one can obtain and install a Kafka and ZooKeeper bundle. We have also started and stopped a basic ZooKeeper and Kafka setup, learned about foreground and daemon modes, and surveyed the logs for useful information. Finally, we installed Kafdrop and took a brief look around. The scene is now set; we just need to make use of it all somehow.