**CHAPTER 9**

**Setting Up Flink for Streaming Applications**

Today’s data infrastructures are diverse. Distributed data processing frameworks like Apache Flink need to be set up to interact with several components such as resource managers, filesystems, and services for distributed coordination.

In this chapter, we discuss the different ways to deploy Flink clusters and how to con‐ figure them securely and make them highly available. We explain Flink setups for dif‐ ferent Hadoop versions and filesystems and discuss the most important configuration parameters of Flink’s master and worker processes. After reading this chapter, you will know how to set up and configure a Flink cluster.

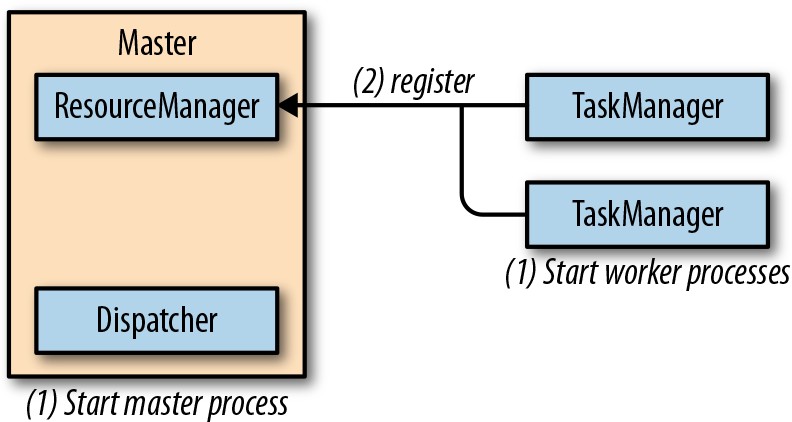
# Deployment Modes

Flink can be deployed in different environments, such as a local machine, a bare- metal cluster, a Hadoop YARN cluster, or a Kubernetes cluster. In [“Components of a](#_bookmark0) [Flink Setup”](#_bookmark0), we introduced the different components of a Flink setup: the JobMan‐ ager, TaskManager, ResourceManager, and Dispatcher. In this section, we explain how to configure and start Flink in different environments—including standalone clusters, Docker, Apache Hadoop YARN, and Kubernetes—and how Flink’s compo‐ nents are assembled in each setup.

## Standalone Cluster

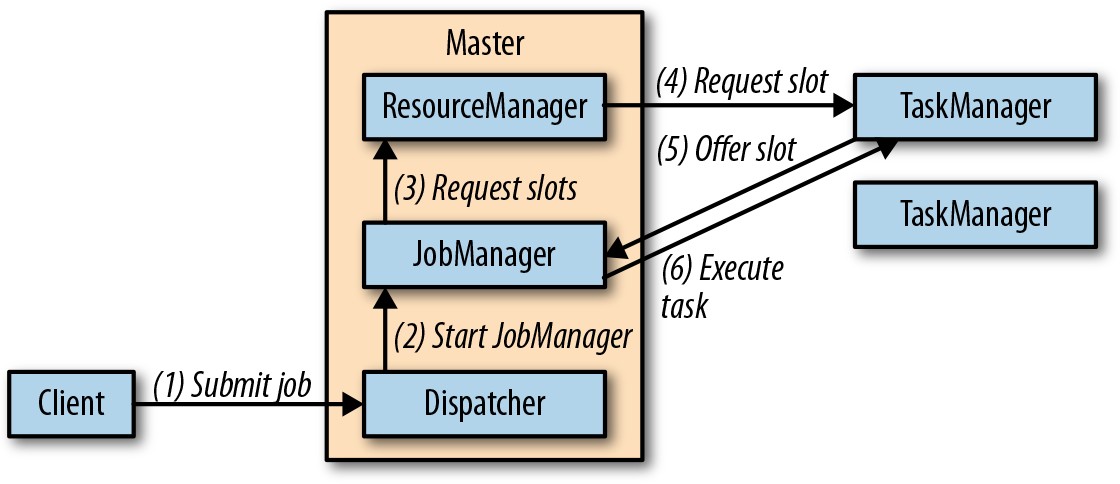
A standalone Flink cluster consists of at least one master process and at least one TaskManager process that run on one or more machines. All processes run as regular Java JVM processes. [Figure 9-1](#_bookmark1) shows a standalone Flink setup.

**221**



*Figure 9-1. Starting a standalone Flink cluster*

The master process runs a Dispatcher and a ResourceManager in separate threads. Once they start running, the TaskManagers register themselves at the ResourceMan‐ ager. [Figure 9-2](#_bookmark3) shows how a job is submitted to a standalone cluster.



*Figure 9-2. Submitting an application to a Flink standalone cluster*

A client submits a job to the Dispatcher, which internally starts a JobManager thread and provides the JobGraph for execution. The JobManager requests the necessary processing slots from the ResourceManager and deploys the job for execution once the requested slots have been received.

In a standalone deployment, the master and workers are not automatically restarted in the case of a failure. A job can recover from a worker failure if a sufficient number of processing slots is available. This can be ensured by running one or more standby workers. Job recovery from a master failure requires a highly available setup as dis‐ cussed later in this chapter.

In order to set up a standalone Flink cluster, download a binary distribution from the Apache Flink website and extract the tar archive with the command:

tar xfz ./flink-1.7.1-bin-scala\_2.12.tgz

The extracted directory includes a *./bin* folder with bash scripts[1](#_bookmark18) to start and stop Flink processes. The *./bin/start-cluster.sh* script starts a master process on the local machine and one or more TaskManagers on the local or remote machines.

Flink is preconfigured to run a local setup and start a single master and a single Task‐ Manager on the local machine. The start scripts must be able to start a Java process. If

the java binary is not on the PATH, the base folder of a Java installation can be speci‐ fied by exporting the JAVA\_HOME environment variable or setting the env.java.home parameter in *./conf/flink-conf.yaml*. A local Flink cluster is started by calling ./bin/ start-cluster.sh. You can visit Flink’s Web UI at *http://localhost:8081* and check the

number of connected TaskManagers and available slots.

In order to start a distributed Flink cluster that runs on multiple machines, you need to adjust the default configuration and complete a few more steps.

* The hostnames (or IP addresses) of all machines that should run TaskManagers need to be listed in the *./conf/slaves* file.
* The *start-cluster.sh* script requires a passwordless SSH configuration on all machines to be able to start the TaskManager processes.
* The Flink distribution folder must be located on all machines at the same path. A common approach is to mount a network-shared directory with the Flink distri‐ bution on each machine.
* The hostname (or IP address) of the machine that runs the master process needs to be configured in the *./conf/flink-conf.yaml* file with the config key jobman ager.rpc.address.

Once everything has been set up, you can start the Flink cluster by calling ./bin/ start-cluster.sh. The script will start a local JobManager and start one TaskMan‐ ager for each entry in the *slaves* file. You can check if the master process was started

and all TaskManager were successfully registered by accessing the Web UI on the machine that runs the master process. A local or distributed standalone cluster is

stopped by calling ./bin/stop-cluster.sh.

## Docker

Docker is a popular platform used to package and run applications in containers. Docker containers are run by the operating system kernel of the host system and are therefore more lightweight than virtual machines. Moreover, they are isolated and

1. In order to run Flink on Windows, you can use a provided bat script or you can use the regular bash scripts on the Windows Subsystem for Linux (WSL) or Cygwin. All scripts only work for local setups.

communicate only through well-defined channels. A container is started from an image that defines the software in the container.

Members of the Flink community configure and build Docker images for Apache Flink and upload them to Docker Hub, a public repository for Docker images.[2](#_bookmark19) The repository hosts Docker images for the most recent Flink versions.

Running Flink in Docker is an easy way to set up a Flink cluster on your local machine. For a local Docker setup you have to start two types of containers, a master container that runs the Dispatcher and ResourceManager, and one or more worker containers that run the TaskManagers. The containers work together like a stand‐ alone deployment (see [“Standalone Cluster”](#_bookmark17)). After starting, a TaskManager registers itself at the ResourceManager. When a job is submitted to the Dispatcher, it spawns a JobManager thread, which requests processing slots from the ResourceManager. The ResourceManager assigns TaskManagers to the JobManager, which deploys the job once all required resources are available.

Master and worker containers are started from the same Docker image with different parameters as shown in [Example 9-1](#_bookmark8).

*Example 9-1. Starting a master and a worker container in Docker*

// start master process

docker run -d --name flink-jobmanager **\**

-e JOB\_MANAGER\_RPC\_ADDRESS=jobmanager **\**

-p 8081:8081 flink:1.7 jobmanager

// start worker process (adjust the name to start more than one TM) docker run -d --name flink-taskmanager-1 **\**

--link flink-jobmanager:jobmanager **\**

-e JOB\_MANAGER\_RPC\_ADDRESS=jobmanager flink:1.7 taskmanager

Docker will download the requested image and its dependencies from Docker Hub and start the containers running Flink. The Docker internal hostname of the Job‐

Manager is passed to the containers via the JOB\_MANAGER\_RPC\_ADDRESS variable,

which is used in the entry point of the container to adjust Flink’s configuration.

The -p 8081:8081 parameter of the first command maps port 8081 of the master container to port 8081 of the host machine to make the Web UI accessible from the

host. You can access the Web UI by opening *http://localhost:8081* in your browser. The Web UI can be used to upload application JAR files and run the application. The port also exposes Flink’s REST API. Hence, you can also submit applications using

1. Flink Docker images are not part of the official Apache Flink release.

Flink’s CLI client at *./bin/flink*, manage running applications, or request information about the cluster or running applications.

Note that it is currently not possible to pass a custom configuration into the Flink Docker images. You need to build your own Docker image if you want to adjust some of the parameters. The build scripts of the available Docker Flink images are a good starting point for customized images.

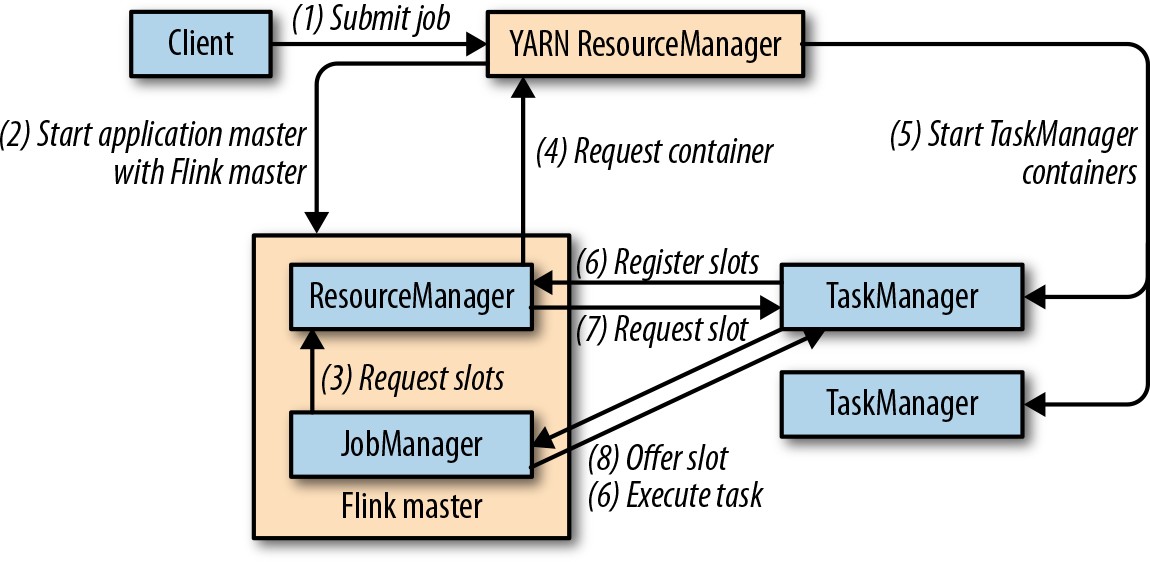
Instead of manually starting two (or more) containers, you can also create a Docker Compose configuration script, which automatically starts and configures a Flink clus‐ ter running in Docker containers and possibly other services such as ZooKeeper and Kafka. We will not go into the details of this mode, but among other things, a Docker Compose configuration needs to specify the network configuration so that Flink pro‐ cesses that run in isolated containers can communicate with each other. We refer you to Apache Flink’s documentation for details.

## Apache Hadoop YARN

YARN is the resource manager component of Apache Hadoop. It manages compute resources of a cluster environment—CPU and memory of the cluster’s machines— and provides them to applications that request resources. YARN grants resources as containers[3](#_bookmark19) that are distributed in the cluster and in which applications run their pro‐ cesses. Due to its origin in the Hadoop ecosystem, YARN is typically used by data processing frameworks.

Flink can run on YARN in two modes: the job mode and the session mode. In job mode, a Flink cluster is started to run a single job. Once the job terminates, the Flink cluster is stopped and all resources are returned. [Figure 9-3](#_bookmark1) shows how a Flink job is submitted to a YARN cluster.

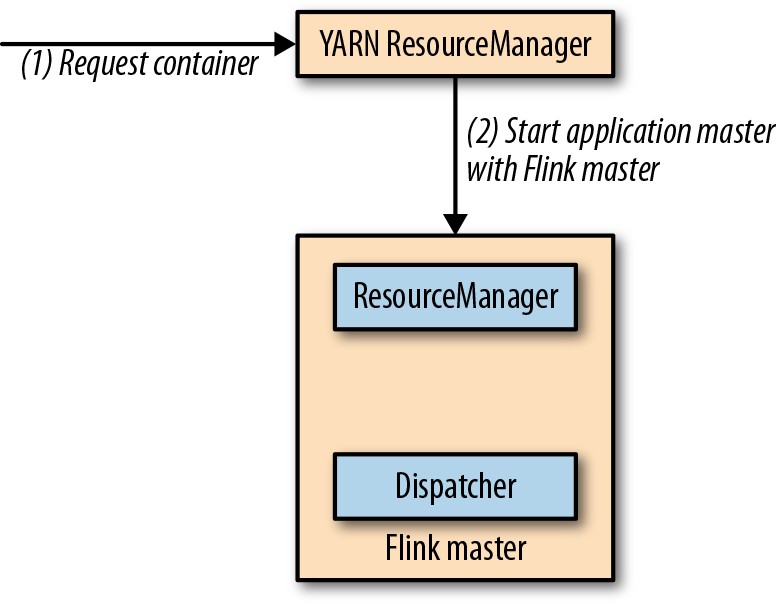
1. Note that the concept of a container in YARN is different from a container in Docker.



*Figure 9-3. Starting a Flink cluster on YARN in job mode*

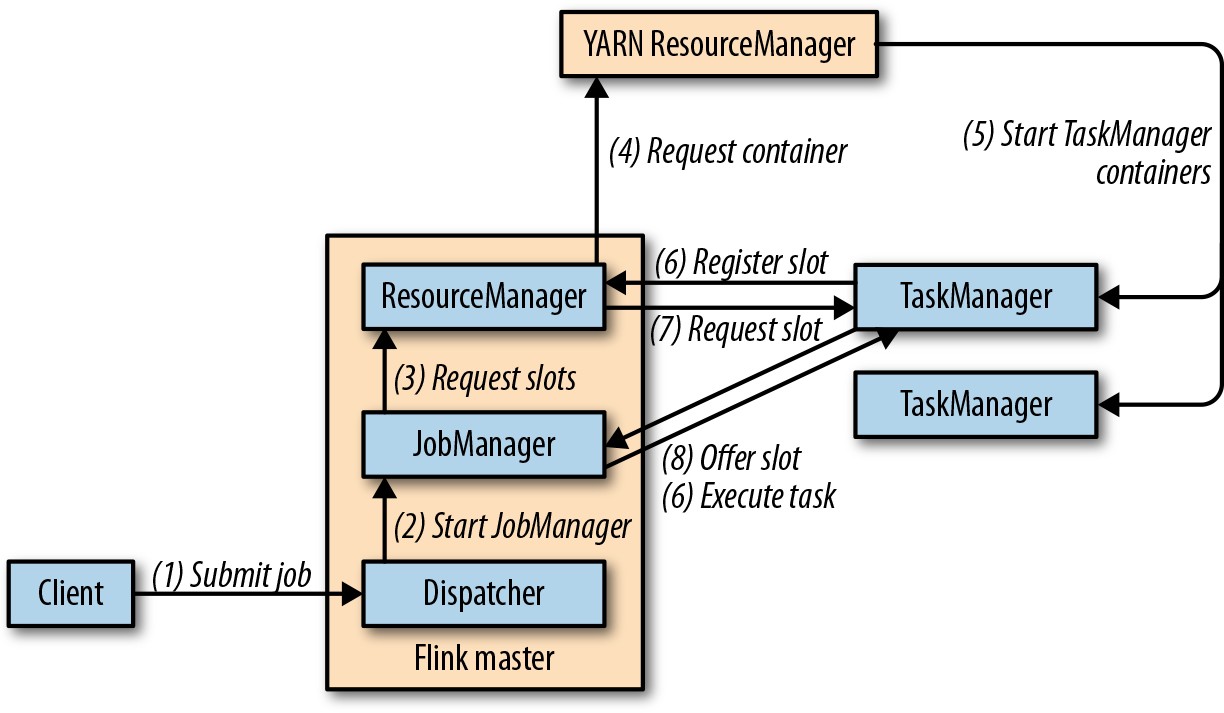
When the client submits a job for execution, it connects to the YARN ResourceMan‐ ager to start a new YARN application master process that consists of a JobManager thread and a ResourceManager. The JobManager requests the required slots from the ResourceManager to run the Flink job. Subsequently, Flink’s ResourceManager requests containers from YARN’s ResourceManager and starts TaskManager pro‐ cesses. Once started, the TaskManagers register their slots at Flink’s ResourceMan‐ ager, which provides them to the JobManager. Finally, the JobManager submits the job’s tasks to the TaskManagers for execution.

The session mode starts a long-running Flink cluster that can run multiple jobs and needs to be manually stopped. If started in session mode, Flink connects to YARN’s ResourceManager to start an application master that runs a Dispatcher thread and a Flink ResourceManager thread. [Figure 9-4](#_bookmark13) shows an idle Flink YARN session setup.



*Figure 9-4. Starting a Flink cluster on YARN in session mode*

When a job is submitted for execution, the Dispatcher starts a JobManager thread, which requests slots from Flink’s ResourceManager. If not enough slots are available, Flink’s ResourceManager requests additional containers from the YARN Resource‐ Manager to start TaskManager processes, which register themselves at the Flink ResourceManager. Once enough slots are available, Flink’s ResourceManager assigns them to the JobManager and the job execution starts. [Figure 9-5](#_bookmark4) shows how a job is executed in Flink’s YARN session mode.



*Figure 9-5. Submitting a job to a Flink YARN session cluster*

For both setups—job and session mode—failed TaskManagers will be automatically restarted by Flink’s ResourceManager. There are a few parameters in the *./conf/flink- conf.yaml* configuration file you can use to control Flink’s recovery behavior on YARN. For example, you can configure the maximum number of failed containers until an application is terminated. In order to recover from master failures, a highly available setup needs to be configured as described in a later section.

Regardless of whether you run Flink in job or session mode on YARN, it needs to have access to Hadoop dependencies in the correct version and the path to the Hadoop configuration. [“Integration with Hadoop Components” on page 236](#_bookmark5) describes the required configuration in detail.

Given a working and well-configured YARN and HDFS setup, a Flink job can be sub‐ mitted to be executed on YARN using Flink’s command-line client with the following command:

./bin/flink run -m yarn-cluster ./path/to/job.jar

The parameter -m defines the host to which the job is submitted. If set to the keyword

yarn-cluster, the client submits the job to the YARN cluster as identified by the

Hadoop configuration. Flink’s CLI client supports many more parameters, such as the ability to control the memory of TaskManager containers. Refer to the documenta‐ tion for a reference of available parameters. The Web UI of the started Flink cluster is served by the master process running on some node in the YARN cluster. You can access it via YARN’s Web UI, which provides a link on the Application Overview page under “Tracking URL: ApplicationMaster.”

A Flink YARN session is started with the ./bin/yarn-session.sh script, which also uses various parameters to control the size of containers, the name of the YARN

application, or provide dynamic properties. By default, the script prints the connec‐ tion information of the session cluster and does not return. The session is stopped and all resources are freed when the script is terminated. It is also possible to start a

YARN session in detached mode using the -d flag. A detached Flink session can be

terminated using YARN’s application utilities.

Once a Flink YARN session is running, you can submit jobs to the session with the command ./bin/flink run ./path/to/job.jar.



## Kubernetes

Note that you do not need to provide connection information, as Flink memorized the connection details of the Flink session run‐ ning on YARN. Similar to job mode, Flink’s Web UI is linked from the Application Overview of YARN’s Web UI.

Kubernetes is an open source platform that enables users to deploy and scale contain‐ erized applications in a distributed environment. Given a Kubernetes cluster and an application that is packaged into a container image, you can create a deployment of the application that tells Kubernetes how many instances of the application to start. Kubernetes will run the requested number of containers anywhere on its resources and restart them in the case of a failure. Kubernetes can also take care of opening net‐ work ports for internal and external communication and can provide services for process discovery and load balancing. Kubernetes runs on-premise, in cloud environ‐ ments, or on hybrid infrastructure.

Deploying data processing frameworks and applications on Kubernetes has become very popular. Apache Flink can be deployed on Kubernetes as well. Before diving into the details of how to set up Flink on Kubernetes, we need to briefly explain a few Kubernetes terms:

* + A *pod* is a container that is started and managed by Kubernetes.[4](#_bookmark19)

1. Kubernetes also supports pods consisting of multiple tightly linked containers.
   * A *deployment* defines a specific number of pods, or containers, to run. Kuber‐ netes ensures the requested number of pods is continuously running, and auto‐ matically restarts failed pods. Deployments can be scaled up or down.
   * Kubernetes may run a pod anywhere on its cluster. When a pod is restarted after a failure or when deployments are scaled up or down, the IP address can change. This is obviously a problem if pods need to communicate with each other. Kuber‐ netes provides services to overcome the issue. A *service* defines a policy for how a certain group of pods can be accessed. It takes care of updating the routing when a pod is started on a different node in the cluster.

### Running Kubernetes on a Local Machine

Kubernetes is designed for cluster operations. However, the Kuber‐ netes project provides Minikube, an environment to run a single- node Kubernetes cluster locally on a single machine for testing or daily development. We recommend setting up Minikube if you would like to try to run Flink on Kubernetes and do not have a Kubernetes cluster at hand.

In order to successfully run applications on a Flink cluster that is deployed on Minikube, you need to run the following command

before deploying Flink: minikube ssh 'sudo ip link set docker0 promisc on'.

A Flink setup for Kubernetes is defined with two deployments—one for the pod run‐ ning the master process and the other for the worker process pods. There is also a service that exposes the ports of the master pod to the worker pods. The two types of pods—master and worker—behave just like the processes of a standalone or Docker deployment we described before. The master deployment configuration is shown in [Example 9-2](#_bookmark14).

*Example 9-2. A Kubernetes deployment for a Flink master*

apiVersion: extensions/v1beta1 kind: Deployment

metadata:

name: flink-master spec:

replicas: 1 template: metadata:

labels:

app: flink component: master

spec:

containers:

* name: master image: flink:1.7 args:
  + jobmanager ports:
  + containerPort: 6123 name: rpc
  + containerPort: 6124 name: blob
  + containerPort: 6125 name: query
  + containerPort: 8081 name: ui

env:

* + name: JOB\_MANAGER\_RPC\_ADDRESS

value: flink-master

This deployment specifies that a single master container should be run (replicas: 1). The master container is started from the Flink 1.7 Docker image (image: flink: 1.7) with an argument that starts the master process (args: - jobmanager). More‐ over, the deployment configures which ports of the container to open for RPC com‐

munication, the blob manager (to exchange large files), the queryable state server, and the Web UI and REST interface. [Example 9-3](#_bookmark9) shows the deployment for worker pods.

*Example 9-3. A Kubernetes deployment for two Flink workers*

apiVersion: extensions/v1beta1 kind: Deployment

metadata:

name: flink-worker spec:

replicas: 2 template: metadata:

labels:

app: flink component: worker

spec:

containers:

* name: worker image: flink:1.7 args:
  + taskmanager ports:
  + containerPort: 6121 name: data
  + containerPort: 6122 name: rpc
  + containerPort: 6125 name: query

env:

* + name: JOB\_MANAGER\_RPC\_ADDRESS

value: flink-master

The worker deployment looks almost identical to the master deployment with a few differences. First of all, the worker deployment specifies two replicas, which means that two worker containers are started. The worker containers are based on the same

Flink Docker image but started with a different argument (args: -taskmanager).

Moreover, the deployment also opens a few ports and passes the service name of the

Flink master deployment so that the workers can access the master. The service defi‐ nition that exposes the master process and makes it accessible to the worker contain‐ ers is shown in [Example 9-4](#_bookmark6).

*Example 9-4. A Kubernetes service for the Flink master*

apiVersion: v1 kind: Service metadata:

name: flink-master spec:

ports:

* name: rpc port: 6123
* name: blob port: 6124
* name: query port: 6125
* name: ui port: 8081 selector: app: flink

component: master

You can create a Flink deployment for Kubernetes by storing each definition in a sep‐ arate file, such as *master-deployment.yaml*, *worker-deployment.yaml*, or *master- service.yaml*. The files are also provided in our repository. Once you have the

definition files, you can register them to Kubernetes using the kubectl command:

kubectl create -f master-deployment.yaml kubectl create -f worker-deployment.yaml kubectl create -f master-service.yaml

When running these commands, Kubernetes starts to deploy the requested contain‐ ers. You can show the status of all deployments by running the following command:

kubectl get deployments

When you create the deployments for the first time, it will take a while until the Flink container image is downloaded. Once all pods are up, you will have a Flink cluster running on Kubernetes. However, with the given configuration, Kubernetes does not export any port to external environments. Hence, you cannot access the master con‐ tainer to submit an application or access the Web UI. You first need to tell Kubernetes to create a port forwarding from the master container to your local machine. This is done by running the following command:

kubectl port-forward deployment/flink-master 8081:8081

When port forwarding is running, you can access the Web UI at *http://localhost:8081*.

Now you can upload and submit jobs to the Flink cluster running on Kubernetes. Moreover, you can submit applications using the Flink CLI client (*./bin/flink*) and access the REST interface to request information about the Flink cluster or manage running applications.

When a worker pod fails, Kubernetes will automatically restart the failed pod and the application will be recovered (given that checkpointing was activated and properly configured). In order to recover from a master pod failure, you need to configure a highly available setup.

You can shut down a Flink cluster running on Kubernetes by running the following commands:

kubectl delete -f master-deployment.yaml kubectl delete -f worker-deployment.yaml kubectl delete -f master-service.yaml

It is not possible to customize the configuration of the Flink deployment with the Flink Docker images we used in this section. You would need to build custom Docker images with an adjusted configuration. The build script for the provided image is a good starting point for a custom image.

# Highly Available Setups

Most streaming applications are ideally executed continuously with as little downtime as possible. Therefore, many applications must be able to automatically recover from failure of any process involved in the execution. While worker failures are handled by the ResourceManager, failures of the JobManager component require the configura‐ tion of a highly available (HA) setup.

Flink’s JobManager holds metadata about an application and its execution, such as the application JAR file, the JobGraph, and pointers to completed checkpoints. This information needs to be recovered in case of a master failure. Flink’s HA mode relies on Apache ZooKeeper, a service for distributed coordination and consistent storage, and a persistent remote storage, such as HDFS, NFS, or S3. The JobManager stores all

relevant data in the persistent storage and writes a pointer to the information—the storage path—to ZooKeeper. In case of a failure, a new JobManager looks up the pointer from ZooKeeper and loads the metadata from the persistent storage. We pre‐ sented the mode of operation and internals of Flink’s HA setup in more detail in [“Highly Available Setup” on page 42](#_bookmark1). In this section, we will configure this mode for different deployment options.

A Flink HA setup requires a running Apache ZooKeeper cluster and a persistent remote storage, such as HDFS, NFS, or S3. To help users start a ZooKeeper cluster quickly for testing purposes, Flink provides a helper script for bootstrapping. First, you need to configure the hosts and ports of all ZooKeeper processes involved in the

cluster by adjusting the *./conf/zoo.cfg* file. Once that is done, you can call ./bin/ start-zookeeper-quorum.sh to start a ZooKeeper process on each configured node.

### Do Not Use start-zookeeper-quorum.sh for Production Setups

You should not use Flink’s ZooKeeper script for production environments but instead carefully configure and deploy a Zoo‐ Keeper cluster yourself.

The Flink HA mode is configured in the *./conf/flink-conf.yaml* file by setting the parameters as shown in [Example 9-5](#_bookmark10).

*Example 9-5. Configuration of a HA Flink cluster*

# REQUIRED: enable HA mode via ZooKeeper high-availability: zookeeper

# REQUIRED: provide a list of all ZooKeeper servers of the quorum high-availability.zookeeper.quorum: address1:2181[,...],addressX:2181

# REQUIRED: set storage location for job metadata in remote storage high-availability.storageDir: hdfs:///flink/recovery

# RECOMMENDED: set the base path for all Flink clusters in ZooKeeper. # Isolates Flink from other frameworks using the ZooKeeper cluster. high-availability.zookeeper.path.root: /flink

**HA Standalone Setup**

A Flink standalone deployment does not rely on a resource provider, such as YARN or Kubernetes. All processes are manually started, and there is no component that monitors these processes and restarts them in case of a failure. Therefore, a stand‐ alone Flink cluster requires standby Dispatcher and TaskManager processes that can take over the work of failed processes.

Besides starting standby TaskManagers, a standalone deployment does not need addi‐ tional configuration to be able to recover from TaskManager failures. All started TaskManager processes register themselves at the active ResourceManager. An appli‐ cation can recover from a TaskManager failure as long as enough processing slots are on standby to compensate for the lost TaskManager. The ResourceManager hands out the previously idling processing slots and the application restarts.

If configured for HA, all Dispatchers of a standalone setup register at ZooKeeper. ZooKeeper elects a leader Dispatcher responsible for executing applications. When an application is submitted, the responsible Dispatcher starts a JobManager thread that stores its metadata in the configured persistent storage and a pointer in Zoo‐ Keeper as discussed before. If the master process that runs the active Dispatcher and JobManager fails, ZooKeeper elects a new Dispatcher as the leader. The leading Dis‐ patcher recovers the failed application by starting a new JobManager thread that looks up the metadata pointer in ZooKeeper and loads the metadata from the persis‐ tent storage.

In addition to the previously discussed configuration, an HA standalone deployment requires the following configuration changes. In *./conf/flink-conf.yaml* you need to set a cluster identifier for each running cluster. This is required if multiple Flink clusters rely on the same ZooKeeper instance for failure recovery:

# RECOMMENDED: set the path for the Flink cluster in ZooKeeper. # Isolates multiple Flink clusters from each other.

# The cluster id is required to look up the metadata of a failed cluster. high-availability.cluster-id: /cluster-1

If you have a ZooKeeper quorum running and Flink properly configured, you can use the regular *./bin/start-cluster.sh* script to start a HA standalone cluster by adding addi‐ tional hostnames and ports to the *./conf/masters* file.

## HA YARN Setup

YARN is a cluster resource and container manager. By default, it automatically restarts failed master and TaskManager containers. Hence, you do not need to run standby processes in a YARN setup to achieve HA.

Flink’s master process is started as a YARN ApplicationMaster.[5](#_bookmark19) YARN automatically restarts a failed ApplicationMaster but tracks and limits the number of restarts to pre‐ vent infinite recovery cycles. You need to configure the number of maximum Appli‐ cationManager restarts in the YARN configuration file *yarn-site.xml* as shown:

**<property>**

**<name>**yarn.resourcemanager.am.max-attempts**</name>**

1. ApplicationMaster is YARN’s master process of an application.

**<value>**4**</value>**

**<description>**

The maximum number of application master execution attempts. Default value is 2, i.e., an application is restarted at most once.

**</description>**

**</property>**

Moreover, you need to adjust Flink’s configuration file *./conf/flink-conf.yaml* and con‐ figure the number of application restart attempts:

# Restart an application at most 3 times (+ the initial start).

# Must be less or equal to the configured maximum number of attempts. yarn.application-attempts: 4

YARN only counts the number of restarts due to application failures—restarts due to preemption, hardware failures, or reboots are not taken into account for the number of application attempts. If you run Hadoop YARN version 2.6 or later, Flink automat‐ ically configures an attempt failure’s validity interval. This parameter specifies that an application is only completely canceled if it exceeds its restart attempts within the val‐ idity interval, meaning attempts that predate the interval are not taken into account.

Flink configures the interval to the same value as the akka.ask.timeout parameter

in *./conf/flink-conf.yaml* with a default value of 10 seconds.

Given a running ZooKeeper cluster and properly configured YARN and Flink setups, you can start a Flink cluster in job mode or session mode as if HA were not enabled—

by using ./bin/flink run -m yarn-cluster and ./bin/yarn-session.sh.

Note that you must configure different cluster IDs for all Flink ses‐ sion clusters that connect to the same ZooKeeper cluster. When starting a Flink cluster in job mode, the cluster ID is automatically set to the ID of the started application and is therefore unique.

## HA Kubernetes Setup

When running Flink on Kubernetes with a master deployment and a worker deploy‐ ment as described in [“Kubernetes” on page 228](#_bookmark11), Kubernetes will automatically restart failed containers to ensure the right number of pods is up and running. This is suffi‐ cient to recover from worker failures, which are handled by the ResourceManager. However, recovering from master failures requires additional configuration as dis‐ cussed before.

In order to enable Flink’s HA mode, you need to adjust Flink’s configuration and pro‐ vide information such as the hostnames of the ZooKeeper quorum nodes, a path to a persistent storage, and a cluster ID for Flink. All of these parameters need to be added to Flink’s configuration file (*./conf/flink-conf.yaml*).

### Custom Configuration in Flink Images

Unfortunately, the Flink Docker image we used in the Docker and Kubernetes examples before does not support setting custom con‐ figuration parameters. Hence, the image cannot be used to set up a HA Flink cluster on Kubernetes. Instead, you need to build a cus‐ tom image that either “hardcodes” the required parameters or is flexible enough to adjust the configuration dynamically through parameters or environment variables. The standard Flink Docker images are a good starting point to customize your own Flink images.

# Integration with Hadoop Components

Apache Flink can be easily integrated with Hadoop YARN and HDFS and other com‐ ponents of the Hadoop ecosystem, such as HBase. In all of these cases, Flink requires Hadoop dependencies on its classpath.

There are three ways to provide Flink with Hadoop dependencies:

1. Use a binary distribution of Flink that was built for a particular Hadoop version. Flink provides builds for the most commonly used vanilla Hadoop versions.
2. Build Flink for a specific Hadoop version. This is useful if none of Flink’s binary distributions works with the Hadoop version deployed in your environment; for example, if you run a patched Hadoop version or a Hadoop version of a distribu‐ tor, such as Cloudera, Hortonworks, or MapR.

In order to build Flink for a specific Hadoop version, you need Flink’s source code, which can be obtained by downloading the source distribution from the website or cloning a stable release branch from the project’s Git repository, a Java JDK of at least version 8, and Apache Maven 3.2. Enter the base folder of Flink’s source code and run one of the commands in the following:

// build Flink **for** a specific official Hadoop version mvn clean install -DskipTests -Dhadoop.version=2.6.1

// build Flink **for** a Hadoop version of a distributor mvn clean install -DskipTests -Pvendor-repos **\**

-Dhadoop.version=2.6.1-cdh5.0.0

The completed build is located in the *./build-target* folder.

1. Use the Hadoop-free distribution of Flink and manually configure the classpath for Hadoop’s dependencies. This approach is useful if none of the provided builds work for your setup. The classpath of the Hadoop dependencies must be declared

in the HADOOP\_CLASSPATH environment variable. If the variable is not configured,

you can automatically set it with the following command if the hadoop command is accessible: export HADOOP\_CLASSPATH=`hadoop classpath`.

The classpath option of the hadoop command prints its configured classpath.

In addition to configuring the Hadoop dependencies, you need to provide the loca‐ tion of Hadoop’s configuration directory. This should be done by exporting either the

HADOOP\_CONF\_DIR (preferred) or HADOOP\_CONF\_PATH environment variable. Once

Flink knows about Hadoop’s configuration, it can connect to YARN’s ResourceMan‐

ager and HDFS.

# Filesystem Configuration

Apache Flink uses filesystems for various tasks. Applications can read their input from and write their results to files (see [“Filesystem Source Connector”](#_bookmark1)), application checkpoints and metadata are persisted in remote filesystems for recovery (see [“Checkpoints, Savepoints, and State Recovery”](#_bookmark15)), and some internal components lev‐ erage filesystems to distribute data to tasks, such as application JAR files.

Flink supports a wide variety of filesystems. Since Flink is a distributed system and runs processes on cluster or cloud environments, filesystems typically need to be globally accessible. Hence, Hadoop HDFS, S3, and NFS are commonly used filesys‐ tems.

Similar to other data processing systems, Flink looks at the URI scheme of a path to identify the filesystem the path refers to. For example, *file:///home/user/data.txt* points to a file in the local filesystem and *hdfs:///namenode:50010/home/user/data.txt* to a file in the specified HDFS cluster.

A filesystem is represented in Flink by an implementation of the org.apache.flink.core.fs.FileSystem class. A FileSystem class implements file‐ system operations, such as reading from and writing to files, creating directories or

files, and listing the contents of a directory. A Flink process (JobManager or Task‐ Manager) instantiates one FileSystem object for each configured filesystem and shares it across all local tasks to guarantee that configured constraints such as limits

on the number of open connections are enforced.

Flink provides implementations for the most commonly used filesystems as follows:

*Local filesystem*

Flink has built-in support for local filesystems, including locally mounted net‐ work filesystems, such as NFS or SAN, and does not require additional configu‐ ration. Local filesystems are referenced by the *file://* URI scheme.

*Hadoop HDFS*

Flink’s connector for HDFS is always in the classpath of Flink. However, it requires Hadoop dependencies on the classpath in order to work. [“Integration](#_bookmark5) [with Hadoop Components” on page 236](#_bookmark5) explains how to ensure Hadoop depen‐ dencies are loaded. HDFS paths are prefixed with the *hdfs://* scheme.

*Amazon S3*

Flink provides two alternative filesystem connectors to connect to S3, which are based on Apache Hadoop and Presto. Both connectors are fully self-contained and do not expose any dependencies. To install either of both connectors, move the respective JAR file from the *./opt* folder into the *./lib* folder. The Flink docu‐ mentation provides more details on the configuration of S3 filesystems. S3 paths are specified with the *s3://* scheme.

*OpenStack Swift FS*

Flink provides a connector to Swift FS, which is based on Apache Hadoop. The connector is fully self-contained and does not expose any dependencies. It is installed by moving the swift-connector JAR file from the *./opt* to the *./lib* folder. Swift FS paths are identified by the *swift://* scheme.

For filesystems for which Flink does not provide a dedicated connector, Flink can del‐ egate to the Hadoop filesystem connector if it is correctly configured. This is why Flink is able to support all HCFSs.

Flink provides a few configuration options in *./conf/flink-conf.yaml* to specify a default filesystem and limit the number of filesystem connections. You can specify a default filesystem scheme (*fs.default-scheme*) that is automatically added as a prefix if a path does not provide a scheme. If you, for example, specify *fs.default-scheme: hdfs://nnode1:9000*, the path */result* will be extended to *hdfs://nnode1:9000/result*.

You can limit the number of connections that read from (input) and write to (output) a filesystem. The configuration can be defined per URI scheme. The relevant configu‐ ration keys are:

fs.<scheme>.limit.total: (number, 0/-1 mean no limit) fs.<scheme>.limit.input: (number, 0/-1 mean no limit) fs.<scheme>.limit.output: (number, 0/-1 mean no limit) fs.<scheme>.limit.timeout: (milliseconds, 0 means infinite) fs.<scheme>.limit.stream-timeout: (milliseconds, 0 means infinite)

The number of connections are tracked per TaskManager process and path authority

—*hdfs://nnode1:50010* and *hdfs://nnode2:50010* are separately tracked. The connec‐ tion limits can be configured either separately for input and output connections or as the total number of connections. When the filesystem reaches its connection limit and tries to open a new connection, it will block and wait for another connection to close. The timeout parameters define how long to wait until a connection request fails

(fs.<scheme>.limit.timeout) and how long to wait until an idle connection is closed (fs.<scheme>.limit.stream-timeout).

You can also provide a custom filesystem connector. Take a look at the [Flink docu‐](http://bit.ly/2HxINvi) [mentation](http://bit.ly/2HxINvi) to learn how to implement and register a custom filesystem.

# System Configuration

Apache Flink offers many parameters to configure its behavior and tweak its perfor‐ mance. All parameters can be defined in the *./conf/flink-conf.yaml* file, which is organized as a flat YAML file of key-value pairs. The configuration file is read by dif‐ ferent components, such as the start scripts, the master and worker JVM processes, and the CLI client. For example, the start scripts, such as *./bin/start-cluster.sh*, parse the configuration file to extract JVM parameters and heap size settings, and the CLI client (*./bin/flink*) extracts the connection information to access the master process. Changes in the configuration file are not effective until Flink is restarted.

To improve the out-of-the-box experience, Flink is preconfigured for a local setup. You need to adjust the configuration to successfully run Flink in distributed environ‐ ments. In this section, we discuss different aspects that typically need to be config‐ ured when setting up a Flink cluster. We refer you to the [official documentation](http://bit.ly/2O4T7fv) for a comprehensive list and detailed descriptions of all parameters.

## Java and Classloading

By default, Flink starts JVM processes using the Java executable linked by the PATH environment variable. If Java is not on the PATH or if you want to use a different Java version you can specify the root folder of a Java installation via the JAVA\_HOME envi‐ ronment variable or the env.java.home key in the configuration file. Flink’s JVM

processes can be started with custom Java options—for example, to fine-tune the garbage collector or to enable remote debugging, with the keys env.java.opts, env.java.opts.jobmanager, and env.java.opts.taskmanager.

Classloading issues are not uncommon when running jobs with external dependen‐ cies. In order to execute a Flink application, all classes in the application’s JAR file must be loaded by a classloader. Flink registers the classes of every job into a separate user-code classloader to ensure the dependencies of the job do not infer with Flink’s runtime dependencies or the dependencies of other jobs. User-code class loaders are disposed of when the corresponding job terminates. Flink’s system class loader loads all JAR files in the *./lib* folder and the user-code classloaders are derived from the sys‐ tem classloader.

By default, Flink looks up user-code classes first in the child (user-code) classloader and then in parent (system) classloader to prevent version clashes in case a job uses

the same dependency as Flink. However, you can also invert the lookup order with the classloader.resolve-order configuration key.



## CPU

Note that some classes are always resolved first in the parent class‐ loader (classloader.parent-first-patterns.default). You can extend the list by providing a whitelist of classname patterns that

are first resolved from the parent classloader (classloader.parent-first-patterns.additional).

Flink does not actively limit the amount of CPU resources it consumes. However, it uses processing slots (see [“Task Execution” on page 40](#_bookmark12) for a detailed discussion) to control the number of tasks that can be assigned to a worker process (TaskManager). A TaskManager provides a certain number of slots that are registered at and governed by the ResourceManager. A JobManager requests one or more slots to execute an application. Each slot can process one slice of an application, one parallel task of every operator of the application. Hence, the JobManager needs to acquire at least as many slots as the application’s maximum operator parallelism.[6](#_bookmark19) Tasks are executed as threads within the worker (TaskManager) process and take as much CPU resources as they need.

The number of slots a TaskManager offers is controlled with the taskmanager.num berOfTaskSlots key in the configuration file. The default is one slot per TaskMan‐ ager. The number of slots usually only needs to be configured for standalone setups as

running Flink on a cluster resource manager (YARN, Kubernetes, Mesos) makes it easy to spin up multiple TaskManagers (each with one slot) per compute node.

## Main Memory and Network Buffers

Flink’s master and worker processes have different memory requirements. A master process mainly manages compute resources (ResourceManager) and coordinates the execution of applications (JobManager), while a worker process takes care of the heavy lifting and processes potentially large amounts of data.

Usually, the master process has moderate memory requirements. By default, it is started with 1 GB JVM heap memory. If the master process needs to manage several applications or an application with many operators, you might need to increase the

JVM heap size with the jobmanager.heap.size configuration key.

1. It is possible to assign operators to different slot-sharing groups and thus assign their tasks to distinct slots.

Configuring the memory of a worker process is a bit more involved because there are multiple components that allocate different types of memory. The most important

parameter is the size of the JVM heap memory, which is set with the key taskman ager.heap.size. The heap memory is used for all objects, including the TaskMan‐ ager runtime, operators and functions of the application, and in-flight data. The state

of an application that uses the in-memory or filesystem state backend is also stored on the JVM. Note that a single task can potentially consume the whole heap memory of the JVM that it is running on. Flink does not guarantee or grant heap memory per task or slot. Configurations with a single slot per TaskManager have better resource isolation and can prevent a misbehaving application from interfering with unrelated applications. If you run applications with many dependencies, the JVM’s nonheap memory can also grow significantly because it stores all TaskManager and user-code classes.

In addition to the JVM, there are two other major memory consumers, Flink’s net‐ work stack and RocksDB, when it is used as a state backend. Flink’s network stack is based on the Netty library, which allocates network buffers from native (off-heap) memory. Flink requires a sufficient number of network buffers to be able to ship records from one worker process to the other. The number of buffers depends on the total number of network connections between operator tasks. For two operators con‐ nected by a partitioning or broadcasting connection the number of network buffers depends on the product of the sending and receiving operator parallelism. For appli‐ cations with several partitioning steps, this quadratic dependency can quickly sum up to a significant amount of memory that is required for network transfer.

Flink’s default configuration is only suitable for a smaller scale distributed setup and needs to be adjusted for more serious scale. If the number of buffers is not appropri‐

ately configured, a job submission will fail with a java.io.IOException: Insufficient number of network buffers. In this case, you should provide more memory to the network stack.

The amount of memory assigned for network buffers is configured with the taskman ager.network.memory.fraction key, which determines the fraction of the JVM size allocated for network buffers. By default, 10% of the JVM heap size is used. Since the

buffers are allocated as off-heap memory, the JVM heap is reduced by that amount. The configuration key taskmanager.memory.segment-size determines the size of a network buffer, which is 32 KB by default. Reducing the size of a network buffer

increases the number of buffers but can reduce the efficiency of the network stack. You can also specify a minimum (taskmanager.network.memory.min) and a maxi‐ mum (taskmanager.network.memory.max) amount of memory that is used for net‐

work buffers (by default 64 MB and 1 GB, respectively) to set absolute limits for the

relative configuration value.

RocksDB is another memory consumer that needs to be taken into consideration when configuring the memory of a worker process. Unfortunately, figuring out the memory consumption of RocksDB is not straightforward because it depends on the number of keyed states in an application. Flink creates a separate (embedded) RocksDB instance for each task of a keyed operator. Within each instance, every dis‐ tinct state of the operator is stored in a separate column family (or table). With the default configuration, each column family requires about 200 MB to 240 MB of off- heap memory. You can adjust RocksDB’s configuration and tweak its performance with many parameters.

When configuring the memory setting of a TaskManager, you should size the JVM heap memory so there is enough memory left for the JVM nonheap memory (classes and metadata) and RocksDB if it is configured as a state backend. Network memory is automatically subtracted from the configured JVM heap size. Keep in mind that some resource managers, such as YARN, will immediately kill a container if it exceeds its memory budget.

## Disk Storage

A Flink worker process stores data on the local filesystem for multiple reasons, including receiving application JAR files, writing log files, and maintaining applica‐

tion state if the RocksDB state backend is configured. With the io.tmp.dirs configu‐

ration key, you can specify one or more directories (separated by colons) that are

used to store data in the local filesystem. By default, data is written to the default tem‐ porary directory as determined by the Java system property java.io.tmpdir, or /tmp on Linux and MacOS. The io.tmp.dirs parameter is used as the default value for the

local storage path of most components of Flink. However, these paths can also be

individually configured.

### Ensure Temporary Directories Are Not Automatically Cleaned

Some Linux distribution periodically clean the temporary direc‐ tory /tmp. Make sure to disable this behavior or configure a differ‐ ent directory if you plan to run continuous Flink applications. Otherwise job recovery might miss metadata that was stored in the temporary directory and fail.

The blob.storage.directory key configures the local storage directory of the blob server, which is used to exchange larger files such as the application JAR files. The env.log.dir key configures the directory into which a TaskManager writes its log files (by default, the *./log* directory in the Flink setup). Finally, the RocksDB state

backend maintains application state in the local filesystem. The directory is config‐

ured using the state.backend.rocksdb.localdir key. If the storage directory is not explicitly configured, RocksDB uses the value of the io.tmp.dirs parameter.

## Checkpointing and State Backends

Flink offers a few options to configure how state backends checkpoint their state. All parameters can be explicitly specified within the code of an application as described in [“Tuning Checkpointing and Recovery”](#_bookmark16). However, you can also provide default set‐ tings for a Flink cluster through Flink’s configuration file, which are applied if job- specific options are not declared.

An important choice that affects the performance of an application is the state back‐ end that maintains its state. You can define the default state backend of a cluster with

the state.backend key. Moreover, you can enable asynchronous checkpointing (state.backend.async) and incremental checkpointing (state.backend.incremen tal). Some backends do not support all options and might ignore them. You can also configure the root directories at the remote storage to which checkpoints (state.checkpoints.dir) and savepoints (state.savepoints.dir) are written.

Some checkpointing options are backend specific. For the RocksDB state backend you can define one or more paths at which RocksDB stores its local files (state.back end.rocksdb.localdir) and whether timer state is stored on the heap (default) or in RocksDB (state.backend.rocksdb.timer-service.factory).

Finally, you can enable and configure local recovery for a Flink cluster by default.[7](#_bookmark19) To enable local recovery, set the parameter state.backend.local-recovery to true. The storage location of the local state copy can be specified as well (taskmanager.state.local.root-dirs).

## Security

Data processing frameworks are sensitive components of a company’s IT infrastruc‐ ture and need to be secured against unauthorized use and access to data. Apache Flink supports Kerberos authentication and can be configured to encrypt all network communication with SSL.

Flink features Kerberos integration with Hadoop and its components (YARN, HDFS, HBase), ZooKeeper, and Kafka. You can enable and configure the Kerberos support for each service separately. Flink supports two authentication modes—keytabs and Hadoop delegation tokens. Keytabs are the preferred approach because tokens expire after some time, which can cause problems for long-running stream processing appli‐ cations. Note that the credentials are tied to a Flink cluster and not to a running job;

1. See [“Configuring Recovery”](#_bookmark7) for details on this feature.

all applications that run on the same cluster use the same authentication token. If you need to work with different credentials, you should start a new cluster. Consult the [Flink documentation](http://bit.ly/2Fc4i3e) for detailed instructions on enabling and configuring Kerberos authentication.

Flink supports mutual authentication of communication partners and encryption of network communication with SSL for internal and external communication. For internal communication (RPC calls, data transfer, and blob service communication to distribute libraries or other artifacts) all Flink processes (Dispatcher, ResourceMan‐ ager, JobManager, and TaskManager) perform mutual authentication— senders and receivers validate each other via an SSL certificate. The certificate acts as a shared secret and can be embedded into containers or attached to a YARN setup.

All external communication with Flink services—submitting and controlling applica‐ tions and accessing the REST interface—happens over REST/HTTP endpoints.[8](#_bookmark19) You can enable SSL encryption for these connections as well. Mutual authentication can also be enabled. However, the recommended approach is setting up and configuring a dedicated proxy service that controls access to the REST endpoint. The reason is that proxy services offer more authentication and configuration options than Flink. Encryption and authentication for communication to queryable state is not sup‐ ported yet.

By default, SSL authentication and encryption is not enabled. Since the setup requires several steps, such as generating certificates, setting up TrustStores and KeyStores, and configuring cipher suites, we refer you to the official [Flink documentation](http://bit.ly/2Fc4i3e). The documentation also includes how-tos and tips for different environments, such as standalone clusters, Kubernetes, and YARN.

# Summary

In this chapter we discussed how Flink is set up in different environments and how to configure HA setups. We explained how to enable support for various filesystems and how to integrate them with Hadoop and its components. Finally, we discussed the most important configuration options. We did not provide a comprehensive configu‐ ration guide; instead, we refer you to [the official documentation of Apache Flink](http://bit.ly/2O5ikGP) for a complete list and detailed descriptions of all configuration options.

1. [Chapter 10](#_bookmark2) discusses job submission and the REST interface.