# **Spark Standalone Mode**

* [Installing Spark Standalone to a Cluster](http://spark.apache.org/docs/latest/spark-standalone.html" \l "installing-spark-standalone-to-a-cluster)
* [Starting a Cluster Manually](http://spark.apache.org/docs/latest/spark-standalone.html" \l "starting-a-cluster-manually)
* [Cluster Launch Scripts](http://spark.apache.org/docs/latest/spark-standalone.html" \l "cluster-launch-scripts)
* [Connecting an Application to the Cluster](http://spark.apache.org/docs/latest/spark-standalone.html" \l "connecting-an-application-to-the-cluster)
* [Launching Spark Applications](http://spark.apache.org/docs/latest/spark-standalone.html" \l "launching-spark-applications)
* [Resource Scheduling](http://spark.apache.org/docs/latest/spark-standalone.html" \l "resource-scheduling)
* [Monitoring and Logging](http://spark.apache.org/docs/latest/spark-standalone.html" \l "monitoring-and-logging)
* [Running Alongside Hadoop](http://spark.apache.org/docs/latest/spark-standalone.html" \l "running-alongside-hadoop)
* [Configuring Ports for Network Security](http://spark.apache.org/docs/latest/spark-standalone.html" \l "configuring-ports-for-network-security)
* [High Availability](http://spark.apache.org/docs/latest/spark-standalone.html" \l "high-availability)
  + [Standby Masters with ZooKeeper](http://spark.apache.org/docs/latest/spark-standalone.html" \l "standby-masters-with-zookeeper)
  + [Single-Node Recovery with Local File System](http://spark.apache.org/docs/latest/spark-standalone.html" \l "single-node-recovery-with-local-file-system)

In addition to running on the Mesos or YARN cluster managers, Spark also provides a simple standalone deploy mode. You can launch a standalone cluster either manually, by starting a master and workers by hand, or use our provided [launch scripts](http://spark.apache.org/docs/latest/spark-standalone.html" \l "cluster-launch-scripts). It is also possible to run these daemons on a single machine for testing.

# **Installing Spark Standalone to a Cluster**

To install Spark Standalone mode, you simply place a compiled version of Spark on each node on the cluster. You can obtain pre-built versions of Spark with each release or [build it yourself](http://spark.apache.org/docs/latest/building-spark.html).

# **Starting a Cluster Manually**

You can start a standalone master server by executing:

./sbin/start-master.sh

Once started, the master will print out a spark://HOST:PORT URL for itself, which you can use to connect workers to it, or pass as the “master” argument to SparkContext. You can also find this URL on the master’s web UI, which is [http://localhost:8080](http://localhost:8080/) by default.

Similarly, you can start one or more workers and connect them to the master via:

./sbin/start-slave.sh <master-spark-URL>

Once you have started a worker, look at the master’s web UI ([http://localhost:8080](http://localhost:8080/) by default). You should see the new node listed there, along with its number of CPUs and memory (minus one gigabyte left for the OS).

Finally, the following configuration options can be passed to the master and worker:

|  |  |
| --- | --- |
| **Argument** | **Meaning** |
| -h HOST, --host HOST | Hostname to listen on |
| -i HOST, --ip HOST | Hostname to listen on (deprecated, use -h or --host) |
| -p PORT, --port PORT | Port for service to listen on (default: 7077 for master, random for worker) |
| --webui-port PORT | Port for web UI (default: 8080 for master, 8081 for worker) |
| -c CORES, --cores CORES | Total CPU cores to allow Spark applications to use on the machine (default: all available); only on worker |
| -m MEM, --memory MEM | Total amount of memory to allow Spark applications to use on the machine, in a format like 1000M or 2G (default: your machine's total RAM minus 1 GB); only on worker |
| -d DIR, --work-dir DIR | Directory to use for scratch space and job output logs (default: SPARK\_HOME/work); only on worker |
| --properties-file FILE | Path to a custom Spark properties file to load (default: conf/spark-defaults.conf) |

# **Cluster Launch Scripts**

To launch a Spark standalone cluster with the launch scripts, you should create a file called conf/slaves in your Spark directory, which must contain the hostnames of all the machines where you intend to start Spark workers, one per line. If conf/slaves does not exist, the launch scripts defaults to a single machine (localhost), which is useful for testing. Note, the master machine accesses each of the worker machines via ssh. By default, ssh is run in parallel and requires password-less (using a private key) access to be setup. If you do not have a password-less setup, you can set the environment variable SPARK\_SSH\_FOREGROUND and serially provide a password for each worker.

Once you’ve set up this file, you can launch or stop your cluster with the following shell scripts, based on Hadoop’s deploy scripts, and available in SPARK\_HOME/sbin:

* sbin/start-master.sh - Starts a master instance on the machine the script is executed on.
* sbin/start-slaves.sh - Starts a slave instance on each machine specified in the conf/slaves file.
* sbin/start-slave.sh - Starts a slave instance on the machine the script is executed on.
* sbin/start-all.sh - Starts both a master and a number of slaves as described above.
* sbin/stop-master.sh - Stops the master that was started via the bin/start-master.sh script.
* sbin/stop-slaves.sh - Stops all slave instances on the machines specified in the conf/slaves file.
* sbin/stop-all.sh - Stops both the master and the slaves as described above.

Note that these scripts must be executed on the machine you want to run the Spark master on, not your local machine.

You can optionally configure the cluster further by setting environment variables in conf/spark-env.sh. Create this file by starting with theconf/spark-env.sh.template, and *copy it to all your worker machines* for the settings to take effect. The following settings are available:

|  |  |
| --- | --- |
| **Environment Variable** | **Meaning** |
| SPARK\_MASTER\_IP | Bind the master to a specific IP address, for example a public one. |
| SPARK\_MASTER\_PORT | Start the master on a different port (default: 7077). |
| SPARK\_MASTER\_WEBUI\_PORT | Port for the master web UI (default: 8080). |
| SPARK\_MASTER\_OPTS | Configuration properties that apply only to the master in the form "-Dx=y" (default: none). See below for a list of possible options. |
| SPARK\_LOCAL\_DIRS | Directory to use for "scratch" space in Spark, including map output files and RDDs that get stored on disk. This should be on a fast, local disk in your system. It can also be a comma-separated list of multiple directories on different disks. |
| SPARK\_WORKER\_CORES | Total number of cores to allow Spark applications to use on the machine (default: all available cores). |
| SPARK\_WORKER\_MEMORY | Total amount of memory to allow Spark applications to use on the machine, e.g. 1000m, 2g (default: total memory minus 1 GB); note that each application's *individual* memory is configured using its spark.executor.memoryproperty. |
| SPARK\_WORKER\_PORT | Start the Spark worker on a specific port (default: random). |
| SPARK\_WORKER\_WEBUI\_PORT | Port for the worker web UI (default: 8081). |
| SPARK\_WORKER\_INSTANCES | Number of worker instances to run on each machine (default: 1). You can make this more than 1 if you have have very large machines and would like multiple Spark worker processes. If you do set this, make sure to also set SPARK\_WORKER\_CORES explicitly to limit the cores per worker, or else each worker will try to use all the cores. |
| SPARK\_WORKER\_DIR | Directory to run applications in, which will include both logs and scratch space (default: SPARK\_HOME/work). |
| SPARK\_WORKER\_OPTS | Configuration properties that apply only to the worker in the form "-Dx=y" (default: none). See below for a list of possible options. |
| SPARK\_DAEMON\_MEMORY | Memory to allocate to the Spark master and worker daemons themselves (default: 1g). |
| SPARK\_DAEMON\_JAVA\_OPTS | JVM options for the Spark master and worker daemons themselves in the form "-Dx=y" (default: none). |
| SPARK\_PUBLIC\_DNS | The public DNS name of the Spark master and workers (default: none). |

****Note:**** The launch scripts do not currently support Windows. To run a Spark cluster on Windows, start the master and workers by hand.

SPARK\_MASTER\_OPTS supports the following system properties:

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Default** | **Meaning** |
| spark.deploy.retainedApplications | 200 | The maximum number of completed applications to display. Older applications will be dropped from the UI to maintain this limit. |
| spark.deploy.retainedDrivers | 200 | The maximum number of completed drivers to display. Older drivers will be dropped from the UI to maintain this limit. |
| spark.deploy.spreadOut | true | Whether the standalone cluster manager should spread applications out across nodes or try to consolidate them onto as few nodes as possible. Spreading out is usually better for data locality in HDFS, but consolidating is more efficient for compute-intensive workloads. |
| spark.deploy.defaultCores | (infinite) | Default number of cores to give to applications in Spark's standalone mode if they don't setspark.cores.max. If not set, applications always get all available cores unless they configurespark.cores.max themselves. Set this lower on a shared cluster to prevent users from grabbing the whole cluster by default. |
| spark.worker.timeout | 60 | Number of seconds after which the standalone deploy master considers a worker lost if it receives no heartbeats. |

SPARK\_WORKER\_OPTS supports the following system properties:

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Default** | **Meaning** |
| spark.worker.cleanup.enabled | false | Enable periodic cleanup of worker / application directories. Note that this only affects standalone mode, as YARN works differently. Only the directories of stopped applications are cleaned up. |
| spark.worker.cleanup.interval | 1800 (30 minutes) | Controls the interval, in seconds, at which the worker cleans up old application work dirs on the local machine. |
| spark.worker.cleanup.appDataTtl | 7 \* 24 \* 3600 (7 days) | The number of seconds to retain application work directories on each worker. This is a Time To Live and should depend on the amount of available disk space you have. Application logs and jars are downloaded to each application work dir. Over time, the work dirs can quickly fill up disk space, especially if you run jobs very frequently. |

# **Connecting an Application to the Cluster**

To run an application on the Spark cluster, simply pass the spark://IP:PORT URL of the master as to the [SparkContext constructor](http://spark.apache.org/docs/latest/programming-guide.html" \l "initializing-spark).

To run an interactive Spark shell against the cluster, run the following command:

./bin/spark-shell --master spark://IP:PORT

You can also pass an option --total-executor-cores <numCores> to control the number of cores that spark-shell uses on the cluster.

# **Launching Spark Applications**

The [spark-submit script](http://spark.apache.org/docs/latest/submitting-applications.html) provides the most straightforward way to submit a compiled Spark application to the cluster. For standalone clusters, Spark currently supports two deploy modes. In client mode, the driver is launched in the same process as the client that submits the application. In cluster mode, however, the driver is launched from one of the Worker processes inside the cluster, and the client process exits as soon as it fulfills its responsibility of submitting the application without waiting for the application to finish.

If your application is launched through Spark submit, then the application jar is automatically distributed to all worker nodes. For any additional jars that your application depends on, you should specify them through the --jars flag using comma as a delimiter (e.g. --jars jar1,jar2). To control the application’s configuration or execution environment, see [Spark Configuration](http://spark.apache.org/docs/latest/configuration.html).

Additionally, standalone cluster mode supports restarting your application automatically if it exited with non-zero exit code. To use this feature, you may pass in the --supervise flag to spark-submit when launching your application. Then, if you wish to kill an application that is failing repeatedly, you may do so through:

./bin/spark-class org.apache.spark.deploy.Client kill <master url> <driver ID>

You can find the driver ID through the standalone Master web UI at http://<master url>:8080.

# **Resource Scheduling**

The standalone cluster mode currently only supports a simple FIFO scheduler across applications. However, to allow multiple concurrent users, you can control the maximum number of resources each application will use. By default, it will acquire *all* cores in the cluster, which only makes sense if you just run one application at a time. You can cap the number of cores by setting spark.cores.max in your [SparkConf](http://spark.apache.org/docs/latest/configuration.html" \l "spark-properties). For example:

**val** conf **=** **new** **SparkConf**()

.setMaster(...)

.setAppName(...)

.set("spark.cores.max", "10")**val** sc **=** **new** **SparkContext**(conf)

In addition, you can configure spark.deploy.defaultCores on the cluster master process to change the default for applications that don’t setspark.cores.max to something less than infinite. Do this by adding the following to conf/spark-env.sh:

export SPARK\_MASTER\_OPTS="-Dspark.deploy.defaultCores=<value>"

This is useful on shared clusters where users might not have configured a maximum number of cores individually.

# **Monitoring and Logging**

Spark’s standalone mode offers a web-based user interface to monitor the cluster. The master and each worker has its own web UI that shows cluster and job statistics. By default you can access the web UI for the master at port 8080. The port can be changed either in the configuration file or via command-line options.

In addition, detailed log output for each job is also written to the work directory of each slave node (SPARK\_HOME/work by default). You will see two files for each job, stdout and stderr, with all output it wrote to its console.

# **Running Alongside Hadoop**

You can run Spark alongside your existing Hadoop cluster by just launching it as a separate service on the same machines. To access Hadoop data from Spark, just use a hdfs:// URL (typically hdfs://<namenode>:9000/path, but you can find the right URL on your Hadoop Namenode’s web UI). Alternatively, you can set up a separate cluster for Spark, and still have it access HDFS over the network; this will be slower than disk-local access, but may not be a concern if you are still running in the same local area network (e.g. you place a few Spark machines on each rack that you have Hadoop on).

# **Configuring Ports for Network Security**

Spark makes heavy use of the network, and some environments have strict requirements for using tight firewall settings. For a complete list of ports to configure, see the [security page](http://spark.apache.org/docs/latest/security.html" \l "configuring-ports-for-network-security).

# **High Availability**

By default, standalone scheduling clusters are resilient to Worker failures (insofar as Spark itself is resilient to losing work by moving it to other workers). However, the scheduler uses a Master to make scheduling decisions, and this (by default) creates a single point of failure: if the Master crashes, no new applications can be created. In order to circumvent this, we have two high availability schemes, detailed below.

## **Standby Masters with ZooKeeper**

****Overview****

Utilizing ZooKeeper to provide leader election and some state storage, you can launch multiple Masters in your cluster connected to the same ZooKeeper instance. One will be elected “leader” and the others will remain in standby mode. If the current leader dies, another Master will be elected, recover the old Master’s state, and then resume scheduling. The entire recovery process (from the time the the first leader goes down) should take between 1 and 2 minutes. Note that this delay only affects scheduling *new* applications – applications that were already running during Master failover are unaffected.

Learn more about getting started with ZooKeeper [here](http://zookeeper.apache.org/doc/trunk/zookeeperStarted.html).

****Configuration****

In order to enable this recovery mode, you can set SPARK\_DAEMON\_JAVA\_OPTS in spark-env using this configuration:

|  |  |
| --- | --- |
| **System property** | **Meaning** |
| spark.deploy.recoveryMode | Set to ZOOKEEPER to enable standby Master recovery mode (default: NONE). |
| spark.deploy.zookeeper.url | The ZooKeeper cluster url (e.g., 192.168.1.100:2181,192.168.1.101:2181). |
| spark.deploy.zookeeper.dir | The directory in ZooKeeper to store recovery state (default: /spark). |

Possible gotcha: If you have multiple Masters in your cluster but fail to correctly configure the Masters to use ZooKeeper, the Masters will fail to discover each other and think they’re all leaders. This will not lead to a healthy cluster state (as all Masters will schedule independently).

****Details****

After you have a ZooKeeper cluster set up, enabling high availability is straightforward. Simply start multiple Master processes on different nodes with the same ZooKeeper configuration (ZooKeeper URL and directory). Masters can be added and removed at any time.

In order to schedule new applications or add Workers to the cluster, they need to know the IP address of the current leader. This can be accomplished by simply passing in a list of Masters where you used to pass in a single one. For example, you might start your SparkContext pointing to spark://host1:port1,host2:port2. This would cause your SparkContext to try registering with both Masters – if host1 goes down, this configuration would still be correct as we’d find the new leader, host2.

There’s an important distinction to be made between “registering with a Master” and normal operation. When starting up, an application or Worker needs to be able to find and register with the current lead Master. Once it successfully registers, though, it is “in the system” (i.e., stored in ZooKeeper). If failover occurs, the new leader will contact all previously registered applications and Workers to inform them of the change in leadership, so they need not even have known of the existence of the new Master at startup.

Due to this property, new Masters can be created at any time, and the only thing you need to worry about is that *new* applications and Workers can find it to register with in case it becomes the leader. Once registered, you’re taken care of.

## **Single-Node Recovery with Local File System**

****Overview****

ZooKeeper is the best way to go for production-level high availability, but if you just want to be able to restart the Master if it goes down, FILESYSTEM mode can take care of it. When applications and Workers register, they have enough state written to the provided directory so that they can be recovered upon a restart of the Master process.

****Configuration****

In order to enable this recovery mode, you can set SPARK\_DAEMON\_JAVA\_OPTS in spark-env using this configuration:

|  |  |
| --- | --- |
| **System property** | **Meaning** |
| spark.deploy.recoveryMode | Set to FILESYSTEM to enable single-node recovery mode (default: NONE). |
| spark.deploy.recoveryDirectory | The directory in which Spark will store recovery state, accessible from the Master's perspective. |

****Details****

* This solution can be used in tandem with a process monitor/manager like [monit](http://mmonit.com/monit/), or just to enable manual recovery via restart.
* While filesystem recovery seems straightforwardly better than not doing any recovery at all, this mode may be suboptimal for certain development or experimental purposes. In particular, killing a master via stop-master.sh does not clean up its recovery state, so whenever you start a new Master, it will enter recovery mode. This could increase the startup time by up to 1 minute if it needs to wait for all previously-registered Workers/clients to timeout.
* While it’s not officially supported, you could mount an NFS directory as the recovery directory. If the original Master node dies completely, you could then start a Master on a different node, which would correctly recover all previously registered Workers/applications (equivalent to ZooKeeper recovery). Future applications will have to be able to find the new Master, however, in order to register.