[](http://spark.apache.org/docs/latest/index.html)2.2.0

* [Overview](http://spark.apache.org/docs/latest/index.html)
* [Programming Guides](http://spark.apache.org/docs/latest/running-on-yarn.html)
* [API Docs](http://spark.apache.org/docs/latest/running-on-yarn.html)
* [Deploying](http://spark.apache.org/docs/latest/running-on-yarn.html)
* [More](http://spark.apache.org/docs/latest/api.html)

**Running Spark on YARN**

Support for running on [YARN (Hadoop NextGen)](http://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html) was added to Spark in version 0.6.0, and improved in subsequent releases.

**Launching Spark on YARN**

Ensure that HADOOP\_CONF\_DIR or YARN\_CONF\_DIR points to the directory which contains the (client side) configuration files for the Hadoop cluster. These configs are used to write to HDFS and connect to the YARN ResourceManager. The configuration contained in this directory will be distributed to the YARN cluster so that all containers used by the application use the same configuration. If the configuration references Java system properties or environment variables not managed by YARN, they should also be set in the Spark application’s configuration (driver, executors, and the AM when running in client mode).

There are two deploy modes that can be used to launch Spark applications on YARN. In cluster mode, the Spark driver runs inside an application master process which is managed by YARN on the cluster, and the client can go away after initiating the application. In client mode, the driver runs in the client process, and the application master is only used for requesting resources from YARN.

Unlike [Spark standalone](http://spark.apache.org/docs/latest/spark-standalone.html) and [Mesos](http://spark.apache.org/docs/latest/running-on-mesos.html) modes, in which the master’s address is specified in the --master parameter, in YARN mode the ResourceManager’s address is picked up from the Hadoop configuration. Thus, the --master parameter is yarn.

To launch a Spark application in cluster mode:

$ ./bin/spark-submit --class path.to.your.Class --master yarn --deploy-mode cluster [options] <app jar> [app options]

For example:

$ ./bin/spark-submit --class org.apache.spark.examples.SparkPi \

--master yarn \

--deploy-mode cluster \

--driver-memory 4g \

--executor-memory 2g \

--executor-cores 1 \

--queue thequeue \

lib/spark-examples\*.jar \

10

The above starts a YARN client program which starts the default Application Master. Then SparkPi will be run as a child thread of Application Master. The client will periodically poll the Application Master for status updates and display them in the console. The client will exit once your application has finished running. Refer to the “Debugging your Application” section below for how to see driver and executor logs.

To launch a Spark application in client mode, do the same, but replace cluster with client. The following shows how you can run spark-shell in client mode:

$ ./bin/spark-shell --master yarn --deploy-mode client

**Adding Other JARs**

In cluster mode, the driver runs on a different machine than the client, so SparkContext.addJar won’t work out of the box with files that are local to the client. To make files on the client available to SparkContext.addJar, include them with the --jars option in the launch command.

$ ./bin/spark-submit --class my.main.Class \

--master yarn \

--deploy-mode cluster \

--jars my-other-jar.jar,my-other-other-jar.jar \

my-main-jar.jar \

app\_arg1 app\_arg2

**Preparations**

Running Spark on YARN requires a binary distribution of Spark which is built with YARN support. Binary distributions can be downloaded from the [downloads page](http://spark.apache.org/downloads.html) of the project website. To build Spark yourself, refer to [Building Spark](http://spark.apache.org/docs/latest/building-spark.html).

To make Spark runtime jars accessible from YARN side, you can specify spark.yarn.archive or spark.yarn.jars. For details please refer to [Spark Properties](http://spark.apache.org/docs/latest/running-on-yarn.html#spark-properties). If neither spark.yarn.archive nor spark.yarn.jars is specified, Spark will create a zip file with all jars under $SPARK\_HOME/jars and upload it to the distributed cache.

**Configuration**

Most of the configs are the same for Spark on YARN as for other deployment modes. See the [configuration page](http://spark.apache.org/docs/latest/configuration.html) for more information on those. These are configs that are specific to Spark on YARN.

**Debugging your Application**

In YARN terminology, executors and application masters run inside “containers”. YARN has two modes for handling container logs after an application has completed. If log aggregation is turned on (with the yarn.log-aggregation-enable config), container logs are copied to HDFS and deleted on the local machine. These logs can be viewed from anywhere on the cluster with the yarn logs command.

yarn logs -applicationId <app ID>

will print out the contents of all log files from all containers from the given application. You can also view the container log files directly in HDFS using the HDFS shell or API. The directory where they are located can be found by looking at your YARN configs (yarn.nodemanager.remote-app-log-dir and yarn.nodemanager.remote-app-log-dir-suffix). The logs are also available on the Spark Web UI under the Executors Tab. You need to have both the Spark history server and the MapReduce history server running and configure yarn.log.server.url in yarn-site.xml properly. The log URL on the Spark history server UI will redirect you to the MapReduce history server to show the aggregated logs.

When log aggregation isn’t turned on, logs are retained locally on each machine under YARN\_APP\_LOGS\_DIR, which is usually configured to /tmp/logs or $HADOOP\_HOME/logs/userlogs depending on the Hadoop version and installation. Viewing logs for a container requires going to the host that contains them and looking in this directory. Subdirectories organize log files by application ID and container ID. The logs are also available on the Spark Web UI under the Executors Tab and doesn’t require running the MapReduce history server.

To review per-container launch environment, increase yarn.nodemanager.delete.debug-delay-sec to a large value (e.g. 36000), and then access the application cache through yarn.nodemanager.local-dirs on the nodes on which containers are launched. This directory contains the launch script, JARs, and all environment variables used for launching each container. This process is useful for debugging classpath problems in particular. (Note that enabling this requires admin privileges on cluster settings and a restart of all node managers. Thus, this is not applicable to hosted clusters).

To use a custom log4j configuration for the application master or executors, here are the options:

* upload a custom log4j.properties using spark-submit, by adding it to the --files list of files to be uploaded with the application.
* add -Dlog4j.configuration=<location of configuration file> to spark.driver.extraJavaOptions (for the driver) or spark.executor.extraJavaOptions (for executors). Note that if using a file, the file: protocol should be explicitly provided, and the file needs to exist locally on all the nodes.
* update the $SPARK\_CONF\_DIR/log4j.properties file and it will be automatically uploaded along with the other configurations. Note that other 2 options has higher priority than this option if multiple options are specified.

Note that for the first option, both executors and the application master will share the same log4j configuration, which may cause issues when they run on the same node (e.g. trying to write to the same log file).

If you need a reference to the proper location to put log files in the YARN so that YARN can properly display and aggregate them, use spark.yarn.app.container.log.dir in your log4j.properties. For example, log4j.appender.file\_appender.File=${spark.yarn.app.container.log.dir}/spark.log. For streaming applications, configuring RollingFileAppender and setting file location to YARN’s log directory will avoid disk overflow caused by large log files, and logs can be accessed using YARN’s log utility.

To use a custom metrics.properties for the application master and executors, update the $SPARK\_CONF\_DIR/metrics.properties file. It will automatically be uploaded with other configurations, so you don’t need to specify it manually with --files.

**Spark Properties**

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Default** | **Meaning** |
| spark.yarn.am.memory | 512m | Amount of memory to use for the YARN Application Master in client mode, in the same format as JVM memory strings (e.g. 512m, 2g). In cluster mode, use spark.driver.memory instead.  Use lower-case suffixes, e.g. k, m, g, t, and p, for kibi-, mebi-, gibi-, tebi-, and pebibytes, respectively. |
| spark.yarn.am.cores | 1 | Number of cores to use for the YARN Application Master in client mode. In cluster mode, use spark.driver.cores instead. |
| spark.yarn.am.waitTime | 100s | In cluster mode, time for the YARN Application Master to wait for the SparkContext to be initialized. In client mode, time for the YARN Application Master to wait for the driver to connect to it. |
| spark.yarn.submit.file.replication | The default HDFS replication (usually 3) | HDFS replication level for the files uploaded into HDFS for the application. These include things like the Spark jar, the app jar, and any distributed cache files/archives. |
| spark.yarn.stagingDir | Current user's home directory in the filesystem | Staging directory used while submitting applications. |
| spark.yarn.preserve.staging.files | false | Set to true to preserve the staged files (Spark jar, app jar, distributed cache files) at the end of the job rather than delete them. |
| spark.yarn.scheduler.heartbeat.interval-ms | 3000 | The interval in ms in which the Spark application master heartbeats into the YARN ResourceManager. The value is capped at half the value of YARN's configuration for the expiry interval, i.e. yarn.am.liveness-monitor.expiry-interval-ms. |
| spark.yarn.scheduler.initial-allocation.interval | 200ms | The initial interval in which the Spark application master eagerly heartbeats to the YARN ResourceManager when there are pending container allocation requests. It should be no larger than spark.yarn.scheduler.heartbeat.interval-ms. The allocation interval will doubled on successive eager heartbeats if pending containers still exist, until spark.yarn.scheduler.heartbeat.interval-ms is reached. |
| spark.yarn.max.executor.failures | numExecutors \* 2, with minimum of 3 | The maximum number of executor failures before failing the application. |
| spark.yarn.historyServer.address | (none) | The address of the Spark history server, e.g. host.com:18080. The address should not contain a scheme (http://). Defaults to not being set since the history server is an optional service. This address is given to the YARN ResourceManager when the Spark application finishes to link the application from the ResourceManager UI to the Spark history server UI. For this property, YARN properties can be used as variables, and these are substituted by Spark at runtime. For example, if the Spark history server runs on the same node as the YARN ResourceManager, it can be set to ${hadoopconf-yarn.resourcemanager.hostname}:18080. |
| spark.yarn.dist.archives | (none) | Comma separated list of archives to be extracted into the working directory of each executor. |
| spark.yarn.dist.files | (none) | Comma-separated list of files to be placed in the working directory of each executor. |
| spark.yarn.dist.jars | (none) | Comma-separated list of jars to be placed in the working directory of each executor. |
| spark.executor.instances | 2 | The number of executors for static allocation. With spark.dynamicAllocation.enabled, the initial set of executors will be at least this large. |
| spark.yarn.executor.memoryOverhead | executorMemory \* 0.10, with minimum of 384 | The amount of off-heap memory (in megabytes) to be allocated per executor. This is memory that accounts for things like VM overheads, interned strings, other native overheads, etc. This tends to grow with the executor size (typically 6-10%). |
| spark.yarn.driver.memoryOverhead | driverMemory \* 0.10, with minimum of 384 | The amount of off-heap memory (in megabytes) to be allocated per driver in cluster mode. This is memory that accounts for things like VM overheads, interned strings, other native overheads, etc. This tends to grow with the container size (typically 6-10%). |
| spark.yarn.am.memoryOverhead | AM memory \* 0.10, with minimum of 384 | Same as spark.yarn.driver.memoryOverhead, but for the YARN Application Master in client mode. |
| spark.yarn.am.port | (random) | Port for the YARN Application Master to listen on. In YARN client mode, this is used to communicate between the Spark driver running on a gateway and the YARN Application Master running on YARN. In YARN cluster mode, this is used for the dynamic executor feature, where it handles the kill from the scheduler backend. |
| spark.yarn.queue | default | The name of the YARN queue to which the application is submitted. |
| spark.yarn.jars | (none) | List of libraries containing Spark code to distribute to YARN containers. By default, Spark on YARN will use Spark jars installed locally, but the Spark jars can also be in a world-readable location on HDFS. This allows YARN to cache it on nodes so that it doesn't need to be distributed each time an application runs. To point to jars on HDFS, for example, set this configuration to hdfs:///some/path. Globs are allowed. |
| spark.yarn.archive | (none) | An archive containing needed Spark jars for distribution to the YARN cache. If set, this configuration replaces spark.yarn.jars and the archive is used in all the application's containers. The archive should contain jar files in its root directory. Like with the previous option, the archive can also be hosted on HDFS to speed up file distribution. |
| spark.yarn.access.hadoopFileSystems | (none) | A comma-separated list of secure Hadoop filesystems your Spark application is going to access. For example, spark.yarn.access.hadoopFileSystems=hdfs://nn1.com:8032,hdfs://nn2.com:8032, webhdfs://nn3.com:50070. The Spark application must have access to the filesystems listed and Kerberos must be properly configured to be able to access them (either in the same realm or in a trusted realm). Spark acquires security tokens for each of the filesystems so that the Spark application can access those remote Hadoop filesystems. spark.yarn.access.namenodes is deprecated, please use this instead. |
| spark.yarn.appMasterEnv.[EnvironmentVariableName] | (none) | Add the environment variable specified by EnvironmentVariableName to the Application Master process launched on YARN. The user can specify multiple of these and to set multiple environment variables. In cluster mode this controls the environment of the Spark driver and in client mode it only controls the environment of the executor launcher. |
| spark.yarn.containerLauncherMaxThreads | 25 | The maximum number of threads to use in the YARN Application Master for launching executor containers. |
| spark.yarn.am.extraJavaOptions | (none) | A string of extra JVM options to pass to the YARN Application Master in client mode. In cluster mode, use spark.driver.extraJavaOptions instead. Note that it is illegal to set maximum heap size (-Xmx) settings with this option. Maximum heap size settings can be set with spark.yarn.am.memory |
| spark.yarn.am.extraLibraryPath | (none) | Set a special library path to use when launching the YARN Application Master in client mode. |
| spark.yarn.maxAppAttempts | yarn.resourcemanager.am.max-attempts in YARN | The maximum number of attempts that will be made to submit the application. It should be no larger than the global number of max attempts in the YARN configuration. |
| spark.yarn.am.attemptFailuresValidityInterval | (none) | Defines the validity interval for AM failure tracking. If the AM has been running for at least the defined interval, the AM failure count will be reset. This feature is not enabled if not configured. |
| spark.yarn.executor.failuresValidityInterval | (none) | Defines the validity interval for executor failure tracking. Executor failures which are older than the validity interval will be ignored. |
| spark.yarn.submit.waitAppCompletion | true | In YARN cluster mode, controls whether the client waits to exit until the application completes. If set to true, the client process will stay alive reporting the application's status. Otherwise, the client process will exit after submission. |
| spark.yarn.am.nodeLabelExpression | (none) | A YARN node label expression that restricts the set of nodes AM will be scheduled on. Only versions of YARN greater than or equal to 2.6 support node label expressions, so when running against earlier versions, this property will be ignored. |
| spark.yarn.executor.nodeLabelExpression | (none) | A YARN node label expression that restricts the set of nodes executors will be scheduled on. Only versions of YARN greater than or equal to 2.6 support node label expressions, so when running against earlier versions, this property will be ignored. |
| spark.yarn.tags | (none) | Comma-separated list of strings to pass through as YARN application tags appearing in YARN ApplicationReports, which can be used for filtering when querying YARN apps. |
| spark.yarn.keytab | (none) | The full path to the file that contains the keytab for the principal specified above. This keytab will be copied to the node running the YARN Application Master via the Secure Distributed Cache, for renewing the login tickets and the delegation tokens periodically. (Works also with the "local" master) |
| spark.yarn.principal | (none) | Principal to be used to login to KDC, while running on secure HDFS. (Works also with the "local" master) |
| spark.yarn.config.gatewayPath | (none) | A path that is valid on the gateway host (the host where a Spark application is started) but may differ for paths for the same resource in other nodes in the cluster. Coupled with spark.yarn.config.replacementPath, this is used to support clusters with heterogeneous configurations, so that Spark can correctly launch remote processes.  The replacement path normally will contain a reference to some environment variable exported by YARN (and, thus, visible to Spark containers).  For example, if the gateway node has Hadoop libraries installed on /disk1/hadoop, and the location of the Hadoop install is exported by YARN as the HADOOP\_HOME environment variable, setting this value to /disk1/hadoop and the replacement path to $HADOOP\_HOME will make sure that paths used to launch remote processes properly reference the local YARN configuration. |
| spark.yarn.config.replacementPath | (none) | See spark.yarn.config.gatewayPath. |
| spark.yarn.security.credentials.${service}.enabled | true | Controls whether to obtain credentials for services when security is enabled. By default, credentials for all supported services are retrieved when those services are configured, but it's possible to disable that behavior if it somehow conflicts with the application being run. For further details please see [Running in a Secure Cluster](running-on-yarn.html#running-in-a-secure-cluster) |
| spark.yarn.rolledLog.includePattern | (none) | Java Regex to filter the log files which match the defined include pattern and those log files will be aggregated in a rolling fashion. This will be used with YARN's rolling log aggregation, to enable this feature in YARN side yarn.nodemanager.log-aggregation.roll-monitoring-interval-seconds should be configured in yarn-site.xml. This feature can only be used with Hadoop 2.6.4+. The Spark log4j appender needs be changed to use FileAppender or another appender that can handle the files being removed while its running. Based on the file name configured in the log4j configuration (like spark.log), the user should set the regex (spark\*) to include all the log files that need to be aggregated. |
| spark.yarn.rolledLog.excludePattern | (none) | Java Regex to filter the log files which match the defined exclude pattern and those log files will not be aggregated in a rolling fashion. If the log file name matches both the include and the exclude pattern, this file will be excluded eventually. |

**Important notes**

* Whether core requests are honored in scheduling decisions depends on which scheduler is in use and how it is configured.
* In cluster mode, the local directories used by the Spark executors and the Spark driver will be the local directories configured for YARN (Hadoop YARN config yarn.nodemanager.local-dirs). If the user specifies spark.local.dir, it will be ignored. In client mode, the Spark executors will use the local directories configured for YARN while the Spark driver will use those defined in spark.local.dir. This is because the Spark driver does not run on the YARN cluster in client mode, only the Spark executors do.
* The --files and --archives options support specifying file names with the # similar to Hadoop. For example you can specify: --files localtest.txt#appSees.txt and this will upload the file you have locally named localtest.txt into HDFS but this will be linked to by the name appSees.txt, and your application should use the name as appSees.txt to reference it when running on YARN.
* The --jars option allows the SparkContext.addJar function to work if you are using it with local files and running in cluster mode. It does not need to be used if you are using it with HDFS, HTTP, HTTPS, or FTP files.

**Running in a Secure Cluster**

As covered in [security](http://spark.apache.org/docs/latest/security.html), Kerberos is used in a secure Hadoop cluster to authenticate principals associated with services and clients. This allows clients to make requests of these authenticated services; the services to grant rights to the authenticated principals.

Hadoop services issue *hadoop tokens* to grant access to the services and data. Clients must first acquire tokens for the services they will access and pass them along with their application as it is launched in the YARN cluster.

For a Spark application to interact with any of the Hadoop filesystem (for example hdfs, webhdfs, etc), HBase and Hive, it must acquire the relevant tokens using the Kerberos credentials of the user launching the application —that is, the principal whose identity will become that of the launched Spark application.

This is normally done at launch time: in a secure cluster Spark will automatically obtain a token for the cluster’s default Hadoop filesystem, and potentially for HBase and Hive.

An HBase token will be obtained if HBase is in on classpath, the HBase configuration declares the application is secure (i.e. hbase-site.xml sets hbase.security.authentication to kerberos), and spark.yarn.security.credentials.hbase.enabled is not set to false.

Similarly, a Hive token will be obtained if Hive is on the classpath, its configuration includes a URI of the metadata store in "hive.metastore.uris, and spark.yarn.security.credentials.hive.enabled is not set to false.

If an application needs to interact with other secure Hadoop filesystems, then the tokens needed to access these clusters must be explicitly requested at launch time. This is done by listing them in the spark.yarn.access.hadoopFileSystems property.

spark.yarn.access.hadoopFileSystems hdfs://ireland.example.org:8020/,webhdfs://frankfurt.example.org:50070/

Spark supports integrating with other security-aware services through Java Services mechanism (see java.util.ServiceLoader). To do that, implementations of org.apache.spark.deploy.yarn.security.ServiceCredentialProvider should be available to Spark by listing their names in the corresponding file in the jar’s META-INF/services directory. These plug-ins can be disabled by setting spark.yarn.security.credentials.{service}.enabled to false, where {service} is the name of credential provider.

**Configuring the External Shuffle Service**

To start the Spark Shuffle Service on each NodeManager in your YARN cluster, follow these instructions:

1. Build Spark with the [YARN profile](http://spark.apache.org/docs/latest/building-spark.html). Skip this step if you are using a pre-packaged distribution.
2. Locate the spark-<version>-yarn-shuffle.jar. This should be under $SPARK\_HOME/common/network-yarn/target/scala-<version> if you are building Spark yourself, and under yarn if you are using a distribution.
3. Add this jar to the classpath of all NodeManagers in your cluster.
4. In the yarn-site.xml on each node, add spark\_shuffle to yarn.nodemanager.aux-services, then set yarn.nodemanager.aux-services.spark\_shuffle.class to org.apache.spark.network.yarn.YarnShuffleService.
5. Increase NodeManager's heap size by setting YARN\_HEAPSIZE (1000 by default) in etc/hadoop/yarn-env.sh to avoid garbage collection issues during shuffle.
6. Restart all NodeManagers in your cluster.

The following extra configuration options are available when the shuffle service is running on YARN:

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Default** | **Meaning** |
| spark.yarn.shuffle.stopOnFailure | false | Whether to stop the NodeManager when there's a failure in the Spark Shuffle Service's initialization. This prevents application failures caused by running containers on NodeManagers where the Spark Shuffle Service is not running. |

**Launching your application with Apache Oozie**

Apache Oozie can launch Spark applications as part of a workflow. In a secure cluster, the launched application will need the relevant tokens to access the cluster’s services. If Spark is launched with a keytab, this is automatic. However, if Spark is to be launched without a keytab, the responsibility for setting up security must be handed over to Oozie.

The details of configuring Oozie for secure clusters and obtaining credentials for a job can be found on the [Oozie web site](http://oozie.apache.org/) in the “Authentication” section of the specific release’s documentation.

For Spark applications, the Oozie workflow must be set up for Oozie to request all tokens which the application needs, including:

* The YARN resource manager.
* The local Hadoop filesystem.
* Any remote Hadoop filesystems used as a source or destination of I/O.
* Hive —if used.
* HBase —if used.
* The YARN timeline server, if the application interacts with this.

To avoid Spark attempting —and then failing— to obtain Hive, HBase and remote HDFS tokens, the Spark configuration must be set to disable token collection for the services.

The Spark configuration must include the lines:

spark.yarn.security.credentials.hive.enabled false spark.yarn.security.credentials.hbase.enabled false

The configuration option spark.yarn.access.hadoopFileSystems must be unset.

**Troubleshooting Kerberos**

Debugging Hadoop/Kerberos problems can be “difficult”. One useful technique is to enable extra logging of Kerberos operations in Hadoop by setting the HADOOP\_JAAS\_DEBUG environment variable.

bash export HADOOP\_JAAS\_DEBUG=true

The JDK classes can be configured to enable extra logging of their Kerberos and SPNEGO/REST authentication via the system properties sun.security.krb5.debug and sun.security.spnego.debug=true

-Dsun.security.krb5.debug=true -Dsun.security.spnego.debug=true

All these options can be enabled in the Application Master:

spark.yarn.appMasterEnv.HADOOP\_JAAS\_DEBUG true spark.yarn.am.extraJavaOptions -Dsun.security.krb5.debug=true -Dsun.security.spnego.debug=true

Finally, if the log level for org.apache.spark.deploy.yarn.Client is set to DEBUG, the log will include a list of all tokens obtained, and their expiry details

**Using the Spark History Server to replace the Spark Web UI**

It is possible to use the Spark History Server application page as the tracking URL for running applications when the application UI is disabled. This may be desirable on secure clusters, or to reduce the memory usage of the Spark driver. To set up tracking through the Spark History Server, do the following:

* On the application side, set spark.yarn.historyServer.allowTracking=true in Spark’s configuration. This will tell Spark to use the history server’s URL as the tracking URL if the application’s UI is disabled.
* On the Spark History Server, add org.apache.spark.deploy.yarn.YarnProxyRedirectFilter to the list of filters in the spark.ui.filters configuration.

Be aware that the history server information may not be up-to-date with the application’s state.