# **Monitoring and Instrumentation**

There are several ways to monitor Spark applications: web UIs, metrics, and external instrumentation.

# **Web Interfaces**

Every SparkContext launches a web UI, by default on port 4040, that displays useful information about the application. This includes:

* A list of scheduler stages and tasks
* A summary of RDD sizes and memory usage
* Environmental information.
* Information about the running executors

You can access this interface by simply opening http://<driver-node>:4040 in a web browser. If multiple SparkContexts are running on the same host, they will bind to successive ports beginning with 4040 (4041, 4042, etc).

Note that this information is only available for the duration of the application by default. To view the web UI after the fact, setspark.eventLog.enabled to true before starting the application. This configures Spark to log Spark events that encode the information displayed in the UI to persisted storage.

## **Viewing After the Fact**

Spark’s Standalone Mode cluster manager also has its own [web UI](http://spark.apache.org/docs/latest/spark-standalone.html" \l "monitoring-and-logging). If an application has logged events over the course of its lifetime, then the Standalone master’s web UI will automatically re-render the application’s UI after the application has finished.

If Spark is run on Mesos or YARN, it is still possible to reconstruct the UI of a finished application through Spark’s history server, provided that the application’s event logs exist. You can start the history server by executing:

./sbin/start-history-server.sh

When using the file-system provider class (see spark.history.provider below), the base logging directory must be supplied in thespark.history.fs.logDirectory configuration option, and should contain sub-directories that each represents an application’s event logs. This creates a web interface at http://<server-url>:18080 by default. The history server can be configured as follows:

|  |  |
| --- | --- |
| **Environment Variable** | **Meaning** |
| SPARK\_DAEMON\_MEMORY | Memory to allocate to the history server (default: 1g). |
| SPARK\_DAEMON\_JAVA\_OPTS | JVM options for the history server (default: none). |
| SPARK\_PUBLIC\_DNS | The public address for the history server. If this is not set, links to application history may use the internal address of the server, resulting in broken links (default: none). |
| SPARK\_HISTORY\_OPTS | spark.history.\* configuration options for the history server (default: none). |

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Default** | **Meaning** |
| spark.history.provider | org.apache.spark.deploy.history.FsHistoryProvider | Name of the class implementing the application history backend. Currently there is only one implementation, provided by Spark, which looks for application logs stored in the file system. |
| spark.history.fs.logDirectory | file:/tmp/spark-events | Directory that contains application event logs to be loaded by the history server |
| spark.history.fs.update.interval | 10s | The period at which information displayed by this history server is updated. Each update checks for any changes made to the event logs in persisted storage. |
| spark.history.retainedApplications | 50 | The number of application UIs to retain. If this cap is exceeded, then the oldest applications will be removed. |
| spark.history.ui.port | 18080 | The port to which the web interface of the history server binds. |
| spark.history.kerberos.enabled | false | Indicates whether the history server should use kerberos to login. This is useful if the history server is accessing HDFS files on a secure Hadoop cluster. If this is true, it uses the configsspark.history.kerberos.principal andspark.history.kerberos.keytab. |
| spark.history.kerberos.principal | (none) | Kerberos principal name for the History Server. |
| spark.history.kerberos.keytab | (none) | Location of the kerberos keytab file for the History Server. |
| spark.history.ui.acls.enable | false | Specifies whether acls should be checked to authorize users viewing the applications. If enabled, access control checks are made regardless of what the individual application had set forspark.ui.acls.enable when the application was run. The application owner will always have authorization to view their own application and any users specified via spark.ui.view.acls when the application was run will also have authorization to view that application. If disabled, no access control checks are made. |
| spark.history.fs.cleaner.enabled | false | Specifies whether the History Server should periodically clean up event logs from storage. |
| spark.history.fs.cleaner.interval | 1d | How often the job history cleaner checks for files to delete. Files are only deleted if they are older than spark.history.fs.cleaner.maxAge. |
| spark.history.fs.cleaner.maxAge | 7d | Job history files older than this will be deleted when the history cleaner runs. |

Note that in all of these UIs, the tables are sortable by clicking their headers, making it easy to identify slow tasks, data skew, etc.

Note that the history server only displays completed Spark jobs. One way to signal the completion of a Spark job is to stop the Spark Context explicitly (sc.stop()), or in Python using the with SparkContext() as sc: to handle the Spark Context setup and tear down, and still show the job history on the UI.

## **REST API**

In addition to viewing the metrics in the UI, they are also available as JSON. This gives developers an easy way to create new visualizations and monitoring tools for Spark. The JSON is available for both running applications, and in the history server. The endpoints are mounted at /api/v1. Eg., for the history server, they would typically be accessible at http://<server-url>:18080/api/v1, and for a running application, athttp://localhost:4040/api/v1.

|  |  |
| --- | --- |
| **Endpoint** | **Meaning** |
| /applications | A list of all applications |
| /applications/[app-id]/jobs | A list of all jobs for a given application |
| /applications/[app-id]/jobs/[job-id] | Details for the given job |
| /applications/[app-id]/stages | A list of all stages for a given application |
| /applications/[app-id]/stages/[stage-id] | A list of all attempts for the given stage |
| /applications/[app-id]/stages/[stage-id]/[stage-attempt-id] | Details for the given stage attempt |
| /applications/[app-id]/stages/[stage-id]/[stage-attempt-id]/taskSummary | Summary metrics of all tasks in the given stage attempt |
| /applications/[app-id]/stages/[stage-id]/[stage-attempt-id]/taskList | A list of all tasks for the given stage attempt |
| /applications/[app-id]/executors | A list of all executors for the given application |
| /applications/[app-id]/storage/rdd | A list of stored RDDs for the given application |
| /applications/[app-id]/storage/rdd/[rdd-id] | Details for the storage status of a given RDD |
| /applications/[app-id]/logs | Download the event logs for all attempts of the given application as a zip file |
| /applications/[app-id]/[attempt-id]/logs | Download the event logs for the specified attempt of the given application as a zip file |

When running on Yarn, each application has multiple attempts, so [app-id] is actually [app-id]/[attempt-id] in all cases.

These endpoints have been strongly versioned to make it easier to develop applications on top. In particular, Spark guarantees:

* Endpoints will never be removed from one version
* Individual fields will never be removed for any given endpoint
* New endpoints may be added
* New fields may be added to existing endpoints
* New versions of the api may be added in the future at a separate endpoint (eg., api/v2). New versions are *not* required to be backwards compatible.
* Api versions may be dropped, but only after at least one minor release of co-existing with a new api version

Note that even when examining the UI of a running applications, the applications/[app-id] portion is still required, though there is only one application available. Eg. to see the list of jobs for the running app, you would go to http://localhost:4040/api/v1/applications/[app-id]/jobs. This is to keep the paths consistent in both modes.

# **Metrics**

Spark has a configurable metrics system based on the [Coda Hale Metrics Library](http://metrics.codahale.com/). This allows users to report Spark metrics to a variety of sinks including HTTP, JMX, and CSV files. The metrics system is configured via a configuration file that Spark expects to be present at$SPARK\_HOME/conf/metrics.properties. A custom file location can be specified via the spark.metrics.conf [configuration property](http://spark.apache.org/docs/latest/configuration.html" \l "spark-properties). Spark’s metrics are decoupled into different *instances* corresponding to Spark components. Within each instance, you can configure a set of sinks to which metrics are reported. The following instances are currently supported:

* master: The Spark standalone master process.
* applications: A component within the master which reports on various applications.
* worker: A Spark standalone worker process.
* executor: A Spark executor.
* driver: The Spark driver process (the process in which your SparkContext is created).

Each instance can report to zero or more *sinks*. Sinks are contained in the org.apache.spark.metrics.sink package:

* ConsoleSink: Logs metrics information to the console.
* CSVSink: Exports metrics data to CSV files at regular intervals.
* JmxSink: Registers metrics for viewing in a JMX console.
* MetricsServlet: Adds a servlet within the existing Spark UI to serve metrics data as JSON data.
* GraphiteSink: Sends metrics to a Graphite node.
* Slf4jSink: Sends metrics to slf4j as log entries.

Spark also supports a Ganglia sink which is not included in the default build due to licensing restrictions:

* GangliaSink: Sends metrics to a Ganglia node or multicast group.

To install the GangliaSink you’ll need to perform a custom build of Spark. *****Note that by embedding this library you will include****[LGPL](http://www.gnu.org/copyleft/lesser.html)****-licensed code in your Spark package*****. For sbt users, set the SPARK\_GANGLIA\_LGPL environment variable before building. For Maven users, enable the -Pspark-ganglia-lgpl profile. In addition to modifying the cluster’s Spark build user applications will need to link to the spark-ganglia-lgplartifact.

The syntax of the metrics configuration file is defined in an example configuration file, $SPARK\_HOME/conf/metrics.properties.template.

# **Advanced Instrumentation**

Several external tools can be used to help profile the performance of Spark jobs:

* Cluster-wide monitoring tools, such as [Ganglia](http://ganglia.sourceforge.net/), can provide insight into overall cluster utilization and resource bottlenecks. For instance, a Ganglia dashboard can quickly reveal whether a particular workload is disk bound, network bound, or CPU bound.
* OS profiling tools such as [dstat](http://dag.wieers.com/home-made/dstat/), [iostat](http://linux.die.net/man/1/iostat), and [iotop](http://linux.die.net/man/1/iotop) can provide fine-grained profiling on individual nodes.
* JVM utilities such as jstack for providing stack traces, jmap for creating heap-dumps, jstat for reporting time-series statistics and jconsolefor visually exploring various JVM properties are useful for those comfortable with JVM internals.