Apache Spark 3

Configuring Apache Spark

Date of Publish: 2018-04-01

http://docs.hortonworks.com

Contents

Configuring the Spark Thrift Server

Use the following steps to configure the Apache Spark Thrift server.

About this task

The Apache Spark Thrift server is a service that allows JDBC and ODBC clients to run Spark SQL queries. The Spark Thrift server is a variant of HiveServer2.

Customizing the Spark Thrift Server Port

The default Spark Thrift server port is 10016. To specify a different port, select Spark > Config on the Ambari dashboard, then select Advanced spark-hive-site-override. Set the hive.server2.thrift.port property to the new port number. Click Save, then restart Spark and any other components that require a restart.

Configuring the Livy Server

Use the following steps to configure the Livy server.

Configure the Livy Server

1. Select Spark > Config on the Ambari dashboard, then select Custom livy-conf.
2. Add a livy.superusers property and set it to the Zeppelin service account.

For a non-Ambari cluster, see "Installing and Configuring Livy" in the Spark 2 chapter of the Command Line Installation Guide.

Configure SSL for the Livy Server

To enable SSL for Livy, configure the following parameters for the SSL certificate and key, the keystore password, and the key password, respectively:

```
livy.keystore=<keystore_file>
livy.keystore.password = <storePassword>
livy.key-password = <KeyPassword>
```

For background information about configuring SSL for Spark, see "Configuring Spark for Wire Encryption" in this guide.

Configure High Availability for the Livy Server

By default, if the Livy server fails, all connected Spark clusters are terminated. This means that all jobs and data will disappear immediately.

For deployments that require high availability, Livy supports session recovery, which ensures that a Spark cluster remains available if the Livy server fails. After a restart, the Livy server can connect to existing sessions and roll back to the state before failing.

Livy uses several property settings for recovery behavior related to high availability. These settings are managed by Ambari.

Configuring the Spark History Server

Use the following steps to configure the Apache Spark history server.

The Spark history server is a monitoring tool that displays information about completed Spark applications. This information is pulled from the data that applications by default write to a directory on Hadoop Distributed File System (HDFS).

To access the Spark history server, click Spark2 in the Ambari Dashboard, then click Spark2 History Server UI under "Quick Links".

Ambari Admin users can see the history of all Spark jobs. Other users can only see the history of jobs they submitted.

Related Information

Apache Monitoring and Instrumentation

Configuring Dynamic Resource Allocation

This section describes how to configure dynamic resource allocation for Apache Spark.

When the dynamic resource allocation feature is enabled, an application's use of executors is dynamically adjusted based on workload. This means that an application can relinquish resources when the resources are no longer needed, and request them later when there is more demand. This feature is particularly useful if multiple applications share resources in your Spark cluster.

Dynamic resource allocation is available for use by the Spark Thrift server and general Spark jobs.

> **Note:**
>
> Dynamic Resource Allocation does not work with Spark Streaming.

You can configure dynamic resource allocation at either the cluster or the job level:

- Cluster level:
  - On an Ambari-managed cluster, the Spark Thrift server uses dynamic resource allocation by default. The Thrift server increases or decreases the number of running executors based on a specified range, depending on load. (In addition, the Thrift server runs in YARN mode by default, so the Thrift server uses resources from the YARN cluster.) The associated shuffle service starts automatically, for use by the Thrift server and general Spark jobs.
  - On a manually installed cluster, dynamic resource allocation is not enabled by default for the Thrift server or for other Spark applications. You can enable and configure dynamic resource allocation and start the shuffle service during the Spark manual installation or upgrade process.
- Job level: You can customize dynamic resource allocation settings on a per-job basis. Job settings override cluster configuration settings.

Cluster configuration is the default, unless overridden by job configuration.

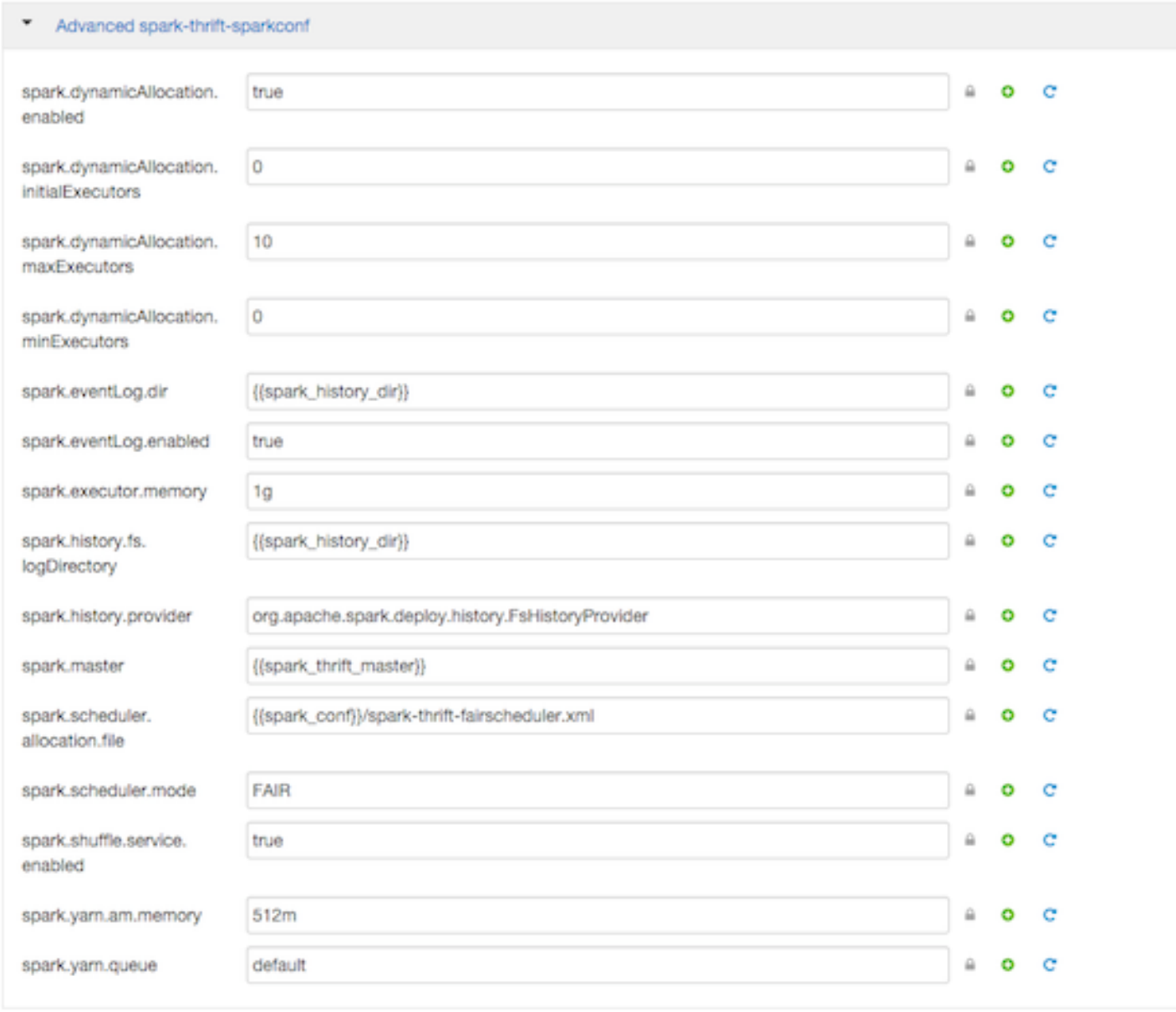
The following subsections describe each configuration approach, followed by a list of dynamic resource allocation properties and a set of instructions for customizing the Spark Thrift server port.

Customize dynamic resource allocation settings on Ambari

Use the following steps to review and customize dynamic resource allocation settings on an Ambari-managed cluster.

On an Ambari-managed cluster, dynamic resource allocation is enabled and configured for the Spark Thrift server as part of the Spark installation process. Dynamic resource allocation is not enabled by default for general Spark jobs.

You can review dynamic resource allocation for the Spark Thrift server, and enable and configure settings for general Spark jobs, by selecting Services > Spark > Advanced spark-thrift-sparkconf:



The "Advanced spark-thrift-sparkconf" group lists required settings. You can specify optional properties in the custom section. For a complete list of DRA properties, see "Dynamic Resource Allocation Properties" in this guide.

Dynamic resource allocation requires an external shuffle service that runs on each worker node as an auxiliary service of NodeManager. If you installed your cluster using Ambari, the service is started automatically for use by the Thrift server and general Spark jobs; no further steps are needed.

Manually configure dynamic resource allocation

Use the following steps to manually configure dynamic resource allocation settings.

Procedure

1. Add the following properties to the spark-defaults.conf file associated with your Spark installation (typically in the $SPARK_HOME/conf directory):
   - Set spark.dynamicAllocation.enabled to true.

- Set spark.shuffle.service.enabled to true.

2. (Optional) To specify a starting point and range for the number of executors, use the following properties:
   - spark.dynamicAllocation.initialExecutors
   - spark.dynamicAllocation.minExecutors
   - spark.dynamicAllocation.maxExecutors

   Note that initialExecutors must be greater than or equal to minExecutors, and less than or equal to maxExecutors.

   For a description of each property, see "Dynamic Resource Allocation Properties" in this guide.
3. Start the shuffle service on each worker node in the cluster:
   a. In the yarn-site.xml file on each node, add spark_shuffle to yarn.nodemanager.aux-services, and then set yarn.nodemanager.aux-services.spark_shuffle.class toorg.apache.spark.network.yarn.YarnShuffleService.
   b. Review and, if necessary, edit spark.shuffle.service.* configuration settings.
   c. Restart all NodeManagers in your cluster.

Related Information

Apache Spark Shuffle Behavior

Configure a job for dynamic resource allocation

Use the following steps to configre dynamic resource allocation for a specific job.

There are two ways to customize dynamic resource allocation properties for a specific job:

- Include property values in the spark-submit command, using the -conf option.

  This approach loads the default spark-defaults.conf file first, and then applies property values specified in your spark-submit command.

  Example:

  spark-submit --conf “property_name=property_value”
- Create a job-specific spark-defaults.conf file and pass it to the spark-submit command.

  This approach uses the specified properties file, without reading the default property file.

  Example:

  spark-submit --properties-file <property_file>

Dynamic resource allocation properties

The following tables provide more information about dynamic resource allocation properties.

Table 3.1. Dynamic Resource Allocation Properties

Property Name	Value	Meaning
spark.dynamicAllocation. enabled	Default is true for the Spark Thrift server, and false for Spark jobs.	Specifies whether to use dynamic resource allocation, which scales the number of executors registered for an application up and down based on workload. Note that this feature is currently only available in YARN mode.
spark.shuffle.service. enabled	true	Enables the external shuffle service, which preserves shuffle files written by executors so that the executors can be safely removed. This property must be set to true if spark.dynamicAllocation. enabledis true.

Property Name	Value	Meaning
spark.dynamicAllocation. initialExecutors	Default is spark.dynamicAllocation. minExecutors	The initial number of executors to run if dynamic resource allocation is enabled. This value must be greater than or equal to the minExecutors value, and less than or equal to the maxExecutors value.
spark.dynamicAllocation. maxExecutors	Default is infinity	Specifies the upper bound for the number of executors if dynamic resource allocation is enabled.
spark.dynamicAllocation. minExecutors	Default is 0	Specifies the lower bound for the number of executors if dynamic resource allocation is enabled.

Table 3.2. Optional Dynamic Resource Allocation Properties

Property Name	Value	Meaning
spark.dynamicAllocation. executorIdleTimeout	Default is 60 seconds (60s)	If dynamic resource allocation is enabled and an executor has been idle for more than this time, the executor is removed.
spark.dynamicAllocation. cachedExecutorIdleTimeout	Default is infinity	If dynamic resource allocation is enabled and an executor with cached data blocks has been idle for more than this time, the executor is removed.
spark.dynamicAllocation. schedulerBacklogTimeout	1 second (1s)	If dynamic resource allocation is enabled and there have been pending tasks backlogged for more than this time, new executors are requested.
spark.dynamicAllocation. sustainedSchedulerBacklogTimeout	Default is schedulerBacklogTimeout	Same as spark.dynamicAllocation. schedulerBacklogTimeout,but used only for subsequent executor requests.

Related Information

Apache Dynamic Resource Allocation

Configuring Spark for Wire Encryption

You can configure Spark to protect sensitive data in transit by enabling wire encryption.

About this task

In general, wire encryption protects data by making it unreadable without a phrase or digital key to access the data. Data can be encrypted while it is in transit and when it is at rest:

- "In transit" encryption refers to data that is encrypted when it traverses a network. The data is encrypted between the sender and receiver process across the network. Wire encryption is a form of "in transit" encryption.
- "At rest" or "transparent" encryption refers to data stored in a database, on disk, or on other types of persistent media.

Apache Spark supports "in transit" wire encryption of data for Apache Spark jobs. When encryption is enabled, Spark encrypts all data that is moved across nodes in a cluster on behalf of a job, including the following scenarios:

- Data that is moving between executors and drivers, such as during a collect() operation.
- Data that is moving between executors, such as during a shuffle operation.

Spark does not support encryption for connectors accessing external sources; instead, the connectors must handle any encryption requirements. For example, the Spark HDFS connector supports transparent encrypted data access from

HDFS: when transparent encryption is enabled in HDFS, Spark jobs can use the HDFS connector to read encrypted data from HDFS.

Spark does not support encrypted data on local disk, such as intermediate data written to a local disk by an executor process when the data does not fit in memory. Additionally, wire encryption is not supported for shuffle files, cached data, and other application files. For these scenarios you should enable local disk encryption through your operating system.

> **Note:**
>
> Enabling Spark wire encryption also enables HTTPS on the History Server UI, for browsing historical job data.

Procedure

1. On each node, create keystore files, certificates, and truststore files.
   a. Create a keystore file:

```
keytool -genkey \
    -alias <host> \
    -keyalg RSA \
    -keysize 1024 \
    –dname CN=<host>,OU=hw,O=hw,L=paloalto,ST=ca,C=us \
    –keypass <KeyPassword> \
    -keystore <keystore_file> \
    -storepass <storePassword>
```

   b. Create a certificate:

```
keytool -export \
    -alias <host> \
    -keystore <keystore_file> \
    -rfc –file <cert_file> \
    -storepass <StorePassword>
```

   c. Create a truststore file:

```
keytool -import \
    -noprompt \
    -alias <host> \
    -file <cert_file> \
    -keystore <truststore_file> \
    -storepass <truststorePassword>
```

2. Create one truststore file that contains the public keys from all certificates.
   a. Log on to one host and import the truststore file for that host:

```
keytool -import \
    -noprompt \
    -alias <hostname> \
    -file <cert_file> \
    -keystore <all_jks> \
    -storepass <allTruststorePassword>
```

   b. Copy the <all_jks> file to the other nodes in your cluster, and repeat the keytool command on each node.
3. Enable Spark authentication.
   a. Set spark.authenticate to true in the yarn-site.xml file:

```
<property>
  <name>spark.authenticate</name>
  <value>true</value>
```

```
</property>
```

b. Set the following properties in the spark-defaults.conf file:

```
spark.authenticate true
spark.authenticate.enableSaslEncryption true
```

4. Enable Spark SSL.

   Set the following properties in the spark-defaults.conf file:

```
spark.ssl.enabled true
spark.ssl.keyPassword <KeyPassword>
spark.ssl.keyStore <keystore_file>
spark.ssl.keyStorePassword <storePassword>
spark.ssl.protocol TLS
spark.ssl.trustStore <all_jks>
spark.ssl.trustStorePassword <allTruststorePassword>
```

5. Enable HTTPS for the Spark UI.

   Set spark.ui.https.enabled to true in the spark-defaults.conf file:

```
spark.ui.https.enabled true
```

6. (Optional) If you want to enable optional on-disk block encryption, which applies to both shuffle and RDD blocks on disk, complete the following steps:

   a. Add the following properties to the spark-defaults.conf file for Spark:

```
spark.io.encryption.enabled true
spark.io.encryption.keySizeBits 128
spark.io.encryption.keygen.algorithm HmacSHA1
```

   b. Enable RPC encryption.

Configuring Spark for a Kerberos-enabled Cluster

This section describes how to configure Apache Spark for a Kerberos-enabled cluster.

About this task

Before running Spark jobs on a Kerberos-enabled cluster, configure additional settings for the following modules and scenarios:

- Spark history server
- Spark Thrift server
- Individuals who submit jobs
- Processes that submit jobs without human interaction

Each of these scenarios is described in the following subsections.

When Kerberos is enabled on an Ambari-managed cluster, Livy configuration for Kerberos is handled automatically.

Configure the Spark history server

On a Kerberos-enabled cluster, the Spark history server daemon must have a Kerberos account and keytab.

When you enable Kerberos for a Hadoop cluster with Ambari, Ambari configures Kerberos for the Spark history server and automatically creates a Kerberos account and keytab for it. Ambari configures the Spark history server

permissions such that Admin users can see the history of all Spark jobs, and other users can only see the history of jobs they submitted. To customize these settings, select Spark2 > Configs > Custom spark2-defaults in Ambari.

For example, to add Admin users:

1. Select Spark2 > Configs > Custom spark2-defaults in Ambari.
2. Add following properties:

Table 1: Apache Spark History Server ACL Settings

Property	Value
spark.history.ui.acls.enable	true
spark.history.ui.admin.acls	Comma-separated list of Admin users.

3. Click Save, then restart Spark and any other services that require a restart.

Related Information

Apache Spark Security
Apache Spark Monitoring and Instrumentation
Enabling Kerberos Authentication Using Ambari
Creating Service Principals and Keytab Files for HDP

Configure the Spark Thrift server

Use the following steps to configure the Apache Spark Thrift server on a Kerberos-enabled cluster.

If you are installing the Spark Thrift server on a Kerberos-enabled cluster, note the following requirements:

- The Spark Thrift server must run in the same host as HiveServer2, so that it can access the hiveserver2keytab.
- Permissions in /var/run/spark and /var/log/spark must specify read/write permissions to the Hive service account.
- You must use the Hive service account to start the thriftserver process.

If you access Hive warehouse files through HiveServer2 on a deployment with fine-grained access control, run the Spark Thrift server as user hive. This ensures that the Spark Thrift server can access Hive keytabs, the Hive metastore, and HDFS data stored under user hive.

Important:

If you read files from HDFS directly through an interface such as the Spark CLI (as opposed to HiveServer2 with fine-grained access control), you should use a different service account for the Spark Thrift server. Configure the account so that it can access Hive keytabs and the Hive metastore. Use of an alternate account provides a more secure configuration: when the Spark Thrift server runs queries as user hive, all data accessible to user hive is accessible to the user submitting the query.

For Spark jobs that are not submitted through the Thrift server, the user submitting the job must have access to the Hive metastore in secure mode, using the kinit command.

Set up access for submitting jobs

Use the following steps to set up access for submitting Spark jobs on a Kerberos-enabled cluster.

About this task

Accounts that submit jobs on behalf of other processes must have a Kerberos account and keytab. End users should use their own keytabs (instead of using a headless keytab) when submitting a Spark job. The following sections describe both scenarios.

Set Up Access for an Account

When access is authenticated without human interaction (as happens for processes that submit job requests), the process uses a headless keytab. Security risk is mitigated by ensuring that only the service that should be using the headless keytab has permission to read it.

The following example creates a headless keytab for a spark service user account that will submit Spark jobs on node blue1@example.com:

1. Create a Kerberos service principal for user spark:

   kadmin.local -q "addprinc -randkey spark/blue1@EXAMPLE.COM"
2. Create the keytab:

```
kadmin.local -q "xst -k /etc/security/keytabs/spark.keytab
              spark/blue1@EXAMPLE.COM"
```

3. For every node of your cluster, create a spark user and add it to the hadoop group:

   useradd spark -g hadoop
4. Make spark the owner of the newly created keytab:

   chown spark:hadoop /etc/security/keytabs/spark.keytab
5. Limit access by ensuring that user spark is the only user with access to the keytab:

   chmod 400 /etc/security/keytabs/spark.keytab

In the following example, user spark runs the Spark Pi example in a Kerberos-enabled environment:

```
su spark
kinit -kt /etc/security/keytabs/spark.keytab spark/blue1@EXAMPLE.COM
cd /usr/hdp/current/spark-client/
./bin/spark-submit --class org.apache.spark.examples.SparkPi \
    --master yarn-cluster \
    --num-executors 1 \
    --driver-memory 512m \
    --executor-memory 512m \
    --executor-cores 1 \
    lib/spark-examples*.jar 10
```

Set Up Access for an End User

Each person who submits jobs must have a Kerberos account and their own keytab; end users should use their own keytabs (instead of using a headless keytab) when submitting a Spark job. This is a best practice: submitting a job under the end user keytab delivers a higher degree of audit capability.

In the following example, end user $USERNAME has their own keytab and runs the Spark Pi job in a Kerberos-enabled environment:

```
su $USERNAME
kinit USERNAME@YOUR-LOCAL-REALM.COM
cd /usr/hdp/current/spark-client/
./bin/spark-submit --class org.apache.spark.examples.SparkPi \
    --master yarn-cluster \
    --num-executors 3 \
    --driver-memory 512m \
    --executor-memory 512m \
    --executor-cores 1 \
    lib/spark-examples*.jar 10
```