Cloudera Runtime 7.3.3

# Configuring Apache Spark

**Date published: 2020-07-28**
**Date modified: 2026-12-13**



**https://docs.cloudera.com/**

# Legal Notice

© Cloudera Inc. 2026. All rights reserved.

The documentation is and contains Cloudera proprietary information protected by copyright and other intellectual property rights. No license under copyright or any other intellectual property right is granted herein.

Unless otherwise noted, scripts and sample code are licensed under the Apache License, Version 2.0.

Copyright information for Cloudera software may be found within the documentation accompanying each component in a particular release.

Cloudera software includes software from various open source or other third party projects, and may be released under the Apache Software License 2.0 (“ASLv2”), the Affero General Public License version 3 (AGPLv3), or other license terms. Other software included may be released under the terms of alternative open source licenses. Please review the license and notice files accompanying the software for additional licensing information.

Please visit the Cloudera software product page for more information on Cloudera software. For more information on Cloudera support services, please visit either the Support or Sales page. Feel free to contact us directly to discuss your specific needs.

Cloudera reserves the right to change any products at any time, and without notice. Cloudera assumes no responsibility nor liability arising from the use of products, except as expressly agreed to in writing by Cloudera.

Cloudera, Cloudera Altus, HUE, Impala, Cloudera Impala, and other Cloudera marks are registered or unregistered trademarks in the United States and other countries. All other trademarks are the property of their respective owners.

Disclaimer: EXCEPT AS EXPRESSLY PROVIDED IN A WRITTEN AGREEMENT WITH CLOUDERA, CLOUDERA DOES NOT MAKE NOR GIVE ANY REPRESENTATION, WARRANTY, NOR COVENANT OF ANY KIND, WHETHER EXPRESS OR IMPLIED, IN CONNECTION WITH CLOUDERA TECHNOLOGY OR RELATED SUPPORT PROVIDED IN CONNECTION THEREWITH. CLOUDERA DOES NOT WARRANT THAT CLOUDERA PRODUCTS NOR SOFTWARE WILL OPERATE UNINTERRUPTED NOR THAT IT WILL BE FREE FROM DEFECTS NOR ERRORS, THAT IT WILL PROTECT YOUR DATA FROM LOSS, CORRUPTION NOR UNAVAILABILITY, NOR THAT IT WILL MEET ALL OF CUSTOMER’S BUSINESS REQUIREMENTS. WITHOUT LIMITING THE FOREGOING, AND TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, CLOUDERA EXPRESSLY DISCLAIMS ANY AND ALL IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, QUALITY, NON-INFRINGEMENT, TITLE, AND FITNESS FOR A PARTICULAR PURPOSE AND ANY REPRESENTATION, WARRANTY, OR COVENANT BASED ON COURSE OF DEALING OR USAGE IN TRADE.

# Contents

**Configuring dynamic resource allocation ..... 4**
Customize dynamic resource allocation settings ..... 4
Configure a Spark job for dynamic resource allocation ..... 4
Dynamic resource allocation properties ..... 5

**Spark security ..... 6**
Enabling Spark authentication ..... 6
Enabling Spark Encryption ..... 7
Running Spark applications on secure clusters ..... 7
Configuring HSTS for Spark ..... 8
Spark and Livy on FIPS 140-3 clusters ..... 8
Configuring Spark for Wire Encryption ..... 9
Encryption in transit for Spark and Livy in Cloudera Runtime 7.3.2.10000 SP1 ..... 11
Configuring TLS/SSL encryption manually for Spark ..... 12
Configuring TLS/SSL encryption manually for Livy ..... 12

**Accessing compressed files in Spark ..... 13**

**Configuring Livy ..... 14**
Configuring the Livy server ..... 14
Livy active/passive high availability ..... 15
Adding independent Livy services with active/passive HA ..... 16
Livy load balancing with HAProxy ..... 18
Preparing Livy service instances for HAProxy load balancing ..... 20
Enabling the Livy /ha-status endpoint ..... 21
Configuring HAProxy for Livy ..... 22
Configuring Knox topologies for multiple Livy for Spark 3 services ..... 25
Making HAProxy highly available with Keepalived ..... 26
Finding the Livy server that owns a session ..... 28
Livy session capacity and YARN queues ..... 29
Livy and HAProxy tuning reference ..... 30
Configuring TLS/SSL encryption manually for Livy ..... 33
Using Spark Hive Warehouse and HBase Connector Client .jar files with Livy ..... 33

**Using Spark History Servers with high availability ..... 35**
Limitations for Spark History Server with high availability ..... 35
Configuring high availability for Spark History Server with an external load balancer ..... 36
Configuring high availability for Spark History Server with an internal load balancer ..... 37
Configuring high availability for Spark History Server with multiple Knox Gateways ..... 38

**How to access Spark files on Ozone ..... 39**

# 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.



**Important:** 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: Dynamic resource allocation is enabled by default. The associated shuffle service starts automatically.
- 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.

## Customize dynamic resource allocation settings

Use the following steps to review and customize dynamic resource allocation settings.

### About this task

Dynamic resource allocation requires an external shuffle service that runs on each worker node as an auxiliary service of NodeManager. This service is started automatically; no further steps are needed.

Dynamic resource allocation is enabled by default. To modify dynamic allocation settings, use the following procedure.

### Procedure

1. From the Cloudera interface, click through to Cloudera Manager for the cluster running Spark.
2. Go to the Spark service page (ClustersSpark service).
3. Click on the Configuration tab.
4. In the Search box, enter dynamicAllocation.
5. After making the changes you want, enter the reason for the change in the Reason for change... box, and then click Save Changes.
6. Restart the Spark service (Spark serviceActionsRestart).

## Configure a Spark job for dynamic resource allocation

Use the following steps to configure 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. These properties can be accessed by clicking through to Cloudera Manager from the Cloudera interface for the cluster running Spark.

### Table 1: Dynamic Resource Allocation Properties

| Property Name | Default Value | Description |
|---|---|---|
| spark.dynamicAllocation.enabled | false for Spark jobs | Enable dynamic allocation of executors in Spark applications. |
| spark.shuffle.service. enabled | true | Enables the external shuffle service. The external shuffle service preserves shuffle files written by executors so that the executors can be deallocated without losing work. Must be enabled if dynamic allocation is enabled. |
| spark.dynamicAllocation. initialExecutors | Same value as spark.dynamicAllocation. minExecutors | When dynamic allocation is enabled, number of executors to allocate when the application starts.<br>Must be greater than or equal to the minExecutors value, and less than or equal to the maxExecutors value. |
| spark.dynamicAllocation. maxExecutors | infinity | When dynamic allocation is enabled, maximum number of executors to allocate. By default, Spark relies on YARN to control the maximum number of executors for the application. |
| spark.dynamicAllocation. minExecutors | 0 | When dynamic allocation is enabled, minimum number of executors to keep alive while the application is running. |
| spark.dynamicAllocation. executorIdleTimeout | 60 seconds (60s) | When dynamic allocation is enabled, time after which idle executors will be stopped. |
| spark.dynamicAllocation. cachedExecutorIdleTimeout | infinity | When dynamic allocation is enabled, time after which idle executors with cached RDD blocks will be stopped. |
| spark.dynamicAllocation. schedulerBacklogTimeout | 1 second (1s) | When dynamic allocation is enabled, timeout before requesting new executors when there are backlogged tasks. |
| spark.dynamicAllocation. sustainedSchedulerBacklogTimeout | Same value as schedulerBacklogTimeout | When dynamic allocation is enabled, timeout before requesting new executors after the initial backlog timeout has already expired. By default this is the same value as the initial backlog timeout. |

### Related Information

Apache Dynamic Resource Allocation

# Spark security

**Note:**

The Spark and Spark 3 History Server web UI does not send HTTP Content Security Policy (CSP) headers. Some vulnerability scanners may therefore flag the Spark History Server endpoints for missing CSP headers.

The Spark History Server UI is a read#only interface and does not accept user input that could be used for cross#site scripting (XSS) attacks. In this context, the absence of CSP headers is not a security vulnerability, and these scanner findings can be safely ignored for the Spark History Server UI.

If your internal policies require CSP headers on all HTTP services, you can deploy the Spark History Server behind an HTTP reverse proxy or load balancer and configure CSP headers there according to your organization's security guidelines.

When you create an environment in Cloudera Management Console, it automatically creates a Kerberos- and TLS-enabled data lake cluster. Data hub clusters are linked to environments, and therefore also have Kerberos enabled by default. You do not need to do anything to enable Kerberos or TLS for Apache Spark in Cloudera. Disabling security is not supported.

To submit Spark jobs to the cluster, users must authenticate with their Kerberos credentials. For more information, see the Environments documentation.

### Encryption in transit (7.3.2.10000 SP1)

For TLS 1.3, Encrypt all ports, and Spark network encryption in service pack 1, see Encryption in transit for Spark and Livy in Cloudera Runtime 7.3.2.10000 SP1.

## Enabling Spark authentication

Spark authentication here refers to an internal authentication mechanism, and not to Kerberos authentication, which is enabled automatically for all Cloudera deployments.

### Before you begin

Minimum Required Role: Security Administrator (also provided by Full Administrator)

### About this task

Spark has an internal mechanism that authenticates executors with the driver controlling a given application. This mechanism is enabled using the Cloudera Manager Admin Console, as detailed below. Cluster administrators can enable the spark.authenticate mechanism to authenticate the various processes that support a Spark application.

To enable this feature on the cluster:

### Procedure

1. In the Cloudera Management Console, go to Data Hub Clusters.
2. Find and select the cluster you want to configure.
3. Click the link for the Cloudera Manager URL.
4. Go to  Clusters <Cluster Name>Spark serviceConfiguration .
5. Scroll down to the Spark Authentication setting, or search for spark.authenticate to find it.
6. In the Spark Authentication setting, click the checkbox next to the Spark (Service-Wide) property to activate the setting.
7. Enter the reason for the change at the bottom of the screen, and then click Save Changes.

8. Restart YARN:
   a) Select Clusters YARN .
   b) Select Restart from the Actions drop-down selector.
9. Re-deploy the client configurations:
   a) Select Clusters *CLUSTER_NAME*
   b) Select Deploy Client Configurations from the Actions drop-down selector.
10. Restart stale services.

## Enabling Spark Encryption

### Before you begin

Before enabling encryption, you must first enable Spark authentication.

### Procedure

1. In the Cloudera Management Console, go to Data Hub Clusters.
2. Find and select the cluster you want to configure.
3. Click the link for the Cloudera Manager URL.
4. Go to Clusters <Cluster Name>Spark serviceConfiguration .
5. Search for the Enable Network Encryption property. Use the checkbox to enable encrypted communication between Spark processes belonging to the same application.
6. Search for the Enable I/O Encryption property. Use the checkbox to enabled encryption for temporary shuffle and cache files stored by Spark on local disks.
7. Enter the reason for the change at the bottom of the screen, and then click Save Changes.
8. Re-deploy the client configurations:
   a) Select Clusters *CLUSTER_NAME*
   b) Select Deploy Client Configurations from the Actions drop-down selector.
9. Restart stale services.

## Running Spark applications on secure clusters

All Cloudera clusters are secure by default. Disabling security on Cloudera clusters is not supported. To run a Spark application on a secure cluster, you must first authenticate using Kerberos.

Users running Spark applications must first authenticate to Kerberos, using kinit, as follows:

```
kinit username@EXAMPLE.COM
```

After authenticating to Kerberos, users can submit their applications using spark-submit as usual, as shown below. This command submits one of the default Spark sample jobs using an environment variable as part of the path, so modify as needed for your own use:

```
$ spark-submit --class org.apache.spark.examples.SparkPi --master yarn \
--deploy-mode cluster $SPARK_HOME/lib/spark-examples.jar 10
```

For information on creating user accounts in Cloudera, see Onboarding Users.

# Configuring HSTS for Spark

You can configure Apache Spark to include HTTP headers to prevent Cross-Site Scripting (XSS), Cross-Frame Scripting (XFS), MIME-Sniffing, and also enforce HTTP Strict Transport Security (HSTS).

## Procedure

1. Go to the Spark service.
2. Click the Configuration tab.
3. Select History Server under Scope.
4. Select Advanced under Category.
5. Set the following HSTS credentials in History Server Advanced Configuration Snippet (Safety Valve) for spark3-conf/spark-history-server.conf.

```
spark.ui.strictTransportSecurity=max-age=31536000;includeSubDomains
```

6. Restart the Spark service.

# Spark and Livy on FIPS 140-3 clusters

Spark 3 and Livy for Spark 3 support Federal Information Processing Standards (FIPS) 140-3 on FIPS-enabled clusters when you use the supported Cloudera Runtime service pack and platform prerequisites.

## Overview

Spark 3 and Livy for Spark 3 run on clusters that are already configured for FIPS 140-3. Platform preparation (host FIPS mode, validated cryptographic modules, and JDK configuration) is documented in the Cloudera platform FIPS installation guides, not in this Runtime library.

In Cloudera Runtime 7.3.2.10000 SP1, Spark 3 includes updated cryptographic modules and Cloudera SASL support for FIPS 140-3. Livy for Spark 3 uses the same cluster TLS and Kerberos settings as Spark 3.

## Default JVM trust store on BCFKS clusters

On FIPS-enabled hosts, the JVM default trust store (java.home/lib/security/cacerts) is often a Bouncy Castle FIPS (BCFKS) keystore. Spark and Livy must supply the trust store password when the JVM does not already have java x.net.ssl.trustStorePassword set.

**Spark 3**

Set spark.ssl.defaultTrustStorePassword. When this property is set, Spark adds -Djavax.net.ssl.trus tStorePassword=<password> to driver and executor JVM options at submit time and when a Secu rityManager is created (for example, Spark History Server).

**Livy for Spark 3**

Set livy.ssl.defaultTrustStorePassword in Livy server configuration. When set, the Livy server sets javax.net.ssl.trustStorePassword on the Livy JVM if it is not already defined.

On Cloudera Manager-managed clusters, these properties may be populated automatically when the default cacerts keystore is BCFKS. If Spark History Server or Livy fail to start with BCFKS MAC or invalid keystore errors, verify the trust store password configuration and the BCFKS cacerts conversion on cluster hosts.

**Important:**

Interactive Livy sessions submit Spark applications. Configure Spark trust store settings as well as Livy server settings.

### Spark Atlas Connector

The Spark Atlas Connector does not ship a separate FIPS cryptographic module. Lineage collection uses Spark 3 and Apache Atlas on the cluster (TLS, Kerberos, and Atlas REST). Ensure Spark 3 and Atlas meet your FIPS requirements.

## 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.

**Note:**

TLS and Spark network encryption: TLS protects HTTP and HTTPS endpoints such as the Spark UI and History Server. Spark network encryption (when spark.authenticate and spark.network.crypto.enabled are enabled) protects RPC between Spark processes and is separate from TLS. In Cloudera Runtime 7.3.2.10000 SP1, Cloudera Manager can enable both when you turn on Encrypt all ports. See Encryption in transit for Spark and Livy in Cloudera Runtime 7.3.2.10000 SP1.

### 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
```

   **Note:**

   When you configure optional on-disk IO encryption through Cloudera Manager in Cloudera Runtime 7.3.2.10000 SP1, the default spark.io.encryption.keySizeBits is 256 and the default spark.io.encryption.keygen.algorithm is HmacSHA256. The values in the example above apply to manual spark-defaults.conf edits only.

   b. Enable RPC encryption.

## Encryption in transit for Spark and Livy in Cloudera Runtime 7.3.2.10000 SP1

In service pack 1, Cloudera Manager can enable Spark and Livy encryption settings automatically when you turn on encryption for all ports. Transport Layer Security (TLS) protects HTTP and REST endpoints; Spark network encryption protects RPC between Spark processes.

### Overview

Cloudera Runtime 7.3.2.10000 SP1 aligns Spark and Livy with encryption-on-the-wire requirements. Two mechanisms work together:

- TLS — Encrypts Spark History Server, Spark UI, and Livy Server HTTPS listeners. Configure TLS protocol versions (including TLS 1.3 where supported) through Cloudera Manager global TLS settings or service TLS properties.
- Spark network encryption — Encrypts communication between Spark drivers, executors, and the external shuffle service when authentication is enabled. This uses AES encryption, not TLS.

When you enable the CMF_FF_ENCRYPT_ALL_PORTS (Encrypt all ports) feature flag in Cloudera Manager, Cloudera Manager sets related Spark properties automatically. For a full list of behavioral changes, see Behavioral Changes in SparkBehavioral Changes in Spark and Behavioral Changes in LivyBehavioral Changes in Livy.

### Encrypt all ports

Several Spark security parameters are disabled by default. When Encrypt all ports is active, Cloudera Manager enables them, including:

- spark.authenticate and spark.network.crypto.enabled for RPC encryption
- spark.ssl.historyServer.disableHttpPort and spark.ssl.ui.disableHttpPort when TLS is enabled for HTTPS-only Jetty listeners
- spark.cloudera.auto.disable.insecure.spark.ui.enabled to disable the Spark UI when SSL is not active
- spark.cloudera.ui.yarn.autoCertificate.clusterMode.enabled for YARN ApplicationMaster certificates in cluster deploy mode (not client mode)

For manual configuration of network encryption, see Configuring Spark for Wire Encryption.

### TLS 1.3

Cloudera recommends TLS 1.3 when your JDK and Cloudera Manager TLS policy allow it. Use TLS 1.2 only as an exception with approved cipher suites. For Spark History Server TLS, see Configuring TLS/SSL encryption manually for Spark. For Livy Server TLS, see Configuring TLS/SSL encryption manually for Livy.

Lineage through the Spark Atlas Connector uses TLS on Apache Atlas and Kafka connections; the connector does not provide a separate TLS stack. Configure Atlas TLS as described in Configuring TLS/SSL for Apache Atlas.

### FIPS 140-3

FIPS 140-3 support and BCFKS trust store settings are documented separately. See Spark and Livy on FIPS 140-3 clusters.

## Configuring TLS/SSL encryption manually for Spark

You can enable TLS manually for the Spark History Server.

### About this task

In Cloudera Runtime 7.3.2.10000 SP1, Spark History Server TLS inherits protocol and cipher settings from Cloudera Manager when you use global TLS configuration. Cloudera recommends TLS 1.3 when your JDK and cluster policy allow it. For Encrypt all ports and related Spark defaults, see Encryption in transit for Spark and Livy in Cloudera Runtime 7.3.2.10000 SP1.

### Procedure

1. In Cloudera Manager, select the Spark service from the Clusters drop-down menu.
2. In the Configuration tab, enter tls into the search box.
3. Edit the following property fields as needed for your cluster and environment:

| Property | Description |
|---|---|
| TLS/SSL Protocol | The version of the TLS/SSL protocol to use when TLS/SSL is enabled. |
| TLS protocol versions | When the tls-advanced-control feature flag is enabled in Cloudera Manager, Spark History Server TLS protocol settings can inherit from global TLS configuration (supported_tls_versions or adv_supported_tls_versions). Include TLSv1.3 to use TLS 1.3. When the flag is disabled, Cloudera Manager uses the legacy TLS 1.2 configuration for the service. |
| Enabled SSL/TLS Algorithms | A comma-separated list of algorithm names to enable when TLS/SSL is enabled. By default, all algorithms supported by the JRE are enabled. |
| TLS/SSL Port Number | Port where to listen for TLS/SSL connections. HTTP connections will be redirected to this port when TLS/SSL is enabled. |
| Enable TLS/SSL for History Server | Encrypt communication between clients and History Server using Transport Layer Security (TLS) (formerly known as Secure Socket Layer (SSL)). |
| History Server TLS/SSL Server JKS Keystore File Location | The path to the TLS/SSL keystore file containing the server certificate and private key used for TLS/SSL. Used when History Server is acting as a TLS/SSL server. The keystore must be in JKS format. |
| History Server TLS/SSL Server JKS Keystore File Password | The password for the History Server JKS keystore file. |

4. Click Save Changes.
5. Restart the Spark service.

## Configuring TLS/SSL encryption manually for Livy

You can enable TLS manually for the Apache Livy Server.

### About this task

In Cloudera Runtime 7.3.2.10000 SP1, Livy Server TLS inherits protocol and cipher settings from Cloudera Manager when you use global TLS configuration. Cloudera recommends TLS 1.3 when your JDK and cluster policy allow it. For related Spark and Livy encryption defaults, see Encryption in transit for Spark and Livy in Cloudera Runtime 7.3.2.10000 SP1.

### Procedure

1. In Cloudera Manager, select the Livy service from the Clusters drop-down menu.
2. In the Configuration tab, enter tls into the search box.
3. Edit the following property fields as needed for your cluster and environment:

| Property | Description |
|---|---|
| Gateway TLS/SSL Trust Store File | The location on disk of the trust store, in .jks format, used to confirm the authenticity of TLS/SSL servers that Gateway might connect to. This trust store must contain the certificate(s) used to sign the service(s) connected to. If this parameter is not provided, the default list of well-known certificate authorities is used instead. |
| Gateway TLS/SSL Trust Store Password | The password for the Gateway TLS/SSL Trust Store File. This password is not required to access the trust store; this field can be left blank. This password provides optional integrity checking of the file. The contents of trust stores are certificates, and certificates are public information. |
| Enable TLS/SSL for Livy Server | Encrypt communication between clients and Livy Server using Transport Layer Security (TLS) (formerly known as Secure Socket Layer (SSL)). |
| TLS protocol versions | When the tls-advanced-control feature flag is enabled in Cloudera Manager, Livy Server TLS protocol settings can inherit from global TLS configuration (supported_tls_versions or adv_supported_tls_versions). Include TLSv1.3 to use TLS 1.3. When the flag is disabled, Cloudera Manager uses the legacy TLS 1.2 configuration for the service. |
| Livy Server TLS/SSL Server JKS Keystore File Location | The path to the TLS/SSL keystore file containing the server certificate and private key used for TLS/SSL. Used when Livy Server is acting as a TLS/SSL server. The keystore must be in JKS format. |
| Livy Server TLS/SSL Server JKS Keystore File Password | The password for the Livy Server JKS keystore file. |

4. Click Save Changes.
5. Restart the Livy service.

# Accessing compressed files in Spark

You can read compressed files using one of the following methods:

- textFile(*PATH*)
- hadoopFile(*PATH*,*OUTPUTFORMATCLASS*)

You can save compressed files using one of the following methods:

- saveAsTextFile(*PATH*, compressionCodecClass="*CODEC_CLASS*")
- saveAsHadoopFile(*PATH*,*OUTPUTFORMATCLASS*, compressionCodecClass="*CODEC_CLASS*")

where *CODEC_CLASS* is one of these classes:

- gzip - org.apache.hadoop.io.compress.GzipCodec
- bzip2 - org.apache.hadoop.io.compress.BZip2Codec

- LZO - com.hadoop.compression.lzo.LzopCodec
- Snappy - org.apache.hadoop.io.compress.SnappyCodec
- Deflate - org.apache.hadoop.io.compress.DeflateCodec

For examples of accessing Avro and Parquet files, see Spark with Avro and Parquet.

# Configuring Livy

Livy is a proxy service for Apache Spark on Cloudera. Use this section to configure the Livy server, high availability, TLS, and client libraries such as Hive Warehouse Connector (HWC) and HBase-Spark connector jars.

Clients such as notebooks, REST applications, and JDBC tools submit Spark work through Livy. Configure Livy in Cloudera Manager and tune service properties for your cluster security and workload requirements.

## Configuring the Livy server

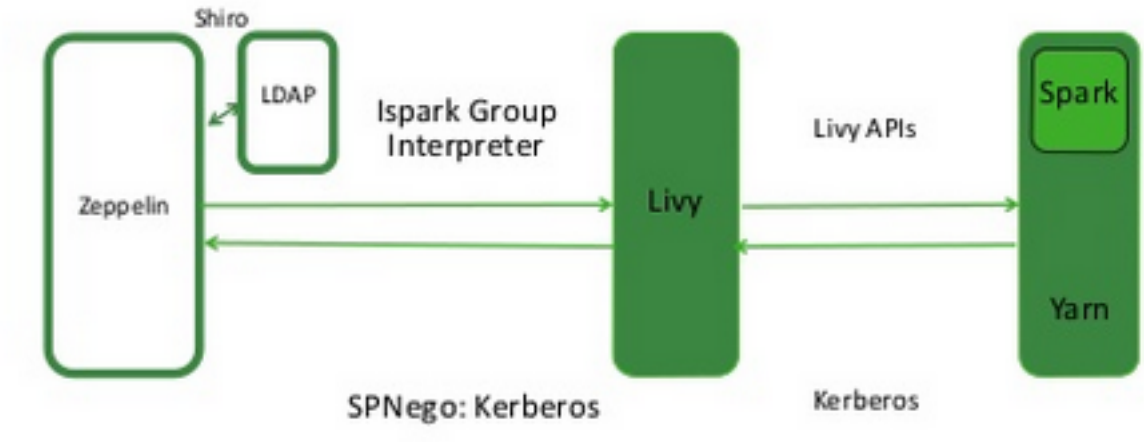
Configure Livy user access control and verify the Livy server on Cloudera.

### About this task

This section describes how to configure Livy in Cloudera.

Livy is a proxy service for Apache Spark. It accepts session and statement requests from clients, launches Spark applications on the cluster, and returns results over HTTP.

Livy supports user impersonation and access control so that multiple users can run isolated Spark sessions on a shared cluster. The following graphic shows how clients, Livy, and Spark interact:




The following sections describe optional configuration steps.

### Configure Livy user access control

You can use the livy.server.access-control.enabled property to configure Livy user access.

When this property is set to false, only the session owner and the superuser can access (both view and modify) a given session. Users cannot access sessions that belong to other users. ACLs are disabled, and any user can send any request to Livy.

When this property is set to true, ACLs are enabled, and the following properties are used to control user access:

- livy.server.access-control.allowed-users – A comma-separated list of users who are allowed to access Livy.

- livy.server.access-control.view-users – A comma-separated list of users with permission to view other users' infomation, such as submitted session state and statement results.
- livy.server.access-control.modify-users – A comma-separated list of users with permission to modify the sessions of other users, such as submitting statements and deleting the session.

### Apply configuration changes

After you change Livy configuration in Cloudera Manager, restart the Livy service so that the new settings take effect. If you use a notebook client that caches interpreter settings, restart or refresh that client according to its documentation.

### Verify that the Livy server is running

To verify that the Livy server is running, access the Livy web UI in a browser window. The default port is 8998:

```
http://<livy-hostname>:8998/
```

## Livy active/passive high availability

Livy high availability is active/passive: one Livy server is active and handles sessions; additional servers are passive and redirect requests to the active server with HTTP 307. When more than one Livy Server role is deployed in a cluster and Cloudera Manager Enable active-passive HA mode (livy_ha_enabled_if_possible, default true) is enabled, Cloudera Manager sets `livy.server.recovery.mode` to `ha` and configures ZooKeeper.

In active/passive high availability mode, multiple Livy instances run in a cluster. Both active and passive instances run continuously; passive instances take the active role when required. There is only one active server at a time, and the remaining instances are passive. Passive instances redirect requests to the active server using HTTP 307.

### Cloudera Manager control of HA

Cloudera Manager can enable active/passive HA automatically when your cluster runs more than one Livy Server role instance. The behavior is controlled by Enable active-passive HA mode (livy_ha_enabled_if_possible). The default value is true.

When livy_ha_enabled_if_possible is true and Cloudera Manager detects more than one Livy Server instance, Cloudera Manager sets `livy.server.recovery.mode` to `ha`, configures ZooKeeper for Livy, and writes the effective settings to livy.conf when you restart Livy or deploy client configuration. While this setting is enabled, values you add for `livy.server.recovery.mode` in the Livy Server for Spark 3 Advanced Configuration Snippet (Safety Valve) for livy3-conf/livy.conf do not take effect if they conflict with HA auto-enablement. The Cloudera Manager UI can show your Safety Valve edit, but the deployed livy.conf can still use `ha`.

To run Livy with `livy.server.recovery.mode` set to `recovery` or to manage recovery mode only through the Safety Valve, set livy_ha_enabled_if_possible to false, add `livy.server.recovery.mode` in the Safety Valve for livy3-conf/livy.conf, restart the Livy service, and deploy client configuration.

### Limitations

- **JDBC connection**: The JDBC connections are not redirected. More details about this limitation are available below.
- **Load balancing**: The active/passive high availability model does not provide additional parallelism or capacity. Interactive sessions are only handled by the active server; adding more servers does not distribute load across servers.

To distribute sessions across several Livy services, you can place HAProxy in front of independent Livy for Spark 3 service instances. This is a validated deployment pattern, not a core Livy feature; each service instance keeps the active/passive behavior described on this page. For details, see Livy load balancing with HAProxy.

### Using Livy without Knox gateway

If you are not using Livy through Knox gateway, clients must follow HTTP redirects and resend authentication. The logic the clients can use is to make a list of the Livy servers by obtaining them from the Cloudera Manager API and if any of the servers do not respond, the clients should retry sending the request to another server in the list. For example, in the case of cURL, the `--location-trusted` flag has to be specified to follow redirects and resend authentication.

### Active/passive high availability and JDBC connections

Livy provides high availability through an active/passive setup. Only the active Livy server node can accept JDBC connections, and the passive nodes reject connection attempts. Therefore, if a client wants to connect using JDBC, it has to iterate through all servers and check which one accepts connections. If the active server goes down, the connection is broken and another node takes over the active role. This behavior is the same for both HTTP and binary mode connections.

### Multiple independent Livy services

When you run more than one Livy service in a cluster (for example, Livy-for-Spark3-1 and Livy-for-Spark3-2), each service must use unique values for livy.server.port, livy.server.recovery.state-store.url, and livy.server.zk.key-prefix. Without unique HDFS recovery paths and ZooKeeper key prefixes, session recovery and leader election can conflict between services.

For step-by-step instructions, see Adding independent Livy services with active/passive HA.

Independent Livy services do not share sessions or a client URL. When you place them behind HAProxy, each service becomes one HAProxy backend, and clients use the single URL of the load balancer.

## Adding independent Livy services with active/passive HA

Each Livy service uses active/passive high availability (not load-balanced active/active). This procedure adds Livy Server roles or a separate Livy service and configures unique ports, session state store paths, and ZooKeeper key prefixes.

### About this task

Use this procedure when you need more than one independent Livy service in a cluster, each with its own HA configuration. For example, you might run Livy-for-Spark3-1 on port 28998 and add Livy-for-Spark3-2 as a separate service. Adding Livy Server roles to one service creates passive instances that redirect requests to the single active instance; see Livy active/passive high availability.

Independent Livy services are also the backends of the HAProxy load-balancing pattern. If you plan to place the services behind HAProxy, complete this procedure first, and then see Preparing Livy service instances for HAProxy load balancing.

With more than one Livy for Spark 3 service, Cloudera Manager supports the Livy gateway role only on the first service, and Knox service discovery adds the Livy Server roles of every service to the Knox topologies. To control which Livy servers Knox exposes, see Configuring Knox topologies for multiple Livy for Spark 3 services.

Prerequisites

- An existing Livy service (for example, Livy-for-Spark3-1) is running.
- Administrative access to Cloudera Manager.
- Required hosts and resources are available in the cluster.

### Procedure

Add Livy roles to the existing Livy service

1. In Cloudera Manager, select  Clusters Livy-for-Spark3-1 Instances .

To enable active/passive HA within an existing Livy service, add Livy Server role instances. One instance is active at a time; the others are passive redirectors.

2. Click Add Role Instances.
3. For Livy Server, click Select Hosts, select the desired hosts, and click OK.
4. Click Continue to deploy the roles.
5. On the Instances tab, verify that the new Livy Server roles are listed.
6. Select the status checkboxes for all Livy instances, then from the Actions for Selected menu, select Restart and confirm.

Add a new Livy service

7. In Cloudera Manager, select Clusters Cluster 1 .
8. From the Actions menu, select Add Service.
9. Select Livy-for-Spark3 as the service type and click Continue.
10. On the dependencies step, select No dependencies, then click Continue.
11. Assign gateways and hosts for the new Livy service roles.
12. Click Continue and complete the Add Service wizard. The new Livy service starts automatically.

Resolve port conflicts

13. If the new Livy service fails to start with an error similar to the following, the default port is already in use by another Livy service:

```
Exception in thread "main" java.io.IOException: Failed to bind to /0.0.0
.0:28998
Caused by: java.net.BindException: Address already in use
```

14. In Cloudera Manager, select the new Livy service (for example, Livy-for-Spark3-2) and open Configuration.
15. Search for livy.server.port.
16. Change the port from the default 28998 to a unique value (for example, 28999).
17. Click Save Changes, confirm the configuration changes, and click Continue.
18. When the service restarts with the updated port, click Finish.

Configure unique session state store paths

19. Each independent Livy service must use a unique HDFS recovery path (livy.server.recovery.state-store.url) to prevent conflicts and enable independent session recovery.
20. For Livy-for-Spark3-1, open Configuration, search for livy.server.recovery.state-store.url, and verify that the path is unique (for example, /user/livy/recovery_for_livy3).
21. For Livy-for-Spark3-2, open Configuration, search for livy.server.recovery.state-store.url, and set a different path (for example, /user/livy/recovery_for_livy3_1).
22. Click Save Changes for each service.
23. Restart both Livy services to apply the configuration.

Configure unique ZooKeeper key prefixes

24. Each independent Livy service must use a unique ZooKeeper key prefix (livy.server.zk.key-prefix) so that leader election namespaces do not conflict.
25. For Livy-for-Spark3-1, open Configuration, search for livy.server.zk.key-prefix, and verify that the prefix is unique (for example, livy3).
26. For Livy-for-Spark3-2, open Configuration, search for livy.server.zk.key-prefix, and set a different prefix (for example, livy3_1).
27. Click Save Changes for each service.
28. Restart both Livy services to apply the configuration.

# Livy load balancing with HAProxy

You can spread Livy sessions across several Livy for Spark 3 service instances by placing HAProxy in front of them. Each service instance keeps its own active/passive high availability, and HAProxy distributes new sessions between the instances and keeps each existing session on the instance that created it.

**Important:** This load-balancing setup is a validated deployment pattern, not a core Livy feature. HAProxy is third-party software that Cloudera does not build or maintain. If security vulnerabilities or other issues are found in HAProxy itself, you must update the package directly on your operating system. Cloudera only supports HAProxy when it is configured and used as described in this guide.

## How this differs from Livy active/passive high availability

In Livy active/passive high availability, ZooKeeper elects one leader per Livy service. Only the leader runs sessions; the other Livy Server roles of the same service are passive and redirect requests to the leader. Adding roles to a service improves availability, but does not add capacity.

The HAProxy pattern adds capacity by running several independent Livy for Spark 3 service instances, each with its own leader. HAProxy treats each service instance as one backend and sends traffic only to the current leader of that backend. Active/passive high availability still protects each backend.

## Architecture

### Figure 1: Livy load balancing with HAProxy and Keepalived

- **Clients** use a single URL, for example https://livy-lb.example.com:8888. The same URL is set as the Livy Load Balancer Address in Cloudera Manager.
- **Backends**: each Livy for Spark 3 service instance in Cloudera Manager is one HAProxy backend. Every server line in a backend is one Livy Server role of that service instance.
- **Health checks**: HAProxy polls the /ha-status endpoint of each Livy server. Only the leader returns HTTP 200, so HAProxy sends requests directly to the leader instead of relying on HTTP 307 redirects.
- **Gateway redundancy** (optional): two gateway hosts run HAProxy with identical configuration, and Keepalived moves a virtual IP (VIP) between them, so HAProxy is not a single point of failure.

### Sticky sessions

Livy session IDs are local to one service instance; backends do not share sessions. HAProxy therefore keeps each client on one backend with a cookie:

1. A request without a cookie, such as the request that creates a new session, goes to the default backend, which contains every Livy server.
2. HAProxy forwards the request to a leader and returns a SRV_ID cookie that identifies the server.
3. The client sends the cookie with every later request. HAProxy maps the cookie value to the backend that owns the session.

The client must store and resend the cookie, for example with a cookie jar. A client that reuses its cookie stays on the same backend; to place sessions on other backends, the client creates new sessions without the cookie. Listing sessions through HAProxy returns only the sessions of the backend that receives the request.

### Failover behavior

- **Livy leader failure**: ZooKeeper elects a new leader within the same service instance, and the next health check marks it as available. Sessions recover from the instance's own recovery state store.
- **Whole service instance unavailable**: HAProxy stops sending new sessions to that backend. Existing sessions of that instance are unavailable until the instance recovers; they do not move to another backend.
- **Gateway failure**: with Keepalived, the VIP moves to the other gateway within seconds. Requests that are in flight during the switch can fail and must be retried.

### Limitations with multiple Livy for Spark 3 services

Because this pattern adds Livy for Spark 3 services to the cluster, be aware of the following limitations:

- **Gateway role**: Cloudera Manager supports the Livy gateway role only on the first Livy for Spark 3 service. You cannot add the gateway role to the second or any later Livy for Spark 3 service. Add the gateway role to the first service only, and point clients to the load balancer URL instead of to an individual service instance.
- **Knox topologies**: Knox service discovery adds the Livy Server roles of every Livy for Spark 3 service to the cdp-proxy, cdp-proxy-api, and cdp-proxy-token topologies. To expose only the Livy servers that you intend, set the Livy URLs explicitly, as described in Configuring Knox topologies for multiple Livy for Spark 3 services.

## Preparing Livy service instances for HAProxy load balancing

Each Livy for Spark 3 service instance behind HAProxy needs its own unique settings and the same Livy Load Balancer Address, so that every instance accepts requests and Kerberos authentication for the load balancer hostname.

### Before you begin

- At least two Livy for Spark 3 service instances, each with unique values for livy.server.zk.key-prefix, livy.server.recovery.state-store.url, and, if roles of different instances share a host, livy.server.port. For instructions, see Adding independent Livy services with active/passive HA.
- Two or more Livy Server roles in each service instance, so that each instance has a passive role that can take over as leader.
- A Cloudera Manager release that provides the Livy Load Balancer Address property for Livy for Spark 3.
- The client-facing URL of the load balancer, for example https://livy-lb.example.com:8888. If you use Keepalived, this is the fully qualified domain name (FQDN) that resolves to the virtual IP.



**Note:** Cloudera Manager supports the Livy gateway role only on the first Livy for Spark 3 service. Do not plan to add the gateway role to the other service instances behind HAProxy.

### Procedure

Set the load balancer address

1. In Cloudera Manager, select the first Livy for Spark 3 service instance and click Configuration.
2. Search for Livy Load Balancer Address and enter the client-facing URL of the load balancer.

   Enter exactly one URI; a comma-separated list is not supported. The value must match the scheme, host, and port that HAProxy exposes. The property corresponds to the Livy configuration key livy.lb.uri.
3. Click Save Changes.
4. Repeat these steps for every other Livy for Spark 3 service instance behind HAProxy, using the same value.

Update Kerberos credentials

5. If the cluster uses Kerberos, stop all Livy Server roles of every service instance behind HAProxy.

6. Generate the missing Kerberos credentials in Cloudera Manager, or regenerate the keytabs of the Livy Server roles.

   When the Livy Load Balancer Address is set, Cloudera Manager adds the HTTP/*LOAD-BALANCER-FQDN@REALM* principal to the Livy keytabs. Do not create or merge keytabs manually with `kadmin` or `ktutil`.

Restart and verify

7. Start or restart all Livy Server roles. If other services were stopped, start ZooKeeper, HDFS, and YARN before the Livy Server roles.

   Until the roles restart, Cloudera Manager shows the Livy services with a stale configuration. For more information, see Stale Configurations.

8. On each Livy Server host, verify that the Livy process received the load balancer address:

   ```
   pid=$(pgrep -f 'org.apache.livy.server.LivyServer' | head -1)
   tr '\0' '\n' < /proc/$pid/environ | grep LIVY_LOAD_BALANCER_URL
   ```

   The output must show the same URL on every host, for example LIVY_LOAD_BALANCER_URL=https://livy-lb.example.com:8888. The process environment is authoritative; the property can be absent from the generated livy.conf file.

   **Note:** If the variable is missing after a restart, the Livy for Spark 3 CSD on the cluster might not export the property. As a workaround, add LIVY_LOAD_BALANCER_URL=*LOAD-BALANCER-URL* to the Livy Server environment safety valve of each service instance and restart the roles.

9. If the cluster uses Kerberos, verify on each Livy Server host that the keytab contains the load balancer principal:

   ```
   pid=$(pgrep -f 'org.apache.livy.server.LivyServer' | head -1)
   proc_dir=$(readlink -f /proc/$pid/cwd)
   klist -kt "$proc_dir/livy_for_spark3.keytab" | grep -i HTTP/
   ```

   The output must include HTTP/*LOAD-BALANCER-FQDN@REALM*.

### Results

Every Livy for Spark 3 service instance accepts requests that clients send to the load balancer URL. Next, enable the /ha-status endpoint so that HAProxy can detect the leader of each instance.

## Enabling the Livy /ha-status endpoint

The /ha-status endpoint reports whether a Livy server is the leader of its service instance. HAProxy uses it as a health check, so that requests go to the leader instead of to a passive server that answers with an HTTP 307 redirect.

### About this task

When the endpoint is enabled, GET /ha-status returns the following responses:

| Livy server state | HTTP status | Response body |
|---|---|---|
| Leader | 200 | {"status":"leader"} |
| Passive (follower) | 503 | {"status":"follower"} |

The endpoint is disabled by default. Livy exposes it only when high availability recovery is active, that is, when livy.server.recovery.mode is ha. Cloudera Manager sets this mode when a Livy service has more than one Livy Server role. If you enable the endpoint while the recovery mode is not ha, Livy logs a warning and does not expose the endpoint.

The endpoint does not require authentication, so that load balancers can probe it without Kerberos credentials. It returns only the leader or follower state of the server.

### Procedure

1. In Cloudera Manager, select a Livy for Spark 3 service instance and click Configuration.
2. Search for Livy Server for Spark 3 Advanced Configuration Snippet (Safety Valve) for livy3-conf/livy.conf and add the following property:

   ```
   livy.server.ha-status.enabled=true
   ```
3. Click Save Changes.
4. Repeat these steps for every Livy for Spark 3 service instance behind HAProxy.
5. Restart the Livy Server roles of each service instance.
6. Verify the endpoint on each Livy server:

   ```
   curl -s -o - -w ' %{http_code}\n' --cacert CA-CERTIFICATE.PEM \
     https://LIVY-HOST.EXAMPLE.COM:LIVY-PORT/ha-status
   ```

   Exactly one server in each service instance returns 200; every other server in the same instance returns 503. If a server returns another status, such as 401 or 404, the endpoint is not enabled on that server or the service instance does not use ha recovery mode.

## Configuring HAProxy for Livy

Configure HAProxy with one backend per Livy for Spark 3 service instance, cookie-based sticky sessions, and /ha-status health checks, and then verify that sessions stay on the backend that created them.

### Before you begin

**Important:** This load-balancing setup is a validated deployment pattern, not a core Livy feature. HAProxy is third-party software that Cloudera does not build or maintain. If security vulnerabilities or other issues are found in HAProxy itself, you must update the package directly on your operating system. Cloudera only supports HAProxy when it is configured and used as described in this guide.

- The Livy for Spark 3 service instances are prepared and the /ha-status endpoint is enabled on every Livy server.
- One or two gateway hosts outside the Livy roles, with HAProxy installed from the operating system packages. HAProxy is not managed by Cloudera Manager.
- A TLS certificate and private key for the load balancer FQDN in one PEM file, and the CA certificate that signed the Livy server certificates.

### Procedure

1. On the gateway host, edit /etc/haproxy/haproxy.cfg. The following example configures two Livy for Spark 3 service instances with two Livy Server roles each:

   ```
   global
       log         127.0.0.1 local2
       chroot      /var/lib/haproxy
       maxconn     4000
       user        haproxy
       group       haproxy
       daemon

   defaults
       mode                    http
       log                     global
       option                  httplog
       option                  http-server-close
       option                  forwardfor       except 127.0.0.0/8
       option                  redispatch
       timeout http-request    10s
       timeout connect         10s
   ```

```
    timeout client          1m
    timeout server          1m

frontend livy_gateway
    bind *:8888 ssl crt /path/to/gateway.pem

    acl cookie_livy1 cook(SRV_ID) -m str s1 s2
    use_backend livy1 if cookie_livy1
    acl cookie_livy2 cook(SRV_ID) -m str s3 s4
    use_backend livy2 if cookie_livy2

    default_backend livy_default
backend livy_default
    balance roundrobin
    cookie SRV_ID insert indirect nocache
    option httpchk GET /ha-status
    http-check expect status 200
    option log-health-checks
    retries 3
    server livy1-a livy1-a.example.com:28998 check inter 5s fall 3 rise 2
 cookie s1 ssl ca-file /path/to/ca.pem
    server livy1-b livy1-b.example.com:28998 check inter 5s fall 3 rise 2
 cookie s2 ssl ca-file /path/to/ca.pem
    server livy2-a livy2-a.example.com:28998 check inter 5s fall 3 rise 2
 cookie s3 ssl ca-file /path/to/ca.pem
    server livy2-b livy2-b.example.com:28998 check inter 5s fall 3 rise 2
 cookie s4 ssl ca-file /path/to/ca.pem

backend livy1
    balance roundrobin
    cookie SRV_ID insert indirect nocache
    option httpchk GET /ha-status
    http-check expect status 200
    option log-health-checks
    retries 3
    server livy1-a livy1-a.example.com:28998 check inter 5s fall 3 rise 2
 cookie s1 ssl ca-file /path/to/ca.pem
    server livy1-b livy1-b.example.com:28998 check inter 5s fall 3 rise 2
 cookie s2 ssl ca-file /path/to/ca.pem

backend livy2
    balance roundrobin
    cookie SRV_ID insert indirect nocache
    option httpchk GET /ha-status
    http-check expect status 200
    option log-health-checks
    retries 3
    server livy2-a livy2-a.example.com:28998 check inter 5s fall 3 rise 2
 cookie s3 ssl ca-file /path/to/ca.pem
    server livy2-b livy2-b.example.com:28998 check inter 5s fall 3 rise 2
 cookie s4 ssl ca-file /path/to/ca.pem
```

Replace the host names, ports, and file paths with the values for your cluster. Each Livy Server role needs one server line with a unique cookie value, and the cookie values in each acl line must match the server lines of the corresponding backend. If roles of different service instances share a host, use the livy.server.port value of each instance.

The key directives work as follows:

| **Directive** | **Purpose** |
|---|---|
| bind *:8888 ssl crt | Accepts client connections over TLS on the port of the Livy Load Balancer Address. |

| Directive | Purpose |
|---|---|
| acl cookie_livy1 cook(SRV_ID) -m str s1 s2 and use_backend livy1 if cookie_livy1 | Sends a client whose SRV_ID cookie is s1 or s2 to the livy1 backend. HAProxy evaluates these rules first, so returning clients always follow their cookie. |
| default_backend livy_default | Sends clients without a matching cookie, such as clients that create a new session, to a backend that contains every Livy server. |
| cookie SRV_ID insert indirect nocache | Adds the SRV_ID cookie to responses so that the client stays on the same server. |
| option httpchk GET /ha-status and http-check expect status 200 | Marks only the leader of each service instance as available. |
| check inter 5s fall 3 rise 2 | Runs the health check every 5 seconds, marks a server unavailable after 3 failed checks, and available again after 2 successful checks. |

For connection limits and timeouts for many concurrent sessions, see Livy and HAProxy tuning reference.

2. Check the configuration file for errors:

```
haproxy -c -f /etc/haproxy/haproxy.cfg
```

3. Enable and start HAProxy:

```
systemctl enable --now haproxy
```

After later changes to the configuration file, apply them with `systemctl reload haproxy`.

4. If you use two gateway hosts, copy the same haproxy.cfg file to the second gateway host and repeat the previous steps there.
5. Verify sticky sessions through the load balancer. Obtain a Kerberos ticket for a user that is allowed to use Livy, and then create a session with a cookie file:

```
curl --location --location-trusted -k -X POST --negotiate -u : \
  -c cookies.dat -b cookies.dat \
  -H "Content-Type: application/json" \
  -d '{"kind": "spark"}' \
  https://livy-lb.example.com:8888/sessions
```

Use the same cookie file for every request in the test.

6. Submit a statement and poll for its result, replacing *SESSION-ID* with the ID from the previous response:

```
curl --location --location-trusted -k -X POST --negotiate -u : \
  -c cookies.dat -b cookies.dat \
  -H "Content-Type: application/json" \
  -d '{"code": "2+3"}' \
  https://livy-lb.example.com:8888/sessions/SESSION-ID/statements
curl --location --location-trusted -k --negotiate -u : \
  -c cookies.dat -b cookies.dat \
  https://livy-lb.example.com:8888/sessions/SESSION-ID/statements/0
```

7. Check the cookie file.

The file contains an SRV_ID entry for the load balancer host, for example s1. This value identifies the server line, and therefore the backend, that owns the session.

### Results

The session is created, the statement reaches the available state, and all requests for the session use the same SRV_ID cookie. To place a session on another backend, create it with a new cookie file.

## Configuring Knox topologies for multiple Livy for Spark 3 services

When a cluster has more than one Livy for Spark 3 service, Knox service discovery adds the Livy Server roles of every service to the Knox topologies. Set the Livy URLs explicitly in Knox Simplified Topology Management, so that Knox exposes only the Livy servers that you intend.

### About this task

Knox discovers services through Cloudera Manager. For the LIVY_FOR_SPARK3 service, discovery collects every Livy Server role of every Livy for Spark 3 service in the cluster and merges them into one service entry in the following topologies:

- cdp-proxy
- cdp-proxy-api
- cdp-proxy-token

For example, with three Livy for Spark 3 services of three Livy Server roles each, the LIVY_FOR_SPARK3 entry of each topology lists nine URLs instead of the URLs of one service. Knox then routes Livy requests to roles that you did not intend to expose through Knox, and the Livy UI can be empty or show sessions from the wrong service.

When you set the LIVY_FOR_SPARK3 URLs explicitly in Knox Simplified Topology Management, the explicit URLs override the URLs that Knox discovers. Typically, you expose the Livy Server roles of the first Livy for Spark 3 service, which is also the only service that supports the Livy gateway role.



**Note:** Clients that use HAProxy connect to the load balancer URL, not to Knox. The Knox configuration in this topic controls only the Livy access that Knox provides, such as the Livy UI on the Knox Homepage.

### Before you begin

- The Livy Server hosts and REST ports of the Livy for Spark 3 service that you want to expose through Knox. The port is the value of livy.server.port for that service.
- Permission to change the Knox configuration in Cloudera Manager.

### Procedure

1. In Cloudera Manager, select the Knox service and click Configuration.
2. Search for Knox Simplified Topology Management.
3. In Knox Simplified Topology Management - cdp-proxy, add one LIVY_FOR_SPARK3 entry for each Livy Server role that you want to expose.
   For example, to expose the two Livy Server roles of the first Livy for Spark 3 service:

   ```
   LIVY_FOR_SPARK3:url=https://livy-primary-a.example.com:28998
   LIVY_FOR_SPARK3:url=https://livy-primary-b.example.com:28998
   ```

4. Add the same entries to Knox Simplified Topology Management - cdp-proxy-api and Knox Simplified Topology Management - cdp-proxy-token.
5. Click Save Changes.
6. Click the Refresh needed indicator next to the Knox service, and wait until the refresh finishes.
7. Verify that each topology lists only the URLs that you set.

   Open the topology through the Knox Admin API, for example https://knox.example.com:8443/gateway/admin/api/v1/topologies/cdp-proxy, and check that the LIVY_FOR_SPARK3 service contains one URL for each entry that you added. Repeat the check for cdp-proxy-api and cdp-proxy-token.
8. Verify that the Livy UI opens through Knox.

   Open the Livy tile on the Knox Homepage, or go to https://knox.example.com:8443/gateway/cdp-proxy/livy_for_spark3/ui, and check that the Sessions and Batches pages list the sessions of the service that you exposed.

### Results

Knox routes Livy requests only to the Livy servers that you set, regardless of how many Livy for Spark 3 services the cluster has.

### What to do next

Custom Knox service definitions for Livy

If you maintain a custom Knox service definition for Livy, for example to show a separate Knox Homepage tile for each Livy for Spark 3 service, follow these guidelines:

- In service.xml, include the metadata element with the context, description, short description, and type of the service. The Knox Homepage uses this metadata to display the service.
- In rewrite.xml, keep the rewrite rules minimal. The Livy UI uses relative paths, and rules that rewrite HTML, CSS, or JavaScript content can break the Sessions and Batches pages.

## Making HAProxy highly available with Keepalived

To remove HAProxy as a single point of failure, run HAProxy on two gateway hosts and use Keepalived to move a virtual IP (VIP) between them. Clients keep using the same load balancer URL when the VIP moves.

### About this task

Keepalived implements the Virtual Router Redundancy Protocol (VRRP). The gateway with the higher priority holds the VIP. A tracking script checks whether HAProxy is running; if HAProxy stops, Keepalived lowers the priority of that gateway and the VIP moves to the other gateway, typically within 3 to 15 seconds.

### Before you begin

- Two gateway hosts on the same network segment, each running HAProxy with the same haproxy.cfg file.
- An unused IP address on that segment for the VIP, and a VRRP router ID (1 to 255) that no other VRRP group on the segment uses.
- A DNS A record that resolves the load balancer FQDN, for example livy-lb.example.com, to the VIP. Use this FQDN in the Livy Load Balancer Address.
- Network rules that allow VRRP traffic (IP protocol 112) between the two gateway hosts.

### Procedure

1. On both gateway hosts, install and enable Keepalived from the operating system packages, for example:

```
yum install -y keepalived
systemctl enable keepalived
```

2. On the first gateway host, create /etc/keepalived/keepalived.conf with the following content:

```
global_defs {
    script_user root
    enable_script_security
}
vrrp_script chk_haproxy {
    script "/usr/bin/systemctl is-active haproxy"
    interval 2
    weight -30
    fall 2
    rise 2
}

vrrp_instance VI_LIVY {
    state MASTER
    interface NIC
```

```
    virtual_router_id ROUTER-ID
    priority 101
    advert_int 1
    preempt_delay 5

    authentication {
        auth_type PASS
        auth_pass VRRP-PASSWORD
    }

    virtual_ipaddress {
        VIP/PREFIX-LENGTH dev NIC
    }

    track_script {
        chk_haproxy
    }
}
```

Replace *NIC* with the network interface that carries the VIP, *ROUTER-ID* with the VRRP router ID, *VIP* and *PREFIX-LENGTH* with the VIP and its network prefix length, and *VRRP-PASSWORD* with a shared password.

The weight -30 setting lowers the priority of this gateway from 101 to 71 when HAProxy is not running, which is below the priority of the second gateway.

3. On the second gateway host, create the same file with two changes: set state BACKUP and priority 100.

   All other values, including virtual_router_id, auth_pass, and the VIP, must be identical on both hosts.

4. Start Keepalived on both gateway hosts:

```
systemctl restart keepalived
systemctl is-active keepalived
```

5. Verify that only the first gateway host holds the VIP. Run the following command on each gateway host:

```
ip -4 addr show NIC | grep VIP
```

   The VIP appears on the first gateway host and not on the second.

6. To monitor HAProxy through the VIP, add a statistics listener to haproxy.cfg on both gateway hosts and reload HAProxy:

```
listen admin
    bind *:8000
    stats enable
    stats uri /stats
```

   You can then check the active gateway with curl http://*VIP*:8000/stats;csv.

7. Test failover. On the first gateway host, stop HAProxy:

```
systemctl stop haproxy
```

   Within a few seconds, the VIP moves to the second gateway host. While the VIP moves, requests through the load balancer can fail, for example with HTTP 504. This is expected. Do not create sessions during the transition; retry requests when the load balancer responds again.

8. Start HAProxy again on the first gateway host:

```
systemctl start haproxy
```

   After the preemption delay, the VIP returns to the first gateway host. Existing sessions and SRV_ID cookies remain valid, because both gateways use the same HAProxy configuration.

### What to do next

Use the following table to troubleshoot common problems:

| Symptom | Action |
|---|---|
| The VIP is present on both gateway hosts. | Check that the priorities differ (101 and 100), that VRRP traffic can pass between the hosts, and that no other VRRP group on the network segment uses the same virtual_router_id. |
| Requests through the VIP fail with HTTP 401 after a failover. | Verify that LIVY_LOAD_BALANCER_URL is set in every Livy Server process, and restart the roles where it is missing. |
| Requests through the VIP fail with HTTP 403. | Verify that the keytab of every Livy Server role contains the HTTP/*LOAD-BALANCER-FQDN* principal. |
| Listing sessions through the VIP does not return all sessions. | This is expected. Each request reaches one backend, and each backend lists only its own sessions. |

To review recent events, run journalctl -u keepalived -n 50 --no-pager and journalctl -u haproxy -n 50 --no-pager.

## Finding the Livy server that owns a session

Behind HAProxy, all Livy servers share one URL. To connect another client, such as Beeline, to an existing interactive session, find the Livy server that owns the session from the livyHost field and map it to the matching SRV_ID cookie.

### About this task

For interactive sessions, Livy adds a livyHost entry to the appInfo map of the session. The entry contains the host name of the Livy server that answered the request, which is the leader of the backend that owns the session.

### Procedure

1. Retrieve the session through the load balancer, using the cookie file of the client that created the session:

   ```
   curl -sk --negotiate -u : -c cookies.dat -b cookies.dat \
     https://livy-lb.example.com:8888/sessions/SESSION-ID | jq '.appInfo'
   ```

   The output contains the livyHost entry, for example:

   ```
   {
     "driverLogUrl": "...",
     "sparkUiUrl": "...",
     "livyHost": "livy1-a.example.com"
   }
   ```

2. Find the server line for that host in haproxy.cfg and note its cookie value.

   For example, the line server livy1-a livy1-a.example.com:28998 ... cookie s1 means that the session belongs to SRV_ID=s1. Depending on the host configuration, livyHost can contain a short host name or an FQDN. Client tools that connect often can keep the host-to-cookie mapping in their own configuration.

3. Send the cookie with every request that must reach the session. For example, connect Beeline to the session through the Livy Thrift server:

   ```
   beeline -u "jdbc:hive2://livy-lb.example.com:HAPROXY-THRIFT-
   PORT/;principal=HTTP/livy-lb.example.com@REALM;ssl=true;sslTrustSt
   ore=TRUSTSTORE.JKS;trustStorePassword=TRUSTSTORE-PASSWORD;trustStoreType=
   jks;transportMode=http;httpHeader.Cookie=SRV_ID=s1;httpPath=cliservice?l
   ivy.session.name=SESSION-NAME"
   ```

For details about the connection parameters, see Connecting to the Livy Thrift server through HAProxy.

### Results

Requests that carry the SRV_ID cookie reach the backend that owns the session. If the leader of that backend changes, HAProxy sends the requests to the new leader in the same backend.

## Livy session capacity and YARN queues

Adding Livy service instances behind HAProxy increases the number of sessions that Livy can manage, but every session also needs YARN resources. The YARN queue that runs the sessions often limits concurrency before Livy does.

### Application Master resource limit

Each interactive Livy session runs a Spark driver in a YARN Application Master (AM) container. The Capacity Scheduler limits the share of a queue that AM containers can use with yarn.scheduler.capacity.*QUEUE-PATH*.maximum-am-resource-percent. When AM containers reach this limit, new sessions stay in the ACCEPTED state in YARN and in the starting state in Livy, even if the cluster has free memory.

For example, if the sessions run in a queue with 16 GB of resources and the AM limit of the queue is 0.2, AM containers can use 3.2 GB in total.

### Container size of a session

YARN allocates more memory than the requested driver memory:

1. Spark adds memory overhead to the driver memory. The overhead is at least 384 MB.
2. YARN rounds the request up to a multiple of yarn.scheduler.minimum-allocation-mb.

For example, a session with 1024 MB of driver memory requests 1024 MB + 384 MB = 1408 MB. If yarn.scheduler.minimum-allocation-mb is 1024 MB, YARN allocates a 2048 MB container. With the 3.2 GB AM limit from the previous example, only one such session can start; the second one would raise AM usage to 4 GB.

To estimate the number of concurrent sessions in a queue, divide the AM resources of the queue by the rounded container size of one session.

### Resource calculator

The Capacity Scheduler resource calculator determines which resources YARN counts:

- DefaultResourceCalculator counts only memory. More sessions can start, but drivers can stay in the starting state when the hosts run out of CPU.
- DominantResourceCalculator counts memory and vCores. A session starts only when both are available, which avoids session start timeouts caused by CPU contention.

### Using a dedicated queue for Livy sessions

Cloudera recommends running Livy sessions in a dedicated YARN queue, so that interactive sessions do not compete with batch jobs and the limits are predictable. Size the capacity, maximum-applications, and maximum-am-resource-percent properties of that queue for the number of concurrent sessions that you expect across all Livy service instances.

The following Capacity Scheduler properties show an example with a child queue named livy:

```
<property>
  <name>yarn.scheduler.capacity.root.queues</name>
  <value>default,livy</value>
</property>
<property>
  <name>yarn.scheduler.capacity.root.default.capacity</name>
  <value>20</value>
```

```
</property>
<property>
  <name>yarn.scheduler.capacity.root.livy.capacity</name>
  <value>80</value>
</property>
<property>
  <name>yarn.scheduler.capacity.root.livy.maximum-applications</name>
  <value>1200</value>
</property>
<property>
  <name>yarn.scheduler.capacity.root.livy.maximum-am-resource-percent</name>
  <value>0.8</value>
</property>
```

The capacities of the child queues of a parent queue must add up to 100%, and all queues at the same level must use the same capacity mode. Apply the properties in the Capacity Scheduler configuration snippet of the YARN service or with the YARN Queue Manager, and then restart or refresh the ResourceManager as Cloudera Manager prompts. For more information, see Setting Application-Master resource-limit for a specific queue.

### How Livy selects the queue

Livy submits a session to the first queue that is set in the following order:

1. The queue field of the session or batch request.
2. The livy.spark.yarn.queue property in the Livy server configuration, which sets a default queue for all sessions of a Livy service instance.
3. The spark.yarn.queue property in the conf map of the request.
4. The default queue of the cluster.

**Note:** When livy.spark.yarn.queue is set, it overrides a spark.yarn.queue value in the conf map of a request. To select another queue for one session, use the queue field of the request.

Set livy.spark.yarn.queue to the same value on every Livy service instance behind HAProxy, and use the full queue path that the ResourceManager reports, for example root.livy.

## Livy and HAProxy tuning reference

Use these settings as a starting point when many interactive sessions are created at the same time through HAProxy. Adjust the values to the size of your cluster and validate them with your own workload.

### HAProxy connection limits and timeouts

The connection limits form a hierarchy: global, defaults, frontend, backend, and server. Low limits cause clients to stall while they wait for a connection. Long timeouts allow slow session creation and deletion to complete.

| Setting | Example value | Where |
|---|---|---|
| maxconn | 50000 | global section |
| maxconn | 5000 | defaults section |
| maxconn | 10000 | frontend section |
| maxconn | 5000 | Default backend that contains all servers |
| maxconn | 2000 | Each backend for a service instance |
| maxconn | 800 | Each server line |
| timeout connect | 30s | defaults section |

| Setting | Example value | Where |
|---|---|---|
| timeout client, timeout server, time out http-request | 600s | defaults section |
| timeout http-keep-alive | 60s | defaults section |
| timeout check | 5s | defaults section |

Use GET /ha-status as the health check of every backend. Do not use GET /sessions as a health check, because it does not identify the leader of a service instance.

### Why leader-only routing matters at scale

A passive Livy server answers with an HTTP 307 redirect to the leader. Without /ha-status health checks, a round-robin load balancer sends part of the requests to passive servers. Each redirect adds a request and a Kerberos authentication, can break SRV_ID stickiness, and can cause HTTP 401 or HTTP 504 errors. These failures are rare with few clients but become frequent when hundreds of sessions start at the same time.

### Livy server properties

Add the following properties to Livy Server for Spark 3 Advanced Configuration Snippet (Safety Valve) for livy3-conf/livy.conf on every Livy for Spark 3 service instance behind HAProxy:

```
livy.server.ha-status.enabled=true
livy.server.yarn.poll-interval=5s
livy.server.session.timeout-check=true
livy.server.session.timeout-check.skip-busy=true
livy.server.session.timeout=3600s
livy.server.session.max-creation=700
livy.server.session.state-retain.sec=600
livy.server.yarn.app-lookup-timeout=600s
livy.spark.yarn.queue=LIVY-YARN-QUEUE
livy.rsc.server.connect.timeout=1200s
livy.rsc.client.connect.timeout=600s
livy.rsc.launcher.port.range=30000~60000
```

| Property | Default | Example | Purpose |
|---|---|---|---|
| livy.server.ha-status.enabled | false | true | Exposes the /ha-status endpoint for HAProxy health checks. |
| livy.server.session.timeout-check.skip-busy | false | true | Does not time out sessions that are running statements. |
| livy.server.session.max-creation | 100 | 700 | Maximum number of sessions that one Livy server creates at the same time. Set it above the expected number of concurrent session creations. |
| livy.server.yarn.app-lookup-timeout | 120s | 600s | Time that Livy waits for YARN to report the application of a new session. A busy queue needs more time. |
| livy.spark.yarn.queue | None | Queue path | Default YARN queue for the sessions of the service instance. See Livy session capacity and YARN queues. |
| livy.rsc.server.connect.timeout | 90s | 1200s | Time that Livy waits for a new Spark driver to connect back. Drivers start more slowly under load. |

| Property | Default | Example | Purpose |
|---|---|---|---|
| livy.rsc.client.connect.timeout | 10s | 600s | Time that Livy waits to connect to a Spark driver. |
| livy.rsc.launcher.port.range | 10000~10010 | 30000~60000 | Ports that Livy uses to communicate with Spark drivers. A narrow range causes BindException errors when many sessions start. The setting applies to the whole Livy server; do not rely on overriding it per session. |

The livy.rsc.* properties apply only to interactive sessions; they do not affect batch jobs. Livy ignores property names that it does not recognize without reporting an error, so check the spelling of each property.

### Related service settings

| Setting | Where to set it | Purpose |
|---|---|---|
| Livy Server Java heap size: 4 GB | Livy for Spark 3 service configuration in Cloudera Manager | Reduces garbage collection pauses that can expire the ZooKeeper session of the Livy server and cause leader changes. |
| maxSessionTimeout=300000 | ZooKeeper zoo.cfg safety valve | Allows longer ZooKeeper sessions, which prevents leader flapping under load. |
| spark.yarn.am.waitTime=600s | Spark 3 spark-defaults.conf safety valve | Gives the Spark driver more time to initialize. Without it, slow drivers fail with Application Master exit code 13. Setting this property in livy.conf has no effect. |
| kdc_tcp_listen_backlog = 128 in the [kdcdefaults] section | kdc.conf on each MIT Kerberos KDC of the realm | A small TCP listen backlog causes LoginException: Connection reset errors when many clients authenticate at the same time. Restart the KDC after the change. Clients that authenticate in large numbers can also set udp_preference_limit = 1 in krb5.conf to use TCP. |

After you change these settings, restart the services in the following order: ZooKeeper, Spark 3, and Livy for Spark 3. If you changed the KDC configuration, restart the KDC. Then reload HAProxy and verify that the HAProxy statistics show one available server per backend.

### Client queue and retry settings

Clients should connect through the load balancer URL and retry briefly when a request fails during a leader or gateway failover. For example, a SparkMagic config.json file can set the queue and a retry policy:

```
{
  "session_configs_defaults": {
    "queue": "LIVY-YARN-QUEUE"
  },
  "retry_policy": "configurable",
  "retry_seconds_to_sleep_list": [0.2, 0.5, 1, 3, 5],
  "configurable_retry_policy_max_retries": 8
}
```

### Reference results

In Cloudera testing with six Livy service instances behind HAProxy, 400 concurrent interactive sessions were created, used, and deleted without errors after all of the settings on this page were applied. These results show the effect of the tuning; they are not a performance guarantee.

# Configuring TLS/SSL encryption manually for Livy

You can enable TLS manually for the Apache Livy Server.

## About this task

In Cloudera Runtime 7.3.2.10000 SP1, Livy Server TLS inherits protocol and cipher settings from Cloudera Manager when you use global TLS configuration. Cloudera recommends TLS 1.3 when your JDK and cluster policy allow it. For related Spark and Livy encryption defaults, see Encryption in transit for Spark and Livy in Cloudera Runtime 7.3.2.10000 SP1.

## Procedure

1. In Cloudera Manager, select the Livy service from the Clusters drop-down menu.
2. In the Configuration tab, enter tls into the search box.
3. Edit the following property fields as needed for your cluster and environment:

| Property | Description |
|---|---|
| Gateway TLS/SSL Trust Store File | The location on disk of the trust store, in .jks format, used to confirm the authenticity of TLS/SSL servers that Gateway might connect to. This trust store must contain the certificate(s) used to sign the service(s) connected to. If this parameter is not provided, the default list of well-known certificate authorities is used instead. |
| Gateway TLS/SSL Trust Store Password | The password for the Gateway TLS/SSL Trust Store File. This password is not required to access the trust store; this field can be left blank. This password provides optional integrity checking of the file. The contents of trust stores are certificates, and certificates are public information. |
| Enable TLS/SSL for Livy Server | Encrypt communication between clients and Livy Server using Transport Layer Security (TLS) (formerly known as Secure Socket Layer (SSL)). |
| TLS protocol versions | When the tls-advanced-control feature flag is enabled in Cloudera Manager, Livy Server TLS protocol settings can inherit from global TLS configuration (supported_tls_versions or adv_supported_tls_versions). Include TLSv1.3 to use TLS 1.3. When the flag is disabled, Cloudera Manager uses the legacy TLS 1.2 configuration for the service. |
| Livy Server TLS/SSL Server JKS Keystore File Location | The path to the TLS/SSL keystore file containing the server certificate and private key used for TLS/SSL. Used when Livy Server is acting as a TLS/SSL server. The keystore must be in JKS format. |
| Livy Server TLS/SSL Server JKS Keystore File Password | The password for the Livy Server JKS keystore file. |

4. Click Save Changes.
5. Restart the Livy service.

# Using Spark Hive Warehouse and HBase Connector Client .jar files with Livy

This section describes how to use Spark Hive Warehouse Connector (HWC) and HBase-Spark connector client .jar files with Livy. These steps are required to ensure token acquisition and avoid authentication errors.

Use the following steps to use Spark HWC and HBase-Spark client .jar files with Livy:

1. Copy the applicable HWC or HBase-Spark .jar files to the Livy server node and add these folders to the livy.file.local-dir-whitelist property in the livy.conf file.
2. Add the required Hive and HBase configurations in the Spark client configuration folder:

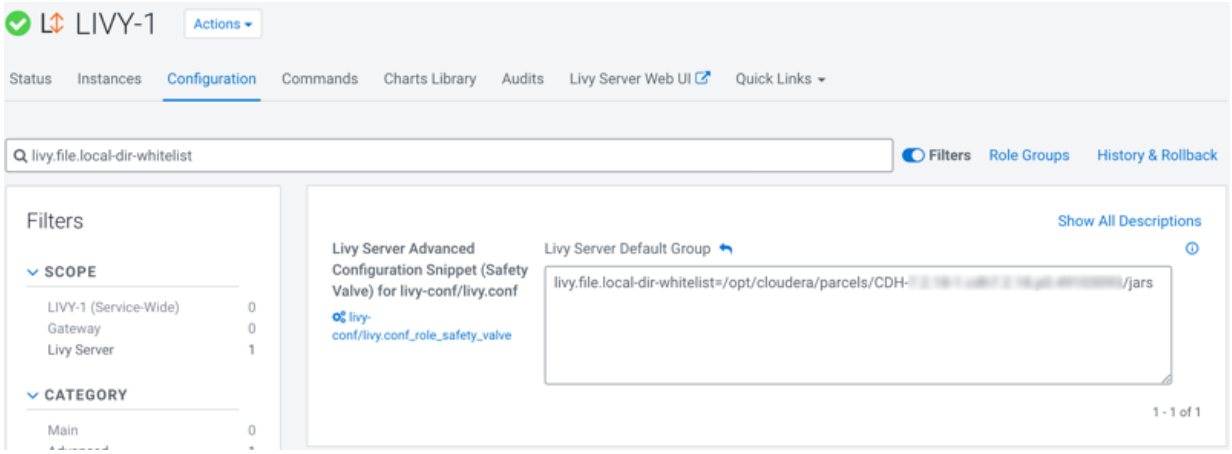
- Hive: /etc/spark3/conf/hive-site.xml
- HBase: /etc/spark3/conf/hbase-site.xml

Or add the required configurations using the conf field in the session creation request. This is equivalent to using "--conf" in spark-submit.

3. Reference these local .jar files in the session creation request using the file:/// URI format.

HWC Example

1. In Cloudera Manager, go to Clusters Livy .
2. Click the Configuration tab and search for the "Livy Server Advanced Configuration Snippet (Safety Valve) for livy-conf/livy.conf" property.
3. Add the /opt/cloudera/parcels/CDH-*[*** CLOUDERA VERSION ***]*/jars folder to the livy.file.local-dir-whit elist property in the livy.conf file.




4. Save the changes and restart the Livy service.
5. Log in to the Zeppelin Server Web UI and restart the Livy interpreter.
6. When running the Zeppelin Livy interpreter, reference the HWC .jar file as shown below.

- For Spark 3:

```
%livy2.conf
livy.spark.jars file:///opt/cloudera/parcels/CDH-[*** CLOUDERA VERSION
 ***]/jars/hive-warehouse-connector-spark3-assembly-[*** PARCEL VERSION
 ***].jar
```

HBase-Spark Connector Example

1. When running the Zeppelin Livy interpreter, reference the following HBase .jar files as shown below. Note that some of these .jar files have 644/root permissions, and therefore may throw an exception. If this happens, you may need to change the permissions of the applicable .jar files on the Livy node.

```
%livy.conf
livy.spark.jars file:///opt/cloudera/parcels/CDH-[*** CLOUDERA VERSION
 ***]/jars/hbase-spark3-[*** PARCEL VERSION ***].jar
file:///opt/cloudera/parcels/CDH-[*** CLOUDERA VERSION ***]/jars/hbase-
shaded-protobuf-[*** PARCEL VERSION ***].jar,
file:///opt/cloudera/parcels/CDH-[*** CLOUDERA VERSION ***]/jars/hbase-
shaded-miscellaneous-[*** PARCEL VERSION ***].jar,
file:///opt/cloudera/parcels/CDH-[*** CLOUDERA VERSION ***]/jars/hbase-
protocol-shaded-[*** PARCEL VERSION ***].jar,
file:///opt/cloudera/parcels/CDH-[*** CLOUDERA VERSION ***]/jars/hbase-
shaded-netty-[*** PARCEL VERSION ***].jar,
file:///opt/cloudera/parcels/CDH-[*** CLOUDERA VERSION ***]/jars/hbase-
shaded-client-[*** PARCEL VERSION ***].jar,
```

```
file:///opt/cloudera/parcels/CDH-[*** CLOUDERA VERSION ***]/jars/hbase-
shaded-mapreduce-[*** PARCEL VERSION ***].jar,
file:///opt/cloudera/parcels/CDH-[*** CLOUDERA VERSION ***]/jars/hbase-
common-[*** PARCEL VERSION ***].jar,
file:///opt/cloudera/parcels/CDH-[*** CLOUDERA VERSION ***]/jars/hbase-
server-[*** PARCEL VERSION ***].jar,
file:///opt/cloudera/parcels/CDH-[*** CLOUDERA VERSION ***]/jars/hbase-
client-[*** PARCEL VERSION ***].jar,
file:///opt/cloudera/parcels/CDH-[*** CLOUDERA VERSION ***]/jars/hbase-
protocol-[*** PARCEL VERSION ***].jar,
file:///opt/cloudera/parcels/CDH-[*** CLOUDERA VERSION ***]/jars/hbase-
mapreduce-[*** PARCEL VERSION ***].jar,
ile:///opt/cloudera/parcels/CDH-[*** CLOUDERA VERSION ***]/jars/guava-32.1
.3-jre.jar
```

**Note:** The references to HBase-Spark connector jar file are included because HBase-Spark connector was used in this example. They are not required for token acquisition.

# Using Spark History Servers with high availability

You can configure the load balancer for Spark History Server to ensure high availability, so that users can access and use the Spark History Server UI without any disruption. Learn how you can configure the load balancer for Spark History Server and the limitations associated with it.

You can access the Spark History Server for your Spark cluster from the Cloudera Management Console interface. The Spark History Server has two main functions:

- Reads the Spark event logs from the storage and displays them on the Spark History Server's user interface.
- Cleans the old Spark event log files.

Introducing high availability enables an extra load balancing layer so that users can see the Spark History Server's user interface without failure or disruption when there are two Spark History Servers in a cluster. Additionally, users are able to use the rolling restart feature for Spark History Server.

There are three supported ways to configure the load balancer for Spark History Server:

- Using an external load balancer for example, HAProxy.
- Using an internal load balancer which requires Apache Knox Gateway.
- Using multiple Apache Knox Gateways and external load balancers, for example, HAProxy.

.

## Limitations for Spark History Server with high availability

You must be aware of a limitation related to how Spark History Server with high availability operates in Cloudera Manager.

- The second Spark History Server in the cluster does not clean the Spark event logs. Cleaning Spark event logs is automatically disabled from the Custom Service Descriptor. The second server can only read logs. This limitation ensures that two Spark History Servers do not try to delete the same files. If the first Spark History Server is down, the second one does not take over the cleaner task. This is not a critical issue because if the first Spark History Server starts again, it will delete those old Spark event logs. The default event log cleaner interval (spar k.history.fs.cleaner.interval) is 1 day in Cloudera Manager which means that the first Spark History Server only deletes the old logs once per day by default.

# Configuring high availability for Spark History Server with an external load balancer

Learn how to configure high availability for Spark History Server using an external load balancer such as HAProxy. The authentication method used is SPNEGO (Kerberos) which requires an external load balancer.

## Before you begin

Download and configure an external load balancer. If you do not have a load-balancing proxy, you can experiment your configurations with HAProxy which is a free open-source load balancer. HAProxy is not a Cloudera Runtime component, and Cloudera does not provide support for HAProxy. Ensure that SPNEGO authentication is enabled in Spark History Server.

## Procedure

1. In Cloudera Manager, navigate to Spark Configuration and configure the history_server_load_balancer_url property.

   The history_server_load_balancer_url property configures the following automatically:

   - The load balancer SPNEGO principal is automatically generated and added to the keytab file. For example: HTTP/loadbalancer.example.com/EXAMPLE.com
   - The value of SPNEGO_PRINCIPAL is automatically set to * , and all HTTP principals are loaded from the automatic generated keytab
   - The value of spark.yarn.historyServer.address is set to this URL in the appropriate gateway files:
     - Spark3: /etc/spark3/conf/spark-defaults.conf
2. Use the following test commands for this configuration:

   curl --negotiate -u : --location-trusted -c cookies.dat -b cookies.dat -k "https://loadbalancer.example.com:18488"

   curl --negotiate -u : --location-trusted -c cookies.dat -b cookies.dat -k "https://loadbalancer.example.com:18489"

## Example

This is the sample haproxy.cfg file that is used in the example below:

```
defaults
    mode                    http
    log                     global
    option                  httplog
    option                  dontlognull
    option http-server-close
    option forwardfor       except 127.0.0.0/8
    option                  redispatch
    retries                 5
    timeout http-request    10s
    timeout queue           1m
    timeout connect         3s
    timeout client          1m
    timeout server          1m
    timeout http-keep-alive 10s
    timeout check           10s
    maxconn                 3000

#---------------------------------------------------------------------
# Spark2 frontend which proxys to the Spark2 backends
#---------------------------------------------------------------------
frontend                        spark_front
```

```
    bind                        *:18488 ssl crt /var/lib/cloudera-scm-agent/
agent-cert/cdep-host_key_cert_chain_decrypted.pem
    default_backend             spark2

#---------------------------------------------------------------------
# round robin balancing between the various backends
#---------------------------------------------------------------------
backend spark2
    balance                     source
    server spark2-1 shs1.example.com:18488 check ssl ca-file /var/lib/clo
udera-scm-agent/agent-cert/cm-auto-global_cacerts.pem
    server spark2-2 shs2.example.com:18488 check ssl ca-file /var/lib/clou
dera-scm-agent/agent-cert/cm-auto-global_cacerts.pem

#---------------------------------------------------------------------
# Spark3 frontend which proxys to the Spark3 backends
#---------------------------------------------------------------------
frontend                        spark3_front
    bind                        *:18489 ssl crt /var/lib/cloudera-scm-agent/
agent-cert/cdep-host_key_cert_chain_decrypted.pem
    default_backend             spark3
#---------------------------------------------------------------------
# round robin balancing between the various backends
#---------------------------------------------------------------------
backend spark3
    balance                     source
    server spark3-1 shs3.example.com:18489 check ssl ca-file /var/lib/clou
dera-scm-agent/agent-cert/cm-auto-global_cacerts.pem
    server spark3-2 shs4.example.com:18489 check ssl ca-file /var/lib/cloud
era-scm-agent/agent-cert/cm-auto-global_cacerts.pem
```

**Related Information**

HAProxy - The Reliable, High Performance TCP/HTTP Load Balancer

Configuring Load Balancer for Impala

## Configuring high availability for Spark History Server with an internal load balancer

Learn how to configure high availability for Spark History Server using an internal load balancer. The authentication method for the internal load balancer uses a username and password through Apache Knox Gateway. The Cloudera stack includes the Apache Knox Gateway which has a built-in load balancer and failover mechanism.

### Before you begin

One Knox service must be installed in Cloudera Manager.

### About this task

This internal load balancer is only recommended for testing purposes because only one Knox Gateway is a single point of failure.

The following Knox topology configuration is automatically generated if there are two Spark History Server clusters in a Cloudera Manager cluster:

```
knox.example.com:/var/lib/knox/gateway/conf/topologies/cdp-proxy.xml

    <service>
        <role>SPARK3HISTORYUI</role>
        <url>https://shs1.example.com:18489</url>
        <url>https://shs2.example.com:18489</url>
    </service>
```

### Procedure

To use the Knox load balancing feature, you must use the Knox Gateway URL. If one of the Spark History Servers is down, the connection will be automatically redirected to the other server. See the example Knox Gateway URLs below:

Spark3: https://knox.example.com:8443/gateway/cdp-proxy/spark3history/

## Configuring high availability for Spark History Server with multiple Knox Gateways

Learn how to configure high availability for Spark History Server using an external load balancer such as HAProxy with multiple Apache Knox Gateway services. If you have only one Knox Gateway, this will be a single point of failure. For example, if the Knox Gateway goes down, and you are using the Knox Gateway's URL to access the Spark History Server's user interface, it will fail.

### Before you begin

You must have installed two or more Knox Gateway services in Cloudera Manager, and one external load balancer, for example, HAProxy.

### About this task

The following is an example HAProxy configuration for Knox Gateways. This is the sample haproxy.cfg file that is used in this example:

```
defaults
    mode                    http
    log                     global
    option                  httplog
    option                  dontlognull
    option http-server-close
    option forwardfor       except 127.0.0.0/8
    option                  redispatch
    retries                 5
    timeout http-request    10s
    timeout queue           1m
    timeout connect         3s
    timeout client          1m
    timeout server          1m
    timeout http-keep-alive 10s
    timeout check           10s
    maxconn                 3000

frontend knox-frontend
    bind :8443 ssl crt /etc/haproxy/cm-auto-host_cert_chain_unenckey.pem
    mode http
    stats enable
    option forwardfor
    http-request redirect location https://%[req.hdr(Host)]/gateway/homepag
e/home/ if { path / }
    default_backend knox-backend

backend knox-backend
    mode http
    option redispatch
    balance leastconn
    option forwardfor
```

```
    stick-table type ip size 1m expire 24h
    stick on src
    option httpchk HEAD /gateway/knoxsso/knoxauth/login.html HTTP/1.1\r\nHo
st:\ hwx.site
    http-check expect status 200
    # Knox Gateway instances (8443)
    server knox1.example.com knox1.example.com:8443 check ssl ca-file /var/
lib/cloudera-scm-agent/agent-cert/cm-auto-global_cacerts.pem
    server knox2.example.com knox2.example.com:8443 check ssl ca-file /var/
lib/cloudera-scm-agent/agent-cert/cm-auto-global_cacerts.pem
    server knox3.example.com knox3.example.com:8443 check ssl ca-file /var/
lib/cloudera-scm-agent/agent-cert/cm-auto-global_cacerts.pem
    server knox4.example.com knox4.example.com:8443 check ssl ca-file /var/
lib/cloudera-scm-agent/agent-cert/cm-auto-global_cacerts.pem
    server knox5.example.com knox5.example.com:8443 check ssl ca-file /var/
lib/cloudera-scm-agent/agent-cert/cm-auto-global_cacerts.pem

frontend stats
    bind *:8404
    stats enable
    stats uri /stats
    stats refresh 1s
```

### Procedure

Add the following HAProxy's URL to the Spark gateway configurations:

Spark3: Cloudera Manager > Spark3 > Configuration > history_server_load_balancer_url https://loadbalancer.example.com:8443/gateway/cdp-proxy/spark3history/

### Related Information

Apache Knox

HAProxy - The Reliable, High Performance TCP/HTTP Load Balancer

Configuring Load Balancer for Impala

# How to access Spark files on Ozone

Learn how you can connect Spark to the Ozone object store with the help of a sample script.

This script, in Scala, counts the number of word occurrences in a text file. The key point in this example is to use the following string to refer to the text file: ofs://omservice1/s3v/hivetest/spark/jedi_wisdom.txt

Word counting example in Scala

```
import sys.process._

// Put the input file into Ozone
//"hdfs dfs -put data/jedi_wisdom.txt ofs://omservice1/s3v/hivetest/spark" !
// Set the following spark setting in the file "spark-defaults.conf" on the
 CML session using terminal
//spark.kerberos.access.hadoopFileSystems=ofs://omservice1/s3v/hivetest
//count lower bound
val threshold = 2
// this file must already exist in hdfs, add a
// local version by dropping into the terminal.
val tokenized = sc.textFile("ofs://omservice1/s3v/hivetest/spark/jedi_wisd
om.txt").flatMap(_.split(" "))
// count the occurrence of each word
val wordCounts = tokenized.map((_ , 1)).reduceByKey(_ + _)
```

```
// filter out words with fewer than threshold occurrences
val filtered = wordCounts.filter(_._2 >= threshold)
System.out.println(filtered.collect().mkString(","))
```