IBM DATA SCIENCE PLATFORM

INTEGRATION WITH HADOOP PLATFORMS

VERSION: 1.3

DATE: 2018-09-27

AUTHOR: FRANK KETELAARS

TABLE OF CONTENTS

INTRODUCTION

There are several integration points between Hadoop and the Data Science component included in DSX and ICP for Data. When implementing for a proof of concept or in a real customer environment, there are several factors to consider to ensure the solutions can be integrated and that it happens in a secure manner.

The document covers 4 integration points specifically:

- Accessing the HDFS file system to read and write files
- Connecting to the Hive database to access table data
- Using Big SQL on the Hortonworks Data Platform to access table data
- Push down Spark processing to the Spark2 server on the HDP

AVAILABLE SERVICES AND SECURITY

When a Hadoop cluster (Hortonworks Data Platform or Cloudera Data Hub) is created, it exposes several services that do not require authentication, such as:

- HDFS native file system access, via port 8020
- WebHDFS file system access, via port 50070
- HCatalog, via port 50111
- Livy for Spark, via port 8998
- Livy for Spark2, via port 8999

Please note that not all services may be deployed on the cluster that is being accessed.

If not protected using a firewall or hidden in a private network, an external HDFS client can establish a native connection to the HDFS using port 8020 and will be granted access per the user who created the connection. So, for example if the user starting the client is “root” or some other super-user like “hdfs” or “hive”, the entire HDFS directory structure and the files it holds is accessible and can be queried and even tampered with. This is a real risk as spoofing a user is trivial if you have administrator access on the client system.

A common method for preventing spoofing of client users is to require them to authenticate to Kerberos. This means that the identity of the user or service is independently checked and they can only connect to the service if they are who they say they are. In HDP implementations, authentication is often implemented via Apache Knox, which then validates user connections with an LDAP service (either an external LDAP service or the demo LDAP service that is included in HDP).

For connecting to Hadoop services, a specialized integration package has been created, DSXHI. This document first covers installation and configuration of DSXHI to securely connect to Hadoop from a notebook. The second part discusses direct connections to Hadoop which can be used when no authentication is required.

SECURE ACCESS TO HADOOP SERVICES

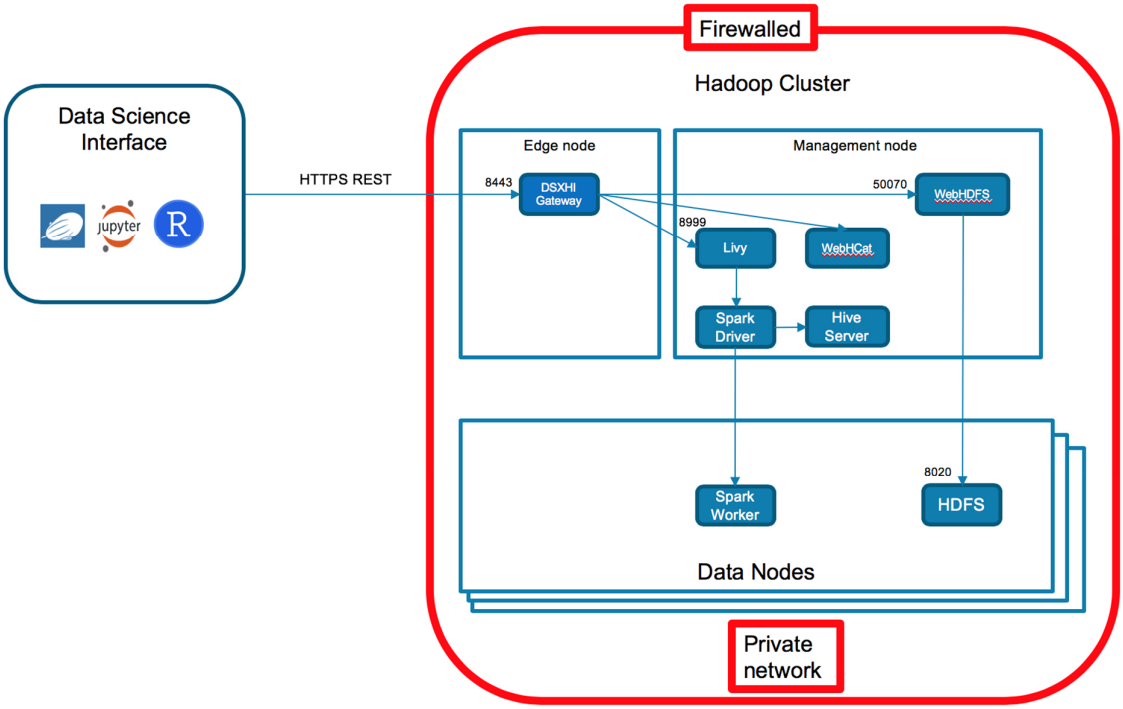
When the Hadoop services are not exposed publicly because they are behind a firewall, or authentication via Knox and/or Kerberos is implemented, it is recommended to implement the Hadoop Integration Service package (DSXHI) on one of the edge nodes of the cluster.

This chapter discusses the installation and setup of the DSXHI package which is required for securely connecting to the Hadoop services which are exposed (WebHDFS, WebHCat, Livy for Spark and Livy for Spark2).

Concealing services behind the DSXHI gateway

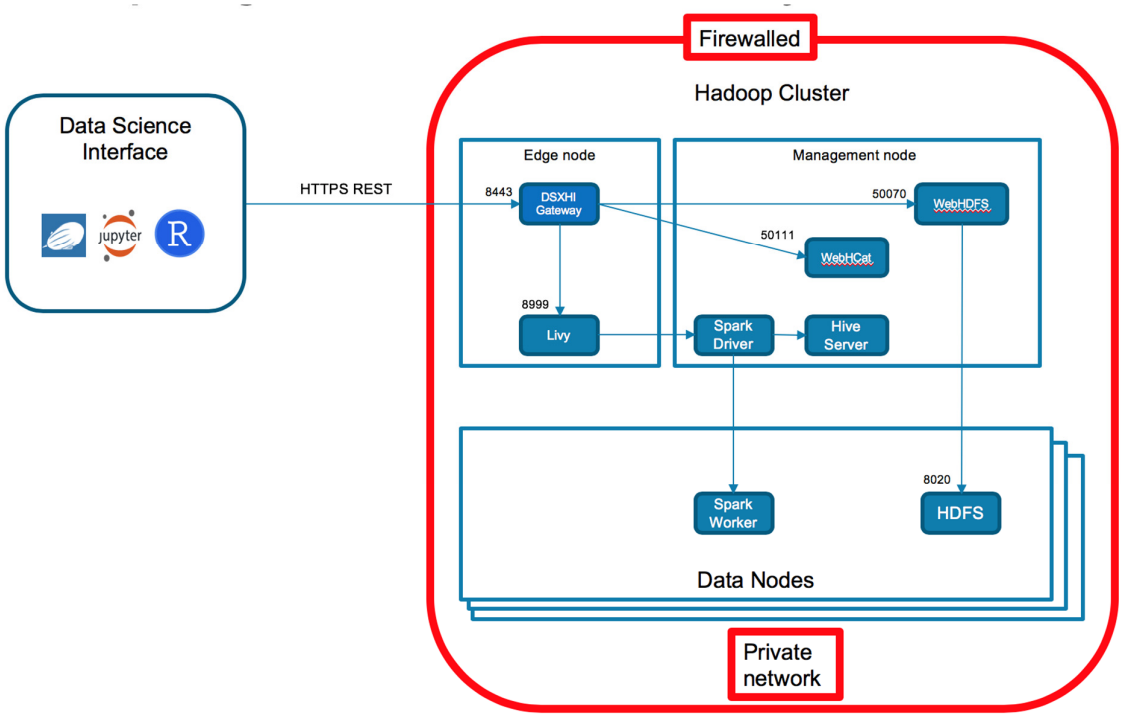
In a secure Hadoop cluster, only the services which support authentication are exposed to the outside world. Ports for unsecure services such as native HDFS, WebHDFS, WebHCat, Livy, etc. are hidden behind a firewall. Additionally, the Hadoop data nodes are typically connected to the management nodes via a private network and therefore not externally visible.

DSXHI sets up a gateway (based on Apache Knox) on the Hadoop edge node to expose the WebHDFS, WebHCat, Livy for Spark and Livy for Spark2 services to DSX. The gateway can – after authentication – redirect the requests to a service that already exists on the Hadoop platform. Additionally, DSXHI can create its own instance of the Livy for Spark and Livy for Spark2 services in case these have not been implemented as part of the Hadoop installation.



In the above picture, the DSXHI gateway will handle requests from the notebook interface and authenticate the user to the Hadoop cluster. The gateway directs the traffic to the appropriate service, HDFS, WebHCat or Livy for Spark/Spark2 which are already running on the Hadoop cluster.

Some Hadoop distributions do not provide Livy by default. DSXHI can instantiate this service on the edge node so that the notebooks can push down processing to the Hadoop cluster. Below you will find the diagram that outlines this configuration.



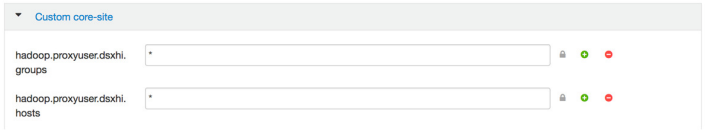
Preparing the Hadoop cluster for DSXHI

The DSXHI services will connect to the Hadoop cluster with special privileges to impersonate other users to ensure that YARN jobs are submitted on behalf of the user and that HDFS authorization settings are respected. In the examples below we assume that user is called “dsxhi”.

For the HDFS component the following settings must be applied to core-site. In HDP, this configuration can be set in HDFS → Configs → Advanced → Custom core-site.

```
hadoop.proxyuser.dsxhi.groups=*
hadoop.proxyuser.dsxhi.hosts=*
```

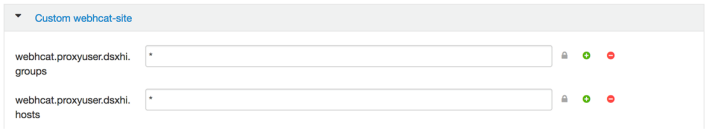
Example:



If you plan to access Hive tables, you must add dsxhi as a WebHCat proxy user. In HDP, you can set this configuration in Hive → Configs → Advanced → Custom webhcat-site.

```
webhcat.proxyuser.dsxhi.hosts=*
webhcat.proxyuser.dsxhi.groups=*
```

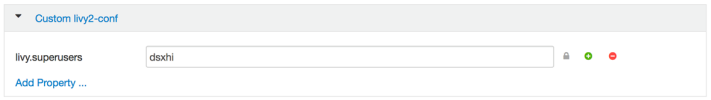
Example:



Finally, if you intend to push down Spark jobs to the Hadoop cluster, you must define dsxhi as the super-user for Spark. In HDP, you can configure Spark or Spark2 → Configs → Advanced → Custom livy2-conf.

```
livy.superusers=dsxhi
```

Example:



When you have set all properties, you must restart the services marked in Ambari.

Ensure users are created

The dsxhi user and also users configured in Data Science Experience or ICP for Data must be known on the Hadoop cluster, and they must have a home directory in the HDFS. Hadoop requires that all users in the cluster must have the same uid, hence we specify this when creating the Linux account.

To create the dsxhi user, execute the following on all Hadoop nodes:

```
useradd -u 3001 -g hdfs dsxhi
```

Subsequently, create the dsxhi HDFS home directory (as user hdfs):

```
hdfs dfs -mkdir /user/dsxhi
hdfs dfs -chown dsxhi:hdfs /user/dsxhi
```

Subsequently, you must also create a user for every data science user registered in DSX or ICP for Data; in the example below we create user fk.

Execute the following on all Hadoop nodes:

```
useradd -u 3002 fk
```

Subsequently, create the user's HDFS home directory (as user hdfs):

```
hdfs dfs -mkdir /user/fk
hdfs dfs -chown fk:fk /user/fk
```

Installing the DSXHI package

You will need to install the DSXHI package on one of the edge nodes of the Hadoop cluster. If no edge node is available and if this is a testing cluster, you can choose to install the package on one of the Hadoop management nodes, but beware that you will probably have to change some of the default ports as they may be conflicting with existing Hadoop services.

To install the package, use yum.

```
yum install -y dsxhi_x86_64.rpm
```

Configure DSXHI

DSXHI includes sample configurations for Hortonworks (HDP) and Cloudera (CDH). Copy the template to dsxhi_install.conf.

```
cp /opt/ibm/dsxhi/conf/dsxhi_install.conf.template.HDP /opt/ibm/dsxhi/conf/dsxhi_install.conf
```

In the below snippet, we have only marked the changed properties. In the example, DSXHI is configured to run the Livy and Livy for Spark 2 service on the edge node. Also, Kerberos is not active on this HDP cluster. See Appendix A – Using Kerberos for DSXHI for more details on activating Kerberos and configuring DSXHI to use Kerberos access.

```
dsxhi_license_acceptance=a
dsxhi_serviceuser=dsxhi
dsxhi_serviceuser_group=hdfs

# dsxhi_serviceuser_keytab=/etc/security/keytabs/dsxhi.keytab
# dsxhi_spnego_keytab=/etc/security/keytabs/spnego.service.keytab
cluster_manager_url=http://hdpfk-hdp.fyre.ibm.com:8080
cluster_admin=admin

existing_webhcat_url=http://hdpfk-hdp.fyre.ibm.com:50111/templeton/v1

# existing_livyspark_url= <-- not to be used, using Livy for Spark server as part of dsx-hi
# existing_livyspark2_url= <-- not to be used, using Livy for Spark2 server as part of dsx-hi

dsxhi_livyspark_port=8998
dsxhi_livyspark2_port=8999

# known_dsx_list=https://dsxlcluster1.ibm.com,https://dsxlcluster2.ibm.com:31843
```

Once the properties have been configured, the install script can be run. This will configure and also start the required services. In the below example we have chosen the same password for the gateway, the certificate password and the Ambari admin password.

```
cd /opt/ibm/dsxhi/bin
./install.py --dsxhi_gateway_master_password=passw0rd --dsxhi_self_signed_cert_pass=passw0rd
--password=passw0rd
```

Output:

```
IBM Data Science Experience Hadoop Integration Service
------------------------------------------------------
Terms and Conditions: http://www14.software.ibm.com/cgi-
bin/weblap/lap.pl?la_formnum=&li_formnum=L-KLSY-
AVZW2D&title=IBM+Data+Science+Experience+Hadoop+Integration&l=en

--Determining properties
--Running the prechecks

--Install Livy for Spark / Spark 2
--Configure gateway
--Create template for gateway
--Setting up known DSX Local clusters
--Install dsxhi_rest
--Start all services
--Install finished. Check status in /var/log/dsxhi/dsxhi.log
```

Register the DSX or ICP for Data service within DSXHI

To prevent “any” DSX or ICP for Data from accessing the services, a set of certificates must be exchanged. DSXHI provides a command that will register a DSX/ICP for Data cluster.

```
cd /opt/ibm/dsxhi/bin
./manage_known_dsx.py --add https://fkicp4d-master-1.fyre.ibm.com:31843
```

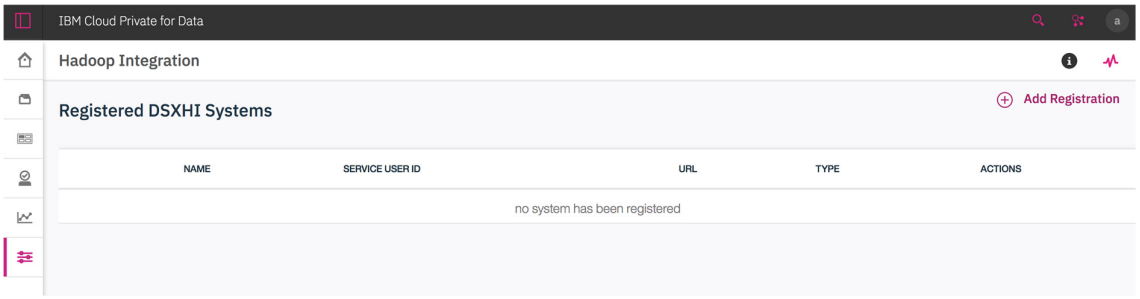
Output:

```
Successfully added:
DSX Local Cluster URL                                                    DSXHI Service URL
https://fkicp4d-master-1.fyre.ibm.com:31843                  https://hdpfk-
hdpe.fyre.ibm.com:8443/gateway/fkicp4d-master-1
```

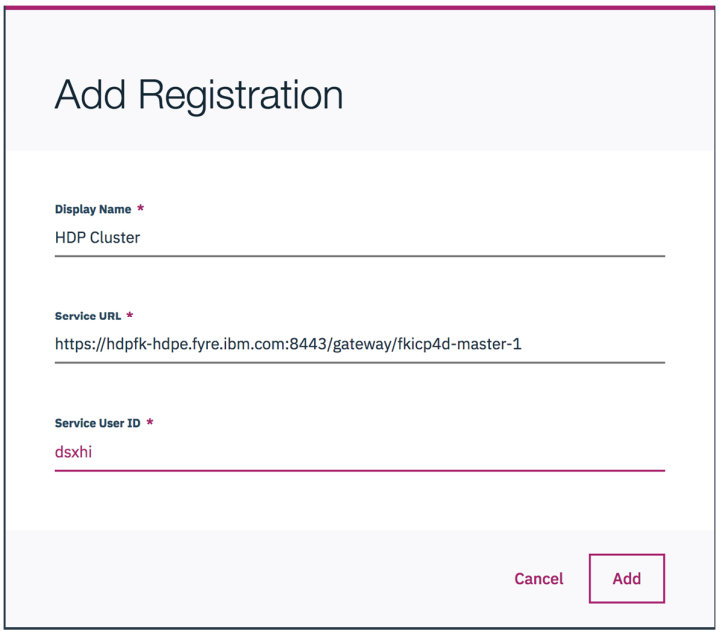
In the above example, the https:// address is the URL for accessing ICP for Data. For DSX, you would choose the URL that provides access to DSX.

Register the Hadoop cluster within DSX or ICP for Data

Now that the data science service has been registered in , you can access Hadoop from DSX or ICP for Data. In DSX, go to the Admin Console and select “Hadoop integration” from the hamburger menu. For ICP for Data, select “Administer” from the left menu and then choose “Hadoop integration”.

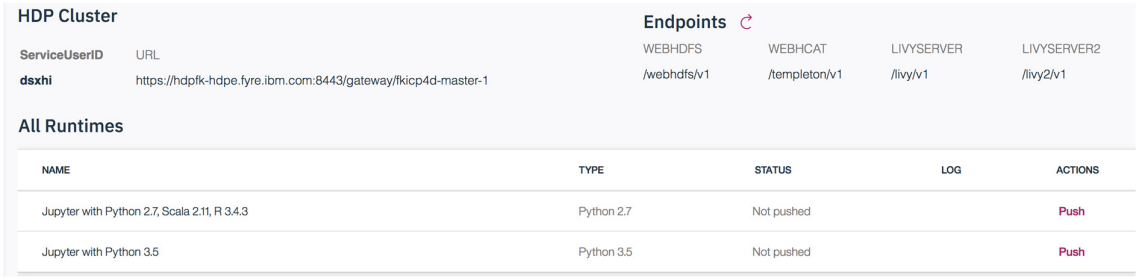


Click on “Add Registration”, pick a name and enter the DSXHI Service URL in the “Service URL” field. Finally, select the user (dsxhi) you will use to log on to the Hadoop cluster and click Add.



You should see the following message:

When you click on the three dots under Actions, you can find the details and which service endpoints are exposed.



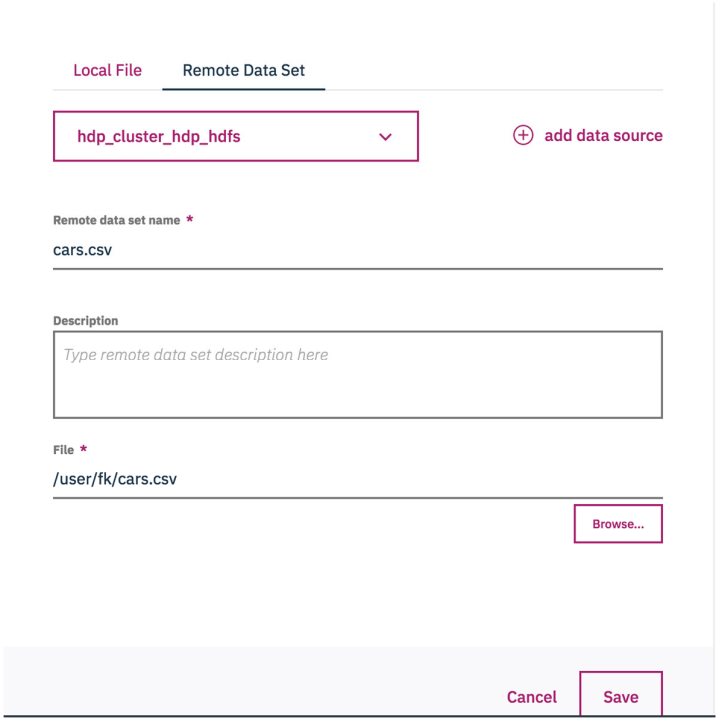
This completes the setup of and integration with the data science component.

Using the Hadoop integration

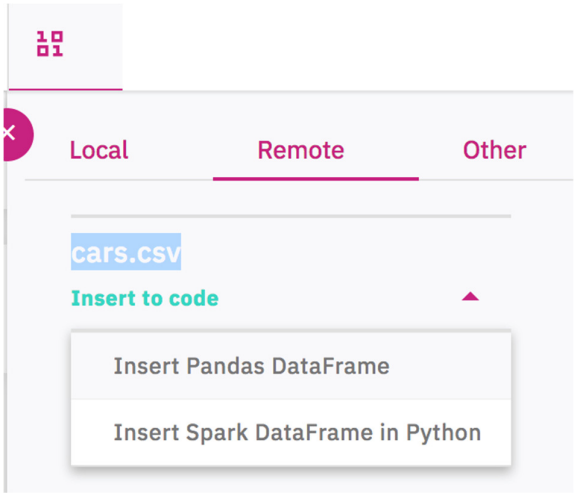
Now that the Hadoop integration has been set up, you can access HDFS and Hive data sources from within a notebook and also push down processing to the Hadoop cluster.

Retrieving data from the HDFS

As part of the Hadoop integration registration process, a data source for HDFS has already been created. We can utilize this to access a new data set.



Now, you can access the file from within a Jupyter notebook by inserting code in a cell.



We have selected “Insert Pandas DataFrame” which renders the below output:

```
In [1]: import dsx_core_utils, requests, jaydebeapi, os, io, sys
from pyspark.sql import SparkSession
import pandas as pd
df1 = None
dataSet = dsx_core_utils.get_remote_data_set_info('cars.csv')
dataSource = dsx_core_utils.get_data_source_info(dataSet['datasource'])
# Access HDFS dataset using RPC port with hdfs protocol
if ("HDFS" in dataSource['type'] and not dataSource['URL']):
    url = 'hdfs://' + dataSource['host'] + ':' + str(dataSource['port'])
    file_fullpath = url + dataSet['file']
    sparkSession = SparkSession(sc).builder.getOrCreate()
    df1 = sparkSession.read.csv(file_fullpath, header = "true", inferSchema = "true").toPandas()
# Access HDFS dataset using HTTP Port with a webhdfs URL
elif ("HDFS" in dataSource['type'] and dataSource['URL']):
    url = (dataSource['URL'][:-1] if (dataSource['URL'].endswith('/')) else dataSource['URL']) + dataSet['file'] + "?op=OPEN"
    headers = {"Authorization": os.environ.get('DSX_TOKEN')}
    response = requests.request("GET", url, headers=headers, timeout=10, verify=False, allow_redirects=True)
    if response.status_code != 200:
        raise Exception("get_data_source_info: " + str(response.status_code) + " returned when sending a request to \"" + url+"\"")
    else:
        if not(dataSet['file'].endswith('csv') or dataSet['file'].endswith('txt')):
            raise Exception("Invalid file type that is not txt or csv")
        df1 = pd.read_csv(io.StringIO(response.text, newline=None), sep=',', engine='python')
df1.head()
```

Out[1]:

	mpg	cylinders	engine	horsepower	weight	acceleration	year	origin	name
0	18.0	8	307.0	130.0	3504	12.0	70	American	chevrolet chevelle malibu
1	15.0	8	350.0	165.0	3693	11.5	70	American	buick skylark 320
2	18.0	8	318.0	150.0	3436	11.0	70	American	plymouth satellite
3	16.0	8	304.0	150.0	3433	12.0	70	American	amc rebel sst
4	17.0	8	302.0	140.0	3449	10.5	70	American	ford torino

Alternatively you can call the WebHDFS REST API directly from a cell in your notebook.

```
import dsx_core_utils, requests, jaydebeapi, os, io, sys
import pandas as pd
url = 'https://hdpfk-hdpe.fyre.ibm.com:8443/gateway/fkicp4d-master-
1/webhdfs/v1/user/fk/cars.csv?OP=OPEN'
headers = {'Authorization': os.environ.get('DSX_TOKEN')}
response = requests.request("GET", url, headers=headers, timeout=10, verify=False,
allow_redirects=True)
df = pd.read_csv(io.StringIO(response.text, newline=None), sep=',')
df.head()
```

Yet another option is to use the pywebhdfs package, which offers a few useful functions to access the HDFS.

```
from pywebhdfs.webhdfs import PyWebHdfsClient
auth_header = {"Authorization": os.environ.get('DSX_TOKEN')}
uri='https://hdpfk-hdpe.fyre.ibm.com:8443/gateway/fkicp4d-master-1/webhdfs/v1'
hdfs = PyWebHdfsClient(base_uri_pattern=uri, request_extra_opts={'verify': False},
request_extra_headers=auth_header)
hdfs.list_dir('/')
```

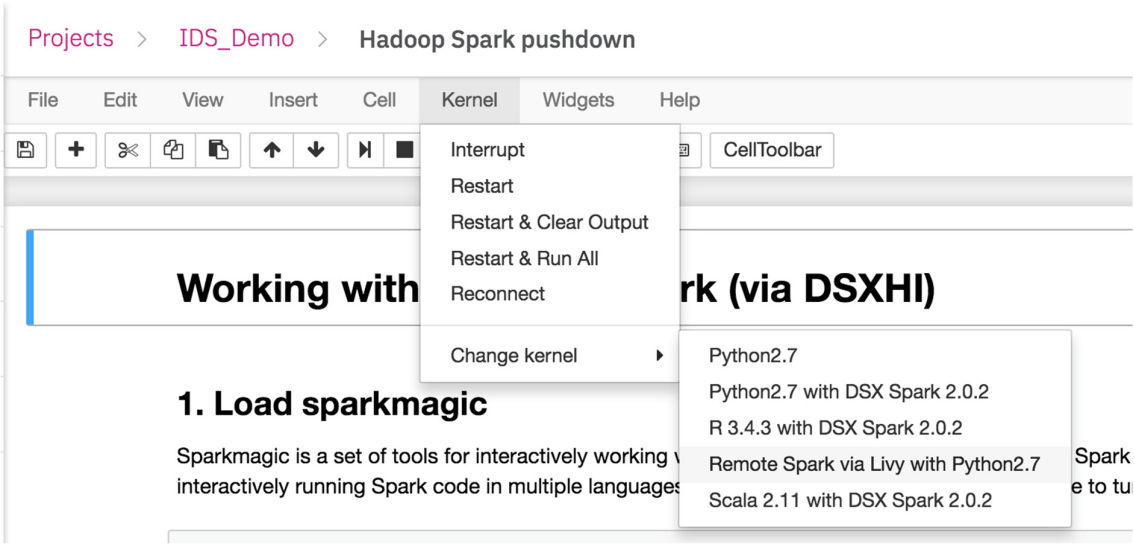
Output:

```
In [36]: from pywebhdfs.webhdfs import PyWebHdfsClient
         auth_header = {"Authorization": os.environ.get('DSX_TOKEN')}
         uri='https://hdpfk-hdpe.fyre.ibm.com:8443/gateway/fkicp4d-master-1/webhdfs/v1'
         hdfs = PyWebHdfsClient(base_uri_pattern=uri, request_extra_opts={'verify': False}, request_extra_headers=auth_header)
         hdfs.list_dir('/')

Out[36]: {u'FileStatuses': {u'FileStatus': [{u'accessTime': 0,
             u'blockSize': 0,
             u'childrenNum': 3,
             u'fileId': 16400,
             u'group': u'hadoop',
             u'length': 0,
             u'modificationTime': 1533616647183,
             u'owner': u'yarn',
             u'pathSuffix': u'app-logs',
             u'permission': u'777',
             u'replication': 0,
             u'storagePolicy': 0,
             u'type': u'DIRECTORY'},
            {u'accessTime': 0,
```

Push down Spark processing

You can create an endpoint in the Jupyter notebook and push down processing via the Livy for Spark or Livy for Spark 2 service. In the dsx_samples project you will find some sample notebooks which will set up the Livy connection and push down Spark processing. Below you will find 2 methods for starting a Spark session, followed by an example of pushed-down Python code.

After having started a remote Spark session, you can use the %%spark magic in every cell to ensure that the code is pushed down to the Hadoop cluster. However, you would probably want to execute most of the code on the remote session anyway. Therefore – once your notebook has opened – change the kernel to “Remote Spark with Livy with Python2.7” or the equivalent for Python 3.



Starting a Spark session using UI mode

The Spark integration library includes a user interface in which you can add and endpoint and start a session. For Python, enter the following code to load the package.

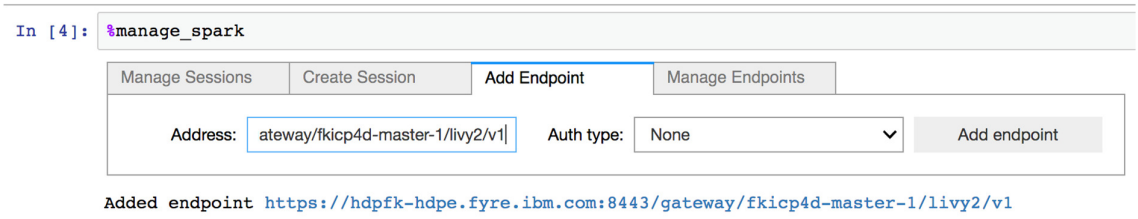
```
%load_ext sparkmagic.magics
import dsx_core_utils
dsx_core_utils.setup_livy_sparkmagic()
```

Then, in the next cell, enter:

```
%manage_spark
```

This will bring up the following window where you add the endpoint given in the Hadoop cluster details; when is installed on the Hadoop edge node, the Livy for Spark 2 endpoint is typically:

https://<edge_node>:8443/gateway/<dsx_icp4d_host>/livy2/v1

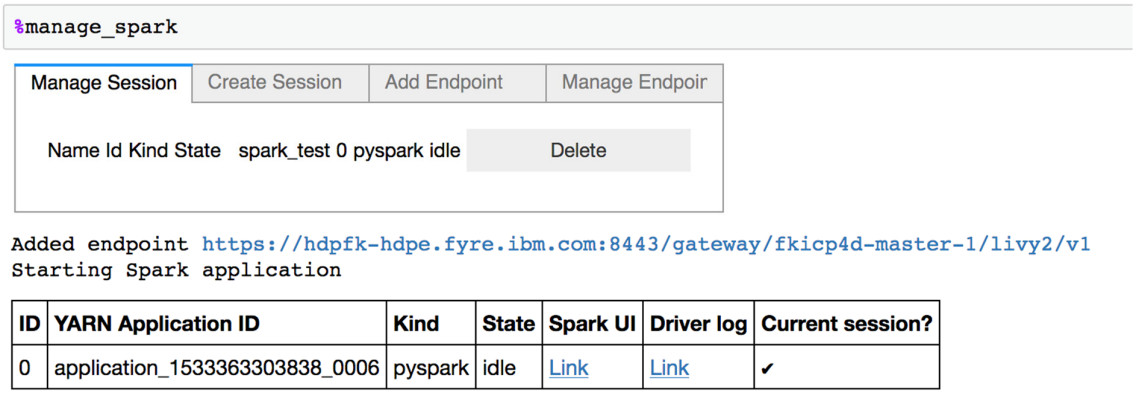


You can list the defined endpoints using the following code:

```
dsx_core_utils.list_dsxhi_livy_endpoints()
```

We entered the following endpoint:
https://hdpfk-hdpe.fyre.ibm.com:8443/gateway/fkicp4d-master-1/livy2/v1

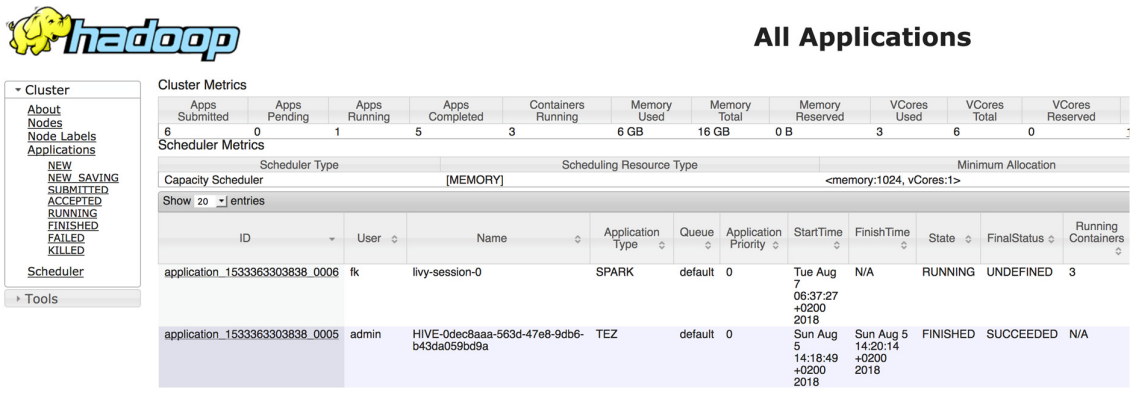
On the “Create Session” tab, create a Scala or Python session for the selected endpoint, as appropriate. This will start a YARN application on the Hadoop cluster.



ID	YARN Application ID	Kind	State	Spark UI	Driver log	Current session?
0	application_1533363303838_0006	pyspark	idle	Link	Link	✔

```
SparkSession available as 'spark'.
```

You can use the links under Spark UI and Driver log to find details about the Spark session and YARN application. Also, the YARN ResourceManager UI will show the application that is running:



Starting a Spark session using command mode

The other method for starting a Spark session is using command mode. Again we need to import the appropriate libraries.

```
%load_ext sparkmagic.magics
import dsx_core_utils
dsx_core_utils.setup_livy_sparkmagic()
```

You can list the defined endpoints using the following code:

```
dsx_core_utils.list_dsxhi_livy_endpoints()
```

In a new cell, enter the following to start a Spark session:

```
%spark add -s spark_test_2 -l python -u https://hdpfk-hdpe.fyre.ibm.com:8443/gateway/fkicp4d-master-1/livy2/v1
```

The Spark session is now started; when ready, it will return the same detail information, including the Spark UI and YARN log links.

```
In [4]: %spark add -s spark_test_2 -l python -u https://hdpfk-hdpe.fyre.ibm.com:8443/gateway/fkicp4d-master-1/livy2/v1
Starting Spark application
```

ID	YARN Application ID	Kind	State	Spark UI	Driver log	Current session?
1	application_1533363303838_0007	pyspark	idle	Link	Link	✔

```
SparkSession available as 'spark'.
```

If you're finished with a Spark session, you can delete it as follows.

```
%spark delete -s spark_test_2
```

Running processing on the remote session

Now that the Spark session has been initialized, you can run processing on it. An initial simple check is to display the Spark version using the SparkContext (sc).

Note: In the below code examples we assume that the notebook is using the Remote Spark via Livy kernel. If you did not change the kernel after opening the notebook, you will have to use the %%spark magic in each cell to ensure the code is executed in the remote session.

```
sc.version
```

Next, run some "real" code

```
import random
NUM_SAMPLES=10000000
def inside(p):
    x, y = random.random(), random.random()
    return x*x + y*y < 1

count = sc.parallelize(xrange(0, NUM_SAMPLES)) \
             .filter(inside).count()
print "Pi is roughly %f" % (4.0 * count / NUM_SAMPLES)
```

The result:

```
In [17]: %%spark
import random
NUM_SAMPLES=10000000
def inside(p):
    x, y = random.random(), random.random()
    return x*x + y*y < 1

count = sc.parallelize(xrange(0, NUM_SAMPLES)) \
             .filter(inside).count()
print "Pi is roughly %f" % (4.0 * count / NUM_SAMPLES)

Pi is roughly 3.142001
```

Access Hive from within a notebook

Once a Spark session has been set up on the Hadoop cluster, you can also access Hive tables.

```
%%spark -c sql -s spark_test_2
SHOW TABLES
```

Output:

```
In [6]: %%spark -c sql -s spark_test_2
        SHOW TABLES

Out[6]:
```

	database	tableName	isTemporary
0	default	cars	False

Then, you can retrieve the Hive table into a dataframe within the Jupyter notebook environment like this:

```
%%spark -c sql -o carsdf -s spark_test_2
SELECT * FROM cars
```

Output:

```
In [7]: %%spark -c sql -o carsdf -s spark_test_2
        SELECT * FROM cars

Out[7]:
```

	mpg	cylinders	engine	horsepower	weight	acceleration	year	origin	name
0	18.0	8	307	130.0	3504	12.0	70	American	chevrolet chevelle malibu
1	15.0	8	350	165.0	3693	11.5	70	American	buick skylark 320
2	18.0	8	318	150.0	3436	11.0	70	American	plymouth satellite
3	16.0	8	304	150.0	3433	12.0	70	American	amc rebel sst

If you then want to get a count of cars per origin, you can do this using SQL.

```
%%spark -c sql -o carscountdf -s spark_test_2
SELECT origin, count(origin) as counts FROM cars GROUP BY origin ORDER BY origin
```

Output:

```
In [28]: %%spark -c sql -o carscountdf -s spark_test_2
         SELECT origin, count(origin) as counts FROM cars GROUP BY origin ORDER BY origin

Out[28]:
```

	origin	counts
0	American	254
1	European	73
2	Japanese	79

If you don't want the data to be retrieved to your local environment but process it on the Hadoop cluster, you can retrieve use the following construct:

```
carshdp = sqlContext.sql("select * from cars")
carshdp.show()
```

```
In [11]: carshdp = sqlContext.sql("select * from cars")
         carshdp.show()
```

mpg	cylinders	engine	horsepower	weight	acceleration	year	origin	name
18	8	307	130	3504	12.0	70	American	chevrolet chevell...
15	8	350	165	3693	11.5	70	American	buick skylark 320
18	8	318	150	3436	11.0	70	American	plymouth satellite
16	8	304	150	3433	12.0	70	American	amc rebel sst
17	8	302	140	3449	10.5	70	American	ford torino
15	8	429	198	4341	10.0	70	American	ford galaxie 500
14	8	454	220	4354	9.0	70	American	chevrolet impala
14	8	440	215	4312	8.5	70	American	plymouth fury iii
14	8	455	225	4425	10.0	70	American	pontiac catalina
15	8	390	190	3850	8.5	70	American	amc ambassador dpl

The resulting Spark dataframe will only exist within the YARN application that runs on the Hadoop cluster. If you try to access the data from within the local Jupyter environment, you cannot.

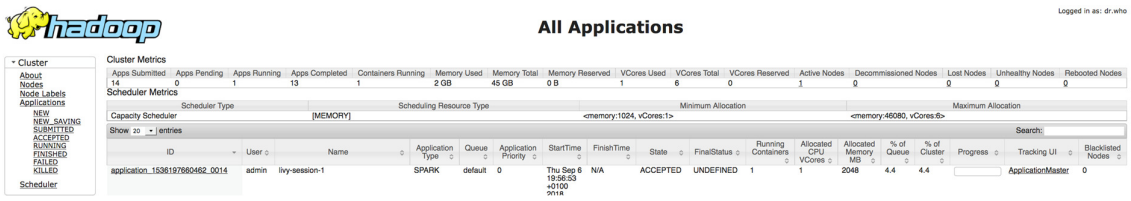
```
%%local
carshdp.show()
```

```
In [12]: %%local
         carshdp.show()

NameErrorTraceback (most recent call last)
<ipython-input-12-d23b8accc7d3> in <module>()
----> 1 carshdp.show()

NameError: name 'carshdp' is not defined
```

Monitoring the remote Spark session using YARN ResourceManager UI

The remote Spark session kicks off an application in YARN (currently using the "default" queue), which you can monitoring using the ResourceManager UI. From Ambari, click on YARN and then use Quick Links to go to the ResourceManager UI.

You should see the Spark application that was submitted on behalf of the user connected to DSX or ICP for Data.



From the ResourceManager UI, you can view the logs, monitor the status, etc.

Specifying a different YARN queue

YARN applications started by DSXHI are submitted to the **default** YARN queue. If you need to use a different queue, you can do so within the notebook.

```
%%spark config
{"name": "IBM_Session",
"proxyUser": "user1", "driverMemory": "2G", "numExecutors": 1, "conf": {
"spark.yarn.queue":"devs" },
"executorCores": 2,
"executorMemory": "2G" }
```

The above code must be executed before the Spark session is started.

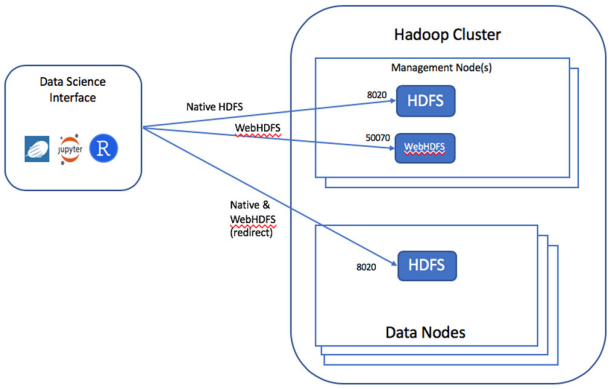
UNSECURE ACCESS TO HADOOP SERVICES

In case of a POC or just a basic test, you may choose not to implement the secure gateway and just access the unsecure services directory from within the notebook. This chapter provides a couple of examples on accessing the Hadoop services directly. Please note that this will probably not work in environments where some level of security has been implemented on the Hadoop cluster.

When accessing the services listed below, the connection will be established with the user that is logged on in DSX or ICP for Data. If you have logged on as “admin”, the activity on the Hadoop cluster will be run as the “admin” account, if it exists.

Unsecure HDFS access

In an unsecure Hadoop cluster, the management node(s) and data nodes are accessible from the outside and there is no firewall blocking access to any ports.



Native HDFS access

HDFS can be directly accessed via port 8020. If you want to read a file from the HDFS, simply specify the following for the URI:

```
hdfs://<Address_of_name_node>:8020/<path_to_file>
```

The name node address can be a host name that is found through the DNS service or the IP address. When the name node receives a file read request, it will redirect the client to the data node that holds the data.

You can read the data into a Spark or Pandas DataFrame.

Example:

```
import dsx_core_utils, requests, jaydebeapi, os, io, sys
from pyspark.sql import SparkSession
import pandas as pd
url = 'hdfs://hdpfk-hdp.fyre.ibm.com:8020/user/fk/cars.csv'
sparkSession = SparkSession(sc).builder.getOrCreate()
df1 = sparkSession.read.csv(url, header = "true", inferSchema = "true").toPandas()
df1.head()
```

WebHDFS access

WebHDFS is accessed via port 50070. If you want to read a file via WebHDFS, use the following URL:

```
http://<Address_of_WebHDFS_service>:50070/webhdfs/v1/<full_path_to_file>?OP=OPEN
```

Example:

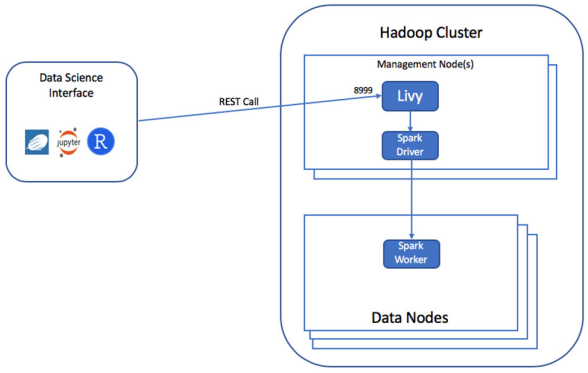
```
import dsx_core_utils, requests, jaydebeapi, os, io, sys
import pandas as pd
url = 'http://hdpfk-hdp.fyre.ibm.com:50070/webhdfs/v1/user/fk/cars.csv?OP=OPEN'
response = requests.request("GET", url, timeout=10, verify=False, allow_redirects=True)
df2 = pd.read_csv(io.StringIO(response.text, newline=None), sep=',')
df2.head()
```

In the case of WebHDFS, the address of the server hosting WebHDFS and the data nodes must be resolvable via DNS or the hosts table; when retrieving an HDFS file (OP=OPEN parameter), an HTTP redirect which contains the host name of the data node where the file is retrieved.

Note:	When using one of the templates that are available on Cloud Concierge, the HDP are not resolvable through DNS and unsecure WebHDFS requests will fail as it uses an HTTP redirect with the hostname it received from the HDP image.

Push down Spark processing via Livy

Livy for Spark 2 is a web service that listens on port 8999 by default. In an unsecure cluster, the client directly connects to the Livy service, which in turn spins up a Spark driver that will push down all the workload to the Spark workers.



When trying to push down Spark processing via this service in an unsecure manner, you must disable CSRF protection to ensure the service can be accessed from a non-local client.

1. Go to Ambari and click on the **Spark2** service
2. Click on **Configs**, expand **Advanced livy2-conf** and change the **livy.server.csrf_protection.enabled** setting to false

3. Restart the Spark2 service

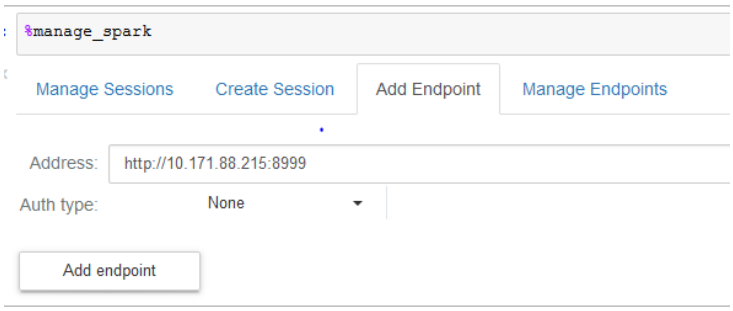
Now you can create an endpoint in the DSX notebook and push down processing via the Livy for Spark 2 service. DSX provides some sample notebooks which will set up the Livy connection and push down Spark processing. In short, enter the following in a cell to set up Livy:

```
%load_ext sparkmagic.magics
import dsx_core_utils
dsx_core_utils.setup_livy_sparkmagic()
%reload_ext sparkmagic.magics
```

Then, in the next cell, enter:

```
%manage_spark
```

You will be presented with a table where you can create an endpoint and establish the session:



Create an endpoint and specify the following:

- Address: http://<Livy for Spark 2 IP address>:8999
- Auth: None

Then, create a session (Scala or Python) using this endpoint. A new Spark session will be started on the Hadoop cluster and when finished a message will be shown that the SparkSession is available as 'spark'. Now, enter the following in a cell and run:

```
%%spark
sc.version
u'2.2.0.2.6.4.0-91'
```

%%spark indicates that processing is pushed down to the specified endpoint. The Spark version displayed above is the one from the Hadoop cluster DSX connected to. Any cell that starts with the %%spark magics will run on the HDP cluster and can access the files in the HDFS as if they are local.

Instead of creating the session via the %manage_spark magics, you can also establish it directly using the %spark magics. Please note that we are specifying a session name below which can be referred to in the %%spark magics.

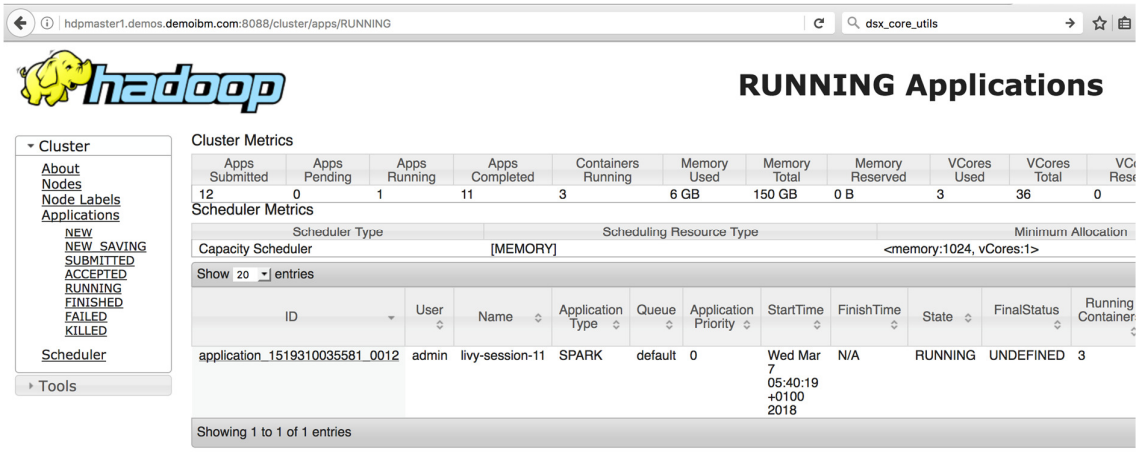
```
%spark add -s session2 -l python -u http://10.175.93.141:8999
```

```
Starting Spark application
```

ID	YARN Application ID	Kind	State	Spark UI	Driver log	Current session?
11	application_1519310035581_0012	pyspark	idle	Link	Link	✔

```
SparkSession available as 'spark'.
```

The session is also visible in the YARN ResourceManager UI. Please note that you do not see a reference to ‘session2’, just the YARN application ID.



Now that session2 has been created, you can run a cell in the remote Spark context.

```
%%spark -s session2
sc.version
```

Once finished with the session, delete it and it will kill the YARN application.

```
%spark delete -s session2
```

APPENDIX A – USING KERBEROS FOR DSXHI

DSXHI was specifically designed to allow Kubernetes-based applications DSX and ICP for Data to work with “Kerberized” Hadoop clusters. Applications on the Kubernetes platform imply that services run in “ephemeral” containers which typically take up temporary names and this is not compatible with Kerberos’s configuration where server names are part of the service principals configured for authentication.

By running the DSXHI service as part of the Hadoop cluster (on an edge node) as a user with impersonation permissions, the DSXHI service can run jobs on behalf of other users defined in DSX or ICP for Data.

The installation and configuration of DSXHI is mandatory for Kerberized Hadoop clusters. Initial installation must also be done via the procedure documented here: Installing the DSXHI package.

Preparing DSXHI Kerberos configuration on the edge node

We’re assuming that you are installing DSXHI on an existing edge node of the Hadoop cluster and that the node is already part of the Kerberos realm. This means that the Kerberos workstation libraries are installed and the Kerberos client configuration is already in place (typically in /etc/krb5.conf).

Additionally, the DSXHI user (dsxhi or other) must have been created on the Hadoop cluster.

First you need to extract the keytab information from the KDC, both for the host (SPNEGO) and the DSXHI service that is going to run on the edge node. Start by creating a directory that will hold the keytab files (you must do this as root).

```
mkdir -p /etc/security/keytabs
```

Then, connect to the KDC.

```
kadmin -p kadmin/admin
```

Create the host Kerberos principle if it doesn’t exist yet.

```
# List the principals
listprincs

# Create the host principle if it doesn’t exist yet
addprinc -randkey host/fkhdp-hdpe.fyre.ibm.com
# Create the DSXHI service principle
addprinc -randkey dsxhi/fkhdp-hdpe.fyre.ibm.com
```

Then, extract the principles into a keytab file.

```
ktadd -k /etc/security/keytabs/dsxhi.keytab host/fkhdp-hdpe.fyre.ibm.com
ktadd -k /etc/security/keytabs/dsxhi.keytab dsxhi/fkhdp-hdpe.fyre.ibm.com
```

Quit from the Kerberos administrator.

```
q
```

The following step is optional as the install.py script will copy the keytab file to the DSXHI directory structure.

```
chown dsxhi:root /etc/security/keytabs/dsxhi.keytab
```

To validate that Kerberos has been set up correctly, initialize and display the contents of the /tmp HDFS directory.

```
su - dsxhi
kinit -k /etc/security/keytabs/dsxhi.keytab dsxhi/fkhdp-hdpe.fyre.ibm.com
# List the HDFS directory
hdfs dfs -ls /tmp
```

Ensure users are registered in the KDC

For every user who accesses the Hadoop cluster, a KDC principal must be in place. Normally this would already have been done as part of the Kerberos configuration. To check that user principals have been created, check the KDC.

```
kadmin -p kadmin/admin
# List the principals
listprincs

# Quit
q
```

Configure DSXHI with Kerberos

Before you run the install.py script, you must ensure that the DSXHI and SPNEGO keytabs are available on the edge node that running DSXHI. See above steps for details.

Edit the /opt/ibm/dsxhi/conf/dsxhi_install.conf file and set it up as documented in section Configure DSXHI, making sure that the DSXHI service user and SPNEGO keytabs have been configured.

```
dsxhi_license_acceptance=a
dsxhi_serviceuser=dsxhi
dsxhi_serviceuser_group=hdfs

dsxhi_serviceuser_keytab=/etc/security/keytabs/dsxhi.keytab
dsxhi_spnego_keytab=/etc/security/keytabs/spnego.service.keytab
cluster_manager_url=http://hdpfk-hdp.fyre.ibm.com:8080
cluster_admin=admin

existing_webhcat_url=http://hdpfk-hdp.fyre.ibm.com:50111/templeton/v1

# existing_livyspark_url= <-- not to be used, using Livy for Spark server as part of dsx-hi
# existing_livyspark2_url= <-- not to be used, using Livy for Spark2 server as part of dsx-hi

dsxhi_livyspark_port=8998
dsxhi_livyspark2_port=8999

# known_dsx_list=https://dsxlcluster1.ibm.com,https://dsxlcluster2.ibm.com:31843
```

Once the properties have been configured, the install script can be run. This will configure and also start the required services. In the below example we have chosen the same password for the gateway, the certificate password and the Ambari admin password.

```
cd /opt/ibm/dsxhi/bin
./install.py --dsxhi_gateway_master_password=passw0rd --dsxhi_self_signed_cert_pass=passw0rd
--password=passw0rd
```

Now that DSXHI has been configured with Kerberos, you can continue by registering the DSX or ICP for Data service within DSXHI as documented here: Register the DSX or ICP for Data service within DSXHI.