How-to: Use the HBase Thrift Interface, Part 1

September 23, 2013 (https://blog.cloudera.com/blog/2013/09/how-to-use-the-hbase-thrift-interface-part-1/) | By Jesse Anderson (https://blog.cloudera.com/blog/author/janderson/) (@jessetanderson) (https://twitter.com/@jessetanderson) | No Comments (https://blog.cloudera.com/blog/2013/09/how-to-use-the-hbase-thrift-interface-part-1/#respond)
Categories: HBase (https://blog.cloudera.com/blog/category/hbase/) How-to (https://blog.cloudera.com/blog/category/how-to/)

There are various way to access and interact with Apache HBase (http://hbase.apache.org). Most notably, the Java API (http://archive.cloudera.com/cdh4/cdh/4/hbase-0.92.1-cdh4.1.2/apidocs/index.html) provides the most functionality. But some people want to use HBase without Java.

Those people have two main options: One is the Thrift interface (http://wiki.apache.org/hadoop/Hbase/ThriftApi) (the more lightweight and hence faster of the two options), and the other is the REST interface (http://wiki.apache.org/hadoop/Hbase/Stargate) (aka Stargate). A REST interface uses HTTP verbs to perform an action. By using HTTP, a REST interface offers a much wider array of languages and programs that can access the interface. (If you'd like more information about the REST interface, you can go to my series of how-to's (http://blog.cloudera.com/blog/2013/03/how-to-use-the-apache-hbase-rest-interface-part-1/) about it.)

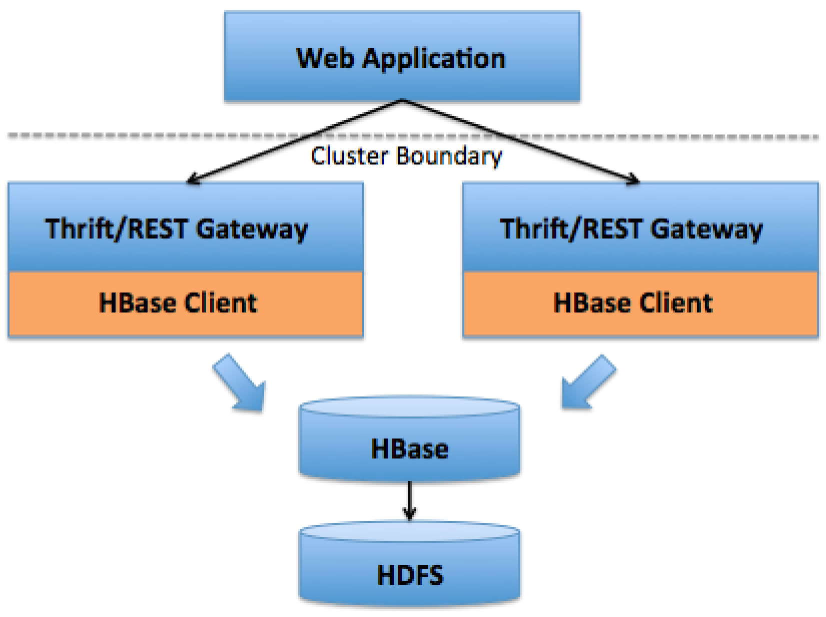
In this series of how-to's you'll learn your way around the Thrift interface and explore Python code samples for doing that. This first post will cover HBase Thrift, working with Thrift, and some boilerplate code for connecting to Thrift. The second post will show how to insert and get multiple rows at a time. The third post will explain how to use scans and some considerations when choosing between REST and Thrift.

The full code samples can be found on my GitHub account (https://github.com/eljefe6a/HBaseThrift).

HBase Thrift

Thrift (http://thrift.apache.org/) is a software framework that allows you to create cross-language bindings. In the context of HBase, Java is the only first-class citizen. However, the HBase Thrift interface allows other languages to access HBase over Thrift by connecting to a Thrift server that interfaces with the Java client.

For both Thrift and REST to work, another HBase daemon needs to be running to handle these requests. These daemons can be installed with the hbase-thrift and hbase-rest packages. The diagram below shows how Thrift and REST are placed in the cluster.



Note that the Thrift and REST client hosts usually don't run any other services (such as DataNodes or RegionServers) to keep the overhead low and responsiveness high for REST or Thrift interactions.

Make sure to install and start these daemons on nodes that have access to both the Hadoop cluster and the application that needs access to HBase. The Thrift interface doesn't have any built-in load balancing, so all load balancing will need to be done with external tools such a DNS round-robin, a virtual IP address, or in code. Cloudera Manager (http://blog.cloudera.com//content/cloudera/en/products/cloudera-manager.html) also makes it really easy to install and manage (https://ccp.cloudera.com/display/FREE45DOC/New+Features+in+Cloudera+Manager+Free+Edition+4#NewFeaturesinClouderaManagerFreeEdition4-What%27sNewinClouderaManagerFreeEdition4.5.0) the HBase REST and Thrift services. You can download (http://blog.cloudera.com//downloads) and try it out for free in Cloudera Standard!

The downside to Thrift is that it's more difficult to set up than REST. You will need to compile Thrift and generate the language-specific bindings. These bindings are nice because they give you code for the language you are working in — there's no need to parse XML or JSON like in REST; rather, the Thrift interface gives you direct access to the row data. Another nice feature is that the Thrift protocol has native binary transport; you will not need to base64 encode and decode data.

To start using the Thrift interface, you need to figure out which port it's running on. The default port for CDH is port 9090. For this post, you'll see the host and port variables used, here are the values we'll be using:

```
host = "localhost"
port = "9090"
```

You can set up (http://hbase.apache.org/book/security.html#d1949e4457) the Thrift interface to use Kerberos credentials for better security.

For your code, you'll need to use the IP address or fully qualified domain name of the node and the port running the Thrift daemon. I highly recommend making this URL a variable as it could change with network changes.

Language Bindings

Before you can create Thrift bindings, you must download and compile Thrift. There are no binary packages for Thrift that I could find, except on Windows. You will have to follow Thrift's instructions for the installation (http://thrift.apache.org/docs/install/) on your platform of choice.

Once Thrift is installed, you need to find the Hbase.thrift file. To define the services and data types in Thrift, you have to create an IDL (http://thrift.apache.org/docs/idl/) file. Fortunately, the HBase developers already created one (https://github.com/eljefe6a/HBaseThrift/blob/master/Hbase.thrift) for us. Unfortunately, the file isn't distributed as part of the CDH binary packages. (We will be fixing that in a future CDH release.) You will need to download the source package (http://archive.cloudera.com/cdh4/cdh/4/hbase-0.94.2-cdh4.2.0.tar.gz) of the HBase version you are using. Be sure to use the correct version of HBase as this IDL could change. In the compressed file, the path to the IDL is hbase-VERSION/src/main/resources/org/apache/hadoop/hbase/thrift/Hbase.thrift.

Thrift supports generating language bindings for more than 14 languages including Java, C++, Python, PHP, Ruby, and C#. To generate the bindings for Python, you would use the following command:

```
thrift -gen py /path/to/hbase/source/hbase-VERSION/src/main/resources/org/apache/hadoop/hb
```

Next, you will need to get the Thrift code for your language that contains all the classes for connection to Thrift and its protocols. This code can be found at /path/to/thrift/thrift-0.9.0/lib/py/src/.

Here are the commands I ran to create a Python project to use HBase Thrift:

```
$ mkdir HBaseThrift
$ cd HBaseThrift/
$ thrift -gen py ~/Downloads/hbase-0.94.2-cdh4.2.0/src/main/resources/org/apache/hadoop/hb
$ mv gen-py/* .
$ rm -rf gen-py/
$ mkdir thrift
$ cp -rp ~/Downloads/thrift-0.9.0/lib/py/src/* ./thrift/
```

I like to keep a copy of the Hbase.thrift file in the project to refer back to. It has a lot of "Javadoc" on the various calls, data objects, and return objects.

```
$ cp ~/Downloads/hbase-0.94.2-cdh4.2.0/src/main/resources/org/apache/hadoop/hbase/thrift/H
```

Boilerplate Code

You'll find that all your Python Thrift scripts will look very similar. Let's go through each part.

```
from thrift.transport import TSocket
from thrift.protocol import TBinaryProtocol
from thrift.transport import TTransport
from hbase import Hbase
```

These will import the Thrift and HBase modules you need.

```
# Connect to HBase Thrift server
transport = TTransport.TBufferedTransport(TSocket.TSocket(host, port))
protocol = TBinaryProtocol.TBinaryProtocolAccelerated(transport)
```

This creates the socket transport and line protocol and allows the Thrift client to connect and talk to the Thrift server.

```
# Create and open the client connection
client = Hbase.Client(protocol)
transport.open()
```

These lines create the Client object you will be using to interact with HBase. From this client object, you will issue all your Gets and Puts. Next, open the socket to the Thrift server.

```
# Do Something
```

Next you'll actually work with the HBase client. Everything is constructed, initialized, and connected. First, start using the client.

```
transport.close()
```

Finally, close the transport. This closes up the socket and frees up the resources on the Thrift server. Here is the code in its entirety for easy copying and pasting:

```
from thrift.transport import TSocket
from thrift.protocol import TBinaryProtocol
from thrift.transport import TTransport
from hbase import Hbase

# Connect to HBase Thrift server
transport = TTransport.TBufferedTransport(TSocket.TSocket(host, port))
protocol = TBinaryProtocol.TBinaryProtocolAccelerated(transport)

# Create and open the client connection
client = Hbase.Client(protocol)
transport.open()

# Do Something

transport.close()
```

In HBase Thrift's Python implementation, all values are passed around as strings. This includes binary data like an integer. All column values are held in the TCell object. Here is the definition in the Hbase.thrift file:

```
struct TCell{
  1:Bytes value,
  2:i64 timestamp
}
```

Notice the change to a string when the Python code is generated:

```
thrift_spec = (
    None, # 0
    (1, TType.STRING, 'value', None, None, ), # 1
    (2, TType.I64, 'timestamp', None, None, ), # 2
)
```

I wrote a helper method (https://github.com/eljefe6a/HBaseThrift/blob/master/common.py) to make it easier to deal with 32-bit integers. To change an integer back and forth between a string, you use these two methods.

```
# Method for encoding ints with Thrift's string encoding
def encode(n):
     return struct.pack("i", n)

# Method for decoding ints with Thrift's string encoding
def decode(s):
     return struct.unpack('i', s)[0]
```

Keep this caveat in mind as you work with binary data in Thrift. You will need to convert binary data to strings and vice versa.

Erroring Out

It's not as easy as it could be to understand errors in the Thrift interface. For example, here's the error that comes out of Python when a table is not found:

```
Traceback (most recent call last):
  File "./get.py", line 17, in &lt;module&gt;
    rows = client.getRow(tablename, "shakespeare-comedies-000001")
  File "/mnt/hgfs/jesse/repos/DevHivePigHBaseVM/training_materials/hbase/exercises/python_t
    return self.recv_getRow()
  File "/mnt/hgfs/jesse/repos/DevHivePigHBaseVM/training_materials/hbase/exercises/python_t
    raise result.io
hbase.ttypes.IOError: IOError(_message='doesnotexist')
```

All is not lost though because you can look at the HBase Thrift log file. On CDH, this file is located at /var/log/hbase/hbase-hbase-thrift-localhost.localdomain.log. In the missing table example, you would see an error in the Thrift log saying the table does not exist. It's inconvenient, but you can debug from there.

In the next installment, I'll cover inserting and getting rows.

Jesse Anderson is an instructor for Cloudera University.

apache (https://blog.cloudera.com/blog/tag/apache/) Apache HBase (https://blog.cloudera.com/blog/tag/apache-hbase/) CDH (https://blog.cloudera.com/blog/tag/cdh/) CDH Release (https://blog.cloudera.com/blog/tag/cdh-release/) CDH4 (https://blog.cloudera.com/blog/tag/cdh4/) cloudera (https://blog.cloudera.com/blog/tag/cloudera/) Cloudera Manager (https://blog.cloudera.com/blog/tag/cloudera-manager/) data (https://blog.cloudera.com/blog/tag/data/) developers (https://blog.cloudera.com/blog/tag/developers/) Hadoop (https://blog.cloudera.com/blog/tag/hadoop/) HBase (https://blog.cloudera.com/blog/tag/hbase/) installation (https://blog.cloudera.com/blog/tag/installation/) java (https://blog.cloudera.com/blog/tag/java/) log (https://blog.cloudera.com/blog/tag/log/) platform (https://blog.cloudera.com/blog/tag/platform/) python (https://blog.cloudera.com/blog/tag/python/) release (https://blog.cloudera.com/blog/tag/release/) REST (https://blog.cloudera.com/blog/tag/rest/) ruby (https://blog.cloudera.com/blog/tag/ruby/) security (https://blog.cloudera.com/blog/tag/security/) thrift (https://blog.cloudera.com/blog/tag/thrift/) Training (https://blog.cloudera.com/blog/tag/training-2/)