Oracle NoSQL Database

Javascript Driver Quick Start

12c Release 1

(Library Version 12.1.3.3)

ORACLE®

NOSQL DATABASE

Legal Notice

5/18/2015

Table of Contents

Introduction

This article provides a quick introduction to the Oracle NoSQL Database Javascript driver. This driver provides native Javascript client access to data stored in Oracle NoSQL Database tables.

The Javascript driver is available as a separate download from the Oracle NoSQL Database server package. You can obtain both the server and the driver download packages from:

http://www.oracle.com/technetwork/database/database-technologies/nosqldb/downloads/index.html

Note

The Javascript driver is compatible with Node.js 0.10.26 and later. It is also compatible with io.js 1.7.1 and later.

To work, the Javascript driver requires use of a proxy server which translates network activity between the Javascript client and the Oracle NoSQL Database store. The proxy is written in Java, and can run on any machine that is network accessible by both your Javascript client code and the Oracle NoSQL Database store. However, for performance and security reasons, Oracle recommends that you run the proxy on the same local host as your driver, and that the proxy be used in a 1:1 configuration with your drivers (that is, each instance of the proxy should be used with just a single driver instance).

This quick start assumes that you have read and understood the concepts described in the *Oracle NoSQL Database Getting Started with the Table API* guide. You can find that guide in your Oracle NoSQL Database server installation package, or find it here:

- HTML:

  http://docs.oracle.com/cd/NOSQL/html/GettingStartedGuideTables/index.html

- PDF:

  http://docs.oracle.com/cd/NOSQL/html/GettingStartedGuideTables/Oracle-NoSQLDB-GSG-Tables.pdf

The entirety of the API used by the Javascript driver is described in the *Javascript API Reference Guide*. This document can be accessed in the Javascript driver package by pointing your browser to `.../nosqldb-oraclejs-driver-`*`X.Y.Z`*`/doc/html/index.html`.

Installation

To run your Javascript clients, you need to have the `nosqldb-oraclejs` module installed locally or globally. You can install `nosqldb-oraclejs` in the Javascript driver directory by using the following command:

```
npm install [-g] nosqldb-oraclejs
```

`Nosqldb-oraclejs` is the module name of the Oracle NoSQL Database Javascript Client API published on http://www.npmjs.org. If `-g` is specified, the command installs the specified module so that it is globally accessible.

Then, to use the module, include `nosqldb-oraclejs` in your client code:

```
var nosqldb = require('nosqldb-oraclejs');
```

Using the Proxy Server

The proxy server is a Java application that accepts network traffic from the Table Javascript API, translates it into requests that the Oracle NoSQL Database store can understand, and then forwards the translated request to the store. The proxy also provides the reverse translation service by interpreting store responses and forwarding them to the client.

The proxy server can run on any network-accessible machine. It has minimal resource requirements and, in many cases, can run on the same machine as the client code is running.

Before your Javascript client can access the store, the proxy server must be running. It requires the following jar files to be in its class path, either by using the `java -cp` command line option, or by using the `CLASSPATH` environment variable:

Note

The proxy server, kvclient and their dependencies reside in the `kvproxy` directory.

- *KVHOME*/lib/kvclient.jar
- .../nosqldb-oraclejs-driver-*X.Y.Z*/kvproxy/kvproxy.jar

The proxy server itself is started using the `oracle.kv.proxy.KVProxy` command. At a minimum, the following information is required when you start the proxy server:

- `-helper-hosts`

  This is a list of one or more host:port pairs representing Oracle NoSQL Database storage nodes that the proxy server can use to connect to the store.

- `-port`

  The port where your client code can connect to this instance of the proxy server.

- `-store`

  The name of the store to which the proxy server is connecting.

A range of other command line options are available. In particular, if you are using the proxy server with a secure store, you must provide authentication information to the proxy server. In addition, you will probably have to identify a store name to the proxy server. For a complete description of the proxy server and its command line options, see Proxy Server Reference (page 16).

The simple examples provided in this quick start guide were written to work with an proxy server that is connected to a `kvlite` instance which was started with default values. The location of the `kvclient.jar` and `kvproxy.jar` files were provided using a `CLASSPATH` environment variable. The command line call used to start the proxy server was:

```
nohup java oracle.kv.proxy.KVProxy -port 7010 \
-helper-hosts localhost:5000 -store kvstore
```

Connecting to the Store

To perform store operations, you must establish a network connection between your client code and the store. There are two pieces of information that you must provide:

1. Identify the store's name, host and port using a configuration object. The host and port that you provide is for any machine hosting a node in the store. (Because the store is comprised of many hosts, there should be multiple host/port pairs for you to choose from).

   You create the configuration object using `nosqldb.Configuration()`.

2. Identify the host and port where the proxy is running. You also do this using the configuration object.

For example, suppose you have a Oracle NoSQL Database store named "kvstore" and it has a node running on n1.example.org at port 5000. Further, suppose you are running your proxy on the localhost using port 7010. Then you would open and close a connection to the store in the following way:

```
//Include nosqldb-oraclejs module
var nosqldb = require('nosqldb-oraclejs');

// Create a configuration object
var configuration = new nosqldb.Configuration();
configuration.proxy.startProxy = false;
configuration.proxy.host = 'localhost:7010';
configuration.storeHelperHosts = ['n1.example.org:5000'];
configuration.storeName = 'kvstore';

// Create a store with the specified configuration
var store = nosqldb.createStore(configuration);

store.on('open', function () {
  console.log('Store opened.');
  // Perform store operations here
  store.close();
}).on('close', function() {
  console.log('Store closed.');
  store.shutdownProxy();
}).on('error', function(error) {
  console.log('Error in the store.');
  console.log(error);
});
store.open();
```

If you are using a secure store, then your proxy server must first be configured to authenticate to the store. See Securing Oracle NoSQL Database Proxy Server (page 17)for details.

Once your proxy server is capable of accesing the secure store, you must indicate which user your driver wants to authenticate as when it performs store access. To do this, use `configuration.username`.

```
//Include nosqldb-oraclejs module
var nosqldb = require('nosqldb-oraclejs');

// Create a configuration object
var configuration = new nosqldb.Configuration();
configuration.proxy.startProxy = false;
configuration.proxy.host = 'localhost:7010';
configuration.storeHelperHosts = ['n1.example.org:5000'];
configuration.storeName = 'kvstore';
configuration.username = 'jsapp-username';

// Create a store with the specified configuration
var store = nosqldb.createStore(configuration);

store.on('open', function () {
  console.log('Store opened.');
  // Perform store operations here
  store.close();
}).on('close', function() {
  console.log('Store closed.');
  store.shutdownProxy();
}).on('error', function(error) {
  console.log('Error in the store.');
  console.log(error);
});
store.open();
```

Automatically Starting the Proxy Server

Your client code can start the proxy server on the local host when it opens the store using `configuration.proxy.startProxy`. To do this, set its value to true and also specify the location of the kvproxy.jar using `configuration.proxy.KVPROXY_JAR`. Also, specify the location of the `kvclient.jar` using `configuration.proxy.KVCLIENT_JAR`.

For example:

```
var nosqldb = require('nosqldb-oraclejs');
var configuration = new nosqldb.Configuration();
configuration.proxy.startProxy = true;
configuration.proxy.host = 'localhost:7010';
configuration.proxy.KVCLIENT_JAR = 'KVHOME/lib/kvclient.jar';
configuration.proxy.KVPROXY_JAR = "kvproxy/lib/kvproxy.jar";
configuration.storeHelperHosts = ['n1.example.org:5000'];
configuration.storeName = 'kvstore';

var store = nosqldb.createStore(configuration);

store.on('open', function () {
  console.log('Store opened.');
  // Perform store operations here
  store.close();
}).on('close', function() {
  console.log('Store closed.');
  store.shutdownProxy();
}).on('error', function(error) {
  console.log('Error in the store.');
  console.log(error);
});
store.open();
```

Be aware that if your proxy is connecting to a secure store, you also must indicate which user to authenticate as, and you must indicate where the security properties file is located on the host where the proxy server is running. To do this, use `configuration.username` and `configuration.proxy.securityFile`.

```
var nosqldb = require('nosqldb-oraclejs');
var configuration = new nosqldb.Configuration();
configuration.proxy.startProxy = true;
configuration.proxy.host = 'localhost:7010';
configuration.proxy.KVCLIENT_JAR = 'KVHOME/lib/kvclient.jar';
configuration.proxy.KVPROXY_JAR = "kvproxy/lib/kvproxy.jar";
configuration.storeHelperHosts = ['n1.example.org:5000'];
configuration.storeName = 'kvstore';
configuration.username = 'jsapp-username';
configuration.proxy.securityFile = 'KVROOT/security/client.security';

var store = nosqldb.createStore(configuration);

store.on('open', function () {
  console.log('Store opened.');
  // Perform store operations here
  store.close();
}).on('close', function() {
  console.log('Store closed.');
  store.shutdownProxy();
}).on('error', function(error) {
  console.log('Error in the store.');
  console.log(error);
});
store.open();
```

For information on configuring your proxy server to connect to a secure store, see Securing Oracle NoSQL Database Proxy Server (page 17).

Creating Table and Index Definitions

Before you can write data to tables in your store, you must define your tables using table DDL statements. You also use DDL statements to define indexes. The table DDL is described in detail in the *Oracle NoSQL Database Getting Started with the Table API* guide.

If you want to submit table DDL statements to the store from your Javascript client code, use `store.execute()`.

For example, to create a table:

```
store.on('open', function () {
console.log('Store opened');

store.execute('CREATE TABLE IF NOT EXISTS myTable ' +
  ' ( id LONG, name STRING, PRIMARY KEY(id) ) ',
  function(err){
    if (err)
      throw err;
    else {
       console.log('Table creation succeeded.');
       store.refreshTables();
       store.close();
    }
  });
}).on('close', function() {
console.log('Store closed.');
store.shutdownProxy();
}).on('error', function(error) {
console.log(error);
});
store.open();
```

Writing to a Table Row

Once you have defined a table in the store, use `store.put()` to create an empty table row. For example, for a table designed like this:

```
"CREATE TABLE IF NOT EXISTS myTable (id LONG, \
                       name STRING, \
                       PRIMARY KEY (id))"
```

You can write a row of table data in the following fashion:

```
store.on('open', function () {
console.log('Store opened');

store.execute('CREATE TABLE IF NOT EXISTS myTable ' +
  ' ( id LONG, name STRING, PRIMARY KEY(id) ) ',
  function(err){
    if (err)
      throw err;
    else {
       console.log('Table creation succeeded.');
       store.refreshTables();
       var row = {id: 0, name:'name'};
       store.put('myTable', row,
        function (err) {
         if (err)
           throw err;
         else
           console.log("Row inserted.");
           store.close();
         });
    }
  });
}).on('close', function() {
console.log('Store closed.');
store.shutdownProxy();
}).on('error', function(error) {
console.log(error);
});
store.open();
```

Other versions of `store.put()` exist which allow you to provide options and version information, and so forth. See the API Javascript documentation for details.

Deleting a Table Row

Use `store.deleteRow()` to delete a table row.

```
var primaryKey = {id:0};
store.delete('myTable', primaryKey,
   function (err) {
    if (err)
      throw err;
    else
      console.log("Row deleted.");
      });
```

Other versions of `store.delete()` exist which allow you to provide options and version information, and so forth. See the API Javascript documentation for details.

Reading a Single Table Row

To read a single table row, use `store.get()`.

For example, to retrieve the table row created in Writing to a Table Row (page 8):

```
var primaryKey = {id: 0};
store.get('myTable', primaryKey,
  function (err, result) {
   if (err)
     throw err;
   else
```

```
      console.log("Returned row:");
      console.log(result.currentRow);
    });
```

Reading Multiple Table Rows

Use `store.multiGet()` to read multiple rows from a table at a time. Rows are returned in primary key order. The primary key used must contain all the table's shard keys. If all of the shard keys are not present, then the function will return without error, but without any results.

For example, suppose you design a table like this:

```
CREATE TABLE newTable (
    itemType STRING,
    itemCategory STRING,
    itemClass STRING,
    itemColor STRING,
    itemSize STRING,
    price FLOAT,
    inventoryCount INTEGER,
    PRIMARY KEY (SHARD(itemType, itemCategory, itemClass), itemColor,
    itemSize)
)
```

You can populate it with data and then retrieve all of the rows in the table by providing just the shard key because, in this example, the shard key is identical for all the rows in the table.

```
store.on('open', function () {
console.log('Store opened');

store.execute('CREATE TABLE IF NOT EXISTS newTable ' +
  ' ( itemType STRING, itemCategory STRING, itemClass STRING,
    itemColor STRING, itemSize STRING, price FLOAT,
    inventoryCount INTEGER,
    primary key (shard(itemType,itemCategory,
                       itemClass),itemColor,itemSize) ) ',
  function(err){
    if (err)
      throw err;
    else {
       console.log('Table creation succeeded.');
       store.refreshTables();
    var row = {itemType:"Hats", itemCategory:"baseball",
    itemClass :"longbill", itemColor:"red",
    itemSize:"small", price:12.07, inventoryCount:127};
    store.put('newTable', row, function (err) {
    if (err)
      throw err;
    else {
      row = {itemType:"Hats", itemCategory:"baseball",
        itemClass :"longbill", itemColor:"red",
        itemSize:"medium", price:13.07,
        inventoryCount:201};
      store.put('newTable', row, function (err) {
        if (err)
          throw err;
        else {
          row = { itemType: "Hats", itemCategory:"baseball",
            itemClass: "longbill", itemColor: "red",
            itemSize: "large", price: 14.07,
            inventoryCount: 309};
          store.put('newTable', row, function (err) {
            if (err)
              throw err;
            else {
              var key = {itemType: "Hats", itemCategory: "baseball",
                itemClass: "longbill"};
              store.multiGet('newTable', key, function(err, result){
                if (err)
                  throw err;
                else {
                  console.log("Returned rows:");
            for (var key in result.rowsWithMetadata)
                  console.log(result.rowsWithMetadata[key].row);
                  store.close();
                }
              });
            }
          });
        }
      });
    }
  });
}
  })
}).on('close', function() {
console.log('Store closed.');
store.shutdownProxy();
}).on('error', function(error) {
console.log(error);
});
store.open();
```

Reading Using Indexes

Use `fieldRange` to read table rows based on a specified index. To use this function, the index must first be created using the `CREATE INDEX` statement.

For example, suppose you have a table defined like this:

```
CREATE TABLE table_index (
     surname STRING,
     familiarName STRING,
     userID STRING,
     phonenumber STRING,
     address STRING,
     email STRING,
     dateOfBirth STRING,
     PRIMARY KEY (SHARD(surname, familiarName), userID))
```

With this index:

```
CREATE INDEX DoB ON table_index (dateOfBirth)
```

You can populate it with data and then you can read using the `DoB` index like this:

```
store.on('open', function () {
console.log('Store opened');

store.execute('CREATE TABLE IF NOT EXISTS table_index ' +
  ' ( surname STRING, familiarname STRING,
    userID STRING, phonenumber STRING,
    address STRING, email STRING, dateofbirth STRING,
    primary key (shard(surname,familiarname),userID)) ',
  function(err){
    if (err)
      throw err;
    else {
       console.log('Table creation succeeded.');
       store.refreshTables();
       store.execute('CREATE INDEX IF NOT EXISTS DoB
                      ON table_index (dateOfBirth)');

    var row = { surname:"Anderson", familiarName:"Pete",
      userID:"panderson", phonenumber:"555-555-5555",
      address:"1122 Somewhere Court",
      email:"panderson@example.com",
      dateOfBirth:"1994-05-01"};
    store.put('table_index', row, function (err) {
      if (err)
        throw err;
      else {
        row = { surname:"Andrews", familiarName:"Veronica",
          userID:"vandrews", phonenumber:"666-666-6666",
          address:"5522 Nowhere Court",
          email:"vandrews@example.com",
          dateOfBirth:"1973-08-21"};
          store.put('table_index', row, function (err) {
          if (err)
            throw err;
          else {
            row = { surname:"Bates", familiarName:"Pat",
              userID:"pbates", phonenumber:"777-777-7777",
              address:"12 Overhere Lane",
              email:"pbates@example.com",
              dateOfBirth:"1988-02-20"};
            store.put('table_index', row,
            function (err) {
              if (err)
                throw err;
              else {
                row = { surname:"Macar", familiarName:"Tarik",
                  userID:"tmacar", phonenumber:"888-888-8888",
                  address:"100 Overthere Street",
                  email:"tmacar@example.com",
                  dateOfBirth:"1990-05-17"};
                store.put('table_index', row,
                function (err) {
                  if (err)
                    throw err;
                  else {
                    var fieldRange =
                      new nosqldb.Types.FieldRange(
                        "dateOfBirth", "1990-01-01", true,
                        "2000-01-01", true
                      );
                    store.indexIterator('table_index', 'DoB', {
                        fieldRange:fieldRange,
                        direction: nosqldb.Types.Direction.FORWARD
                      },
                      function (err, iterator) {
                         iterator.on('done', function () {
                         store.close();
                        });
                        iterator.forEach(function (err, rowWithMetadata) {
                        console.log(rowWithMetadata.row);
                        });
                        });
                  }
                });
              }
            });
          }
        });
      }
    });
}})
}).on('close', function() {
console.log('Store closed.');
store.shutdownProxy();
}).on('error', function(error) {
```

```
console.log(error);
});
store.open();
```

Sequence Execution

Use `store.executeOperations()` to hold a sequence of write operations. All the write operations will execute as a single atomic structure so long as all the operations share the same shard key.

For example, if you had a table populated with data such as is described in Reading Multiple Table Rows (page 10), then you could update the price and inventory values for each row of the table in an atomic operation like this:

```
store.on("open",function (err) {
  if (err)
    throw err;
  var operations = [];
  var row = {
    itemType: "Hats", itemCategory: "baseball",
    itemClass: "longbill",
    itemColor: "red", itemSize: "small",
    price: 12.07, inventoryCount: 127
  };
  operations.push (new nosqldb.Types.Operation('newTable',
    nosqldb.Types.OperationType.PUT, row,
    nosqldb.Types.ReturnChoice.ALL, true, null));

  row = {
    itemType: "Hats", itemCategory: "baseball",
    itemClass: "longbill",
    itemColor: "red", itemSize: "medium",
    price: 13.07, inventoryCount: 201
  };
  operations.push (new nosqldb.Types.Operation('newTable',
  nosqldb.Types.OperationType.PUT, row,
    nosqldb.Types.ReturnChoice.ALL, true, null));

  row = {
    itemType: "Hats", itemCategory: "baseball",
    itemClass: "longbill",
    itemColor: "red", itemSize: "large",
    price: 14.07, inventoryCount: 309
  };
  operations.push (new nosqldb.Types.Operation('newTable',
    nosqldb.Types.OperationType.PUT, row,
    nosqldb.Types.ReturnChoice.ALL, true, null));

  store.executeUpdates(operations, function(err, result){
    if (err)
```

```
      throw err;
    else {
      console.log("Inserted rows.");
      store.close();
    }
  });

}).on('close', function() {
console.log('Store closed.');
store.shutdownProxy();
}).on('error', function(error) {
console.log(error);
});
store.open();
```

Setting Consistency Guarantees

By default, read operations are performed with a consistency of guarantee of `KV_CONSISTENCY_NONE`. Use the following function to create a consistency guarantee that overrides this default: `nosqldb.Types.Consistency()`.

You then use `nosqldb.Types.ReadOptions()` with the specified consistency guarantee when performing a read operation in the store.

For example, the code fragment shown in Reading a Single Table Row (page 9) can be rewritten to use a consistency policy in the following way:

```
var SimpleConsistency = nosqldb.Types.SimpleConsistency;
var readOptions = new nosqldb.Types.ReadOptions(
                                        SimpleConsistency.ABSOLUTE, 1000);
// Setting up the primary key
var primaryKey = {id: 0};
store.get('myTable', primaryKey, readOptions,
  function (err, result) {
   if (err)
     throw err;
   else
     console.log("Reading row:");
     console.log(result.currentRow);
   });
```

Setting Durability Guarantees

By default, write operations are performed with a durability guarantee of `KV_DURABILITY_COMMIT_NO_SYNC`. Use the following function to create a durability guarantee that overrides this default: `nosqldb.Types.Durability()`.

You then use `nosqldb.Types.WriteOptions()` with the specified durability guarantee when performing a write operation in the store.

For example, the code fragment shown in Writing to a Table Row (page 8) can be rewritten to use a durability policy in the following way:

```
var durability = new nosqldb.Types.Durability(
                   nosqldb.Types.SyncPolicy.SYNC,
                   nosqldb.Types.ReplicaAckPolicy.NONE,
                   nosqldb.Types.SyncPolicy.SYNC);
var writeOptions = new nosqldb.Types.WriteOptions(durability, 1000);
// Setting up the row
var row = {id: 0, name:'name'};

store.put('myTable', row, writeOptions,
  function (err) {
   if (err)
     throw err;
   else
     console.log("Inserted row");
   });
```

Proxy Server Reference

The proxy server command line options are:

```
nohup java -cp KVHOME/lib/kvclient.jar:kvproxy/lib/kvproxy.jar
oracle.kv.proxy.KVProxy -help
 -port <port-number> Port number of the proxy server. Default: 5010
 -store <store-name> Required KVStore name. No default.
 -helper-hosts <host:port,host:port,...>   Required list of KVStore
        hosts and ports (comma separated).
 -security <security-file-path>  Identifies the security file used
        to specify properties for login. Required for connecting to
        a secure store.
  -username <user>  Identifies the name of the user to login to the
        secured store. Required for connecting to a secure store.
  -read-zones <zone,zone,...>  List of read zone names.
  -max-active-requests <int> Maximum number of active requests towards
        the store.
  -node-limit-percent <int> Limit on the number of requests, as a
        percentage of the requested maximum active requests.
  -request-threshold-percent <int> Threshold for activating request
        limiting, as a percentage of the requested maximum active
        requests.
  -request-timeout <long> Configures the default request timeout in
        milliseconds.
  -socket-open-timeout <long> Configures the open timeout in
        milliseconds used when establishing sockets to the store.
  -socket-read-timeout <long> Configures the read timeout in
        milliseconds associated with the underlying sockets to the
        store.
  -max-iterator-results <long> A long representing the maximum
```

```
        number of results returned in one single iterator call.
        Default: 100
  -iterator-expiration <long>  Iterator expiration interval in
        milliseconds.
  -max-open-iterators <int>    Maximum concurrent opened iterators.
        Default: 10000
  -num-pool-threads <int>      Number of proxy threads. Default: 20
  -max-concurrent-requests <int>      The maximum number of
        concurrent requests per iterator. Default: <num_cpus * 2>
  -max-results-batches <int>      The maximum number of results
        batches that can be held in the proxy per iterator.
        Default: 0
  -help  Usage instructions.
  -version  Print KVProxy server version number.
  -verbose  Turn verbose flag on.
```

Always start the Oracle NoSQL Database store before starting the proxy server.

When connecting to a non-secured store, the following parameters are required:

- `-helper-hosts`
- `-port`
- `-store`

When connecting to a secured store, the following parameters are also required:

- `-security`
- `-username`

Note

Drivers are able to start and stop the proxy server on the local host if properly configured. See Automatically Starting the Proxy Server (page 6) for details.

Securing Oracle NoSQL Database Proxy Server

If configured properly, the proxy can access a secure installation of Oracle NoSQL Database. To do this, the `-username` and `-security` proxy options must be specified.

The following example describes how to add security to an Oracle NoSQL Database single node deployment. The example also shows how to initiate a connection to the Oracle NoSQL Database replication nodes.

To install Oracle NoSQL Database securely:

```
java -Xmx256m -Xms256m \
-jar KVHOME/lib/kvstore.jar makebootconfig \
-root KVROOT -port 5000 \
-admin 5001 -host node01 -harange 5890,5900 \
-store-security configure -pwdmgr pwdfile -capacity 1
```

1. Run the `makebootconfig` utility with the required `-store-security` option to set up the basic store configuration with security:

2. In this example, `-store-security configure` is used, so the `security configuration` utility is run as part of the makebootconfig process and you are prompted for a password to use for your keystore file:

   ```
   Enter a password for the Java KeyStore:
   ```

3. Enter a password for your store and then reenter it for verification. In this case, the password file is used, and the `securityconfig` tool will automatically generate the following security related files:

   ```
   Enter a password for the Java KeyStore: ***********
   Re-enter the KeyStore password for verification: ***********
   Created files:
   security/client.trust
   security/client.security
   security/store.keys
   security/store.trust
   security/store.passwd
   security/security.xml
   ```

   ## Note

   In a multi-host store environment, the security directory and all files contained in it should be copied to each server that will host a Storage Node. For more information on multiple node deployments see the Oracle NoSQL Database Security Guide.

4. Start the Storage Node Agent (SNA):

   ```
   nohup java -Xmx256m -Xms256m \
   -jar KVHOME/lib/kvstore.jar start -root KVROOT&
   ```

   When a newly created store with a secure configuration is first started, there are no user definitions available against which to authenticate access. To reduce risk of unauthorized access, an admin will only allow you to connect to it from the host on which it is running. This security measure is not a complete safeguard against unauthorized access. It is important that you do not provide local access to machines running KVStore. In addition, you should perform steps 5, 6 and 7 soon after this step to minimize the time period in which the admin might be accessible without full authentication. For more information on maintaining a secure store see the *Oracle NoSQL Database Security Guide*.

5. Start `runadmin` in security mode on the KVStore server host (node01). To do this, use the following command:

   ```
   java -Xmx256m -Xms256m \
   -jar KVHOME/lib/kvstore.jar \
   ```

```
runadmin -port 5000 -host node01 \
-security KVROOT/security/client.security
Logged in admin as anonymous
```

6. Use the `configure -name` command to specify the name of the KVStore that you want to configure:

```
kv-> configure -name mystore
Store configured: mystore
```

7. Configure the KVStore by deploying a Zone, a Storage Node, and an Admin Node. Then, create a Storage Node Pool. Finally, create and deploy a topology.

```
kv-> plan deploy-zone -name mydc -rf 1 -wait
Executed plan 2, waiting for completion...
Plan 2 ended successfully
kv-> plan deploy-sn -zn zn1 -port 5000 -host node01 -wait
Executed plan 3, waiting for completion...
Plan 3 ended successfully
kv-> plan deploy-admin -sn sn1 -port 5001 -wait
Executed plan 4, waiting for completion...
Plan 4 ended successfully
kv-> pool create -name mypool
kv-> pool join -name mypool -sn sn1
Added Storage Node(s) [sn1] to pool mypool
kv-> topology create -name mytopo -pool mypool -partitions 30
Created: mytopo
kv-> plan deploy-topology -name mytopo -wait
Executed plan 5, waiting for completion...
Plan 5 ended successfully
```

8. Create an admin user. In this case, user `root` is defined:

```
kv-> plan create-user -name root -admin -wait
Enter the new password: ********
Re-enter the new password: ********
Executed plan 6, waiting for completion...
Plan 6 ended successfully
```

9. Create a new password file to store the credentials needed to allow clients to login as the admin user (root):

```
java -Xmx256m -Xms256m \
-jar KVHOME/lib/kvstore.jar securityconfig \
pwdfile create -file KVROOT/security/login.passwd
java -Xmx256m -Xms256m \
-jar KVHOME/lib/kvstore.jar securityconfig pwdfile secret \
-file KVROOT/security/login.passwd -set -alias root
Enter the secret value to store: ********
Re-enter the secret value for verification: ********
Secret created
OK
```

Note

The password must match the one set for the admin in the previous step.

10. At this point, it is possible to connect to the store as the root user. To login, you can use either the `-username <user>` runadmin argument or specify the "oracle.kv.auth.username" property in the security file.

    In this example, a security file (mylogin.txt) is used. To login, use the following command:

    ```
    java -Xmx256m -Xms256m \
    -jar KVHOME/lib/kvstore.jar runadmin -port 5000 \
    -host localhost -security mylogin
    Logged in admin as root
    ```

    The file `mylogin.txt` should be a copy of the `client.security` file with additional properties settings for authentication. The file would then contain content like this:

    ```
    oracle.kv.auth.username=root
    oracle.kv.auth.pwdfile.file=KVROOT/security/login.passwd
    oracle.kv.transport=ssl
    oracle.kv.ssl.trustStore=KVROOT/security/client.trust
    oracle.kv.ssl.protocols=TLSv1.2,TLSv1.1,TLSv1
    oracle.kv.ssl.hostnameVerifier=dnmatch(CN\=NoSQL)
    ```

Then, to run KVProxy and access the secure Oracle NoSQL Database deployment:

```
java -cp KVHOME/lib/kvclient.jar:KVPROXY/lib/kvproxy.jar
oracle.kv.proxy.KVProxy -helper-hosts node01:5000 -port 5010
-store mystore -username root -security mylogin
Nov 21, 2014 12:59:12 AM oracle.kv.proxy.KVProxy <init>
INFO: PS: Starting KVProxy server
Nov 21, 2014 12:59:12 AM oracle.kv.proxy.KVProxy <init>
INFO: PS: Connect to Oracle NoSQL Database mystore nodes : localhost:5000
Nov 21, 2014 12:59:13 AM oracle.kv.proxy.KVProxy <init>
INFO: PS:   ... connected successfully
Nov 21, 2014 12:59:13 AM oracle.kv.proxy.KVProxy startServer
INFO: PS: Starting listener ( Half-Sync/Half-Async server - 20
no of threads on port 5010)
```

Note

Because this proxy server is being used with a secure store, you should limit the proxy server's listening port (port 5010 in the previous example) to only those hosts running authorized clients.

Trouble Shooting the Proxy Server

If your client is having trouble connecting to the store, then the problem can possibly be with your client code, with the proxy and its configuration, or with the store. To help determine what might be going wrong, it is useful to have a high level understanding of what happens when your client code is connecting to a store.

1. First, your client code tries to connect to the `ip:port` pair given for the proxy.
2. If the connection attempt is not successful, and your client code indicates that the proxy should be automatically started, then:
    a. The client driver will prepare a command line that starts the proxy on the local host. This command line includes the path to the `java` command, the classpath to the two jar files required to start the proxy, and the parameters required to start the proxy and connect to the store (these include the local port for the proxy to listen on, and the store's connection information).
    b. The driver executes the command line. If there is a problem, the driver might be able to provide some relevant error information, depending on the exact nature of the problem.
    c. Upon command execution, the driver waits for a few seconds for the connection to complete. During this time, the proxy will attempt to start. At this point it might indicate a problem with the classpath.

        Next, it will check the version of `kvclient.jar` and indicate if it is not suited.

        After that, it will check the connection parameters, and indicate problems with those, if any.

        Then the proxy will actually connect to the store, using the `helper-hosts` parameter. At this time, it could report connection errors such as the store is not available, security credentials are not available, or security credentials are incorrect.

        Finally, the proxy tries to listen to the indicated port. If there's an error listening to the port (it is already in use by another process, for example), the proxy reports that.
    d. If any errors occur in the previous step, the driver will automatically repeat the entire process again. It will continue to repeat this process until it either successfully obtains a connection, or it runs out of retry attempts.

        Ultimately, if the driver cannot successfully create a connection, the driver will return with an error.
3. If the driver successfully connects to the proxy, it sends a verify message to the proxy. This verify message includes the helper-host list, the store name, the username (if using a secure store), and the readzones if they are being used in the store.

    If there is anything wrong with the information in the verify message, the proxy will return an error message. This causes the proxy to check the verify parameters so as to ensure that the driver is connected to the right store.
4. If there are no errors seen in the verify message, then the connection is established and store operations can be performed.

To obtain the best error information possible when attempting to troubleshoot a connection problem, start the proxy with the `-verbose` command line option. Also, you can enable assertions in the proxy Java code by using the `java -ea` command line option.

Between these two mechanisms, the proxy will provide a great deal of information. To help you analyze it, you can enable logging to a file. To do this:

Start the proxy with the following parameter:

```
java -cp KVHOME/lib/kvclient.jar:KVPROXY/lib/kvproxy.jar
-Djava.util.logging.config.file=logger.properties
oracle.kv.proxy.KVProxy -helper-hosts node01:5000 -port 5010
-store mystore -verbose
```

The file `logger.properties` would then contain content like this:

```
# Log to file and console
handlers = java.util.logging.FileHandler, java.util.logging.ConsoleHandler
## ConsoleHandler ##
java.util.logging.ConsoleHandler.level = FINE
java.util.logging.ConsoleHandler.formatter =
                                       java.util.logging.SimpleFormatter
## FileHandler ##
java.util.logging.FileHandler.formatter = java.util.logging.SimpleFormatter
# Limit the size of the file to x bytes
java.util.logging.FileHandler.limit = 100000
# Number of log files to rotate
java.util.logging.FileHandler.count = 1
# Location and log file name
# %g is the generation number to distinguish rotated logs
java.util.logging.FileHandler.pattern = ./kvproxy.%g.log
```

Configuration parameters control the size and number of rotating log files used (similar to java logging, see java.util.logging.FileHandler). For a rotating set of files, as each file reaches a given size limit, it is closed, rotated out, and a new file is opened. Successively older files are named by adding "0", "1", "2", etc. into the file name.