Getting Started with SAS® Viya™ 3.2 for Java

Requirements

To use Java with SAS Cloud Analytic Services, the client machine that runs Java must meet the following requirements:

- Use a Java 8 runtime.
- Use the following runtime dependencies:
  - ANTLR runtime (3.5.2)
  - cas-client-3.1.4.jar (SAS Viya 3.1)

    cas-client-3.2.9.jar (SAS Viya 3.2)

    The JAR file is available for download at http://support.sas.com/downloads/package.htm?pid=1976.
  - Google Protocol Buffers (2.6.1)
  - json-20141113 (this dependency is required to use Java with the REST interface)

There is additional information that is common with other programming languages and authenticating with the server.

See Also

- *Client Authentication Using an Authinfo File*

How to Run Actions

View Server Status

The following code shows how to run the serverStatus action.

2

Note: This example, and all the examples in this document require that you have access to a running instance of SAS Cloud Analytic Services. As a document convention, the host is presented as cloud.example.com and the network port is 5570. Contact your system administrator for the values to substitute.

ServerStatusExample1.java

```
import com.sas.cas.CASActionResults;
import com.sas.cas.CASClient;
import com.sas.cas.CASClientInterface;
import com.sas.cas.CASValue;
import com.sas.cas.actions.builtins.ServerStatusOptions;

public class ServerStatusExample1 {
  public static void main(String[] args) throws Exception {
    try (CASClientInterface client = new CASClient("cloud.example.com", 5570)) { // 1

      ServerStatusOptions options = new ServerStatusOptions();                    // 2

      CASActionResults<CASValue> results = client.invoke(options);

      for (int i = 0; i < results.getResultsCount(); i++) {
        System.out.println(results.getResult(i));
      }

      // The CASClientInterface class implements java.lang.AutoCloseable
      // that was introduced in Java 1.7. The client.close() method is
      // automatically called to end the session and close the network socket.
    }
  }
}
```

1 The instance of the CASClient class is used to connect to SAS Cloud Analytic Services and start your session. Substitute your server's host name and port.

2 The instance of the ServerStatusOptions class identifies the action to run. In this case, the ServerStatusOptions class corresponds to the serverStatus action.

Results of the ServerStatus Action

```
{
 About= {
  CAS="Cloud Analytic Services"
  Version="3.01"
  VersionLong="V.03.01M0D09062016"
  Copyright="Copyright 2014-2016 SAS Institute Inc. All Rights Reserved."
  System=  {
   Hostname="cloud"
   OS Name="Linux"
   OS Family="LIN X64"
   OS Release="2.6.32-573.el6.x86_64"
   OS Version="#1 SMP Wed Jul 1 18:23:37 EDT 2015"
   Model Number="x86_64"
  }
  license=  {
   site="SAS Institute Inc."
   siteNum=1
   expires="03Nov2016:00:00:00"
   gracePeriod=62
   warningPeriod=31
  }
 }
}
{
 server=server
Node Count Total Actions
---------- -------------
         8             1
1 row

}
{
 nodestatus=nodes
Node Name             Role       Uptime (Sec) Running Stalled
--------------------- ---------- ------------ ------- -------
cloud1.unx.sas.com    worker            0.044       0       0
cloud2.unx.sas.com    worker            0.042       0       0
cloud3.unx.sas.com    worker            0.045       0       0
cloud4.unx.sas.com    worker            0.042       0       0
cloud5.unx.sas.com    worker            0.045       0       0
cloud6.unx.sas.com    worker            0.043       0       0
cloud7.unx.sas.com    worker            0.045       0       0
cloud.unx.sas.com     controller        0.041       0       0    1
8 rows

}
```

1 The results are shown for a distributed server that uses eight machines. For a single-machine server, the results include the line for the controller node only.

The results of the serverStatus action include three keys, as shown in the output. You can filter the results by the key:

```
...
CASActionResults<CASValue> results = client.invoke(options);

if (null != results) {
  System.out.println(CASValue.getValue(results, "server"));
}
...
```

The output includes the results for the Server key only:

4

```
{
 server=server
Node Count Total Actions
---------- -------------
         8             1
1 row

}
```

Fluent Interface Programming

The SAS Viya 3.2 release adds a fluent interface. For example, the following snippet invokes the builtins.serverStatus action by using the getActionSets method on the client object:

```
CASActionResults<CASValue> result = client.getActionSets()
                                          .builtins()
                                          .serverStatus()
                                          .invoke();
```

An alternative is to invoke the action using the client context alone (this uses the last client that is instantiated or used):

```
CASActionResults<CASValue> result = new ServerStatusOptions().invoke();
```

See Also

See Modification: Use the Fluent Interface on page 8.

Example: View Descriptive Statistics

Setup

This example requires that you have in-memory data to analyze. You can use a sample program that is included with the cas-client JAR file to read a CSV file and then transfer the data to the server.

Download the orsales.csv file from http://support.sas.com/documentation/onlinedoc/viya/examples.htm.

After you download the file, run the following command. You must have the cas-client, ANTLR, and the Google protobuf JAR files in your CLASSPATH:

```
java com.sas.cas.samples.table.AddCSVSample host="cloud.example.com" port=5570 path=orsales.csv
promote=true table=orsales
```

TIP You can run `java com.sas.cas.samples.table.AddCSVSample -help` to list the command-line options.

```
Guessing variable info from orsales.csv
Found 8 variable(s)
{formattedLength=4,length=8,name="Year",offset=0,rType="NUMERIC",type="SAS"}
{length=16,name="Quarter",offset=8,rType="CHAR",type="VARCHAR"}
{length=16,name="Product_Line",offset=24,rType="CHAR",type="VARCHAR"}
{length=16,name="Product_Category",offset=40,rType="CHAR",type="VARCHAR"}
{length=16,name="Product_Group",offset=56,rType="CHAR",type="VARCHAR"}
{formattedLength=3,length=8,name="Quantity",offset=72,rType="NUMERIC",type="SAS"}
{formattedLength=7,length=8,name="Profit",offset=80,rType="NUMERIC",type="SAS"}
{formattedLength=4,length=8,name="Total_Retail_Price",offset=88,rType="NUMERIC",type="SAS"}
[Authenticated to cloud.example.com:5570] User=sasdemo
Table change event received
Processed 912 line(s) and 912 row(s)
```

The promote command-line option is needed to add the table to global scope in your personal caslib. (Information about caslibs is available in *SAS Cloud Analytic Services: Fundamentals* and the Tables action set information in *SAS Viya: System Programming Guide*.)

To confirm that the table is still in-memory on the server, you can run the TableInfoSample utility:

```
java com.sas.cas.samples.table.TableInfoSample host="cloud.example.com" port=5570
```

The key information is to ensure that the results include the Orsales table:

```
[Authenticated to cloud.example.com:5570] User=sasdemo
Disposition Events:
 CASDisposition [time=2016-09-21 15:33:55.057, severity=0, reason=0, statusCode= 0, statusMessage=, debugInfo=]
Performance Events:
 CASPerformanceEvent [time=2016-09-21 15:33:54.986, elapsedTime=0.015880000000000002, cpuTime=0.0,
systemTime=0.0, totalNodes=0, totalCores=0, wallTime=293, stats={system_nodes=8, system_cores=256,
memory=1111040, cpu_system_time=0.021995999999999998, elapsed_time=0.015880000000000002, memory_os=52854784,
memory_quota=52854784, cpu_user_time=0.009998, system_total_memory=2166657482752}]
(No log events)
Result Values:
{
 TableInfo=TableInfo Table Information for Caslib CASUSERHDFS(sasdemo)
Name    Rows Columns Encoding Created            Last Modified      Character Set CreateTime
ModTime           Global Repeated View Loaded Source Source Caslib Compressed Table Creator Last Table Modifier
------- ---- ------- -------- ------------------ ------------------ ------------- -----------------
---------------- ------ -------- ---- ------------- ------------- ---------- ------------- --------------------
ORSALES  912       8 utf-8    21Sep2016:15:23:51 21Sep2016:15:23:51 UTF8          1790090631.241056
1790090631.241056      1        0    0                                      0 sasdemo
1 row

}
```

Run the Summary Action

After the Orsales data is loaded into memory, you can run a program that accesses the in-memory table and runs the summary action on the data.

SummaryExample1.java

```
import com.sas.cas.CASActionResults;
import com.sas.cas.CASClient;
import com.sas.cas.CASClientInterface;
import com.sas.cas.CASValue;

import com.sas.cas.actions.simple.SummaryOptions;
import static com.sas.cas.actions.simple.SummaryOptions.SUBSET.*;
import com.sas.cas.actions.Castable;
```

```
public class SummaryExample1 {
  public static void main(String[] args) throws Exception {
    try (CASClientInterface client = new CASClient("cloud.example.com", 5570)) {

      SummaryOptions so = new SummaryOptions();
      so.setSummarySubset(                                                    // 1
         new SummaryOptions.SUBSET[]{MIN, MAX, MEAN, N, NMISS, STD, STDERR}
      );

      Castable table = new Castable();                                        // 2
      table.setName("orsales");
      so.setTable(table);

      CASActionResults<CASValue> results = client.invoke(so);

      for (int i = 0; i < results.getResultsCount(); i++) {
        System.out.println(results.getResult(i));
      }
    }
  }
}
```

1 A subset of the available descriptive statistics is used with the SummaryOptions class.

2 The Castable class is used to represent in-memory tables. In the program, only the table name is set, but you can set several parameters, such as those used in BY-group processing, filtering with a where parameter, and so on.

Summary Action Results

```
{
 Summary=Summary Descriptive Statistics for ORSALES
Analysis Variable  Min    Max        N   Number Missing Mean           Std Dev.        Std Error
----------------- ----- ---------- --- -------------- -------------- --------------- -------------
Year               1999       2002 912              0         2000.5      1.11864745    0.03704212
Quantity             10       9026 912              0  1465.08552632   1621.72304437   53.70061616
Profit            209.8  552970.51 912              0 64786.23735197  84128.37618423 2785.76891008
Total_Retail_Price 422.3 1159837.26 912              0 122090.6840625 166576.07510888 5515.88503487
4 rows

}
```

Modification: BY-Group Processing

As indicated in the preceding program, you can specify a groupBy parameter for the Castable class. The following modifications to the program demonstrate how to perform BY-group processing:

```
// add the following two imports
import com.sas.cas.actions.Castable;
import com.sas.cas.actions.Casinvardesc;

// these lines are the snippet from the previous program
SummaryOptions so = new SummaryOptions();
so.setSummarySubset(
```

```
  new SummaryOptions.SUBSET[]{MIN, MAX, MEAN, N, NMISS, STD, STDERR}
);

Castable table = new Castable();
table.setName("orsales");
table.setGroupBy(new Casinvardesc[] {                                    // 1
  new Casinvardesc().setName("year").setFormattedLength(32),
  new Casinvardesc().setName("product_line")
});
so.setTable(table);
```

1 The setGroupBy() method for the Castable instance is used to set the variable to use for grouping. The formatted values of the Year and Product_Line variables in the Orsales table are used to form unique groups. The summary action is run on each group and descriptive statistics are generated for each group.

When you perform a BY-group analysis, the results include a table that is named ByGroupInfo. This table shows the unique groups that were formed from the formatted values of the specified variables.

The results also include a result table for each group. The results for the first two groups (ByGroup1.Summary and ByGroup2.Summary) follow the ByGroupInfo information.

Summary Action Results with BY-Group Processing

```
{
 ByGroupInfo=ByGroupInfo ByGroupInfo
Year Year_f       Product_Line    Product_Line_f  _key_
---- ------------ --------------- --------------- ----------------------------------------------
1999         1999 Children        Children                1999                    Children
1999         1999 Clothes & Shoes Clothes & Shoes         1999                    Clothes & Shoes
1999         1999 Outdoors        Outdoors                1999                    Outdoors
1999         1999 Sports          Sports                  1999                    Sports
2000         2000 Children        Children                2000                    Children
2000         2000 Clothes & Shoes Clothes & Shoes         2000                    Clothes & Shoes
2000         2000 Outdoors        Outdoors                2000                    Outdoors
2000         2000 Sports          Sports                  2000                    Sports
2001         2001 Children        Children                2001                    Children
2001         2001 Clothes & Shoes Clothes & Shoes         2001                    Clothes & Shoes
2001         2001 Outdoors        Outdoors                2001                    Outdoors
2001         2001 Sports          Sports                  2001                    Sports
2002         2002 Children        Children                2002                    Children
2002         2002 Clothes & Shoes Clothes & Shoes         2002                    Clothes & Shoes
2002         2002 Outdoors        Outdoors                2002                    Outdoors
2002         2002 Sports          Sports                  2002                    Sports
16 rows

}
{
 ByGroup1.Summary=Summary Descriptive Statistics for ORSALES
Analysis Variable  Min    Max      N  Number Missing Mean           Std Dev.       Std Error
------------------ ----- -------- -- -------------- -------------- -------------- -------------
Year                1999     1999 44              0           1999              0             0
Quantity              14     2458 44              0   650.29545455    593.2126792   89.43017626
Profit             288.8 44840.28 44              0 12308.00204545 12121.86661139  1827.4401504
Total_Retail_Price 580.4 82469.83 44              0 22797.47590909  22253.3548058 3354.81946443
4 rows

}
{
 ByGroup2.Summary=Summary Descriptive Statistics for ORSALES
Analysis Variable  Min    Max       N  Number Missing Mean            Std Dev.       Std Error
------------------ ------ --------- -- -------------- --------------- -------------- --------------
Year                 1999      1999 72              0            1999              0              0
Quantity               25      6853 72              0   1499.33333333  1796.72279913   211.74581253
Profit             1925.1 340341.14 72              0  58189.28652778 73018.65472409  8605.33098475
Total_Retail_Price   3433 686906.59 72              0 111268.78444444 145044.2577739 17093.62970735
4 rows

}
```

Only the results for the first two groups are shown. A total of 16 result tables are available, as shown by the number of rows in the ByGroupInfo table.

Modification: Use the Fluent Interface

The following program is a modification of the BY-group processing program. The highlighted lines demonstrate the fluent interface style that is introduced with the SAS Viya 3.2 release.

SummaryExampleFluent.java

```
import com.sas.cas.CASActionResults;
import com.sas.cas.CASClient;
import com.sas.cas.CASClientInterface;
import com.sas.cas.CASValue;

import com.sas.cas.actions.simple.SummaryOptions;
```

```
import static com.sas.cas.actions.simple.SummaryOptions.SUBSET.*;
import com.sas.cas.actions.Castable;
import com.sas.cas.actions.Casinvardesc;

public class SummaryExampleFluent {
  public static void main(String[] args) throws Exception {
    try {
      CASClientInterface client = new CASClient("cloud.example.com", 5570);

      Castable table = new Castable();
      table.setName("orsales");
      table.setGroupBy(new Casinvardesc[] {
        new Casinvardesc().setName("year").setFormattedLength(32),
        new Casinvardesc().setName("product_line")
      });

      CASActionResults<CASValue> results = client.getActionSets()
         .simple()
         .summary().setTable(table).setSubSet(
              new SummaryOptions.SUBSET[]{MIN, MAX, MEAN, N, NMISS, STD, STDERR}
         )
         .invoke();

      if (null == results) {
        System.err.println("Failed to run the summary action.");
        return;
      }

      for (int i = 0; i < results.getResultsCount(); i++) {
        System.out.println(results.getResult(i));
      }
    } finally {
      // production code should include exception handling
    }
  }
}
```

The results of the program are identical to the BY-group processing program.

Working with Results

The following list identifies the commonly used classes for working with the results of an action:

com.sas.cas.CASActionResults
: After an action runs, the results are wrapped in a CASActionResults object. A CASActionResults object contains a list of CASValue objects, logs, and performance information.

com.sas.cas.CASValue
: A CASValue class is a key/value pair. The key is a string, and the value can be any data type. A common data type is an instance of CASTable. Tabular results are represented as CASTable objects. Other possible value types are primitives (Integer, Long, Double, String, Boolean) as well as lists of type CASValue.

com.sas.cas.CASTable
: The CASTable class provides methods to query metadata and to traverse the data in the result table.

The following code shows one way to display the contents of a CASTable:

```
import com.sas.cas.CASTable;

...

// First, output the column names
StringBuffer sb = new StringBuffer();
for (int i = 0; i < resultTable.getColumnCount(); i++) {
   if (i > 0) sb.append(" ");
   sb.append(resutTable.getColumns()[i].getName());
}
System.out.println(sb);

// Now, display the data
for (int row = 0; row < resultTable.getRowCount(); row++) {
   sb = new StringBuffer();
   for (int i = 0; i < resultTable.getColumnCount(); i++) {
      if (i > 0) sb.append(" ");
      sb.append(resultTable.getStringAt(row, i));
   }
   System.out.println(sb);
}
```

The CASTable class also has a convenience toString() method for formatting the result table as a String value. The following code shows how to return:

- a String that is formatted with aligned columns
- display column labels
- display a maximum of 1000 rows
- display 6 decimals for numeric types

```
String s = resulTable.toString(OutputType.PRETTY, true, 1000, 6);
```

The following code shows how to return:

- a String that is formatted as CSV (comma-separated values)
- display column labels
- display a maximum of 500 rows
- display 8 decimals for numeric types

```
String s = resultTable.toString(OutputType.CSV, true, 500, 8);
```

Working with Events

Events and the CASActionResults Class

By default, all server events are returned as part of the CASActionResults object that is returned by the invoke() method. This means that you have access to all events after the action is complete. Alternatively, you can register a listener for each type of event to receive the event as it is streamed from the server.

Log Events

Log events are generated by the server when information is sent to the log. A log event consists of a type (such as WARN or ERROR) and a message.

If you want to receive log events as they are streamed from the server, you need to register a CASLogEventListener on the action options object before you invoke the action:

```
// Set the log event listener. This handles the log events as they
// come from the server instead of accessing them from the results
// after the action is complete.
options.setLogEventListener(new CASLogEventListener() {

   @Override
   public boolean handleLogEvent(CASActionOptions options, CASLogEvent logEvent) {

      System.out.println("# " + logEvent);

      // Return the propagate flag. If true, the log event
      // is propagated to the result object. Otherwise nothing
      // else will be done with the log event.
      return false;
   }
});
```

Performance Events

Performance events are generated by the server when an action is designed to inform the caller about certain performance characteristics of the action. A performance event can include timings (elapsed time, CPU time, and so on), memory usage, node usage, and CPU core usage.

If you want to receive performance events as they are streamed from the server, you need to register a CASPerformanceEventListener on the action options object before you invoke the action:

```
// Set the performance event listener. This handles the performance events as they
// come from the server instead of accessing them from the results
// after the action is complete.
options.setPerformanceEventListener(new CASPerformanceEventListener() {

   @Override
   public boolean handlePerformanceEvent(CASActionOptions options, CASPerformanceEvent performanceEvent) {

      System.out.println("# " + performanceEvent);

      // Return the propagate flag. If true, the performance event
      // is propagated to the result object. Otherwise nothing
      // else will be done with the performance event.
      return false;
   }
});
```

Disposition Events

Disposition events are generated by the server when an action completes or fails. A disposition event contains a severity, reason, and message.

If you want to receive disposition events as they are streamed from the server, you need to register a CASDispositionEventListener on the action options object before you invoke the action:

```
// Set the disposition event listener. This handles the disposition events as they
// come from the server instead of acessing them from the results
// after the action is complete.
options.setDispositionEventListener(new CASDispositionEventListener() {
```

```
    @Override
    public boolean handleDispositionEvent(CASActionOptions options, CASDispositionEvent dispositionEvent) {

       System.out.println("# " + dispositionEvent);

       // Return the propagate flag. If true, the disposition event
       // is propagated to the result object. Otherwise nothing
       // else will be done with the disposition event.
       return false;
    }
});
```

Message Tag Events

A message tag event is different from the other event types. For a few actions, the action can send a message back to the client during the invocation of the action to request additional information. This is done with a message tag. See com.sas.cas.message.CASMessageHeader for the list of tags.

The addTable action in the table action set is an example of an action that uses message tag events. The addTable action sends a message back to the client so that the client can determine when to send data rows to the server and populate a table. In order to handle this message, which has a DATA tag, you must register a message tag handler (of type CASMessageTagHandler) on the action options.

The following code shows an example of CASMessageTagHandler, which simply returns no data:

```
@Override
public boolean handleMessageTag(CASMessageTagEvent event) throws CASException, IOException {

   // Just create the table; don't send any rows
   CASDataAppender.sendZeroRows(event);

   // Do not propagate the response
   return false;
}
```

To use this message tag handler, you must register the handler on the action options before you invoke the action:

```
options.setMessageTagHandler(CASMessageHeader.TAG_DATA, <the handler>);
```

To send data rows to the server (to append data to the table), use this message handler, which is more sophisticated than the first example:

```
@Override
public boolean handleMessageTag(CASMessageTagEvent event) throws CASException, IOException {

   // Get the variable list
   Addtablevariable[] vars = (Addtablevariable[]) event.getOptions().get(AddTableOptions.KEY_VARS);
   Integer reclen = (Integer) event.getOptions().get(AddTableOptions.KEY_RECLEN);
   if (reclen == null) {
      // This shouldn't happen. The server should have verified.
      throw new CASException("Missing reclen");
   }

   // Create our data appender
   CASDataAppender appender = new CASDataAppender(event, vars, reclen, bufSize);

   // Creates a new random generator with the given seed
```

```
    Random r = new Random(0);

    for (int i = 0; i < nrows; i++) {
       appender.setDouble(0,  i);
       appender.setString(1,  "Some String at " + i);
       appender.setDouble(2,  r.nextDouble());
       appender.appendRecord();
    }

    appender.close();

    // Do not propagate the response
    return false;
}
```

The preceding code is suitable for handling message tag events from the addTable action (com.sas.actions.table.AddTableOptions class). Another action that uses message tag events, the upload action (a specialized com.sas.cas.io.UploadDataTagHandler class), is available. It accepts a filename or an InputStream as the input.

Server Events

You can register for a few types of events generated by the server. These events include:

- Caslib list has changed
- Table list has changed
- Action set has been added
- Data source list has changed
- Permissions have changed

To register for one or more of these events, add an event listener:

```
client.addEventListener(CASConstants.EVENT_FLAG_CASLIBS, new CASEventListener() {
   @Override
   public void handleCASEvent(long flag) {
      // Do something with the event
   }
});
```

You can register for multiple events:

```
client.addEventListener(
  CASConstants.EVENT_FLAG_CASLIBS | CASConstants.EVENT_FLAG_TABLES,
  new CASEventListener() {
    @Override
    public void handleCASEvent(long flag) {
       // Do something with the event
       if (flag == CASConstants.EVENT_FLAG_CASLIBS) {
          // Do something - the caslibs have changed
       }
       else if (flag == CASConstants.EVENT_FLAG_TABLES) {
          // Do something - the tables have changed
       }
    }
  });
```

Server events are packaged as part of an action response, so these are not asynchronous events. Event notification occurs when an action completes. Some types of clients might desire to "poll" for events. This can be done simply by invoking any action, such as com.sas.cas.actions.builtins.PingOptions.

Socket Events

The CASSocketEventListener class provides callbacks for when a new socket connection is established and when a socket connection is closed. This enables you to customize socket settings, such as the time-out.

```
// Register our socket event listener
client.setSocketEventListener(new CASSocketEventListener() {

   @Override
   public void handleSocketConnectionEvent(CASClientInterface client, Socket socket) {
      System.out.println("Socket opened");
   }

   @Override
   public void handleSocketClosedEvent(CASClientInterface client, Socket socket) {
      System.out.println("Socket closed");
   }
});
```

Connections and Actions: Finer Points

TLS (SSL) Connections

SAS Cloud Analytic Services supports encrypted connections between the server and clients such as Java applications. When the server is configured to use encryption between the client and the server, all user data and all authentication data between the client and server is encrypted using SSL.

If the server is configured to use encryption, then it notifies the client during the authentication phase and the CASClient class attempts to connect using standard SSL.

For more information about TLS and SSL, see *Encryption in SAS Viya: Data in Motion*.

REST Connections

By default, and shown in the examples in this document, the CASClient class uses sockets to communicate with the server. You can also use a REST implementation that uses HTTP or HTTPS to communicate with the REST interface that is available on the CAS controller. The REST interface supports username and password authentication. You can specify these values, or they can be retrieved from your .authinfo file (as shown in the following section).

To use the REST interface with username and password authentication, simply instantiate the CASRestClient class instead of the CASClient:

```
// import the following classes
import com.sas.cas.rest.CASRestClient;
import java.util.URL;

// Instantiate a new REST client. Set the URL and optionally, username and password
CASClientInterface client = new CASRestClient(new URL("http://cloud.example.com:5570"));
```

As with the CASClient class, the CASRestClient class searches for credentials in your .authinfo file if you do not specify username and password arguments.

Note: JSON is used to communicate with the server. You must include a json-20141113.jar (or a later version) in your CLASSPATH. The actions that use message tag events to exchange data with the server are not supported with the REST interface. These include, but are not limited to, the addTable and upload actions.

TIP Working with the CASRestClient class does not require any special coding as compared to the CASClient class. However, if you want to see the JSON messages, you can set the following property on the command line:

```
-Dcom.sas.cas.rest.debug=true
```

Working with SAS Missing Values

SAS has a concept of a missing value that is often compared to a null value, but is somewhat different.

CAS actions can return missing value types. For these missing values, bits are set in the standard double NaN value. In order to retrieve the missing value type, use the CASMissingValue class to decode it:

```
if (Double.isNaN(value)) {
  String type = CASMissingValue.getMissingValueType(value);
  ...
}
```

The missing value type is a letter (such as "I") that represents the corresponding missing type created by the CAS action.

Action Parameters

The example code that is shown in this document uses "setter" methods for a class the corresponds to a CAS action. For example, the following code uses the setTable() method in the SummaryOptions class to identify the table to use in the analysis:

```
SummaryOptions so = new SummaryOptions();

Castable table = new Castable();
table.setName("orsales");
so.setTable(table);
```

As shown in the preceding example, it is typical to instantiate a class for the CAS action and then set named parameter values. The action object is essentially a Map with keys and values. This similarity to a Map enables another way to specify parameters and invoke actions. Strictly, the only argument that is necessary is a CASActionOptions object, and this enables an alternative way to invoke actions.

Some CAS actions take parameter lists as arguments. A common example is the table parameter. A parameter list is a sublist of parameters and can be represented either as a Map or as a CASActionOptions instance with no action set or action name. The following code is a rewrite of SummaryExample1.java on page 5 that shows how the actions and parameters are essentially Maps and CASActionOptions instances:

UseOptions.java

```
import com.sas.cas.CASActionResults;
import com.sas.cas.CASClient;
import com.sas.cas.CASClientInterface;
import com.sas.cas.CASValue;

import com.sas.cas.CASActionOptions;
```

```
import java.util.Map;
import java.util.HashMap;

public class UseOptions {
  public static void main(String[] args) throws Exception {
    try {
      CASClientInterface client = new CASClient("cloud.example.com", 5570);

      // Set up the simple.summary action options
      CASActionOptions summaryOptions =
        new CASActionOptions("simple", "summary");                          // 1
      summaryOptions.setParameter("summarySubset",
        new String[] {"MIN", "MAX", "MEAN", "N", "NMISS", "STD", "STDERR"}
      );

      // A table is a parameter list which can be represented as Map or an instance
      // of CASActionOptions (which is a Map)
      Map<String, Object> tableParams = new HashMap<String, Object>();      // 2

      // The table name
      tableParams.put("name", "orsales");

      // An array of parameter names
      tableParams.put("vars",                                               // 3
         new String[] {"year", "quarter", "quantity", "profit", "total_retail_price"}
      );

      // Set the table parameter list
      summaryOptions.setParameter("table",  tableParams);

      // Perform the analysis - generate descriptive statistics
      CASActionResults<CASValue> results = client.invoke(summaryOptions);

      for (int i = 0; i < results.getResultsCount(); i++) {
        System.out.println(results.getResult(i));
      }
    }
    finally {
      // production code should include exception handling
    }
  }
}
```

1 The instance of the CASActionOptions("simple", "summary") class specifies simple as the action set and summary as the action. The instance is equivalent to instantiating an instance of the com.sas.cas.actions.simple.SummaryOptions class.

2 The HashMap instance is equivalent to the Castable class that represents an in-memory table.

3 Specifying the variables is not equivalent to the SummaryExample1.java code. However, it shows how the Map is used to specify the vars parameter for the summary action.

An alternative way to specify the variable names in the vars parameter is to use an instance of the List class:

```
List<String> variableNames = new ArrayList<String>();
variableNames.add("year");
variableNames.add("quarter");
variableNames.add("quantity");
```

```
variableNames.add("profit");
variableNames.add("total_retail_price");
tableParams.put("vars", variableNames);
```

The results are identical to the results for the SummaryExample1.java code.

```
{
 Summary=Summary Descriptive Statistics for ORSALES
Analysis Variable  Min    Max         N   Number Missing Mean            Std Dev.         Std Error
------------------ ----- ---------- --- -------------- --------------- ---------------- -------------
Year                1999       2002 912              0         2000.5       1.11864745    0.03704212
Quantity              10       9026 912              0  1465.08552632   1621.72304437   53.70061616
Profit             209.8  552970.51 912              0 64786.23735197  84128.37618423 2785.76891008
Total_Retail_Price 422.3 1159837.26 912              0 122090.6840625 166576.07510888 5515.88503487
4 rows

}
```

More about Sessions

When a CASClient instance is created, the client does not make a connection to the server immediately. The client/server connection is established when the client invokes any server-side action for the first time. If no session ID is specified on the client-side when the CASClient instance is created, then a new session is created.

The CASClient class provides a getSessionID() method to retrieve the session ID from an existing session. If you want to attach to an existing session, then use the setSessionID() method in a CASClient instance to specify the ID of the session. If the client is not attached to any session, then getSessionID() returns null.

The CASClient class implements the java.lang.AutoCloseable interface. As a result, if the client instance is created in a try block, then the session is terminated and the socket connection is automatically closed at the end of the try block. If you want more control over terminating your session and closing a connection, you can write code to perform those tasks.

As suggested, terminating a session (calling the EndSession action) and closing a connection (calling CASClient.close() method) are two different things. The EndSession action ends the current session on the server. The CASClient.close() method closes the socket connection between the client and the server.

If you call CASClient.close(), the connection is closed, but the session still exists and continues to run on the server. This enables you to re-connect to the same session later. The session is subject to a time-out, so a disconnected session runs until it reaches the time-out. Disconnected sessions do not run indefinitely.

To terminate a session and close a connection, you need to invoke the EndSession action and then call CASClient.close(), in this order. Or you can call CASClient.close(true), which invokes the EndSession action before it closes the connection.

Sample Java Programs

About the Sample Programs

Several sample programs are available for download as source code. These samples demonstrate how to use a wider range of classes than the example programs that are included in this document. They also demonstrate more realistic programming practices such as instantiating instances of classes, modest exception handling, and so on. You can download the programs from the following URL:

http://support.sas.com/downloads/package.htm?pid=1976

The programs are also available as compiled classes in the cas-client JAR file. You can run the sample programs from the command line. The com.sas.cas.samples.table.AddCSVSample is included in the setup on page 4 in another section of this document.

You can run each program with the -help command-line option to learn the parameters to specify as key=value pairs.

```
java com.sas.cas.samples.FileInfoSample -help
```

```
Usage:
        FileInfoSample [caslib=<caslib>] [disablessl=<true|false>] host=<host> [n=<n>]
[nodes=<nodes>] [password=<password>] [path=<path>] [pause=<true|false>] [port=<port>]
[rest=<true|false>] [sessionid=<sessionid>] [username=<username>]
Where:
        caslib     CAS library name
        disablessl disablessl (default is false)
        host       Host name (or host:port)
        n          The number of times to execute the action (default is 1)
        nodes      Number of nodes to use when creating a session (default is 0)
        password   Password
        path       The path of the file or table (default is /)
        pause      'true' to pause before invoking the sample (default is false)
        port       Port number
        rest       'true' to use the REST interface (default is false)
        sessionid  Session ID to join
        username   User name
```

To run the FileInfoSample program, your command line might resemble the following example:

```
java com.sas.cas.samples.FileInfoSample host="cloud.example.com" port=5570
```

TIP To simplify running the samples, you can create a file in the current working directory that is named CASClient.properties and specify frequently used arguments such as host and port. Use the standard key=value syntax for Java properties files. Although you can specify a username key and password key in the properties file, it is more secure to store them in a .authinfo file.

com.sas.cas.samples

These samples cover a wide variety of functionality. They demonstrate the basics of running actions and working with events.

com.sas.cas.samples.ActionEventSample
: shows how to register an action event listener. Action events are fired before an action invocation is sent to the server.

com.sas.cas.samples.DispositionEventSample
: shows how to register a disposition event listener to handle disposition events as they occur, instead of accessing the disposition events from the results after the action completes.

com.sas.cas.samples.FetchSample
: shows how to perform a basic fetch of data from an in-memory table.

com.sas.cas.samples.FileInfoSample
: shows how to get file information from a caslib's data source.

com.sas.cas.samples.LoadActionSetSample
: shows how to load an action set into your session.

com.sas.cas.samples.LogEventSample
: shows how to register a log event listener to handle log events as they occur, instead of accessing log events from the results after the action completes.

com.sas.cas.samples.PerformanceEventSample
shows how to register a performance event listener to handle performance events as they occur, instead of accessing performance events from the results after the action completes.

com.sas.cas.samples.PingSample
shows how to run the builtins.ping action.

com.sas.cas.samples.QueryActionSetSample
shows how to query a server to determine whether an action set is loaded.

com.sas.cas.samples.ReflectionSample
shows how to reflect the loaded action sets and actions.

com.sas.cas.samples.ServerStatusSample
shows how to invoke the server status action.

com.sas.cas.samples.ShutdownSample
shows how to shut down a server.

com.sas.cas.samples.SocketEventSample
shows how to register a socket event listener to handle socket open and close events.

com.sas.cas.samples.TableAttributesSample
shows how to get table attributes.

com.sas.cas.samples.TableEventSample
shows how to register a table event listener to handle CASTable creation events.

com.sas.cas.samples.builtins

These samples demonstrate two system-level actions from the builtins action set.

com.sas.cas.samples.builtins.HelpSample
shows how to retrieve help from the server.

com.sas.cas.samples.builtins.ListNodesSample
shows how to get a list of nodes from the server.

com.sas.cas.sample.simple

These samples demonstrate how to run a few of the analytics actions in the simple action set.

com.sas.cas.samples.simple.CrosstabSample
shows how to execute the simple.crosstab action.

com.sas.cas.samples.simple.NumrowsSample
shows how to get the number of rows in a table.

com.sas.cas.samples.simple.SummarySample
shows how to execute the simple.summary action.

com.sas.cas.samples.table

These samples demonstrate the basics of working with data to add tables into the work and retrieve information about in-memory tables.

com.sas.cas.samples.table.AddCaslibSample
shows how to add a caslib to the server and register a server event listener.

com.sas.cas.samples.table.AddCSVSample
shows how to add a local CSV file into the server.

com.sas.cas.samples.table.AddCSVWithTransformSample
: shows how to add a local CSV file into the server using a data transformer to convert various date and time formats into the appropriate CAS data type.

com.sas.cas.samples.table.AddTableSample
: shows how to add a table to the server and append rows.

com.sas.cas.samples.table.ColumnInfoSample
: shows how to retrieve column information from a table.

com.sas.cas.samples.table.EmptyTableSample
: shows how to add an empty table to the server.

com.sas.cas.samples.table.FetchByCursorSample
: shows how to fetch data from an in-memory table and then process the CASTable with a cursor using CASTable.next().

com.sas.cas.samples.table.TableExistsSample
: shows how to check if a table exists in the server.

com.sas.cas.samples.table.TableInfoSample
: shows how to retrieve table information from the server.

System Properties

You can set system properties to affect how the classes in the cas-client JAR file operate. For example, the following command results in printing debugging information about how the .authinfo file is processed:

```
java -Dcom.sas.cas.authinfo.debug=true YourApplication
```

The following list describes several system properties:

com.sas.cas.append.buffer.size
: If set, this property defines the initial row buffer size (in bytes) used for appending data to a table. The default is 131072 (128K) bytes.

com.sas.cas.authinfo.enabled
: if set, this property defines whether credentials are read from .authinfo if no other user credentials are provided. The valid values are true or false. The default is true.

com.sas.cas.authinfo.debug
: If set, this property controls the printing of debugging information to stderr for .authinfo processing. The valid values are true or false. The default is false.

com.sas.cas.max.table.mem.size
: If set, this property defines the maximum size of a result CASTable (in bytes) before caching the table to disk. The default is 0, meaning never cache to disk.

com.sas.cas.protobuf.size.limit
: If set, this property defines the protobuf message size limit. The default is 67108864 (64 MB) bytes.

com.sas.cas.rest.debug
: If set, this property controls the printing of JSON responses to stdout from the REST client. The valid values are true or false. The default is false.

com.sas.cas.session.close
: If set, this property defines whether sessions are terminated when a CASClient object is closed. The valid values are true or false. The default is false.

These system properties correspond to the fields in com.sas.cas.CASConstants.

Using the CAS Shell

About the Shell

The cas-client JAR file includes the com.sas.cas.Cash class. This class provides a shell-like interface to SAS Cloud Analytic Services that enables you to submit actions using Lua syntax.

Using the CAS shell can help you determine the results of an action before you write the Java code to handle the results.

The requirements for running the CAS shell are identical to the requirements for developing Java programs. See Requirements on page 1.

Connecting to a Server

To use the CAS shell, you must meet all the requirements for developing and running Java applications with CAS. Specifically, the cas-client JAR file must be in your CLASSPATH. Creating a .authinfo file or CASClient.properties file is an alternative to providing credentials interactively.

Start the CAS Shell

```
java com.sas.cas.Cash
```

CASH Prompt

```
  ________  _____ __
 / ___/ _ | / __/ // /
/ /__/ __ |_\ \/ _  /
\___/_/ |_/___/_//_/

cash>
```

Be aware that even though the CAS shell is started, it is not yet connected to a CAS server or started a CAS session. At the shell prompt, you can specify the host name and port to use:

Connect and Run the SessionId Action

```
cash> host=host=cloud.example.com:5570
cash> s:session_sessionId{}
```

Note: If you do not have a .authinfo file or CASClient.properties file, then specify values for username= and password= just like host= is specified.

SessionId Action Results

```
{
   Session:Fri Mar  3 14:12:27 2017="41364ae2-3867-504c-815a-b248681b82af"
}
[Performance] Elapsed 0.000
[Disposition] (OK)
```

After you submit the sessionId action, there is a brief pause as a connection to the specified server is made and a session is started.

For the purpose of comparison, the first Java program in this document demonstrated how to use the com.sas.cas.actions.builtins.ServerStatusOptions class. The equivalent action in Lua syntax is to run the builtins_serverStatus action:

Run the ServerStatus Action

```
cash> s:builtins_serverStatus{}
```

```
{
 About= {
  CAS="Cloud Analytic Services"
  Version="3.02"
  VersionLong="V.03.02M0P02272017"
  Copyright="Copyright © 2014-2017 SAS Institute Inc. All Rights Reserved."
  System=  {
    ...
```

Next Steps

- The CAS shell includes a help command. Many common tasks are available from the shell commands rather than running an action.

```
cash> help
!!               Executes the last command
!<n>             Executes the specified command number from history
caslib           Sets the caslib
caslibs          Shows the currently defined caslibs
charset          Sets the charset for reading action scripts
...
```

- For more information about the Lua syntax for running CAS actions, see *Getting Started with SAS Viya for Lua*.
- Understand that the Lua syntax convention for a CAS session is represented by the variable named ‘s.’ The corresponding Java variable is an instance of the CASClient class.
- Understand that for an action such as builtins_serverStatus, there is a corresponding Java class that is named com.sas.cas.actions.builtins.ServerStatusOptions.