SAS® Viya® Platform Programming: Getting Started with Java for CAS

2020.1 - 2025.06

This document might apply to additional versions of the software. Open this document in SAS Help Center and click on the version in the banner to see all available versions.

Requirements

To use Java with SAS Cloud Analytic Services, the client machine that runs Java must meet the following requirements for Java 11:

Package	Version
cas-client	4.7.7 or later
antlr-runtime	3.5.2

Package	Version
protobuf-java	2.6.1
json	20230227
commons-crypto	1.0.14
jakarta.xml.bind-api	2.3.2

You can download the cas-client package (including the associated Javadoc package) and the commons-crypto package from the following URL:

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

You can download the remaining packages from the Maven repository.

Java applications need to authenticate to SAS Cloud Analytic Services. For user name and password authentication, you can supply credentials by creating an authinfo file. For more information, see *Client Authentication Using an Authinfo File*.

Note: Beginning with the 3.10.x version of the CAS client JAR, you must set the **`com.sas.cas.authinfo.enabled`** system property to true to enable reading credentials from an authinfo file. In previous releases, the default value was true.

How to Run Actions

View Server Status

The following code shows how to run the serverStatus action.

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.

Example Code 1 *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   3
            // 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.

3 The session is ended if the com.sas.cas.session.close Java system property is set to true. The default value is false. The default behavior is to close the connection, but leave the session running until it times out or you connect to it again.

Output 1 *Results of the ServerStatus Action*

```
{
 About= {
  CAS="Cloud Analytic Services"
  Version="4.00"
  VersionLong="V.04.00M0P10052023"
  Copyright="Copyright 2014-2023 SAS Institute Inc. All Rights Reserved."
  ServerTime="2023-10-06T17:43:25Z"
  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"
   Linux Distribution="Red Hat Enterprise Linux Server release 6.7 (Santiago)"
  }
  license=  {
   site="SAS Institute Inc."
   siteNum=1
   expires="30Nov2023:00:00:00"
   gracePeriod=62
   warningPeriod=31
  }
  CASHostAccountRequired="OPTIONAL"
  Transferred="NO"
  CASCacheLocation="CAS Disk Cache"
 }
}
{
 server=server
Node Count Total Actions
---------- -------------
         8             1
1 row

}
{
 nodestatus=nodes
Node Name             Role       Uptime (Sec) Running Stalled
--------------------- ---------- ------------ ------- -------
cloud1.unx.sas.com    backup ctl        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
8 rows

}
```

As shown in the output, the result of the serverStatus action includes three keys: About, server, and nodestatus. In this example, nodestatus shows a distributed server that uses eight machines: a controller, a backup controller, and 6 worker nodes. For a single-machine server, the results include the line for the controller node only.

You can filter the results by the key:

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

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

Note: For more information about the CASValue object, see "Working with Results" on page 12.

The output includes the results for the Server key only:

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

}
```

Fluent Interface Programming

The API provides a fluent programming 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 11.

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 -Dcom.sas.cas.authinfo.enabled=true com.sas.cas.samples.table.AddCSVSample \
host="cloud.example.com" port=5570 path=orsales.csv promote=true caslib=casuser
table=orsales
```

TIP You can run the following command to list the command-line options:

```
java com.sas.cas.samples.table.AddCSVSample -help
```

```
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 Platform: System Programming Guide*.)

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

```
java -Dcom.sas.cas.authinfo.enabled=true com.sas.cas.samples.table.TableInfoSample \
host="cloud.example.com" port=5570  caslib=casuser
```

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

```
...
Result Values:
{
 TableInfo=TableInfo Table Information for Caslib CASUSER(sasdemo)
Name    Rows Columns Indexed Columns Encoding Created                  Last Modified
Last Accessed             Character Set CreateTime        ModTime           AccessTime
Global Repeated View MultiPart Loaded Source Source Caslib Compressed Table Creator Last Table
Modifier Source Modified SourceModTime Table Redistribute Up Policy
------- ---- ------- ---------- -------- ------------------------- -------------------------
------------------------- ------------- ---------- ----------------- ----------------- -----------------
------ -------- ---- --------- ------------- ------------- ---------- -------------
------------------- --------------- ------------- ----------------------------
ORSALES  912       8               0 utf-8    2023-10-11T14:17:39-04:00 2023-10-11T14:17:39-04:00
2023-10-11T14:17:39-04:00 UTF8          2012667458.596361 2012667458.646073 2012667458.645097
1        0    0         0                                        0
casdemo                                                        . Not Specified
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.

Example Code 2 *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");
            table.setCaslib("casuser");
            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.

Output 2 *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:

```
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.Castable;                                    // 1
import com.sas.cas.actions.Casinvardesc;

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_ByGroup {
    public static void main(String[] args) throws Exception {
        try (CASClientInterface client = new CASClient("cloud.example.com", 5570)) {
            SummaryOptions so = new SummaryOptions();
            so.setSummarySubset(
                new SummaryOptions.SUBSET[]{MIN, MAX, MEAN, N, NMISS, STD, STDERR}
            );

            Castable table = new Castable();
            table.setName("orsales");
            table.setCaslib("casuser");
            table.setGroupBy(new Casinvardesc[] {                       // 2
                new Casinvardesc().setName("year").setFormattedLength(32),
                new Casinvardesc().setName("product_line")
            });
            so.setTable(table);

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

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

```
        }
    }
```

1 Castable and Casinvardesc are imported.

2 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.

Output 3 *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.

Example Code 3 *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.setCaslib("casuser");
            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 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 a result table.

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

```
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.Casouttable;
import com.sas.cas.actions.table.TableActions;
import com.sas.cas.actions.table.FetchOptions;
import com.sas.cas.CASTable;                                          // 1
import com.sas.cas.CASTable.OutputType;

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

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

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

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

            CASActionResults<CASValue> results = client.invoke(so);
            com.sas.cas.CASTable resultTable =                        // 2
                (CASTable)results.getResult(0).getValue();

            StringBuffer sb = new StringBuffer();                     // 3
            for (int i = 0; i < resultTable.getColumnCount(); i++) {
                if (i > 0) sb.append(" ");
                sb.append(resultTable.getColumns()[i].getName());
            }
            System.out.println("\nTable " + resultTable.getName() +
                " columns:\n" + sb);

            System.out.println("\nTable " + resultTable.getName() +  // 4
                " 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);
            }
        }
    }
}
```

1 Import CASTable and CASTable.OutputType.

2 Store the output table in a CASTable.object.

3 Display a list of the table column names.

4 Display the table data.

Here is the output.

```
Table Summary columns:
Column Min Max N NMiss Mean Std StdErr

Table Summary data:
Year 1999.0 2002.0 912.0 0.0 2000.5 1.118647450518278 0.037042118608994416
Quantity 10.0 9026.0 912.0 0.0 1465.0855263157894 1621.723044367722 53.70061616157693
Profit 209.8 552970.510000002 912.0 0.0 64786.23735197367 84128.37618423298 2785.76891008392
Total_Retail_Price 422.3 1159837.26 912.0 0.0 122090.68406250014 166576.07510887776 5515.88503486514
```

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 str = resultTable.toString(OutputType.PRETTY, true, 1000, 6);
System.out.println(str);
```

Here is an example of the output.

```
Summary Descriptive Statistics for orsales
Analysis Variable  Minimum Maximum    N   N Miss Mean          Std Dev       Std Error
------------------ ------- ---------- --- ------ ------------- ------------- -----------
Year                  1999       2002 912      0        2000.5      1.118647    0.037042
Quantity                10       9026 912      0   1465.085526   1621.723044   53.700616
Profit               209.8  552970.51 912      0  64786.237352  84128.376184  2785.76891
Total_Retail_Price   422.3 1159837.26 912      0 122090.684063 166576.075109 5515.885035
4 rows
```

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

```
System.out.println(csv);
```

Here is an example of the output.

```
"Summary","Descriptive Statistics for orsales"
"Analysis Variable","Minimum","Maximum","N","N Miss","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
```

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

Overview

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 TLS.

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 TLS.

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

REST Connections with TLS

In a full deployment, the SAS Viya platform uses an HTTP server to proxy requests to services. By default, the HTTP server is configured to use TLS. The HTTP server can use a self-signed certificate that is generated automatically or you can supply a certificate.

You can use the CASRestClient class and connect to the HTTP server. See Example Code 5 on page 21 for an example.

Unless you use the CASRestClient class and your HTTP server uses certificates that are signed by a well-known certificate authority (CA), you need access to the trustedcerts.jks file or a Java truststore that includes the CA certificates that were used to sign your certificates.

Binary Connections with TLS

In a full deployment, CAS is also configured to use TLS. CAS uses a self-signed certificate that is generated automatically by HashiCorp Vault. If you prefer the binary connection to CAS—typically on port 5570—you can use the CASClient class even if the server is configured to use TLS.

In this case, because the certificate is self-signed, your Java application needs access to the trustedcerts.jks file that is located on the CAS controller host.

Certificate Truststore and TLS Related System Properties

If your Java application requires access to the certificate truststore, the default location on the CAS controller host is as follows:

```
/opt/sas/viya/config/etc/SASSecurityCertificateFramework/cacerts/trustedcerts.jks
```

> **TIP** As an alternative to accessing the truststore from the controller, you can use the keytool command to manage trusted certificates in a keystore.

The following two system properties are commonly used to indicate the truststore to use:

-Djavax.net.ssl.trustStore=*path-to-trustedcerts.jks*
: specifies a local path to the trustedcerts.jks file that is created by HashiCorp Vault or the path to a truststore that you create with the CA certificates that signed your certificates.

-Djavax.net.ssl.trustStorePass=*changeit*
: specifies the password for the truststore.

REST Connections

By default, the CASClient class uses sockets to communicate with the server. The examples in this document use the default behavior. 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 user name 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 user name and password authentication, simply instantiate the CASRestClient class instead of the CASClient:

Example Code 4 *Sample REST Connection*

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

// Instantiate a new REST client. Set the URL and optionally, user name and password.
CASClientInterface client =
    new CASRestClient(new URL("http://cloud.example.com:8777"));
```

Example Code 5 *Sample REST Connection with TLS*

```
// import the following classes
import com.sas.cas.rest.CASRestClient;
import java.util.URL;
// For full deployments of the SAS Viya platform that use Apache HTTP Server with
TLS, connect to
// the web server instead of the CAS server.
CASClientInterface client =
    new CASRestClient(new URL("https://webserver.example.com
/cas-shared-default-http/"));
```

With a full deployment and an HTTP server that uses TLS, connect to CAS through the HTTP server. The cas-shared-default-http portion of the URL applies to a typical SAS Viya platform deployment. If the deployment instance did not use default for the deployment name, then that portion of the URL is different.

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

Note: JSON is used to communicate with the server. In your CLASSPATH, you must include json-20230227.jar (or later). 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
```

Authinfo File Details

If a user name or user name and password are not specified, the CASClient class searches for an authinfo file. The search order for authinfo files is as follows:

1. AUTHINFO environment variable, if it is set
2. NETRC environment variable, if it is set
3. %HOMEDRIVE%%HOMEPATH%_authinfo (Windows only)
4. $HOME/.authinfo
5. %HOMEDRIVE%%HOMEPATH%_netrc (Windows only)
6. $HOME/.netrc

Rules about authinfo files:

- The current user must be the owner of the file.
- No other users/groups can have Read/Write access to the file (UNIX only).

To disable authentication with an authinfo file, set the system property com.sas.cas.authinfo.enabled to false.

See Also

- *Client Authentication Using an Authinfo File*
- “System Properties ” on page 31

Prompting for a User Name and Password

A token provider is available that can prompt for the user name and password. If your code runs as a stand-alone console application, the following code registers a token provider to perform the prompting:

```
client.setSecurityTokenProvider(new CASPasswordTokenProvider());
```

When the application is run, if no other credentials are provided, the following prompts appear:

```
User name:
Password:
```

If the application is run within an IDE, such as Eclipse, a Swing pop-up dialog box is presented instead.

Kerberos Authentication

If no user name or password credentials are available—either by specifying them or from an authinfo file—CASClient attempts to retrieve a Kerberos ticket for the current user. A valid JAAS configuration file must be specified using the system property java.security.auth.login.config:

```
-Djava.security.auth.login.config=./jaas.conf
```

The default JAAS configuration name within the configuration file is com.sun.security.jgss.krb5.initiate. To override the default JAAS configuration name, set the name in the property com.sas.cas.kerberos.config.name.

Example Code 6 *Sample JAAS Configuration Entry*

```
com.sun.security.jgss.krb5.initiate {
  com.sun.security.auth.module.Krb5LoginModule
  required
  useTicketCache=true
  renewTGT=true
  doNotPrompt=true;
};
```

When creating the Kerberos context, a service name must be specified. The default service name is sascas. To override the default service name, set the name in the property com.sas.cas.kerberos.service.name. You can also specify the optional realm name by using the system property com.sas.cas.kerberos.realm.name. The service principal name (SPN) is constructed using the service name, target host name, and optional realm name:

service-name
/hostname@realm-name

To override the construction of the SPN, use the system property com.sas.cas.kerberos.spn to specify the full SPN.

Table 1 *Troubleshooting*

Error Message	Suggestion
"Unable to obtain Princpal Name for authentication" exception	If you are using Java 6, you might receive this exception. This might be due to a bug in Java 6. Try running **`kinit -R`** to renew the ticket.
Debug output similar to "unsupported key type found the default TGT: 18"	This message indicates that your Java installation does not support all of the necessary encryption types. You might need to install the Java Cryptography Extension (JCE) Unlimited Strength policy files. To view your currently installed security providers, run the com.sas.cas.samples.ShowSecurityProviders sample. The maximum AES key length should be unlimited (or 2147483647).

Error Message	Suggestion
"javax.security.auth.login.LoginException: No LoginModules configured for com.sun.security.jgss.krb5.initiate" exception	This exception can indicate that Kerberos is not properly configured for your environment.

See Also

"System Properties " on page 31

Forcing an Authentication Method

You can force a specific authentication mechanism for your Java application. The CAS_AUTH_METHOD environment variable can be set to one of the following:

- authinfo (or netrc)
- OAuth
- userpass (or password)
- Kerberos

If you supply enough information for the specified authentication mechanism, that mechanism is used regardless of the default process. If you specify an authentication method that is not possible based on the supplied information (such as specifying 'password' but not supplying a password), authentication fails.

Expired Tickets or Certificates

In certain cases an OAuth ticket can expire. If this happens, the CASClient class attempts to make a callback on the registered securityTokenProvider. A provider must be registered on the client such as this:

```
client.setSecurityTokenProvider(new CASSecurityTokenProvider() {
    public CASSecurityToken getSecurityToken(CASClientInterface client, REQUEST_TYPE
type)
        throws CASException, IOException {

        CASSecurityToken token = ...

        return token;
    }
});
```

- If an OAuth ticket has expired, the REQUEST_TYPE of the getSecurityToken call is set to EXPIRATION. In this case, you need to return a new CASSecurityToken with a new OAuth ticket in the token field.

- If the server requires that the client switch to Kerberos during authentication, the REQUEST_TYPE of the getSecurityToken call is set to EXPIRED_TICKET. In this case, you need to return a new CASSecurityToken with a GSSCredential.

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 com.sas.cas.io.SASMissingValue class to decode it:

```
if (Double.isNaN(value)) {
    String type = SASMissingValue.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 Example Code 2 on page 8 that shows how the actions and parameters are essentially Maps and CASActionOptions instances:

Example Code 7 *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");

           // The table caslib
           tableParams.put("casLib", "casuser");

           // 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));
           }
        }
        catch(Exception e) {
            // Include appropriate exception handling.
            System.out.println("An error occurred:\n" + e);
        }
        finally {
            // Include any finally code that is needed.
        }
    }
}
```

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.

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 6 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 -Dcom.sas.cas.authinfo.enabled=true 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 user name key and password key in the properties file, it is more secure to store them in an 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 prevents any attempt to authenticate with Kerberos:

```
java -Dcom.sas.cas.kerberos.enabled=false 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 false.

com.sas.cas.backup.retries
If set, this property specifies the number of attempts for connecting to the backup controller. The default value is 20.

com.sas.cas.connection.timeout
If set, this property specifies the connection time-out in milliseconds. The default value is 0 (no time-out).

com.sas.cas.kerberos.enabled
If set, this property defines whether the client should attempt to get a Kerberos ticket from the current environment if no other credentials are provided. The valid values are true or false. The default is true.

com.sas.cas.kerberos.realm.name
If set, this property specifies the Kerberos realm name.

com.sas.cas.kerberos.service.name
If set, this property defines the Kerberos service name. The default is sascas.

com.sas.cas.kerberos.spn
: If set, this property overrides the service principal name. By default, the service principal name is constructed with the service name, target host name, and optional realm name as follows:

  service-name /hostname@ realm-name

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 unspecified, meaning no maximum.

com.sas.cas.preserve.varchar.padding
: If set, this property defines whether CASTable VARCHAR columns preserve padding with blank spaces. The default value is false. False indicates to trim leading and trailing whitespace characters from VARCHAR columns.

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.

Logging

Overview of CASClient Logging

The CASClient class that is used to connect to SAS Cloud Analytic Services and run actions uses the java.util.Logger utility. You can enable logging from your Java application through the standard java.util.Logger configuration or with Java system properties.

Logging with a Configuration File

A sample java.util.Logger configuration file is as follows:

Example Code 8 *Sample logging.properties File*

```
handlers=java.util.logging.ConsoleHandler
com.sas.cas.level=FINE
java.util.logging.ConsoleHandler.level=ALL
```

Specify the path to the file with the java.util.logging.config.file system property when you start your application:

```
-Djava.util.logging.config.file=/path/to/
logging.properties
```

Logging with the Java System Property

Another way to enable logging is to specify the partial or full package name to enable as a system property when you start your application:

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

Logging can be written to a file by specifying a system property too:

```
-Dcom.sas.cas.debug.file=/tmp/my.out
```

Logging File Format

The default logging format can be overridden using the standard logging system property java.util.logging.SimpleFormatter.format:

```
-Djava.util.logging.SimpleFormatter.format="%1$tY-%1$tm-%1$td %1$tH:%1$tM:%1$tS.%1$tL
%4$-6s %2$s %5$s%6$s%n"
```

For information about the format string syntax, see the Java documentation for the java.util.Formatter class. The Java 11 documentation is available at the following URL:

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Formatter.html

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 2.

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.

Example Code 9 *Start the CAS Shell*

```
java -Dcom.sas.cas.authinfo.enabled=true com.sas.cas.Cash
```

Output 4 *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:

```
host=cloud.example.com:5570
```

If you do not have a .authinfo file or CASClient.properties file, then specify values for username= and password= just like host= is specified. After you specify the host and port, you can then run actions as shown in the following example:

```
s:session_sessionId{}
```

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.

Output 5 *SessionId Action Results*

```
{
   Session:Tue Oct 10 14:27:16 2023="7332fc32-afc2-fd4c-be99-fa58ae36b655"
}
[Performance] Elapsed 0.056, CPU 0.131, Nodes 6
[Disposition] (OK)
```

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:

Example Code 10 *Run the ServerStatus Action*

```
s:builtins_serverStatus{}
```

```
{
 About= {
  CAS="Cloud Analytic Services"
  Version="4.00"
  VersionLong="V.04.00M0P10092023"
  Copyright="Copyright © 2014-2023 SAS Institute Inc. All Rights Reserved."
  ServerTime="2023-10-10T18:28:39Z"
  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 *SAS Viya Platform Programming: Getting Started with Lua for CAS*.
- 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.

Importing CSV Files to SASHDAT Files

About the CSV to SASHDAT Utility

The cas-client JAR file includes the com.sas.cas.samples.csv.CSVToHdat class. This class acts like a utility for reading CSV files and creating SASHDAT files. The SASHDAT files can be read by the server from caslibs of type Path or DNFS.

The requirements for using the command-line interface are identical to the requirements for developing Java programs. See "Requirements " on page 2.

The utility also offers a graphical user interface. The graphical user interface is available on Windows and to Linux hosts that can access an X Window System server.

Using the Command Line

The simplest use is to specify the two required arguments, path and output:

```
java com.sas.cas.samples.csv.CSVToHdat path="orsales.csv" output="orsales.sashdat"
```

```
Guessing variable info from orsales.csv                                    1
Found 8 variable(s)
 Year (type=SAS, rType=NUMERIC, length=8)
 Quarter (type=SAS, rType=CHAR, length=6)
 Product_Line (type=SAS, rType=CHAR, length=15)
 Product_Category (type=SAS, rType=CHAR, length=24)
 Product_Group (type=SAS, rType=CHAR, length=24)
 Quantity (type=SAS, rType=NUMERIC, length=8)
 Profit (type=SAS, rType=NUMERIC, length=8)
 Total_Retail_Price (type=SAS, rType=NUMERIC, length=8)
Creating orsales.sashdat
orsales.sashdat contains 0 bytes, 0 records, 0 large blocks, 0 extents     2
orsales.sashdat written in 00.079
Large block written with 92288 bytes, 912 records, 1 small block           3
```

1 By default, the utility guesses the number of variables, variable names, and data types.

2 The log line indicates the initial number of bytes, records, and so on, for the output file. For a new file, zeros are shown. When you append to a file, the values reflect the contents of the SASHDAT file before new records are appended.

3 The line indicates that one large block was written with 912 records. With larger files that require more than one large block (the default size is 16M), this line is repeated for each large block.

The output corresponds to the orsales.csv file that is available for download from http://support.sas.com/documentation/onlinedoc/viya/examples.htm.

Examples

Example Code 11 *Create a New File by Overwriting an Existing File*

```
java com.sas.cas.samples.csv.CSVToHdat path=orsales.csv output=orsales.sashdat
replace=true
```

Or, you can specify the replace argument alone:

```
java com.sas.cas.samples.csv.CSVToHdat path=orsales.csv output=orsales.sashdat replace
```

Example Code 12 *Append Records*

```
java com.sas.cas.samples.csv.CSVToHdat path=orsales.csv output=orsales.sashdat append
```

Example Code 13 *Import Specific Variables Only*

```
java com.sas.cas.samples.csv.CSVToHdat path=orsales.csv output=orsales.sashdat \
   keepVars="year,product_line,profit" replace
```

> **TIP** Do not include a space after each comma in the list of variable names.

Example Code 14 *Create a Compressed SASHDAT File*

```
java com.sas.cas.samples.csv.CSVToHdat path=orsales.csv \
   output=orsales.sashdat compress replace
```

Considerations for Appending Records

- Records with missing values (two subsequent delimiters) are acceptable.
- Records with additional variables are not acceptable. From the command line, you can control the variables to import by specifying the names or by specifying the keepVars or dropVars arguments. The graphical user interface on page 40 enables you to specify the variables to keep after clicking the **Guess Variables** button.

An error message like the following indicates that fixed-width characters are used and new records have longer lengths.

```
com.sas.cas.CASException: variable-name exceeds the record length of nn
```

A few strategies are available for avoiding this issue:

1. For CHAR variables that vary widely in size, you can specify those names in the varcharVars argument to use the VARCHAR data type.
2. If the CHAR variables do not vary widely, you can specify a sufficiently large size in the “vars="*variable-1* type= *data-type-1*< , *variable-2* type= *data-type-2*,...>"|vars="*variable-1* as *new-name-1* type= *data-type-1*"” on page 39 argument.
3. Use the user interface and specify the lengths after clicking the **Guess Variables** button.
4. Finally, you can specify defaultStringType=varchar to use the VARCHAR data type for all character variables.

Command-Line Arguments

The command-line arguments are also available from the utility with the ? or -help argument.

Required Arguments

path=*path*
specifies the relative or fully qualified path to the input file.

output=*output*
specifies the relative or fully qualified path for the output. Include the .sashdat filename suffix.

Optional Arguments

append<=true|false >
when set to true, rows from the input file are added to the output file. Specifying the argument sets it to true.

compress<=true|false >
when set to true, the output SASHDAT file uses compression. Specifying the argument sets it to true.

compressionLevel=<1-9 >
1 indicates to compress faster and 9 indicates to compress best.

Default 6

defaultStringLength=*integer*
specifies the default length for character variables.

Default 1

defaultStringType=<varchar|fixed >
specifies the default data type for character variables.

Default fixed

delimiter=*character*
specifies the delimiter. You can specify \t for tab-delimited files.

Default ,

dropVars="*variable-1* <,*variable-2, ...*>"
specifies a comma-separated list of variables from the input file to exclude from the output file.

firstrowheader<=true|false >
when set to true, variable names are read from the first row of the input file.

Default true

fixedVars="*variable-1* <,*variable-2, ...*>"
specifies a comma-separated list of variables from the input file to assign the fixed-width CHAR data type.

guess=*integer*
specifies the number of rows to read in order to guess the variable data types and lengths. Specify -1 to read all rows.

Default 1000

handleEscapedCommas<=true|false >
when set to true, commas that are escaped with a backslash outside of quotation marks are not considered a field separator. For example, if handleEscapedCommas is set to true, then `Last \,First` is handled as a single value rather than two values.

Default false

keepVars="*variable-1* <,*variable-2, ...*>"
specifies a comma-separated list of variables from the input file to include in the output file. By default, all variables are included.

locale=*locale*
specifies the locale to use when parsing numeric values.

maxLargeBlockSize=*integer*
specifies the maximum large block size, in bytes. You can suffix with K, M, or G for larger values.

Default 16M

maxRows=*integer*
specifies the maximum number of rows to import from the input file.

Default -1 (all rows)

maxSmallBlockSize=*integer*
specifies the maximum small block size, in bytes. You can suffix with K or M for larger values.

Default 256K

removeDups<=true|false >
when set to true, variables with duplicate values (VARCHAR or VARBINARY) are removed. Specifying the argument sets it to true.

Default false

replace<=true|false >
when set to true, the output file is replaced. Specifying the argument sets it to true.

Default false

ui<=true|false>
when set to true, the graphical user interface is displayed. Specifying the argument sets it to true.

varcharVars="*variable-1* <,*variable-2, ...*>"
specifies a comma-separated list of variables to use with the VARCHAR data type.

vars="*variable-1* type= *data-type-1*< , *variable-2* type= *data-type-2*,...>" | vars="*variable-1* as *new-name-1* type= *data-type-1*"
specifies a comma-separated list of altered variable names and data types.

- For variable names that include spaces, enclose the name in single quotation marks.
- The data types are as follows:
  - NUMERIC
  - VARCHAR
  - FIXED(*length*)
  - INT32
  - INT64
  - DATE(*MM/dd/yy*)
  - TIME(*HH:mm:ss*)

- DATETIME (*MM/dd/yyyy HH:mm:ss*)

The DATE, TIME, and DATETIME patterns follow the specification for the java.text.SimpleDateFormat class. The pattern acts as a SAS informat and values are converted to SAS date, times, and datetime values.

Examples vars="'Product Line' as product_line type=FIXED(128)"

vars="'Gross profit' as profit type=NUMERIC,year=DATE('yyyy')"

For more information related to informats and SAS dates, see *SAS Formats and Informats: Reference* and "About SAS Date, Time, and Datetime Values" in *SAS Programmer's Guide: Essentials*.

Using the Graphical User Interface

Starting the User Interface

You can start the graphical user interface with the following command:

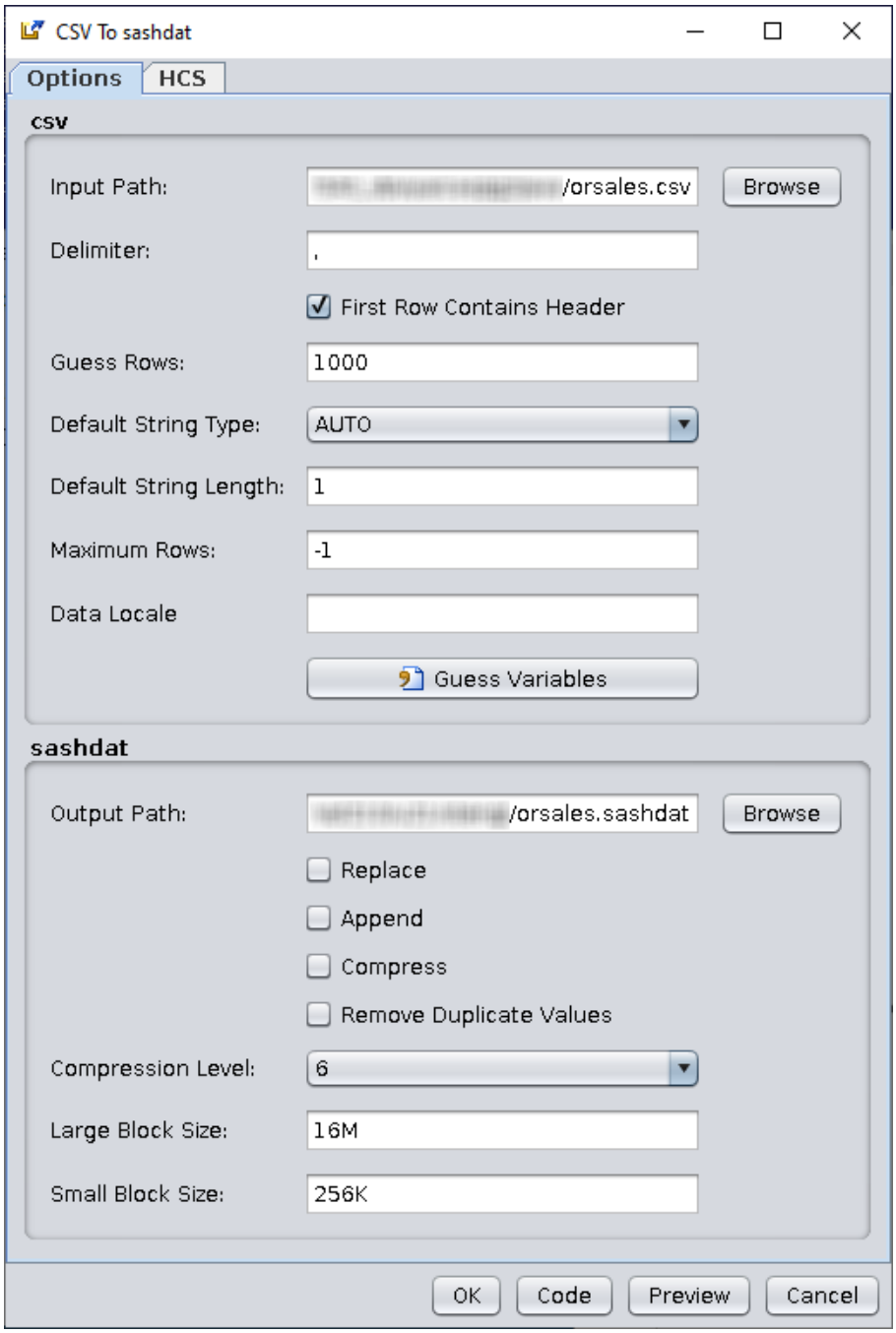
```
java com.sas.cas.samples.csv.CSVToHdat ui
```

- You can specify other command-line arguments. For example, you can specify defaultStringType=varchar before or after the ui argument.
- A .CSVToHdatDialog.properties file in your home directory is used for reading default values.
- If you start the user interface, but do not have an X server or have not configured X11 forwarding, the following error message is displayed:

```
No X11 DISPLAY variable was set, but this program performed an operation which requires it.
```

Specify an Input File and Output File

After starting the user interface, use the **Browse** buttons to specify the input file and the output file. Include the .sashdat filename suffix for the output file.

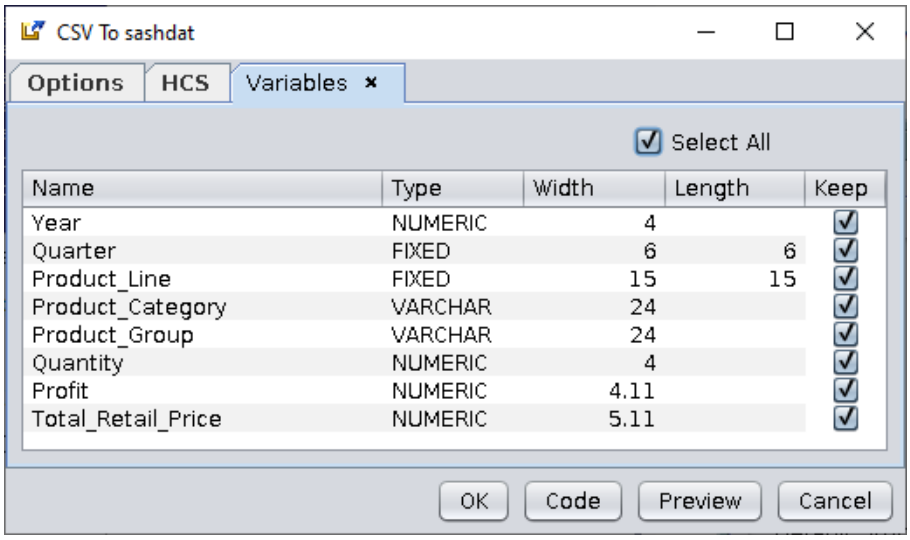


All the fields have corresponding command-line options. See “Command-Line Arguments”.

Selecting and Modifying Variables

After you start the user interface and specify an input file, you can click **Guess Variables** to select and modify variables.

Figure 1 *Variables Tab*



- Use the **Select All** and **Keep** check boxes to control which variables are imported.
- Click on the variable name in the **Name** column to specify a different name for the variable.
- Click the value in the **Type** column to change the data type. FIXED specifies a fixed-width CHAR data type. By default, the **Length** column shows the longest CHAR value that was found for the column. You can click in the cell and specify a larger value. See the vars command-line option for more information.

  Use the VARCHAR data type for character data that varies widely. There is a 16-byte overhead for VARCHAR and if the length for a variable is small, such as a state abbreviation, then a fixed-width CHAR uses less storage.
- Click the **Options** tab to specify additional settings for the import or to increase the number of **Guess Rows** and click **Guess Variables** again. Be aware that clicking **Guess Variables** again overwrites any changes that you made to the variables.

Importing the File

Verify your import settings:

Preview
Click this button to preview the import of the first 100 rows.

Code
Click this button to view the equivalent command-line options for the selections that are made in the user interface.

When you are ready to import the file, click **OK**. When the import completes, a window that is similar to the following appears.