Web Services API Guide

ArcSight Logger 5.5

January 24, 2014

Contact Information

Phone	A list of phone numbers is available on the HP ArcSight Technical Support page: http://www8.hp.com/us/en/software-solutions/software.html?compURI=1345981#.URitMaVwpWI.
Support Web Site	http://support.openview.hp.com
Protect 724 Community	https://protect724.arcsight.com

Revision History

Date	Product Version	Description
01/24/2014	5.5	5.5 release. Minor changes.
03/05/2013	5.3 SP1	5.3 SP1 release. Includes a new search call—getDataforRowIds
07/27/2012	5.3	5.3 release. Includes updated runReports information.
12/09/2011	5.2	5.2 release. Includes a new reports call—getSubGroups()
05/31/2011	5.1	First release of the Web Services API.

Contents

Chapter 1

Logger Web Services

ArcSight Logger provides Web Services for its search and reporting functions. These services enable you to log in, perform searches, or run reports on Logger from a Web Service client that you write using Java, Perl, Python, Ruby, and so on. This guide describes Web Services version 1.0.0.0.2, included with Logger 5.5.

Web Services are not available for trial Loggers. To take advantage of this feature, you need the Enterprise version.

Three Web Services are available:

- Login Service—to log in to a Logger and establish a cookie that is used for all search and report service calls.
- Search Service—to run a search query on Logger.
- Report Service—to run a report on Logger.

The examples provided in this guide are for illustration only and may not work as-is in your environment.

To learn more about writing a Web Service client, refer to the documentation of the language you intend to use to write the client.

Accessing the API

The Logger Web Services API is included with Logger 5.5.

To access the API:

1. Install Logger 5.5 on your Logger.
2. Write a Web Services client using a language of your choice, such as Java, Perl, or Python. Use the following endpoint in the client to access Logger's Web Services:

   ```
   https://<LoggerHost or IP address>/soap/services/<ServiceName>/
   <ServiceName>.wsdl
   ```

   where ServiceName is **LoginService** for logging into your Logger, **ReportService** for reports, and **SearchService** for search.

Obtaining the WSDL for Logger Web Services

Use the following WSDL to access Logger's Web Services:

- On a Logger appliance:
  `https://<LoggerHost or IPaddress>/soap/services/<ServiceName>/<ServiceName>.wsdl`
- On a software Logger:
  `https://<LoggerHost or IPaddress>:<port_number>/soap/services/<ServiceName>/<ServiceName>.wsdl`

where `<LoggerHost or IPaddress>` is the hostname or IP address of the Logger, `<port_number>` is the port that you specify in the URL when connecting to a software Logger and `<ServiceName>` is the name of Web Service you want to access.

Use **LoginService** for logging into your Logger, **ReportService** for reports, and **SearchService** for search.

Setting a Cookie

All API calls require you to input a cookie that identifies a session on Logger on which the call will run. A cookie is set when you log in to a Logger using the Login Service `login` call.

For example, you can set cookie in this way:

```
cookie = LoginService.login("admin", "password", 3600);
```

For more information about the Login Service, see Chapter 2, Login Service, on page 7.

Chapter 2

Login Service

The Login Web Service enables you to log in to a Logger and establish a cookie that is used for all search and report service calls. Additionally, this service enables you to extend or log out of an existing session, and obtain the version of Web Services currently running on your Logger.

extendSession

```
void extendSession(String cookie)
```

This call extends the session identified by the specified cookie.

`cookie` identifies a session on the Logger on which this call will run.

getVersion

```
String getVersion()
```

This call returns the version of the Web Services.

The Web Services version is different from the Logger version. For example, for Loggers running 5.5, the Web Services version is 1.0.0.0.2

login

```
String login(String username, String password, int
sessionTimeoutInSeconds)
```

This call enables you to log in to a Logger and returns a cookie. All API calls require you to input a cookie that identifies a session on Logger on which the call will run.

For example, you can set cookie in this way:

```
cookie = LoginService.login(“admin”, “password”, 3600);
```

`username` is a user configured on Logger. The user must have the appropriate privileges configured for the actions he/she is going to take using the API calls. For example, the user must be configured to “View, run, and schedule reports” for Report folder [Firewall] if he/she needs to run those reports.

`password` is the password associated with the username.

`sessionTimeoutInSeconds` is the number of seconds of inactivity after which the login session will end. You can extend an existing session by using the `extendSession` call.

Example:

```
login(“admin”, “password”, 3600);
```

logout

```
void logout(String cookie)
```

This call ends a session identified by the cookie and expires that cookie. This cookie is the one that was established using the `login` call.

`cookie` identifies a session on the Logger on which this call will run.

Chapter 3

Report Service

This section describes the API calls you can use to run a report on Logger.

Some report formats return results in binary format. Therefore, report results are base-64 encoded. You need to decode these results to display them in human-readable form.

getDeviceGroups

```
String[] getDeviceGroups(String cookie)
```

This call returns an array of the names of device groups configured on the Logger that is identified by the specified cookie.

getDevices

```
String[] getDevices(String cookie)
```

This call returns an array of the names of devices configured on the Logger that is identified by the specified cookie.

getDevicesInDeviceGroup

```
String[] getDevicesInDeviceGroup(String cookie, String
deviceGroupName)
```

This call returns an array of the names of all devices in the specified device group on the Logger that is identified by the specified cookie.

getReportGroups

```
Group[] getReportGroups(String cookie)
```

This call returns an array of report groups, where each group is associated with a name and a unique report group identifier (groupID), on the Logger that is identified by the specified cookie.

The report groups are the same as report categories in the Logger UI.

getSubGroups

```
Group[] getSubGroups(String groupId, String cookie)
```

This call returns an array of groups within the group whose identifier you specified (groupId), on the Logger that is identified by the specified cookie.

The report groups are the same as report categories in the Logger UI.

getReportsInGroup

```
Report[] getReportsInGroup(String groupID, String cookie)
```

This call returns an array of reports in the specified group (identified by the groupID) on the Logger that is identified by the specified cookie. Each report in the returned array is associated with a report name and a unique report identifier (reportID).

The report groups are the same as report categories in the Logger UI.

Use the `getReportGroups` call to obtain groupID.

getStorageGroups

```
String[] getStorageGroups(String cookie)
```

This call returns an array of the storage group names configured on the Logger that is identified by the specified cookie.

runReport

```
String runReport(String reportID, long startTime, long endTime, int
scanlimit, int resultRowLimit, String devices, String deviceGroups,
String storageGroups, String reportParameters, String reportformat,
String cookie)
```

This call runs the report specified by the reportID parameter on the Logger that is identified by the specified cookie. The report fields are arranged in the CSV format according to the order defined in the report on Logger and are base-64 encoded. You must use a decoder to convert this data into human-readable form. To decode a base-64 encoded report, you need the `ws-commons-util-1.0.1.jar` file.

`reportID` is a unique identifier for a report. To obtain reportID, use `getReportsInGroup` call.

`startTime` is the epoch time starting at which events for this report are scanned. For example, if you want to specify `startTime` as ($NOW - 2h) in Java, enter `System.currentTimeMillis() – 2 * 60 * 60 * 1000`.

If you use the specified example, make sure the time on the client machine is synchronized with the time on Logger. Otherwise, search results will contain events that span a different time range than what you wanted.

`endTime` is the epoch time up to which events for this reports are scanned. For example, if you want to specify endTime as ($Now) in Java, enter `System.currentTimeMillis()`.

If you use the specified example, make sure the time on the client machine is synchronized with the time on Logger. Otherwise, search results will contain events that span a different time range than what you wanted.

`scanlimit` is the number of events to scan. If you specify 0, all events are scanned.

`resultRowLimit` is the maximum number of rows of report data to return. If you specify 0, all rows are returned.

`devices` are the names of devices whose events are scanned for this report. If you do not want to specify device names, enter **null**. In that case, all devices are scanned. To specify multiple devices, enter a comma-separated list that is enclosed in double quotes; for example, “finance-2, internal, dev-server3”. To obtain a list of devices configured on a Logger, use the `getDevices` call.

`deviceGroups` are the names of device groups whose events are scanned for this report. If you do not want to specify a device group name, enter **null**. In that case, all device groups are scanned. To specify multiple device groups, enter a comma-separated list that is enclosed in double quotes; for example, “finance-servers, sales-servers, dev-servers.

To obtain a list of device groups configured on a Logger, use the `getDeviceGroups` call.

`storageGroups` are the names of storage groups whose events are scanned for this report. If you do not want to specify a storage group name, enter **null**. In that case, all storage groups on Logger are scanned. To specify multiple storage groups, enter a comma-separated list that is enclosed in double quotes; for example, “storage-group1, storage-group3”.

To obtain a list of storage groups configured on a Logger, use the `getStorageGroups` call.

`reportParameters` are the parameters a report requires to run. If a report does not require any parameter, enter **null**. Even if a parameter has default values assigned, those values are not automatically used when a report is run using this API call. You must specify those values in the API call to use them. If a report requires parameters and you do not specify them, the report will not run.

Use double quotes (“ ”) to separate parameters and single quotes (‘ ’) to separate parameter values.

In Java, double quotes must be escaped by using the backslash (\) character.

`reportformat` is the format in which a report is generated. Only the CSV and PDF formats are supported currently. Enter “CSV” or “csv” for CSV and “PDF” or “pdf” for PDF.

`cookie` identifies a session on the Logger on which the report will run. This is a required parameter.

Example: Running a Report

The following example, which uses a Java client, runs a report on Logger host, logger.companyxyz.com, to determine the most common events in the last 2 hours on that Logger and generates a report in the CSV format. The generated report is decoded with a base-64 decoder. In this example, a login session is established first. If the session is idle for 600 seconds (10 minutes), it is ended.

When running the following search on a software Logger, make sure you specify the port number on which Logger is running in the _loggerHost variable. For example, "user-centos:9000".

```
package com.coolcustomer.logger.webservices;

import java.rmi.RemoteException;

import org.apache.axis2.client.Options;
import org.apache.axis2.client.ServiceClient;
import org.apache.ws.commons.util.Base64;

import com.arcsight.wsclient.logger.login.adb.LoginServiceStub;
import
com.arcsight.wsclient.logger.report.adb.ArcSightReportServiceException;
import com.arcsight.wsclient.logger.report.adb.ReportServiceStub;

public class LoggerReportAPIExample {

       // LoginService needed to make API calls
       private LoginServiceStub _loginService = null;

       // ReportService needed to make API calls
       private ReportServiceStub _reportService = null;

       // IP Address or Hostname (:Port) of the Logger
       private String _loggerHost = “192.0.2.5”;

       private String _login = “admin”;
       private String _password = “password”;
       private int _timeout = 600;

       // Main Method
       // A simple test client to run a report by passing in a Name and
       // finding the ReportId and running the report
       public static void main(String[] args) throws Exception {
              LoggerReportAPIExample example = new LoggerReportAPIExample();
              String result = example.runReport(“Most Common Events”);
              System.out.println(new String(result));
       }

       /**
        * This method runs a report, illustrating how to fetch the ID of a
        * report by name
        * @param reportName the name of the report
        * @return result of the report run
        */
       public String runReport(String reportName) throws Exception {
```

```
        init(_loggerHost);

        // Make a Web Service Call to Login and retrieve an
        //authentication token
        String cookie = _loginService.login(_login, _password,
        _timeout);

        // Fetch the Id for the report from its name, by using a method
        // that recursively loops over all categories and returns the
        // report id
        String id = getReportId(reportName, cookie);

        if (id != null) {
               String result = runReport(id,
                      (System.currentTimeMillis() - 2 * 60 * 60 * 1000),
                      System.currentTimeMillis(), 0, 100, null, null,
                      null, null, “CSV”, cookie);
               byte[] reportBytes = Base64.decode(result);
               return new String(reportBytes);
        }

        return null;
}

/**
 * Run a report by passing all the parameters needed by the API
 * @return result of the report run
 * @throws Exception
 */
public String runReport(String reportId, long startTime, long endTime,
               int scanLimit, int resultRowLimit, String devicesCSV,
               String deviceGroupsCSV, String storageGroupsCSV,
               String reportParameters, String reportformat, String
               cookie)
               throws Exception {

        String result = null;

        // Make a Web Service Call to Run the Report
        result = _reportService.runReport(reportId, startTime, endTime,
               scanLimit, resultRowLimit, devicesCSV, deviceGroupsCSV,
               storageGroupsCSV, reportParameters, reportformat,
               cookie);
        return result;
}

/**
 * One way to find a ReportID, is from the Logger Web UI
 * (ReportCategories menu item) Here's a simple programmatic example of
 * how to recurse over the categories to find the report ID
 * @param reportName the name of the report to search for
 * @param cookie the authentication token
 * @return reportID the ID of the report
 * @throws ArcSightReportServiceException
 * @throws RemoteException
 */
private String getReportId(String reportName, String cookie)
               throws ArcSightReportServiceException, RemoteException {
```

```
            // get the top level report groups
            ReportServiceStub.Group[] groups = _reportService
                        .getReportGroups(cookie);

            for (int i = 0; i < groups.length; i++) {
                  ReportServiceStub.Group group = groups[i];

                  String groupID = group.getId();
                  String groupName = group.getName();

            // Recursively search for the report in all of its subgroups
                  String reportId = depthFirstSearchForReport(groupID,
                        reportName,cookie);
                  if (reportId != null) {
                        return reportId;
                  }

            }
            return null;
      }

      /**
       * Simple depth first example illustrating the use of the
       * _reportService.getSubGroups method of the API
       * @param groupID the group whose subgroups are needed
       * @param reportName the report that we're looking for recursively
       * @param cookie the authentication token
       * @return reportId, if found
       */
      private String depthFirstSearchForReport(String groupID, String
            reportName,
            String cookie) throws ArcSightReportServiceException,
            RemoteException {

            // get the reports
            ReportServiceStub.Report[] reports =
_reportService.getReportsInGroup(
                        groupID, cookie);
            if (reports != null) {
                  for (int j = 0; j < reports.length; j++) {
                        ReportServiceStub.Report report = reports[j];
                        String reportID = report.getId();
                        if (reportName.equals(report.getName())) {
                              return reportID;
                        }
                  }
            }

            // if not found here, start recursing over the subgroups
            ReportServiceStub.Group[] subgroups =
_reportService.getSubGroups(
                        groupID, cookie);
            if (subgroups != null && subgroups.length > 0) {
                  for (int i = 0; i < subgroups.length; i++) {
                        String subGroupID = subgroups[i].getId();
                        String reportId =
depthFirstSearchForReport(subGroupID,
                              reportName, cookie);
                        if (reportId != null) {
```

```
                                    return reportId;
                              }
                        }
                  }

                  return null;
            }

            private void init(String loggerHost) {
                  // Use this class, to make the JRE trust the certificates
                  XTrustProvider.install();
                  if (_reportService != null && _loginService != null) {
                        return;
                  }

                  // Setup the LoginService & ReportService stubs to make API
                  // calls to your Logger
                  try {
                        _reportService = new ReportServiceStub("https://" +
loggerHost
                                    +
"/soap/services/ReportService/ReportService.wsdl");
                        _loginService = new LoginServiceStub("https://" +
loggerHost
                                    +
"/soap/services/LoginService/LoginService.wsdl");

                        // 30 minutes
                        long timeOutInMilliSeconds = 30 * 60 * 1000;

                        // Axis related settings
                        Options axisOptions = new Options();

axisOptions.setTimeOutInMilliSeconds(timeOutInMilliSeconds);
                        ServiceClient serviceClient =
_reportService._getServiceClient();
                        serviceClient.setOverrideOptions(axisOptions);
                  } catch (Exception e) {
                        e.printStackTrace();
                  }
            }

}
```

Example: Passing Parameters when Running a Report

Before you can run a report through the API, the report must be set up. For this example, we will set up a report that allows users to select from a list of products and a list of vendors.

To create the example report:

1. In the in the **Parameter Object Editor**, create two multi-select lists with predefined values.
   - Multi-Select List 1: commonProducts, with the values **Logger** and **ESM**
   - Multi-Select List 2: commonVendors, with the values **Arcsight**, **Cisco**, and **Juniper**

2 Create the following query in the **Query Object Editor**:

```
SELECT arc_name, arc_deviceProduct, arc_deviceVendor
FROM events
Where lower(arc_deviceProduct) IN (<%commonProducts%>)
OR lower(arc_deviceVendor) IN (<%commomVendors%>)
GROUP BY arc_name, arc_deviceProduct, arc_deviceVendor
LIMIT 5
```

3 In the **Adhoc Report Designer**, create a report called Product_Vendor_Option_Report.

4 Add the query that you created in Step 2 to the report.

5 Add the Common Products and Common Vendors multi-select lists you created in Step 1 to this report.

When running the report, users are promoted to enter the parameters based on the query. Logger builds the query based on the user's specification.

For example, if the user selects **Logger** for commonProducts and **Arcsight** and **Cisco** for commonVendors, Logger will fill in the fields like this when running the query:

```
SELECT arc_name, arc_deviceProduct, arc_deviceVendor
FROM events
Where lower(arc_deviceProduct) IN ("logger")
OR lower(arc_deviceVendor) IN ("arcsight", "cisco")
GROUP BY arc_name, arc_deviceProduct, arc_deviceVendor
LIMIT 5
```

In order to run the report through the API, you must pass the parameters that a user would have selected.

To run the example report from the API:

1 Find the Report ID - either using the API or from the UI.

Open **Reports > Report Categories > Deploy Reports and Categories** and note the report ID of the report you want to use.

For example, suppose the Product_Vendor_Option_Report that we created has the Report Id 1C568A25-8458-50E1-2C7E-7605291C5EB4.

2 Run the report from the API as follows:

```
String result = reportService.runReport(
   "1C568A25-8458-50E1-2C7E-7605291C5EB4",  // Report ID
   System.currentTimeMillis() -  60 * 60 * 1000, // Start Time
   System.currentTimeMillis(),  // End Time
   10000, 100, null, // Scan Limit, Row Limit, Devices
   null, null,  // Device Groups, Storage Groups
   "\"commonVendors='arcsight', 'cisco'\",
   \"commonProducts='logger'\"",     //Comma Separated parameters
   "csv",    // Output Format
   cookie);   // Cookie
byte[] data = Base64.decode(result);
String reportResult = new String(data);
```

Be sure to use quotes to identify comma separated parameter strings. In this example, the string that needs to be sent is:
`"commonVendors='arcsight', 'cisco'", "commonProducts='logger'"`

In Java, the quotes must be escaped by using the backslash (\ ) character, like this:

```
String str = “\“commonVendors='arcsight', 'cisco'\”,
\“commonProducts='logger'\””;
```

Chapter 4

Search Service

This section describes the API calls you can use to perform a search on Logger. You can run any query that conforms to the syntax Logger expects.

Use the following guidelines when using the Search API:

- The Search API can only return search results that do not contain binary data. If the search results contain binary data, the following exception is generated:

  `"Unexpected EOF; was expecting a close tag for element <ns1:data>"`

- Searching across peers is not supported.

How the Search API Works

The Search API uses an iterator pattern to search and retrieve events. To search for events, you start a search session first using the `startSearch` API call. This call also specifies the query to run. Next, you check if any matches were found using the `hasMoreTuples` API call. If matches are found, you use the `getNextTuples` call to retrieve those events. Once all events have been retrieved or if you have retrieved the events you were searching for, you terminate the search session using the `endSearch` call.

The following example illustrates how a search is performed on Logger.

```
String cookie = loginService.login("admin", "password", 600);
searchService.startSearch("CEF", System.currentTimeMillis() - 2 * 60 * 60 *
1000, System.currentTimeMillis(), cookie);

String[] arr = searchService.getHeader(cookie);
for (String str : arr) {
            System.out.println(str);
     }
while (searchService.hasMoreTuples(cookie)) {
       Tuple [] tuples = searchService.getNextTuples(10, 600, cookie);
       if (tuples != null) {
              for (Tuple tuple : tuples) {
                     System.out.println (tuple.getData());
              }
       }
     }
searchService.endSearch(cookie);
loginService.logout(cookie);
```

Returning Specific Fields in Search Results

By default, the Search API returns all fields of matching rows. However, if you need to obtain specific fields and not all, define the fields you need using the `cef` command. Doing so creates the new columns and adds them to the tuple's data array. You can refer to the array, arr[*n*] where *n* is the index location, to obtain specific fields.

The following search query creates two new columns, `name` and `deviceVendor`.

```
ICMP* | cef name, deviceVendor
```

The header format of the search results for this query will be:

```
_rowId  _EventTime  _raw  _PeerName  name  deviceVendor
```

where

_rowId is the ID of the row
_EventTime is the epoch time
_raw contains the raw event data
_PeerName is always Local because searching across peers is not supported
name is the cef-defined field in the above query
deviceVendor is the cef-defined field in the above query

In this case, the first element, **_rowId**, is added to tuple's data array at arr[0]. Thus, the new columns, name and deviceVendor, are added at arr[4] and arr[5]. You can refer to these array locations to access these fields.

endSearch

```
void endSearch(String cookie)
```

This call terminates the currently running search session on the Logger identified by the cookie.

`cookie` identifies a session on the Logger on which this call will run.

getDataforRowIds

```
String[] getDataforRowIds(String [] rowIds, String cookie)
```

This call looks up the row IDs passed in as an argument and returns a String array of matching raw event data corresponding to the row IDs, in the order they were passed. If a row ID is not found, then the corresponding result contains "null".

`rowIds` is a String array of row IDs

> You can obtain the Row IDs through search queries. For example, in the search query explained in "Returning Specific Fields in Search Results" on page 20, the header format of the search results for the query `ICMP* | cef name, deviceVendor` was:
>
> ```
> _rowId  _EventTime  _raw  _PeerName  name  deviceVendor
> ```

The `_rowIds`, returned by that search query are the ones to use in `getDataforRowIds`, should you need access to the corresponding `_raw` event data.

Some searches, such as `ICMP* | cef name | top 5 name`, do not return the `_rowId`. Instead they return created columns like `name _count`. Results from these searches cannot be passed to this call.

`cookie` identifies a session on the Logger on which this call will run.

Example use:

```
String [] result = searchService.getDataforRowIds(new String[]
{"100177-0", "invalid"}, cookie);
```

getHeader

```
String getHeader(String cookie)
```

This call gets the header information that specifies the order in which the fields are returned in the matching events.

getNextTuples

```
Tuple[] getNextTuples(int count, long timeOut, String cookie)
```

This call retrieves an array of tuples. Depending on the search query, a tuple might contain rows of matching events or aggregated data. If no data is available at the time the call is made, the return value is “null”.

`count` is the number of tuples (rows of matching events or aggregated data) to retrieve in one iteration of this call.

`timeOut` is the time in milliseconds the call waits to receive tuples from Logger. If a tuple is not received within this time, the call terminates.

`cookie` identifies a session on the Logger on which this call will run.

- If a search operation is in progress but has not found any matching events yet, the `getNextTuples` might not return any data even though `hasMoreTuples` call returned a true value.
- The `getHeader` call specifies the order of fields returned in a matching event.

hasMoreTuples

```
boolean hasMoreTuples(String cookie)
```

This call returns `true` if the search operation (`startSearch`) is searching for or has found matching events that can be retrieved. Once search finishes on the Logger and no more events remain to be retrieved, this call returns `false`.

`cookie` identifies a session on the Logger on which this call will run.

startSearch

```
void startSearch(String queryString, long startTime, long endTime,
String cookie)
```

This call starts a new search session on Logger identified by the cookie.

`queryString` is any search query that conforms to the syntax Logger expects. For example, Error.

`startTime` is the epoch time starting at which events for this search operation are scanned. For example, if you want to specify `startTime` as ($NOW - 2h) in Java, enter `System.currentTimeMillis() – 2 * 60 * 60 * 1000`.

If you use the specified example, make sure the time on the client machine is synchronized with the time on Logger. Otherwise, search results will contain events that span a different time range than what you wanted.

`endTime` is the epoch time up to which events for this search operation are scanned. For example, if you want to specify endTime as ($Now) in Java, enter `System.currentTimeMillis()`.

If you use the specified example, make sure the time on the client machine is synchronized with the time on Logger. Otherwise, search results will contain events that span a different time range than what you wanted.

`cookie` identifies a session on the Logger on which the query will run.

Example: Searching for Events

The following example, which uses a Java client, runs a search on a Logger appliance, 192.0.2.5, to search for CEF events received on Logger in the last 5 hours and extracts the name field from the matching events. In this example, a login session is established first. (If the session is idle for 600 seconds (10 minutes), it is ended.) Then, a search session is started. If matching events are found, they are retrieved, 50 rows at a time using the getNextTuples call. If no rows are returned within 10 minutes of the last retrieval call (getNextTuples), the search session terminates. Or, once all rows have been retrieved, the search session is ended.

If the following search was run on a software Logger, make sure you specify the port number on which software Logger is running for the _loggerHost variable. For example, "192.168.36.5:9000".

```
package com.coolcustomer.logger.webservices;

import org.apache.axis2.client.Options;
import org.apache.axis2.client.ServiceClient;

import com.arcsight.wsclient.logger.login.adb.LoginServiceStub;
import com.arcsight.wsclient.logger.search.adb.SearchServiceStub;
import com.arcsight.wsclient.logger.search.adb.SearchServiceStub.Tuple;

public class LoggerSearchAPIExample {
```

```
private SearchServiceStub _searchService = null;
private LoginServiceStub _loginService = null;
private String _loggerHost = “192.0.2.5”;
private String user = “admin”;
private String password = “password”;
private int timeout = 600;

public String runSearch (String query) {
   init(_loggerHost);

   String cookie = null;

   try {
       String version = _loginService.getVersion();
       System.out.println(version);
       cookie = _loginService.login(user, password, timeout);
       _searchService.startSearch(query,
                    System.currentTimeMillis() - (5 * 60 * 60 * 1000),
                    System.currentTimeMillis(), cookie);

       // See what's the format of the Tuples
       String [] header = _searchService.getHeader(cookie);
       for (String str : header) {
          System.out.println(str);
       }

       int rowNum = 0;
       while (_searchService.hasMoreTuples(cookie)) {
          Tuple [] tuples =
              _searchService.getNextTuples(50, 600, cookie);
          if (tuples != null) {
                 for (Tuple tuple : tuples) {
                 String [] arr = tuple.getData();
                 System.out.println(“ *** Row: ” + ++rowNum + “ *** ”);
                 for (int i=0; i<header.length; i++) {
                        System.out.println(arr[i]);
                 }
                 System.out.println(“\n\n”);
                 }
          }
       }
   } catch (Exception e) {
       e.printStackTrace();
   } finally {
       // clean up
       if (cookie != null) {
          try {
              _searchService.endSearch(cookie);
              _loginService.logout(cookie);
          } catch (Exception e) {
              e.printStackTrace();
          }
       }
   }
   return null;
}

private void init(String loggerHost) {
```

```
        XTrustProvider.install();

        try {
            _loginService =
               new LoginServiceStub("https://" + loggerHost +
                     "/soap/services/LoginService/LoginService.wsdl");

            _searchService =
               new SearchServiceStub("https://" + loggerHost +
                     "/soap/services/SearchService/SearchService.wsdl");

            // read time out from some property file

            // 30 minutes
            long timeOutInMilliSeconds = 30 * 60 * 1000;
            Options axisOptions = new Options();
            axisOptions.setTimeOutInMilliSeconds(timeOutInMilliSeconds);
            ServiceClient serviceClient = _loginService._getServiceClient();
            serviceClient.setOverrideOptions(axisOptions);
            ServiceClient _searchServiceClient =
               _searchService._getServiceClient();
            _searchServiceClient.setOverrideOptions(axisOptions);

        } catch (Exception e) {
            e.printStackTrace();
        }

    }

    public static void main(String[] args) throws Exception {
        LoggerSearchAPIExample example = new LoggerSearchAPIExample();
        example.runSearch("CEF | cef name");
    }
}
```