AceQL HTTP 2.0 - Java Client SDK

Fundamentals

This document describes how to use the AceQL Java Client SDK and gives some details about how it operates with the server side.

The AceQL Java Client SDK allows users to wrap the AceQL HTTP APIs and eliminate the tedious work of handling communication errors and parsing JSON results.

Android and Java Desktop application developers can access remote SQL databases and/or SQL databases in the cloud, simply by including standard JDBC calls in their code, just like they would for a local database.

The AceQL Server operation is described in AceQL HTTP Server Installation and Configuration Guide, whose content is sometimes referred in this User Guide.

Technical operating environment

The AceQL Java Client SDK is entirely written in Java, and functions identically with Microsoft Windows, Linux, and all versions of UNIX supporting Java 7+.

The only required third party installation is a recent JVM. The following JVMs are supported in this version:

OS	JVM (Java Virtual Machine)
Android	Android 4.1+
Windows	Oracle Java SE 7, Java SE 8, Java SE 9
UNIX/Linux	Oracle Java SE 7, Java SE 8, Java SE 9 OpenJDK 7, OpenJDK 8, OpenJDK 9
OS X / mac OS	Oracle Java SE 7 for OS X 10.7.3+ Oracle Java SE 8 for OS X 10.8+ Oracle Java SE 9 for mac OS 10.10+

AceQL Server side compatibility

This 2.0 SDK version is compatible with AceQL HTTP server side v2.0. It is not compatible with AceQL HTTP server side v1.0.

License

The SDK is licensed with the liberal Apache 2.0 license.

SDK instead of JDBC Driver

Note that the SDK is *not *a real JDBC Driver, because it lacks important metadata call capabilities:

- `Connection.getMetaData()`.
- `ResultSet.getMetaData()`.

Because Metadata calls are not supported, we decided not to package the SDK as a real JDBC Driver. It could not be used with third party database query tools, thus it would be misleading to present it as a real JDBC Driver.

Note that we will soon release a real JDBC Driver. Please contact us at contact@kawansoft.com if you would like more information.

AceQL Java Client SDK installation

Go to the download page and follow the instructions to download and install the SDK.

Android Project settings

Add the following3 lines to your AndroidManifest.xml:

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

A full Android project sample is available on GitHub: aceql-http-client-android-sample.

Data transport

Transport format

AceQL transfers the least possible amount of meta-information:

- Request parameters are transported in UTF-8 format.
- JSON format is used for data and class transport (using `javax.json` package).

Content streaming and memory management

All requests are streamed:

- Output requests (from the client side) are streamed directly from the socket to the server to avoid buffering any content body.
- Input responses (for the client side) are streamed directly from the socket to the server to efficiently read the response body.

Large content (ResultSet, Blobs/Clobs...) is transferred using files. It is never loaded in memory. Streaming techniques are always used to read and write this content.

Best practices for fast response time

Every HTTP exchange between the client and server side is time-consuming, because the HTTP call is synchronous and waits for the server's response

Try to avoid coding JDBC calls inside loops, as this can reduce execution speed. Each JDBC call will send an HTTP request and wait for the response from the server.

Note that AceQL is optimized as much as possible:

- A SELECT call returning a huge Result Set will not consume memory on the server or client side: AceQL uses input stream and output stream I/O for ResultSet transfer.
- Result Set retrieval is as fast as possible:
    - The `ResultSet` creation is done once on the server by the `executeQuery()`.
    - The rows are all dumped at once on the servlet output stream by the server.
    - The client side gets the ResultSet content as a file.

- All `ResultSet` navigation commands are executed locally on the client side by navigating through the file: `next()`, `prev()`, `first()`, `last()`, etc.

Using the AceQL Java Client SDK

We will use the same `kawansoft_example` database for all our code samples.

The schema is available here: kawansoft_example.txt.

Connection creation

The `Connection` to the remote database is created using AceQL's AceQLConnection class and passing the URL of the `ServerSqlManager` Servlet of your server configuration:

```
// The URL of the AceQL Server servlet
// Port number is the port number used to start the Web Server:
String url = "https://www.acme.com:9443/aceql";

// The remote database to use:
String database = "kawansoft_example";

// (username, password) for authentication on server side.
// No authentication will be done for our Quick Start:
String username = "MyUsername";
char [] password = { 'M', 'y', 'S', 'e', 'c', 'r', 'e', 't'};

// Attempt to establish a connection to the remote database:
Connection connection = new AceQLConnection(url, database, username,
    password);
```

From now on, you can use the connection to execute updates and queries on the remote database, using standard and unmodified JDBC calls.

Using a Proxy

Communication via a proxy server is done using a `java.net.Proxy` instance.

If proxy requires authentication, pass the credentials using a `java.net.PasswordAuthentication` instance:

```
// Proxy info
String proxyHost = "localhost";
int proxyPort = 8080;
String proxyUsername = "myProxyUsername";
char[] proxyPassword = { 'p', 'a', 's', 's', 'w', 'o', 'r', 'd' };

Proxy proxy = new Proxy(Proxy.Type.HTTP, new InetSocketAddress(
proxyHost, proxyPort));

PasswordAuthentication authentication = new PasswordAuthentication(
proxyUsername, proxyPassword);
```

```
// Attempt to establish a connection to the remote database using a
// Proxy
Connection connection = new AceQLConnection(url, database, username,
password, proxy, authentication);
```

Handling Exceptions

Except for `NullPointerException`, exceptions thrown are always an instance of AceQLException.

The `AceQLException` contains 5 pieces of information :

Info	Description
Reason	The error message. Retrieved with `getMessage()`.
Error Type	See below for description. Retrieved with `getErrorCode()`.
Cause	The `Throwable` cause, if any. Retrieved with `getCause()`.
Http Status Code	See below for description. Retrieved with `getHttpStatusCode(`).
Server Exception	The Exception Stack Trace thrown on the server side, if any. Retrieved with `getRemoteStackTrace()`.

The error type

The error type allows users to get the type of error and where the error occurred. It is retrieved with `AceQLException.getErrorCode()`:

Error Type Value	Description
0	The error occurred locally on the client side. See `getHttpStatusCode()` for more info. Typical cases: no Internet connection, proxy authentication required.
1	The error is due to a JDBC Exception. It was raised by the remote JDBC Driver and is rerouted by AceQL as is. The JDBC error message is accessible via `getMessage()` Typical case: an error in the SQL statement. Examples: wrong table or column name.
2	The error was raised by the AceQL Server. It means that the AceQL Server expected a value or parameter that was not sent by the client side. Typical cases: misspelling in URL parameter, missing required request parameters, JDBC Connection expiration, etc. The detailed error message is accessible via `getMessage()`. See below for the most common AceQL Server error messages.
3	The AceQL Server forbids the execution of the SQL statement for a security reason. For security reasons, `getMessage()` gives access to voluntarily vague details.
4	The AceQL Server is on failure and raised an unexpected Java Exception. The stack track is included and accessible via `getRemoteStackTrace()`.

Most common AceQL Server messages

AceQL Sever Error Messages (AceQLException.getErrorCode() = 2)
AceQL main servlet not found in path
An error occurred during Blob download
An error occurred during Blob upload
Blob directory defined in `DatabaseConfigurator.getBlobDirectory()` does not exist
Connection is invalidated (probably expired).
Database does not exist
Invalid blob_id. Cannot be used to create a file
Invalid blob_id. No Blob corresponding to blob_id
Invalid session_id.
Invalid username or password.
No action found in request.
Operation not allowed in stateless mode.
Unable to get a `Connection`.
Unknown SQL action or not supported by software

HTTP Status Codes

The HTTP Status Code is accessible with `AceQLException.getHttpStatusCode()`.

The HTTP Status Code is 200 (OK) on successful completion calls.

When an error occurs:

- If the error type is 0, the HTTP Status Code is returned by the client side and may take all possible values in a malformed HTTP call.
- If the error type is > 0, the HTTP Status Code can take one the following values returned by server side:

HTTP Status Code	Description
400 (BAD REQUEST)	Missing element in URL path. Missing request parameters. All JDBC errors raised by the remote JDBC Driver.
401 (UNAUTHORIZED)	Invalid username or password in connect. Invalid session_id. The AceQL Server forbade the execution of the SQL statement for security reasons.
404 (NOT_FOUND)	BLOB directory does not exist on server. BLOB file not found on server.
500 (INTERNAL_SERVER_ERROR)	The AceQL Server is on failure and raised an unexpected Java Exception.

Data types

The main JDBC data types for columns are supported:

Boolean, Blob/Clob, Integer, Short, Double, Float, BigDecimal, Long,String, Date, Time, Timestamp, URL and Array.

State Management

The AceQL SDK supports two state management modes:

- The Stateful Mode
- The Stateless Mode

The Stateful Mode is the default when creating a session.

State Management is described in detail in:

AceQL HTTP Server Installation and Configuration Guide.

You can set the session State *before* creating the `AceQLConnection` instance with a call to: AceQLConnection.setStateless(booleanstateless).

Note that transactions and Connections modifiers calls are not allowed in Stateless Mode and will raise an `AceQLException` exception.

SQL Transactions & Connections modifiers

The AceQLSDK support SQL transactions in Stateful Mode with:

- `commit()`
- `rollback()`
- `setAutoCommit(boolean autoCommit)`

The following Connections modifiers calls are supported in this version, only in Stateful Mode:

- `setHoldability(int holdability)`
- `setTransactionIsolation(int level)`
- `setReadOnly(boolean readOnly)`

BLOB management

The AceQL SDK supports BLOB creation and reading. Methods are implemented using streaming techniques to keep memory consumption low, both on the client and server sides.

CLOB are not supported in this version.

BLOB creation

BLOB creation is supported through `PreparedStatement.setBinaryStream()`:

```
    /**
     * An INSERT example with a Blob.
     */
    public void insertOrderWithImage(int customerId, int itemNumber,
        String itemDescription, BigDecimal itemCost, File imageFile)
        throws SQLException, IOException {

    // Some databases require to be in a transaction for BLOB actions
    connection.setAutoCommit(false);

    try {

        String sql = "insert into orderlog "
            + "values ( ?, ?, ?, ?, ?, ?, ?, ?, ? )";

        // We will insert a Blob (the image of the product).
        // The transfer will be done in streaming both on the client
        // and on the Server: we can upload/download very big files.
        InputStream in = new BufferedInputStream(new FileInputStream(
            imageFile));

        // Create a new Prepared Statement
        PreparedStatement prepStatement = connection.prepareStatement(sql);

        int i = 1;
        long theTime = new java.util.Date().getTime();
        java.sql.Date theDate = new java.sql.Date(theTime);
        Timestamp theTimestamp = new Timestamp(theTime);

        prepStatement.setInt(i++, customerId);
        prepStatement.setInt(i++, itemNumber);
        prepStatement.setString(i++, itemDescription);
        prepStatement.setBigDecimal(i++, itemCost);
        prepStatement.setDate(i++, theDate);
        prepStatement.setTimestamp(i++, theTimestamp);
        prepStatement.setBinaryStream(i++, in, (int) imageFile.length());
```

```
        prepStatement.setInt(i++, 0);
        prepStatement.setInt(i++, 1);

        prepStatement.executeUpdate();
        prepStatement.close();
    } catch (Exception e) {
        connection.rollback();
        throw e;
    } finally {
        connection.setAutoCommit(true);
    }
}
```

BLOB reading

BLOB reading is supported through `PreparedStatement.setBinaryStream()`:

```
/**
 * A SELECT example with a BLOB.
 */
public void selectOrdersForCustomerWithImage(int customerId, int itemId,
    File imageFile) throws SQLException, IOException {

// Some databases require to be in a transaction for BLOB actions
connection.setAutoCommit(false);

try {

    String sql = "select customer_if, order_id, jpeg_image "
        + "from orderlog where customer_id = ? and item_id = ?";

    PreparedStatement prepStatement = connection.prepareStatement(sql);
    int i = 1;
    prepStatement.setInt(i++, customerId);
    prepStatement.setInt(i++, itemId);

    ResultSet rs = prepStatement.executeQuery();

    if (rs.next()) {
    int customer_id = rs.getInt("customer_id");
    int item_id = rs.getInt("item_id");

    // Get BLOB from remote server and store it on disk:
    try (InputStream in = rs.getBinaryStream("jpeg_image")) {
        Files.copy(in, imageFile.toPath());
    }

    System.out.println();
    System.out.println("customer_id : " + customer_id);
    System.out.println("item_id     : " + item_id);
    System.out.println("jpeg_image  : " + imageFile);

    }
```

```
            prepStatement.close();
            rs.close();
        } catch (Exception e) {
            connection.rollback();
            throw e;
        } finally {
            connection.setAutoCommit(true);
        }
    }
```

Using Progress Bars with Blobs

Using Progress Bar when inserting Blobs in a background engine requires two atomic variables:

- An `AtomicInteger` that represents the Blob transfer progress between 0 and 100.
- An `AtomicBoolean` that says if the end user has cancelled the Blob transfer.

The atomic variables values will be shared by AceQL download/upload processes and by the Progress Monitor.

The values are to be initialized and passed to `AceQLConnection` before the JDBC actions with the static setters:

- AceQLConnection.setProgress(AtomicInteger progress)
- AceQLConnection.setCancelled(AtomicBoolean cancelled)

Values will then be updated and read:

- Progress value will be updated by the `AceQLConnection`.
- Canceled value will be updated to true if user cancels the task, and `AceQLConnection` will thus interrupt the Blob/Clob transfer.

Remember to always set the progress value to 100 at end of a successful or failed operation in order to close the Progress Monitor.

Example:

The first step is to declare the 2 atomic variables:

```
    /** Progress value between 0 and 100. Will be used by progress monitor. */
    private AtomicInteger progress = new AtomicInteger();

    /** Says if user has cancelled the Blob/Clob upload or download */
    private AtomicBoolean cancelled = new AtomicBoolean();
```

The atomic variables will be passed to the `AceQLConnection` with their setter:

```
    /**
     *SQL insert with BLOB column
     */
    private void doInsert() {
    try {

        // BEGIN MODIFY WITH YOUR VALUES
```

```
        String userHome = System.getProperty("user.home");

        // Port number is the port number used to start the Web Server:
        String url = "https://www.acme.com:9443/aceql";
        String database = "kawansoft_example";
        String username = "username";
        char [] password = { 'p', 'a', 's', 's', 'w', 'o', 'r', 'd'};
        File imageFile =
            new File(userHome + File.separator + "image_1.jpg");
        // END MODIFY WITH YOUR VALUES

        // Attempts to establish a connection to the remote database:
        Connection connection = new AceQLConnection(url, database,
            username, password);

        // Pass the mutable & shareable progress and canceled to the
        // underlying AceQLConnection.
        // - progress value will be updated by the AceQLConnection and
        // retrieved by SwingWorker to increment the progress.
        // - cancelled value will be updated to true if user cancels the
        // task and AceQLConnection will interrupt the Blob upload.

        ((AceQLConnection) connection).setProgress(progress);
        ((AceQLConnection) connection).setCancelled(cancelled);

        // Now run our insert
        BlobExample blobExample = new BlobExample(connection);

        // Delete if duplicate
        blobExample.deleteOrderlog(1, 1);

        blobExample.insertOrderWithImage(1, 1, "description",
            new BigDecimal("99.99"), imageFile);

        System.out.println("Blob upload done.");

    } catch (Exception e) {

        if (e instanceof SQLException && e.getCause() != null
            && e.getCause() instanceof InterruptedException) {
        System.out.println(e.getMessage());
        return;
        }
        e.printStackTrace();
    } finally {
        // Always set progress to maximum/end value to close the progress
        // monitor
        progress.set(100);
    }
}
```

Assuming hat you want to display a progress indicator using `SwingWorker`, you would start the preceding code as a Thread. To update the progress bar, the `SwingWorker.doInBackground()` method would be overridden as follows:

```
    @Override
    public Void doInBackground() {
        cancelled.set(false);
        progress.set(0);

        setProgress(0);

        while (progress.get() < 100) {
        try {
            Thread.sleep(50);
        } catch (InterruptedException ignore) {
        }

        if (isCancelled()) {
            // If end user cancels the task, say it to mutable
                // & shareable cancelled.
                //cancelled will be read by AceQLConnection to
            // interrupt blob upload
            cancelled.set(true);
            break;
        }

        // Get the progress value between 0 and 100 that
        // is updated by doInsert in background thread
        setProgress(Math.min(progress.get(), 100));
        }

        return null;
    }
```

A complete example is available in SqlProgressMonitorDemo.java and BlobExample.java

HTTP session options

You can set the http timeout values with the static setters to be called before `AceQLConnection` creation:

- AceQLConnection.setConnectTimeout(int connectTimeout)
- AceQLConnection.setReadTimeout(int readTimeout)

Limitations

The following JDBC features are not supported nor implemented in this version:

- Metadata calls are not supported:
  - `Connection.getMetaData()`
  - `ResultSet.getMetaData()`.

- Savepoints are not supported.
- Stored Procedures are not supported.
- Batch methods are not supported.
- BLOB syntax is limited in `PreparedStatement` and in `ResultSet`.
- There are no `java.sql.Blob` and `java.sql.Clob` interface implementation.
- CLOB are not supported.
- `ROWID` are not supported.
- Auto-generated keys are not supported.
- Advanced data types: `Struct`, `NClob`, `SQLXML` and `Typemaps`.
- Some Statement methods: `getWarnings`, `isPoolable`/`setPoolable`, `getMoreResults`, `setCursorName`.
- Updatable Result Set.
- `RowSet` Objects.
