AceQL HTTP 1.0 - Swift Client SDK

Fundamentals

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

The Swift SDK allows you to wrap the AceQL HTTP APIs to access remote SQL databases and/or SQL databases in the cloud by simply including standard SQL calls in your code, just like you would do for any local database. There is zero learning curve and usage is straightforward.

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

On the remote side, like the AceQL Server access to the SQL database using Java JDBC, we will sometimes use the JDBC terminology (ResultSet, etc.) in this document. Nevertheless, knowledge of Java or JDBC is *not* a requirement.

Requirements

- iOS 8.0+ / macOS 10.10+
- Xcode 8.3+
- Swift 3.1+

AceQL Server side compatibility

This 1.0 SDK version is compatible with both AceQL HTTP server side v1.0 and AceQL HTTP server side v2.0.

Installation

aceql-swift is available through CocoaPods. To install it, simply add the following line to your Podfile:

```
pod 'aceql-swift'
```

Samples

To run the example project, clone the repo, and run `pod install` from the Example directory first.

License

The SDK is licensed with the liberal Apache 2.0 license.

Author

KawanSoft and Bruno Paul, brunopaul88@outlook.com.

Data transport

HTTP requests parameters are transported in UTF-8 format and JSON format is used for data and class transport

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.

Note that in this version SELECT result sets are stored in memory and not streamed.

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 SQL calls inside loops, as this can reduce execution speed. Each SQL call will send an http request and wait for the response from the server.

Server JDBC ResultSet 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 data reading commands are executed locally on the client side with forward-only reading.

Datatypes

The main server side JDBC data types for columns are supported:

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

Note that the Swift SDK does not allow you to specify data types to use; data types are implicitly chosen by the module.

Parameter values are automatically converted to their SQL equivalent. ate Management

AceQL 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 with the static method:

`AceQLConnection.setStateless(Bool)`

Note that transactions and Connection modifiers calls are not allowed in Stateless mode and will raise an Error exception.

Usage

Quickstart

To use the SDK, just create an `AceQLConnection` object that represents the database, then open it with `AceQLConnection.openAsync()` .

```
let serverUrl = "https://www.acme.com:9443/aceql";
let database = "kawansoft_example";
let username = "username";
let password = "password";

connection = AceQLConnection(server: serverUrl, database: database, username: username,
password: password)

// Establish the connection with the remote server
connection.openAsync() { status in }
```

The schema of the database is here: [kawansoft_example]

Following sample shows how to insert a new customers using a prepared statements

```
let sql = "insert into customer values "
    + "(@param1, @param2, @param3, @param4, @param5, @param6, @param7, @param8)"

let stmt = connection.prepare(sql: sql)
stmt?.run(args: 1, "Sir", "John", "Smith I", "1 Madison Ave",
    "New York", "NY 10010", "+1 212-586-7001") { result, status in }
```

We view the first inserted customer:

```
let sql = "select * from customer"
self.connection.run(sql: sql) { stmt in
    for row in stmt
    {
       for column in row
       {
           print("column: \(column)")
       }
    }
}
```

which returns:

```
column: 1
column: Sir
column: Town_2
column: John
column: Smith I
column: 1 Madison Ave
column: New York
column: NY 10010
column: +1 212-586-7001
```

At end of our session, it is highly recommended to close the `Connection`:

```
// Make sure connection is always closed in order to close and release
// server connection into the pool:
connection.closeAsync() {status in }
```

Handling Exceptions

Exceptions thrown are always an instance of `AceQLException`.

The `AceQLException` contains 5 pieces of information:

Info	Description
Reason	The error message. Retrieved with `Reason` property.
Error Type	See below for description. Retrieved with `ErrorType` property.
Exception	The original Exception that is the cause, if any. Retrieved with `ExceptionCause` property.
Http Status Code	See below for description. Retrieved with `HttpStatusCode` property.
Server Exception	The Java Exception Stack Trace thrown on server side, if any. Retrieved with `RemoteStackTrace` property.

The error type

The error type allows you to get the type of error, and where the error occurred. It is retrieved with the `AceQLException.ErrorType` property.

Error Type Value	Description
0	The error occurred locally on the client side. See `HttpStatusCode` property 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 `Reason` property. Typical case: an error in the SQL statement. Examples: wrong table or column name.
2	The error was raised by the AceQL Server. This 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 `Reason` property. See below for most common AceQL Server error messages.
3	The AceQL Server forbade the execution of the SQL statement for a security reason. For security reasons, `Reason` property gives access to voluntarily vague details.
4	The AceQL Server is on failure and raised an unexpected Java Exception. The stack trace is included and accessible via `RemoteStackTrace` property.

Most common AceQL server messages

AceQL Sever Error Messages (AceQLException.ErrorType = 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 StatusCode is accessible with the `AceQLException.HttpStatusCode` property.

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

When an error occurs:

If `errortype` is 0, the HTTP Status Code is returned by the client side and may take all possible values in a malformed HTTP call.

If `errortype` is > 0, the HTTP Status Code can take one the following values returned by the 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.

Advanced Usage

Setting NULL values

`NULL` values are handled in a specific way, because the remote server must know the type of the `NULL` value.

Use the `AceQLNullType` enum to pass a NULL value. Choose the `SqlNullType` constant that defines the type of the parameter on the SQL side.

Example for a String/VARCHAR value:

```
let sql = "insert into customer values "
    + (@param1, @param2, @param3, @param4, @param5, @param6, @param7, @param8)"

let stmt = connection.prepare(sql: sql)

// We don't know the phone number ==> set it to null
stmt?.run(args: 1, "Sir", "John", "Smith I", "1 Madison Ave",
    "New York", "NY 10010", AceQLNullType.VARCHAR)) { result, status in }
```

Transactions

Transactions are supported by the SDK. Because the remote server executes JDBC code, client code must follow the JDBC requirement to set the auto commit mode to false prior executing a transaction.

Transactions are done using same syntax than JDBC. The transactions methods are:

```
AceQLConnection.setAutoCommitAsync(Bool)
AceQLConnection.commitAsync()
AceQLConnection.rollbackAsync()
```

BLOB management

The Swift SDK module supports BLOB creation and reading. CLOBs are not supported in this version.

BLOB creation example

```
let sql = "insert into orderlog values (@param1, @param2, @param3, @param4, "
    + "@param5, @param6, @param7, @param8, @param9)";

let stmt = self.connection.prepare(sql: sql)
let image = UIImage(named: "tx01_366")
let customer_id = 0

stmt?.run(args: customer_id, customer_id,
    "Description_" + String(customer_id),
     AceQLNullType.DECIMAL, Date(), Date(),
     UIImageJPEGRepresentation(image!, 1.0), 1, 2000 )
    { result, status in
        if (status)
        {
            self.UI() {
                self.showAlert(message: "Success - Blob Insert")
            }
        }
    }
```

BLOB reading example

The BLOB is referred with a `"blobId"` stored as a string in the BLOB column. Then use the value with `AceQLConnection.downloadBlob` method to get the BLOB content:

```
let sql = "select * from orderlog where customer_id = 1"
var blobId: String?

self.connection.run(sql: sql) { stmt in
    for row in stmt
    {
        for column in row
        {
            print("column : \(column)")
        }

        blobId = row[3] as? String
    }


    if (blobId != nil)
```

```
    {
        self.connection.downloadBlob(blobId: blobId) { data in
            if (data != nil) {
                self.UI() {
                    self.blobImageView.image = UIImage(data: data!)
                    self.showAlert(message: "Success - Blob Select")
                }
            }
        }
    }
}
```