AceQL HTTP 2.1

Server Installation and Configuration Guide

Fundamentals

Overview

AceQL HTTP is a library of REST like APIs that allows you access to remote SQL databases over HTTP from any device that supports HTTP. This software has been designed to handle heavy traffic in production environments.

For example, a select command would be called from the client side using this http call with cURL:

```
$ curl\
 --data-urlencode"sql=select lname, fname from customer" \
http://localhost:9090/aceql/session/jqj39jofm3rbr092jkcbuv9f0s/\
execute_query
```

AceQL HTTP is authorized through a license. The license type depends on the edition you test,use, or buy:

Product	License
AceQL HTTP Open Source	AceQL Open Source License (LGPL v2.1)
AceQL HTTP Professional	30 day Trial Singe Server AceQL License Single Site AceQL License

The AceQL HTTP framework consists of:

- The AceQL Web Server.
- User Configuration classes injected at runtime, called "Configurators" in this document. These are server classes that   ensure both security and configuration.
- The AceQL Helper Libraries. These are SDKs for Java and C# that allow you to wrap AceQL HTTP API calls using fluent code.

The execution of each AceQL HTTP API statement is conditioned by optional rules, defined in configuration classes called "Configurators."

The AceQL Web Server embeds its own Tomcat 8.5 servlet container in order to run AceQL without any Java EE servlet container.

Note that AceQL can run inside any Java EE servlet container (see Running AceQL HTTP in a Java EE servlet container).

No Tomcat 8.5 expertise is required in order to configure and use the AceQL Web Server:

- All Tomcat configuration values are optional: AceQL uses Tomcat 8.5 default values if no user configuration is done.
- In some cases you may be required to read portions of the Tomcat 8.5 user documentation: fine tuning, using SSL & Certificates, etc.
- See the Quick Start Guide for an example that uses default Tomcat configuration.

The AceQL Web Server consists of one Java jar. It is started by calling a Java class on the command line. All configuration elements are defined in a Java `.properties` file, named **aceql-server.properties** file in this document.

All communication between the client and the server uses HTTP/ HTTPS protocols. .

This User Guide covers:

- Installation
- Configuring the server side
- Starting/Stopping The AceQL Web Server
- Session Management

- AceQL internals
- Advanced techniques
- Limitations in this version

Technical operating environment

The AceQL HTTP server side is entirely written in Java, and functions identically under Microsoft Windows, Linux, and all versions of UNIX that support Java 7+,Servlet 2.5+ and JDBC 4.0+.

The only required third party installation is a recent JVM.

The following environments are supported in this version:

Operating System	**JVM (Java Virtual Machine)**
Windows	Oracle Java SE 7, Java SE 8 and Java SE 9. (64-bit only).
UNIX/Linux	Oracle Java SE 7, Java SE 8 and Java SE 9 OpenJDK 7, OpenJDK 8 and OpenJDK 9
OS X / macOS	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+

Databases
Actian Ingres 10+
IBM DB2 9.7+
IBM Informix 11.70+.
MariaDB 10.0+
Microsoft SQL Server 2008 R2+
MS Access 2010+
Oracle Database 11g Release 2+
Oracle MySQL 5.5+
PostgreSQL 8.4.1+
Sybase ASE 15.7+
Sybase SQL Anywhere 12+
Teradata Database 13+

Notes:

- All these databases have been intensively tested with AceQL.
- The table designates the tested version. Prior versions *should* work correctly with their corresponding JDBC 4.0 driver.
- AceQL will support all subsequent versions of each database.

Download& Installation

Linux / Unix Installation

Open a terminal and download with Wget.

If you use an Open Source database:

```
wget https://www.aceql.com/rest/soft/2.1/download/aceql-http-2.1.run
```

If you use a commercial database:

```
wget https://www.aceql.com/rest/soft/2.1/download/aceql-http-pro-2.1.run
```

If you get a certificate error message, do one of the following:

1. If the problem is that a known root CA is missing and when you are using Ubuntu or Debian, then you can solve the problem with this one line: `sudo apt-getinstall ca-certificates`. Then retry the Wget call.
2. Retry the Wget call with `--no-check-certificate` at end of command line. Then check the PGP signature of the downloaded file using the corresponding `.asc` signature file available on download page using the PGP hyperlink.

In following lines we will assume that the Open Source edition is chosen. (Operating mode is the same for Professional edition).

Make the file executable and then run it:

```
chmod+x aceql-http-2.1x.run
./aceql-http-2.1x.run
```

This will create the `aceql-http-2.1x` folder that you can move where you want.

The full path to the final `aceql-http-2.1x` installation folder will be surnamed **ACEQL_HOME** in following text.

Example: if you run `aceql-http-2.1.run` from `/home/mike`, then software is installed in `/home/mike/aceql-http-2.1` which is the value of **ACEQL_HOME.**

Note

Skip this section and jump directly to Server side configuration if you plan to use AceQL in a JavaEE servlet container.

Update the PATH (Optional)

Open a shell session and make sure `java` binary is in the PATH by typing

`Java -version` on the command line.

Add `java` to your PATH if the command does not display Java version.

Add to your PATH the path to the bin directory of aceql-http-2.1 installation:

```
$ PATH=$PATH:/path/to/aceql-http-2.1/bin/
export PATH
```

Testing server installation

Open a shell session and make sure java binary is in the PATH by typing

`Java -version` on the command line.

Add java to your PATH if the command does not display Java version.

Call the `aceql-server` script to display the AceQL version:

```
 $ aceql-server -version
```

It will display a line with all version info, like:

```
AceQL HTTP Open Source v2.1 - 18-jun-2018
```

Or:

```
AceQL HTTP Professional v2.1 - 18-jun-2018
```

Windows Installation

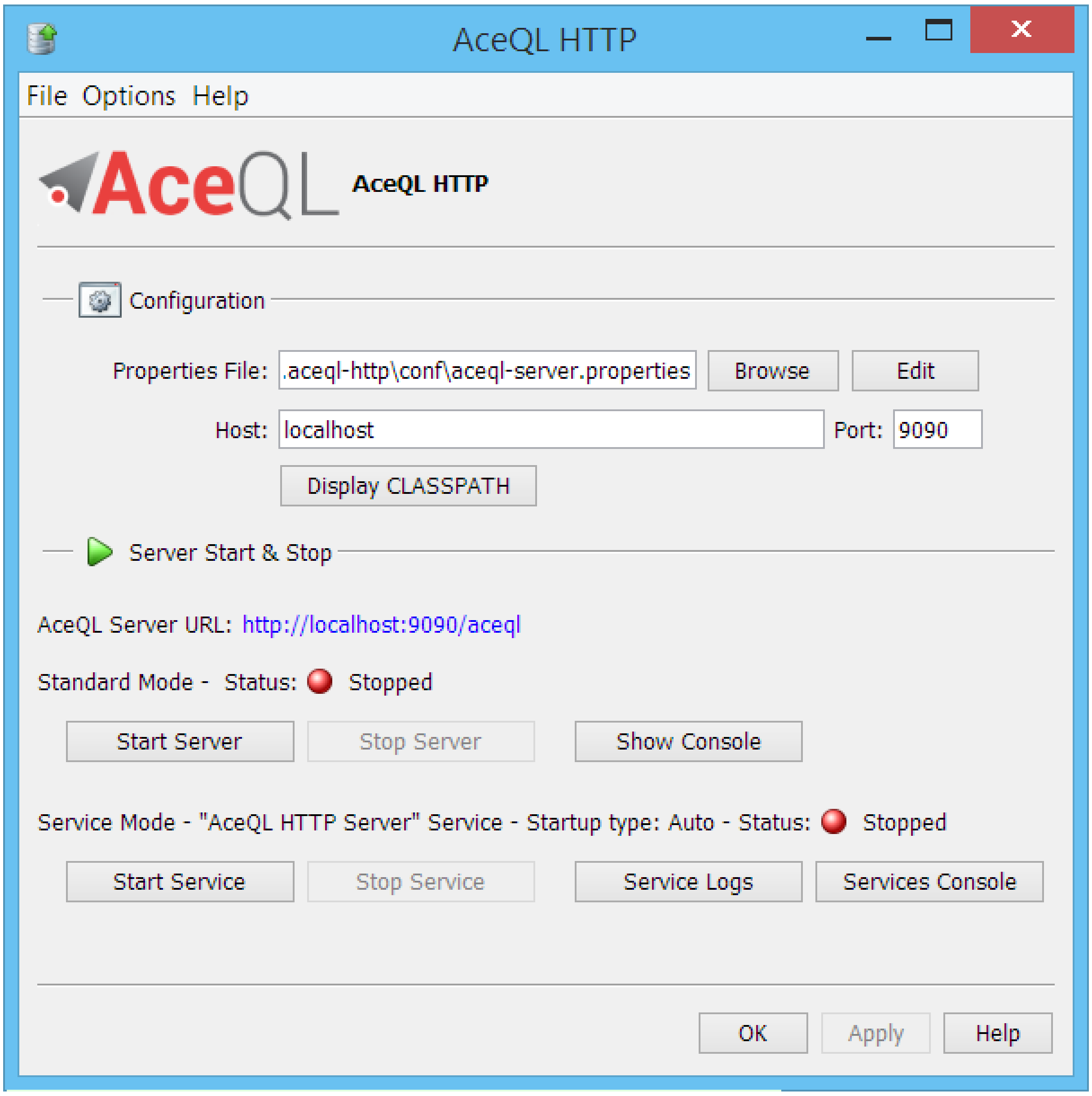
[Download AceQL Windows Installer.]()

Choose Open Source edition if you use an Open Source database, choose Professional edition for commercial database.

Because the software installs and runs a Windows Service, you must be logged as a Windows Administrator to install AceQL.

Run the installer.

It will run AceQL at end of installation and display the Window:



Properties File: .aceql-http\conf\aceql-server.properties

Host: localhost Port: 9090

AceQL Server URL: http://localhost:9090/aceql

Standard Mode - Status: ● Stopped

Service Mode - "AceQL HTTP Server" Service - Startup type: Auto - Status: ● Stopped

N.B: Because of a bug in early versions of Java 9 on Windows, the interface will appear "ugly" or "blurred" on Java 9 if you have increased Windows Screen Resolution Options to 125% or 150%. See https://goo.gl/PAVvrd for more info. Set back Windows Screen Resolution to 100% for clean display.

Server side configuration

Most AceQL configuration is carried out by defining properties in the `aceql-server.properties` file:

- Setting license parameters (only for commercial databases).
- Defining the name of AceQL Manager servlet.
- Defining the JDBC parameters of the Tomcat JDBC Connection Pool that is used to get a Connection on the server side.
- Defining embedded Tomcat settings (Optional).

A more advanced configuration using a "DatabaseConfigurator", which is all optional, will also be detailed:

- Client login and password verification.
- SQL security settings.

The AceQL Manager servlet

All HTTP commands sent by the client side are received by the AceQL Manager servlet. The AceQL Manager servlet then:

- Authenticates the client call
- Extracts a JDBC Connection from the connection pool
- Analyzes the JDBC statement that was received
- Executes the statement if the JDBC statement matches the rules defined by the "DatabaseConfigurator" (see below). Otherwise, an Exception is sent back to the client.
- Sends the result of the statement back to the client side

The aceql-server.properties file

Most AceQL configuration is carried out by defining properties in the

`aceql-server.properties` file, except the hostname and port, which are passed as start parameters.

The file is organized in Sections. Only the first 3 Sections must be configured in order to start the AceQL Manager.

- **License Info Section (Professional Edition only).**
- **AceQL Manager servlet Section.**
- **Tomcat JDBC Connection Pool Section**.
- Configurators Section .
- Default Tomcat HTTP Connector Sections.
- Session Configurator Section.
- Upload & Download Configurators Section.
- System Properties Section.
- Servlets Sections

AceQL Manager servlet Section

This section allows you to define the name of the AceQL SQL Manager Servlet to call from the client side. We recommend using the default value `aceql`:

```
aceQLManagerServletCallName=aceql
```

Tomcat JDBC Connection Pool Section

This section allows you to define:

1. The names of the databases to use

1. The JDBC parameters used to build the embedded The Tomcat 8.5 JDBC Connection Pool for each database.

The databases are defined with the databases property. If there are more than one database, separate each name with a comma:

```
# Database names separated by commas
databases = my_database1, my_database2
```

Only the following four properties must be defined per database name if you want to use the Tomcat JDBC Connection Pool:

Property	Value
driverClassName	The fully qualified Java class name of the JDBC driver to be used.
url	The connection URL to be passed to our JDBC driver to establish a connection.
username	The connection username to be passed to our JDBC driver to establish a connection.
password	The connection password to be passed to our JDBC driver to establish a connection.

Each property must be prefixed with the associated database name defined in databases property and a dot.

Example :

```
# Database names separated by commas
databases = my_database1, my_database2

# Mandatory JDBC properties:
my_database1.driverClassName = org.postgresql.Driver
my_database1.url= jdbc:postgresql://localhost:5432/kawansoft_example
my_database1.username= user1
my_database1.password= password1

my_database2.driverClassName = oracle.jdbc.driver.OracleDriver
my_database2.url= jdbc:oracle:thin:my_database2@//localhost:1521/XE
my_database2.username= user2
my_database2.password= password2
```

You may add other properties supported by Tomcat JDBC Connection Pool,

as defined in Common Attributes and in Enhanced Attributes.

Note: It is not mandatory to use Tomcat JDBC Connection Pool. If you want to use another preferred Connection Pool system, just comment the `driverClassName` property. Implementing another Connection Pool system is described in Database Configurators.

Database Configurators Section

This section is optional. It allows you to define the Database Configurators to use. The role and usage of the Configurators are described in Database Configurators.

Each Database Configurator is defined with:

- Property name: `databaseConfiguratorClassName` prefixed by the database name and a dot
- Property value: the full Database Configurator class name

If no property is defined in this section, the default value used by the Web Server session is for each database the `DefaultDatabaseConfigurator` class name :

```
database.databaseConfiguratorClassName = \
org.kawanfw.sql.api.server.DefaultDatabaseConfigurator
```

Example:

```
my_database1.databaseConfiguratorClassName = \
com.acme.MyDatabaseConfigurator
```

Default Tomcat HTTP Connector Sections (Standalone server only)

Skip this section and jump directly to Running AceQL HTTP in a Java EE servlet container if you plan to use AceQL in a JavaEE servlet container.

Tomcat HTTP Connectors allow fine tuning of Tomcat 8.5.

It is possible to define properties for a unique HTTP Connector that will either accept HTTP or secured HTTP**S** calls.

Default Tomcat HTTP Connector Section - Base attributes

This section is optional. If no value is defined, default Tomcat values will be used for the default HTTP Connector.

You may define all attributes defined in the Tomcat HTTP Connector documentation, *except SSL attributes that must be defined in the next section*.

Each property must be prefixed by connector.

```
# Example: Change default maxThreads from 200 to 300
connector.maxThreads=300
```

Default Tomcat HTTP Connector Section - SSL Attributes

This section is optional. It allows you to configure the HTTP Connector in order to use SSL when calling AceQL Manager Servlet from the client side.

It also allows you to define the SSL Certificate to be used.

Set the `SSLEnabled` property o true, in order to say that the HTTP Connector will accept SSL calls from client side.

Each property must be prefixed by `sslConnector`.

Note: If `SSLEnabled` is set to `true`, AceQL HTTP Web server will accept only SSL connections, i.e. a non SSL call from client side with `http` scheme will fail.

To define SSL HTTP Connector attribute values, refer to the Tomcat 8.5 SSL Support documentation.

The following properties are mandatory and must be defined:

Mandatory Property Name	Property Role
sslConnector.scheme	Scheme to use. Must be set to "https"
sslConnector.keystoreFile	The file containing the SSL/TLS certificates
sslConnector.keystorePass	The keystore password
sslConnector.keyPass	The certificate password

To create an SSL Certificate, refer to:

- Tomcat 8.5 Prepare the Certificate Keystore.
- Oracle JDK Security Tools.

Sample aceql-server.properties file

Here is a documented example of an aceql-server.properties file:

model-aceql-server.properties.

Starting/Stopping the AceQL Web Server from Linux/Unix

Add your JDBC driver to the AceQL installation

Before starting the AceQL Web Server, drop you JDBC driver jar into

`ACEQL_HOME/lib-jdbc directory` or add it to the Java CLASSPATH.

Starting the AceQL Web Server

Open a shell and type:

```
$ aceql-server -start -host <hostname> -port <port number> -properties <file>
```

where:

- `-host <hostname>` hostname of the Web server
- `-port <port number>` port number of the Web server. Defaults to 9090
- `-properties <file>` properties file to use for this SQL Web server Session. Defaults to `ACEQL_HOME/conf/aceql-server.properties`.

The console will display the properties used and will end with this line if everything is OK (assuming you choose 9090 for port.)

```
[ACEQL HTTP START] AceQL HTTP Web Server OK. Running on port 9090
```

If configuration errors occur, they are displayed with the tag

```
[ACEQL HTTP START FAILURE] - USER CONFIGURATION FAILURE]
```

Examples

Starting the AceQL Web Server on port 9090

```
$ aceql-server -start -host localhost
```

The URL to use on the client side will be: `http://localhost:9090/aceql`

assuming the AceQL Manager Servlet Section contains the following line:

```
aceQLManagerServletCallName=aceql
```

Starting the AceQL Web Server on port 9091

```
$ aceql-server -start -host www.acme.org -port 9091
```

The URL to use on the client side will be: `http://www.acme.org:9091/aceql`

assuming the AceQL Manager Servlet Section contains the following line:

```
aceQLManagerServletCallName=aceql
```

Using SSL from the client side

Assuming you have enabled SSL and defined a Certificate in the

aceql-server.propertiesfile, the URL to use on the client side will be:

`https://www.acme.org:9091/aceql`

Stopping the AceQL Web Server

To stop a running instance of the AceQL Web Server:

```
$ aceql-server-stop -port <port number>
```

where:

`-port <port number>` port number of the Web server. Defaults to 9090

Linux: running the AceQL Web server as a service

The `aceqlhttp` wrapper allows to run AceQL program as a Linux service.

- Click here to download `aceqlhttp.sh`
- Copy aceqlhttp.sh to `/etc/init.d/aceqlhttp` (requires root privilege).
- `sudo chmod +x /etc/init.d/aceqlhttp`
- Then edit `/etc/init.d/aceqlhttp` and:
    - Modify JAVA_HOME to the path of you Java installation.
    - Modify ACEQL_HOME to the path of your AceQL installation.
    - Modify ACEQL_HOST and ACEQL_PORT with your hostname and port.
    - Modify CLASSPATH if you plan to use a Database Configurator (See Database Configurators).

Then:

- Test that it runs: `sudo service aceqlhttp start`
- Test that it stop: `sudo service aceqlhttp stop`
- Test that it restarts: `sudo service aceqlhttp restart`

Then check the content of the log file defined by LOG_PATH_NAME and which defaults to: `/var/log/aceqlhttp.out`.

Starting/Stopping the AceQL WebServer from Windows

Server maybe started within the current JVM, or as a Windows Service.

If you wish to run the AceQL HTTP Server as a Windows Service, it is highly recommended to test your configuration by starting once the server in Standard Mode.

The running options are fully described in the user interface help.

Running the AceQL Web Server without Windows Desktop

If you don't have access to the Windows Desktop interface (running in a cloud instance, etc.) you can still run the AceQL HTTP Web Server from the command line.

- Open Sure Edition: see `<installation-directory>\AceQL\bin\aceql-server.bat` script.
- Professional Edition: see `<installation-directory>\AceQLPro\bin\aceql-server.bat` script.

You can also start/top the AceQL Web Server from you java programs, as explained in next section.

Starting/Stopping the AceQL WebServer from a Java program

You may start or stop the AceQL Server from a Java program calling the WebServerApi API.

Running AceQL HTTP in a Java EE servlet container

AceQL server side may be run inside a Java EE servlet container such as Tomcat.

This option may be preferred by users who already have a Java EE servlet container configured with all Connectors & SSL options, etc., and do want to recode the options in the `aceql-server.properties` file.

Installation

Install the files of installation directory `webapp/WEB-INF/lib` in the lib directory of your webapp.

If your JavaEE servlet container is *not* Tomcat >=7, it may not contain the Tomcat JDBC Pool: add `webapp/WEB-INF/lib-tomcat/tomcat-jdbc-8.5.xx.jar` jar in the /lib directory of you webapp.

If you have coded your own Configurators, deploy the classes in the `/classes` directory of your webapp.

AceQL servlet configuration in web.xml

Create and configure the aceql-server.properties file like normal, as described in The aceql-server.properties file. Do not configure the Tomcat Connector sections that will not be used.

In `web.xml`, define the AceQL Manager servlet that is defined in `the aceql-server.properties` file. This dual definition is required. The servlet class is. `org.kawanfw.sql.servlet.ServerSqlManager`.

Example:

Assuming the `aceql-server.properties` file is stored in `c:\Users\Mike` and you have defined the following aceQLManagerServletCallName in `aceql-server.properties`:

```
aceQLManagerServletCallName=aceql
```

then your `web.xml` should contain the following code:

```
<servlet>
    <servlet-name>aceql</servlet-name>
    <servlet-class>org.kawanfw.sql.servlet.ServerSqlManager</servlet-class>

    <init-param>
        <param-name>properties</param-name>
        <param-value>c:\Users\Mike\aceql-server.properties</param-value>
    </init-param>
</servlet>

<!—- Allows you to see immediately in servet container if servlet is OK or KO -->
< load-on-startup>1</load-on-startup >

<servlet-mapping>
    <!-- Note the trailing /* in url-pattern -->
    <servlet-name>aceql</servlet-name>
    <url-pattern>/aceql/*</url-pattern>
</servlet-mapping>
```

Note the trailing `/*` in the URL pattern: this is required by the AceQL Manager that uses both the servlet name and elements in servlet path values to execute actions requested by the client side.

Testing the servlet configuration

After restarting your server, check you web server logs.

AceQL start statuses are written on standard output stream.

Type the HTTP address of each of your AceQL Manager servlets into a browser.

Example corresponding to previous web.xml:

`http://www.yourhost.com/path-to-webapp/aceql`

It will display a JSON string and should display a status of `"OK"` and the current AceQL version:

```
{
  "status":"OK",
  "version":"AceQL HTTP v1.0 - 28-feb-2018"
}
```

If not, the configuration errors are detailed for correction.

Database Configurators

Advanced and security server configurations are implemented via Java classes called "Database Configurators" in our terminology. A Database Configurator is a user-developed Java class that implements the `DatabaseConfigurator` interface built in AceQL. The `DatabaseConfigurator` concrete instance is dynamically loaded by the AceQL Manager at bootstrap time, using Dependency Injection (DI). The methods of the `DatabaseConfigurator` instance are then called internally by the AceQL Manager servlet when necessary.

Note that AceQL comes with a default `DatabaseConfigurator` class: `DefaultDatabaseConfigurator`. You are not required to write your own `DatabaseConfigurator` (e.g. for a simple environment to test AceQL.)

See the Quick Start Guide.

Development Environment

Setting up a development environment will allow you to develop your own DatabaseConfigurators.

Create a "Server" project and add the jars of the:

- `<installation-directory>\AceQL\lib-server` subdirectory & `<installation-directory>\AceQL\lib-jdbc` to your development CLASSPATH if you installed the Open Source edition.
- `<installation-directory>\AceQLPro\lib-server` subdirectory & `<installation-directory>\AceQLPro\lib-jdbc` subdirectory to your development CLASSPATH if you installed the Professional edition.

Or for Maven users:

```
<groupId>com.aceql</groupId>
<artifactId>aceql-http</artifactId>
<version>2.1</version>
```

Database Configurator interface

The DatabaseConfigurator interface allows you to define:

1) Main configuration settings:

- Define how to extract a JDBC Connection from a Connection Pool (if you don't want to use the default Tomcat JDBC Pool).
- Define with login method if a client username and password are allowed for connection.
- Define the directories where the Blobs/Clobs are located for upload & download.
- Define the maximum number of minutes a Connection can live before it's closed and released in the pool.
- Define some Java code to execute before/after a `Connection.close()`.
- Define the Logger to use to trap server Exceptions and warning messages.

2) SQL Security settings:

- Define if a client user has the right to call a `Statement.executeUpdate()` (i.e. call a statement that updates the database).
- Define if a client user has the right to call a raw `Statement` that is not a `PreparedStatement`.
- Define a specific piece of Java code to analyze the source code of the SQL statement before allowing or not allowing its execution.
- Execute a specific piece of Java code if a SQL statement is not allowed.
- Define the maximum number of rows that   may be returned to the client.

AceQL comes with a default `DatabaseConfigurator` implementation that may be extended: DefaultDatabaseConfigurator.

If no `DatabaseConfigurator` is implemented, AceQL loads and uses the `DefaultDatabaseConfigurator` class.

A `DatabaseConfigurator` is associated with a database. Same `DatabaseConfigurator` may be used with all databases.

Passing concrete DatabaseConfigurator classes

Your concrete implementations are passed to the AceQL as properties of the Configurators Section in the `aceql-server.properties` file, as described in Database Configurators Section:

- The `databaseConfiguratorClassName` property lets you define your concrete implementation of `DatabaseConfigurator`.
- You `DatabaseConfigurator` classes must be added to the CLASSPATH before the start of the AceQL Server.

If you don't provide a parameter for a `DatabaseConfigurator`, AceQL will use the `DefaultDatabaseConfigurator` (which is already in the CLASSPATH).

Instances are loaded using a non-args constructor.

Coding Database Configurators

This section will illustrate how to code your own configuration methods in a concrete implementation of the DatabaseConfigurator interface.

Create a class `MyDatabaseConfigurator` that extends `DefaultDatabaseConfigurator`. You will then implement your own methods.

Extracting a Connection from your connection pool system

This is not necessary if you use the default Tomcat JDBC Pool embedded in AceQL: DefaultDatabaseConfigurator.getConnection().

If you want to implement your own connection pool system, overload the `DatabaseConfigurator.getConnection()` method in your concrete class implementation.

Login method - authenticating the client username and password

Of course, you – and your database administrator – don't want your SQL database to be accessible to the whole world. AceQL provides a mechanism that allows you to check the username and password sent by the remote client program. This is done through the login method of the `DatabaseConfigurator` interface. If login returns true, access is granted.

Note that the username and password checked here are not the database username and password. The username and password should be checked via an applicative access mechanism, such as an LDAP directory, a login table in the database, etc.

The following example checks that the username and password entered by the client match an access list defined in an SQL table of the host database. Add the method to your `MyDatabaseConfigurator` class:

```
    /**
     * Our own Acme Company authentication of remote client users.
     * This methods overrides the {@link DefaultDatabaseConfigurator#login}
     * method. <br>
     * The (username, password) values are checked against the
     * user_login table.
     *
     * @param username
     *            the username sent by AceQL client side
     * @param password
     *            the user password sent by AceQL client side
     * @param database
     *            the database name to which the client wants to connect
     * @param ipAddress
     *            the IP address of the client user
     * @return true if access is granted, else false
     */
    @Override
    public boolean login(String username, char[] password, String database,
        String ipAddress) throws IOException, SQLException {

    PreparedStatement prepStatement = null;
    ResultSet rs = null;

    // Always close the Connection so that it is put
    // back into the pool for another user at end of call.

    try (// Extract a Connection from our Pool
    Connection connection = super.getConnection(database);) {

        String hashPassword = null;

        try {
        hashPassword = SimpleSha1.sha1(new String(password), true);
        } catch (Exception e) {
        throw new IOException("Unexpected Sha1 failure", e);
        }

        // Check (username, password) existence in user_login table
        String sql = "SELECT username FROM user_login "
            + "WHERE username = ? AND hash_password = ?";
        prepStatement = connection.prepareStatement(sql);
        prepStatement.setString(1, username);
        prepStatement.setString(2, hashPassword);

        rs = prepStatement.executeQuery();

        boolean ok = false;
        if (rs.next()) {

        ok = true; // Yes! (username, password) are authenticated
        }

        prepStatement.close();
        rs.close();
        return ok;
    }
  }
```

The `DatabaseConfigurator` interface contains other methods that you can implement in your `MyDatabaseConfigurator` class to define your configuration preferences:

`getBlobsDirectory`:

Implement this method if you want to choose the directories where the Blobs/Clobs are located for upload & download.

`getConnectionMaxAge`:

Implement this method if you want to define the maximum number of minutes a `Connection` can live before it's closed and released in the pool.

`close`:

Implement this method if you want to define some Java code to execute before/after a `Connection.close()`.

Please check the Javadoc of DatabaseConfigurator for more information.

Coding SQL security settings

What are the purposes of SQL security settings

The purposes of the SQL security settings on the server side are:

- to decide whether certain JDBC classes or methods should be allowed to execute or be rejected.
- To filter and syntactically analyze all incoming JDBC calls (or SQL statements) from the client.
- To define an action to be executed if the JDBC call is refused.

Note that he basic security settings are sufficient to protect against illegitimate users that do *not* have a valid username and password: such users won't be able to connect to the database if the login is refused.

These SQL security settings are necessary in an environment where unknown users may be present. An example is an Internet desktop application. A legitimate user with a correct (username, password) pair could potentially submit illegitimate SQL statements for your database.

A good dual defense strategy would be:

- To obfuscate your mobile & desktop app code so that it is very hard for the user to discover the table and column   names
- To analyze all incoming SQL statements on the server side and verify that they are correct and legitimate. If not, discard the user and ban his IP address.

The DatabaseConfigurator interface SQL security methods

The SQL security settings are coded in a concrete implementation of the DatabaseConfigurator interface with the following methods:

Method	**Role**
allowExecuteUpdate	Defines if a client user has the right to call a statement that updates the database
allowStatementClass	Defines if a client user has the right to call a raw `Statement` that is not a `PreparedStatement`
allowStatementAfterAnalysis	Defines a specific piece of Java code to get the caller username and IP address and analyze the source code of the SQL statement before allowing or disallowing its execution
runIfStatementRefused	Executes a specific piece of Java code if an SQL statement is not allowed
getMaxRows	Defines the maximum number of rows per request to be returned to the client. If the defined limit is exceeded, the excess rows are silently dropped.

AceQL default implementation (DefaultDatabaseConfigurator) has minimal restrictions and should *not* be used in production code. It allows *all* SQL code execution.

For example, assume that you want more security in your own `DatabaseConfigurator` implementation that defines these requirements:

- Username does not exist in applicative SQL table `BANNED_USERNAMES`

- Statement does not contain SQL comments.
- Statement does not contain the statement separator character: `;`
- Statement is a DML statement: DELETE / INSERT / SELECT / UPDATE.
- DELETE / UPDATE statements that are not a `PreparedStatement` or that have no parameters are rejected.
- Any UPDATE on the `USER_LOGIN` and `PRODUCT_ORDER` tables requires that the USERNAME value is the last parameter of the `PreparedStatement`.
- If an illegitimate SQL statement is detected, the username is inserted in an applicative table   that stores banned usernames.

The StatementAnalyzer utility class allows analysis of the SQL statement.

Create a `MySqlConfigurator` class that extends `DefaultSqlConfigurator` and implements `allowStatementAfterAnalysis`:

```
    /**
     * Allows, for the passed client username, to analyze the string
     * representation of the SQL statement that is received on the server. <br>
     * If the analysis defined by the method returns false, the SQL statement
     * won't be executed.
     *
     * @param username
     *            the client username to check the rule for.
     * @param connection
     *            The current SQL/JDBC <code>Connection</code>
     * @param ipAddress
     *            the IP address of the client user
     * @param isPreparedStatement
     *            Says if the statement is a prepared statement
     * @param sql
     *            the SQL statement
     * @param parameterValues
     *            the parameter values of a prepared statement in the natural
     *            order, empty list for a (non prepared) statement
     * @return <code><b>true</b></code> if all following requirements are met:
     *         <ul>
     *         <li>username does not exists in applicative SQL table
     *         BANNED_USERNAMES.</li>
     *         <li>SQL string must *not* contain SQL comments.</li>
     *         <li>SQL string must *not* contain any semicolon.</li>
     *         <li>SQL string statement must be a DML: DELETE / INSERT / SELECT
     *         / UPDATE.</li>
     *         <li>DELETE / UPDATE statements that are not a PreparedStatement
     *         or that have no parameters are rejected.</li>
     *         <li>Any UPDATE on the USER_LOGIN and PRODUCT_ORDER tables
     *         requires that USERNAME value is the last parameter of the
     *         PreparedStatement.</li>
     *         <li>If an illegitimate SQL statement is detected, discard the
     *         username and log his IP as a banned IP.</li>
     *         </ul>
     *

     * @throws IOException
     *             if an IOException occurs
     * @throws SQLException
     *             if a SQLException occurs
     */
    @Override
    public boolean allowStatementAfterAnalysis(String username,
        Connection connection, String ipAddress, String sql,
        boolean isPreparedStatement, List<Object> parameterValues)
        throws IOException, SQLException {

    // First thing is to test if the username has previously been stored in
    // our applicative BANNED_USERNAME table
    String sqlOrder = "SELECT USERNAME FROM BANNED_USERNAMES WHERE USERNAME = ?";

    PreparedStatement prepStatement = connection.prepareStatement(sqlOrder);
    prepStatement.setString(1, username);
    ResultSet rs = prepStatement.executeQuery();

    boolean usernameBanned = rs.next();
    prepStatement.close();
    rs.close();

    if (usernameBanned) {
        return false;
    }

    // We will start statement analysis on the SQL string.
    StatementAnalyzer statementAnalyzer = new StatementAnalyzer(sql,
        parameterValues);

    // SQL string must *not* contain SQL comments ==> Possible security
    // hole.
    if (statementAnalyzer.isWithComments()) {
        return false;
    }

    // SQL string must *not* contain ";" ==> Possible security hole.
    if (statementAnalyzer.isWithSemicolons()) {
        return false;
    }

    // SQL string statement must be a DML: DELETE / INSERT / SELECT /
    // UPDATE.
    if (!statementAnalyzer.isDml()) {
        return false;
    }

    // Any UPDATE on the USER_LOGIN and PRODUCT_ORDER tables requires that
    // USERNAME value is the last parameter of the PreparedStatement

    if (statementAnalyzer.isUpdate() || statementAnalyzer.isDelete()) {
        String table = statementAnalyzer.getTableNameFromDmlStatement();

        if (table == null) {
```

```
        return false;
        }

        if (!isPreparedStatement) {
        return false;
        }

        if (table.equalsIgnoreCase("USER_LOGIN")
            || table.equalsIgnoreCase("PRODUCT_ORDER")) {
        String lastParamValue = null;

        lastParamValue = statementAnalyzer.getLastParameter()
            .toString();

        if (!lastParamValue.equals(username)) {
            return false;
        }
        }
    }

    // OK, accept the statement!
    return true;
  }
```

We can now implement `runIfStatementRefused`:

```
    /**
     * Insert the username that made an illegal SQL call and its IP address
     * into the BANNED_USERNAMES table. From now on, the username will not be
     * able to make any further AceQL HTTP calls.
     *
     * @param username
     *            the discarded client username
     * @param connection
     *            The current SQL/JDBC <code>Connection</code>
     * @param ipAddress
     *            the IP address of the client user
     * @param sql
     *            the SQL statement
     * @param parameterValues
     *            the parameter values of a prepared statement in the natural
     *            order, empty list for a (non prepared) statement
     *
     * @throws IOException
     *             if an IOException occurs
     * @throws SQLException
     *             if a SQLException occurs
     */
    @Override
    public void runIfStatementRefused(String username, Connection connection,
        String ipAddress, String sql, List<Object> parameterValues)

        throws IOException, SQLException {
```

```
    // Call the parent method that logs the event:
    super.runIfStatementRefused(username, connection, ipAddress, sql,
        parameterValues);

    // Insert the username & its IP into the banned usernames table
    String sqlOrder = "INSERT INTO BANNED_USERNAMES VALUES (?, ?)";

    PreparedStatement prepStatement = connection.prepareStatement(sqlOrder);
    prepStatement.setString(1, username);
    prepStatement.setString(2, ipAddress);
    try {
        prepStatement.executeUpdate();
    } catch (SQLException e) {
        // Case the instance already exists
        System.err.println(e.toString());
    }
    prepStatement.close();

}
```

Session management and security

SessionConfigurator interface

After server authentication succeeds (through the DatabaseConfigurator.login(). method), the AceQL Manager builds an authentication session id that is sent back to the client and will be used by each succeeding client call in order to authenticate the calls.

Session security is managed by implementing the SessionConfigurator interface that defines how to generate and verify the session id for (username, database) sessions.

Interface implementation allows you to:

- Define how to generate a session id after client /login call
- Define the session's lifetime
- Define how to verify that the stored session is valid and not expired

Session management default implementation

The default mechanism that builds an authentication session id is coded in the class

DefaultSessionConfigurator:

- Session ids are generated using a `SecureRandom` with the SessionIdentifierGenerator class.
- Session info (username, database) and session date/time creation are stored in a `HashMap`, whose key is the session id.
- Session id is sent by client side at each API call. AceQL verifies that the `HashMap` contains the username and that the session is not expired to grant access to the API execution.

Benefits of this implementation are:

- Session ids are short and generate less HTTP traffic.
- Because session ids are short, they are easy to use “manually” (with cURL, etc.)

The disadvantage is that session information is stored on the server side.

Session management using JWT

Session management using JWT is coded in JwtSessionConfigurator.

Session management is done using self-contained JWT (JSON Web Token).

See https://jwt.io for more information on JWT.

A benefit of JWT is that no session information is stored on the server and that it allows full stateless mode.

A disadvantage of JWT is that the tokens are much longer and thus generate more http traffic and are less convenient to use "manually" (with cURL, etc.).

Activating JwtSessionConfigurator

Edit the `aceql-server.properties` file and uncomment the two lines:

```
sessionConfiguratorClassName=\
org.kawanfw.sql.api.server.session.JwtSessionConfigurator
jwtSessionConfiguratorSecret=changeit
```

Change the `jwtSessionConfiguratorSecret` property change it value with your own secret value.

Restart the AceQL Web Server for activation.

Creating your own session management

If you want to create your session management using your own session id generation and security rules, you can implement the SessionConfigurator in your own class, and then:

Add your class in the CLASSPATH.

Add you class name in the `SessionConfigurator` section in your `aceql-server.properties` file:

```
sessionConfiguratorClassName=com.acme.MySessionConfigurator
```

Restart the AceQL Web Server for activation.

Interacting with the JDBC Pool at runtime

The Servlets Section in `aceql-server.properties` allow to define you own servlets in order to interact with AceQL Web Server with different actions :

- query info about JDBC pools in use,
- modify a pool size,
- etc.

The API DataSourceStore class allows to retrieve for each database the Tomcat org.apache.tomcat.jdbc.pool.DataSource corresponding to the Tomcat JDBC Pool created at AceQL Web server startup.

State management / Stateful Mode

AceQL runs in "Stateful Mode": when creating a session on the client side with `/login` API, the AceQL servlet that is contacted extracts a JDBC `Connection` from the connection pool (with `DatabaseConfigurator.getConnection(`)) and stores it in memory in a static Java `Map`.

The server's JDBC Connection is persistent, attributed to the client user, and will not be used by other users: the same `Connection` will be used for each JDBC call until the end of the session. This allows you SQL transactions to be created.

The `Connection` will be released from the AceQL Manager Servlet memory and released into the connection pool by a client side `/close` or `/logout` API call.

A server side background thread will release phantom Connections that were not closed by the client side.

Therefore, it is important for client applications to explicitly and systematically call `/logout` API before the application exits, in order to avoid phantom Connections to persist for a period of time on the server.

AceQL internals

This chapter describes some technical and implementation aspects of AceQL.

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 JSON.simple library and Google Gson library)

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.

Managing temporary files

AceQL uses temporary files, these temporary files contain:

- Contents of Result Sets
- Contents of Blobs and Clobs

Temporary files are created to allow streaming and/or to allow the earliest possible release of SQL resources and network resources.

These temporary files are automatically cleaned (deleted) by AceQL on the server side.

If you want to ensure that temporary files will be cleaned, you can access the temporary directories:

1. `ResultSet` data is dumped in `user.home/.kawansoft/tmp` directory

1. The uploaded/downloaded Blob or Clob files are located in the directory defined by `DatabaseConfigurator.getBlobsDirectory()`. Default `DefaultDatabaseConfigurator.getBlobsDirectory()` implementation stores the Blob/Clob files in `user.home/.aceql-server-root/username`.

Where:

- `user.home` = the user.home of the user that started the AceQL Web Server.
- `username` = the username of the client user.