



**SafeJDBC – v2.0
SQL encryption Driver
User Documentation
January 31, 2005**

Table of Contents

1	INTRODUCTION.....	4
1.1	PRESENTATION OF SAFEJDBC	4
1.2	TARGET AUDIENCE	4
1.3	TECHNICAL OPERATING ENVIRONMENT	5
2	INSTALLING SAFEJDBC.....	6
2.1	INSTALLING THE JDK 1.4 FROM SUN MICROSYSTEMS	6
2.2	INSTALLING THE SAFEJDBC BINARY FILES	6
2.2.1	Standard installation	6
2.2.2	Installing the SafeJDBC binaries	6
2.2.3	Customized installation.....	7
2.3	LICENSE FILE	7
2.4	UPDATING THE JAVA CLASSPATH	8
2.4.1	Windows NT/2000/XP Classpath	8
2.4.2	Unix/Linux classpath.....	8
2.5	JAVADOC	8
2.6	JAVADOC ON LINE	8
3	SAFEJDBC OPERATING PRINCIPLES	9
3.1	TERMINOLOGY	9
3.2	LOADING THE SAFEJDBC DRIVER.....	9
3.3	EXAMPLE OF ENCRYPTING VALUES USING AN INSERT	11
3.4	EXAMPLE OF DECRYPTION OF VALUES USING A SELECT.....	12
3.5	CONTENTS OF THE SQL DATABASE AFTER THE INSERT.....	13
4	SQL ASPECTS	14
4.1	TAKING PRODUCTION CONSTRAINTS INTO ACCOUNT	14
4.2	AUTHORIZED SQL SYNTAX	14
4.3	IMPACT OF ENCRYPTION ON DATA SIZE	16
4.4	THE SAFEJDBC_CATALOG SQL DATABASE	16
4.5	INTEGRITY AND BACKUPS OF SAFEJDBC_CATALOG	16

5	UTILIZATION & THE API	17
5.1	CREATING THE SAFEJDBC_CATALOG DATABASE.....	17
5.2	SAFEJDBCSETTER – THE SAFEJDBC DRIVER CONFIGURATION API	17
5.2.1	getInstance() – Creating an instance of SafeJdbcSetter	17
5.2.2	setWrappedDriverName() – Configuring the encapsulated native driver.....	18
5.2.3	setSafeJdbcDbUrl() – Setting the URL for SAFEJDBC_CATALOG	18
5.2.4	Defining the Logfile	19
5.2.5	Loading the SafeJDBC driver	19
5.3	SAFEJDBCSETTER - API FOR CONFIGURING CRYPTOGRAPHIC ELEMENTS	20
5.3.1	Generalities.....	20
5.3.2	Providers.....	20
5.3.3	setAlgorithm() – Setting the algorithm to use	21
5.3.4	setKey() – Setting the secret encryption key	22
5.4	CALLING SAFEJDBC IN PURE JDBC DRIVER MODE	24
5.4.1	Principle.....	24
5.4.2	Configuring con_safejdbc.ini	24
5.5	VERIFYING THE SECURE STATE OF A CONNECTION.....	26
5.6	HOW TO GET THE WRAPPED «NATIVE» CONNECTION	26
5.7	CIPHERCONNECTIONFACTORY – API FOR CREATING A "CONNECTION"	27
5.8	USING DATASOURCES WITH J2EE APPLICATION SERVERS	29
5.8.1	Generalities.....	29
5.8.2	The SafeDataSourceFactory JavaBean	29
5.9	REAL WORLD EXAMPLE: SECURE DATASOURCES WITH TOMCAT 5.0.....	30
5.9.1	Step 1: define the application database DataSource	31
5.9.2	Step 2 : Define the SafeJDBC Catalog DataSource.....	32
5.9.3	Step 3 : Define the Secure SafeDataSource	33
5.10	COLUMNSCIPHER - API FOR SELECTING THE COLUMNS TO ENCRYPT/DECRYPT	35
5.10.1	Generalities.....	35
5.10.2	getInstance () – Creating an instance of ColumnsCipher.....	35
5.10.3	setOutputStream() – Configuring the output stream	35
5.10.4	setFileOutputStream() – Setting up an output stream to a file	36
5.10.5	addColumn() – Adding a column	36
5.10.6	addColumnFromFile() – Adding columns from an ".ini" file	37
5.10.7	defineJoinColumn() – Defining encrypted joins	38
5.10.8	defineJoinColumnsFromFile() – Defining Join columns from an "ini" file.....	39
5.10.9	execute() – Running the encrypting or decrypting process	40
5.11	COLUMNSCIPHERMAIN BATCH FILE	42
5.12	HOW TO DEFINE ENCRYPTED COLUMNS IN A VIEW	44
6	SUPPORT	47

1 INTRODUCTION

1.1 Presentation of SafeJDBC

SafeJDBC is a JDBC driver which is used in conjunction with any driver for JDBC version 2.x or 3.x. SafeJDBC enables SQL columns to be encrypted or decrypted on the fly without adding to Java source code.

SafeJDBC takes care of:

- Encrypting values in SQL columns based on the unencrypted values in Java *before* the database is updated by an SQL 92 INSERT or UPDATE command.
- Decrypting values from SQL columns under the JVM *after* they have been retrieved from the database (SELECT command, or the columns in WHERE conditions of UPDATE or DELETE commands).

SafeJDBC transparently encapsulates the Java application's usual driver, which is simply configured by prior Java instructions.

1.2 Target audience

This document is intended for:

- 1) Java developers who would like to include SafeJDBC in their programs (Java classes, scripts, JSP etc.)
- 2) SQL DBAs (database administrators) who would like to find out about the effects of SafeJDBC on the operation of their databases.

1.3 Technical operating environment

SafeJDBC is 100% written in Java, and functions identically under Windows NT/2000/XP, Linux and all versions of Unix supporting Java and JDBC.

Here are the environments supported in this version:

Technical environment	Versions supported
Windows NT	<ul style="list-style-type: none"> • Windows NT Server V 4.0 SP 4 or above. • Windows NT Workstation V 4.0 SP 4 or above.
Windows 2000/XP	<ul style="list-style-type: none"> • Windows 2000 Server SP 2 or above. • Windows 2000 Professional SP 2 or above. • Windows XP Professional SP 1 or above.
Unix	<ul style="list-style-type: none"> • Sun Solaris 7.x/8.x/9.x (SPARC and Intel). • Linux (Intel) RedHat 6.1/7.1/8.0/9.0. • Other Unix versions which support Java.
JVM (Java Virtual Machine)	<ul style="list-style-type: none"> • JDK 1.4.1 or above from Sun Microsystems.

The API is supported for all these Java environments and subsequent versions.

2 INSTALLING SAFEJDBC

2.1 Installing the JDK 1.4 from Sun Microsystems.

SafeJDBC requires the installation of the JDK 1.4 from Sun Microsystems.

The installation of the JDK 1.4 is described on Sun Microsystems' website at the following address: <http://java.sun.com/j2se>.

This page also contains downloads for other environments: Sun Solaris, Itanium, etc.

2.2 Installing the SafeJDBC binary files

Download `safejdbc_2.0.zip` file at:

http://www.safelogic.com/safejdbc/download/safejdbc_v2.00.zip

2.2.1 Standard installation

We suggest that the following installation directories be used by default:

Windows NT/2000/XP	Unzip and copy the <code>\safelogic</code> directory to <code>c:\</code>
Unix/Linux	Create a user named <code>safelogic</code> , unzip and copy the <code>\safelogic</code> directory in the WinZip file into <code>/home</code>

2.2.2 Installing the SafeJDBC binaries

This is the resulting Windows tree:

Windows directory (by default)	Comments
<code>c:\safelogic\lib</code>	Java common libraries.
<code>c:\safelogic\safejdbc</code>	SafeJDBC base directory.
<code>c:\safelogic\safejdbc\bin</code>	Scripts for classpath.
<code>c:\safelogic\safejdbc\conf</code>	Configuration files.
<code>c:\safelogic\safejdbc\doc</code>	User Documentation.
<code>c:\safelogic\safejdbc\examples</code>	Sample Java programs.
<code>c:\safelogic\safejdbc\javadoc</code>	Javadoc.
<code>c:\safelogic\safejdbc\lib</code>	Java SafeJDBC libraries.

This is the resulting Unix/Linux tree:

Unix/Linux directory	Comments
/home/safelogic/lib	Java common libraries.
/home/safelogic/safejdbc	SafeJDBC base directory.
/home/safelogic/safejdbc/bin	Scripts for classpath.
/home/safelogic/safejdbc/conf	Configuration files.
/home/safelogic/safejdbc/doc	User Documentation.
/home/safelogic/safejdbc/examples	Sample Java programs.
/home/safelogic/safejdbc/javadoc	Javadoc.
/home/safelogic/safejdbc/lib	Java SafeJDBC libraries.

2.2.3 Customized installation

SafeJDBC may be installed in any directory.

The only constraint is to update the CLASSPATH (defined below) with the corresponding files and directories.

2.3 LICENSE FILE

SafeJDBC uses a license file called `safejdbc_license.txt`.

The lines contains :

- The keyword “safejdbc”
- A license code for the type of product.
- The server names list (HOSTNAME), separated by “;”
- A product evaluation end date or 99999999 for full versions.
- Your email address.
- A hexadecimal string corresponding to the signature for the above elements with a private RSA key.

By default, `safejdbc_license.txt` is installed in the `conf` directory. (`c:\safelogic\safejdbc\conf` or `/home/safelogic/safejdbc/conf`)

If you choose a customized installation, please follow these constraints:

- `safejdbc_license.txt` must be installed in a directory defined in the classpath.
- `safejdbc_license.txt` must be installed in the same directory as `license@safelogic.com_1_RSA.pkf`. (This default file installation is in the `conf` directory).

2.4 Updating the Java CLASSPATH

The CLASSPATH Java environment variable must include:

1. Access path to the two libraries `cryptix.jar` and `safejdbc.jar`.
2. Access path to the `conf` directory that contains `safejdbc_license.txt` and `license@safelogic.com_1_RSA.pkf`.

2.4.1 Windows NT/2000/XP Classpath

For the standard installation, add the following string to the CLASSPATH:

```
c:\safelogic\lib\cryptix.jar;c:\safelogic\safejdbc\lib\safejdbc.jar;c:\safelogic\safejdbc\conf
```

2.4.2 Unix/Linux classpath

For the standard installation, add the following string to the CLASSPATH:

```
/home/safelogic/lib/cryptix.jar:/home/safelogic/safejdbc/lib/safejdbc.jar  
:/home/safelogic/safejdbc/conf
```

2.5 Javadoc

SafeJDBC Javadoc is in the `/safelogic/safejdbc/javadoc` directory of the `safejdbc_v2.00.zip` file.

2.6 Javadoc on line

SafeJDBC Javadoc is browsable online at:
<http://www.safelogic.com/safejdbc/v2.00/javadoc>.

3 SAFEJDBC OPERATING PRINCIPLES

3.1 Terminology

From here on, we shall use the following terms to clarify the discussion:

Term	Meaning
Java application	This is the Java application accessing SQL via JDBC, using SafeJDBC for encryption.
Native JDBC driver	The Java application's driver which is usually used to access the SQL engine via JDBC.

3.2 Loading the SafeJDBC driver

SafeJDBC encapsulates the native JDBC driver being used.

So the instruction to load a MySQL native JDBC driver is as follows:

```

/**
 * Load the Driver & return a Connection
 * @param sPassword    the password connection
 *                    to the sj_clients database
 */
public Connection loadJdbcDriver(String sPassword)
    throws Exception
{
    // Set JDBC Driver & application database address.
    String sDriver = "org.gjt.mm.mysql.Driver";
    String sDbUrl
        = "jdbc:mysql://localhost:3306/sj_clients?"
          + "user=safelogic&password="+ sPassword;

    // load the Driver and get a Connection to the database
    Class.forName(sDriver).newInstance();
    Connection connection = DriverManager.getConnection(sDbUrl);

    return connection;
}

```

To activate SafeJDBC encryption, modify this method as follows:

- Define sDriver using the name `com.safejdbc.api`.
- Define the name of the original Driver to be “wrapped” or encapsulated.
- Create an instance of **SafeJdbcSetter**.
- Define a key derived from a passphrase.

This gives the modified **loadJdbcDriver** method:

```
/**
 * Load the Driver & return a Connection
 * @param sUserId      the SafeJDBC UserId
 * @param sPassword    the password connection
 *                    to the sj_clients database
 * @param caPassprhase the passphrase to use as key
 *                    for the SafeJDBC Driver
 */
public Connection loadJdbcDriver(String sUserId,
                                char [] caPassword,
                                char [] caPassprhase)
    throws Exception
{
    // Set JDBC Driver & application database address.

    String sDriver = "com.safejdbc.api.Driver";
    String sDbUrl
        = "jdbc:mysql://localhost:3306/sj_clients?"
          + "user=safelogic&password=" + new String(caPassword);

    String sNativeDriver = "org.gjt.mm.mysql.Driver";

    String sJdbcUrl = "jdbc:mysql://localhost:3306/"
        + "safejdbc_catalog?" + "user=safelogic&password="
        + new String(caPassword);

    // create a SafeJDBC instance
    SafeJdbcSetter sjSetter = SafeJdbcSetter.getInstance();

    // set the Wrapped Driver
    sjSetter.setWrappedDriverName(sNativeDriver);

    // set SafeJdbc's own catalog
    sjSetter.setSafeJdbcDbUrl(sJdbcUrl);

    // set the encryption key for cipher & decipher operations
    sjSetter.setKey(sUserId, caPassprhase);

    // load the Driver and get a Connection to the database
    Class.forName(sDriver).newInstance();

    Connection connection = DriverManager.getConnection(sDbUrl);

    if (! (connection instanceof CipherConnection))
    {
        throw new SQLException("SafeJDBC Driver not loaded!");
    }

    return connection;
}
```

3.3 Example of encrypting values using an INSERT

SafeJDBC receives the SQL instructions to be sent to the DBMS and encrypts the values corresponding to the columns defined as being “to be encrypted”. We will look at how and where to define the columns to be encrypted later.

For example, we define a CLIENTS table, in which we wish to encrypt the last name, first name, telephone and email fields.

```

/**
 * Insert a row in the database
 * @param conn      The database Connection
 * @throws SQLException
 */
public void doInsert(Connection conn) throws SQLException
{
    String sStatement =
        "INSERT INTO CLIENTS VALUES "
        + " (1, 'Smith', 'John', 1, NULL, 'john@smith.com' ,"
        + " '01 45 72 25 15', '01 45 72 25 15', 7, "
        + " 'Bld de Dixmude', 75007, 'Paris', 'fr' )";

    Statement stmt = conn.createStatement();

    stmt.executeUpdate(sStatement);
    stmt.close();
}

```

SafeJDBC intercepts the code in **sStatement** and “rewrites” it as follows before transmitting it to the SQL engine via the original JDBC driver:

```

INSERT INTO CLIENTS VALUES ( 1, 'VpOAdDo=', '+/LHjw==', 1, NULL,
'7qrhr5ZExgVrxkE8PR8=', 'xbQXUnruJQrVRNHkBG0=', '01 45 72 25 15', 7,
'Bld de Dixmude', 75007, 'Paris', 'fr' )

```

The principle is the same for UPDATE instructions.

SafeJDBC follows the same principle with the **PreparedStatement**'s associated parameters: each parameter value is encrypted by the setXxxx() instruction before being sent to the SQL engine.

3.4 Example of decryption of values using a SELECT

SafeJDBC retrieves the contents of the Statement and encrypts the conditions affecting encrypted columns.

```

/**
 * Select rows from the database
 * @param conn      The database Connection
 * @throws SQLException
 */
public void doSelect(Connection conn) throws Exception
{
    String sStatement =
    "SELECT CLI_REF, CLI_LASTNAME, CLI_FIRSTNAME, CLI_GENDER, "
    + " CLI_BIRTHDATE, CLI_EMAIL, CLI_PHONE, CLI_FAX, "
    + " CLI_ADDR_STREET_NUM, CLI_ADDR_STREET_NAME, "
    + " CLI_ADDR_ZIP, CLI_ADDR_TOWN "
    + " FROM CLIENTS WHERE (CLI_REF = ? OR CLI_REF = ?)"
    + "          AND (CLI_EMAIL = 'john@smith.com' "
    + "          OR CLI_EMAIL = 'robert@wesson.com')";

    PreparedStatement prepStmt
    = conn.prepareStatement(sStatement);

    prepStmt.setInt(1, 1);
    prepStmt.setInt(2, 10);

    ResultSet rs = prepStmt.executeQuery();

    while (rs.next())
    {
        String sCliRef      = rs.getString("CLI_REF");
        String sCliLastname = rs.getString("CLI_LASTNAME");
        String sCliFirstname = rs.getString("CLI_FIRSTNAME");
        // Etc..
    }

    rs.close();
    prepStmt.close();
}

```

Before being sent to the SQL engine, the **Statement** is translated into:

```

SELECT CLI_REF, CLI_LASTNAME, CLI_FIRSTNAME, CLI_GENDER, CLI_BIRTHDATE,
CLI_EMAIL, CLI_PHONE, CLI_FAX, CLI_ADDR_STREET_NUM, CLI_ADDR_STREET_NAME,
CLI_ADDR_ZIP, CLI_ADDR_TOWN FROM CLIENTS WHERE (CLI_REF = ? OR CLI_REF = ?)
AND (CLI_EMAIL = '7qrhr5ZExgVrxkE8PR8='
OR CLI_EMAIL = '9qrrpKRD6xufb3uqKVODGbg=');

```

The ResultSet retrieved in the JVM only contains encrypted values for the last name, first name, telephone and email columns.

Each `rs.getXxx("column_name")` decrypts the corresponding value in the ResultSet (of the native JDBC driver) before it is sent via the SafeJDBC ResultSet.

3.5 Contents of the SQL database after the INSERT

The SQL database now contains a mixture of encrypted and unencrypted information.

Here is a sample dump from a MySQL 3.23 database under Windows 2000:

```
***** 1. row *****
      cli_ref: 1
      cli_lastname: VpOAdDo=
      cli_firstname: +/LHjw==
      cli_gender: 1
      cli_birthdate: 20030124174250
      cli_email: 7qrhr5ZExgVrxkE8PR8=
      cli_phone: xbQXUnruJQrVRNHkBG0=
      cli_fax: 01 45 72 25 15
      cli_addr_street_num: 7
      cli_addr_street_name: Bld de Dixmude
      cli_addr_zip: 75007
      cli_addr_town: Paris
      cli_addr_country_code: fr
2 rows in set (0.00 sec)

mysql>
```

4 SQL ASPECTS

4.1 Taking production constraints into account

SafeJDBC was designed to operate in a production SQL environment.

SafeJDBC generates **no** overhead on the SQL engine when Java applications are running:

- SafeJDBC does not access the SQL catalogue.
- SafeJDBC never creates extra SQL requests.
- SafeJDBC never transforms the original requests (except to remove the UPPER() and LOWER() functions).
- SafeJDBC uses an isolated SQL meta-database, which is only called when the client application starts.

4.2 Authorized SQL syntax

We can distinguish between two cases:

- SafeJDBC supports all SQL syntaxes for requests which do not refer to encrypted columns. In these cases, the request is transmitted directly to the native JDBC driver.
- SafeJDBC supports the SQL 92 syntax for requests which refer to encrypted columns, with the restrictions described in the table below.

Area of limitation	Limitations for encrypted columns in SafeJDBC Version 1.0
Column formats	Encrypted columns must be of the following types: INTEGER, CHAR, VARCHAR, LONGVARCHAR, BINARY, VARBINARY, LONGVARBINARY, BLOB, CLOB
UPDATE	If a SET is used on an encrypted column, the value must be alphanumeric between “ quotes ” or numeric.
WHERE clause	<ul style="list-style-type: none"> • The following comparison operators may be used on encrypted columns: =, <>, !=, IN. • Encrypted columns may only be compared with a discrete list of alphanumeric or numeric values. • The following comparison operators are not supported for encrypted columns: >, >=, <, <=, BETWEEN, LIKE • The following operators are not authorized for encrypted columns: ALL, ANY, EXISTS, UNIQUE.
Sorting	An ORDER BY clause for an encrypted column will cause an SQL warning.
Grouping functions	GROUP BY [HAVING] Grouping functions are not authorized on encrypted columns, except for COUNT
Other functions	<ul style="list-style-type: none"> • Only the UPPER, LOWER and TRIM functions are supported for encrypted columns. • The other functions are not supported for encrypted columns: POSITION, CHAR_LENGTH, BIT_LENGTH, EXTRACT, SUBSTRING
Joins	Joins are accepted for encrypted columns with the following restrictions: <ul style="list-style-type: none"> • Only tests for equality and difference are accepted. • All the columns specified in a join must be encrypted.
All	The use of “embedded SELECTs” is not supported for instructions containing encrypted columns.
UPDATE INSERT	Methods of the PreparedStatement.setXxxStream() type are not supported for encrypted columns.
ResultSet	ResultSet.updateXxx() methods are not supported for encrypted columns.

4.3 Impact of encryption on data size

The impact of encryption on data storage is described in the following table:

SQL data type	Impact of encryption on the size of stored data
INTEGER	None.
BINARY, VARBINARY, LONGVARBINARY, BLOB	None.
CHAR, VARCHAR, LONGVARCHAR, CLOB	Expansion of the string of characters by about 25%.

4.4 The safejdbc_catalog SQL database

SafeJDBC has its own “catalogue”, which contains:

- The columns to be encrypted for each table and database, with detailed information (rank, type, length etc.).
- Encryption information for each column.

This catalogue plays a “static” role, and is *not* accessed during the applications’ operation:

- The **safejdbc_catalog** database is accessed once and once only when the JVM is loaded.
- The SafeJDBC driver loads its contents into memory in a dedicated structure which is organized for very rapid access.

Note:

The **safejdbc_catalog** database contains **no confidential information**.

4.5 Integrity and backups of safejdbc_catalog

The **integrity** of safejdbc_catalog must be ensured in order to guarantee that encryption and decryption operations are consistent. We recommend that you make regular backups of the database.

The information contained in the **safejdbc_store** and **safejdbc_users** tables can be regenerated in the event that data is lost from the table.

But:

The information in the safejdbc_iv table cannot be reconstructed, as it contains random values generated when the database is initialized.

5 UTILIZATION & THE API

5.1 Creating the SAFEJDBC_CATALOG database

The catalogue for SafeJDBC's SQL tables must be created before the Driver is used.

The following two files contain the CREATE TABLE statements for the SAFEJDBC_CATALOG database:

Environment	Creation file
Windows NT/2000	c:\safelogic\safejdbc\sql\safejdbc_create_tables.sql
Unix/Linux	/home/safelogic/safejdbc/sql/safejdbc_create_tables.sql

Notes:

- The name of the database can be changed/chosen at will.
- The syntax used in the CREATE TABLE instructions is a restricted subset of SQL 92, which is supported by all modern SQL DBMSs.

5.2 SafeJdbcSetter – the SafeJDBC Driver configuration API

The SafeJDBC Driver must be configured from the Java application *before* the Driver is loaded.

The **com.safejdbc.api.SafeJdbcSetter** class is used to configure the secure Java Connection:

5.2.1 getInstance() – Creating an instance of SafeJdbcSetter

Method	SafeJdbcSetter.getInstance()
Description	Returns an instance of the SafeJdbcSetter class.
Arguments	None.
Returns	A new instance of the SafeJdbcSetter class.

Example:

```
SafeJdbcSetter sjSetter = SafeJdbcSetter.getInstance();
```

The following import declaration is required:

```
import com.safejdbc.api.SafeJdbcSetter;
```

5.2.2 setWrappedDriverName() – Configuring the encapsulated native driver

Method	SafeJdbcSetter.setWrappedDriverName()		
Description	Configures the Driver to be used for accessing SQL databases.		
Arguments	String	SDriver	Name of the class for loading the native JDBC driver.
Returns	Nothing.		

Example:

Configuring the MySQL driver:

```
// Set the Wrapped Driver
sjSetter.setWrappedDriverName("org.gjt.mm.mysql.Driver");
```

5.2.3 setSafeJdbcDbUrl() – Setting the URL for SAFEJDBC_CATALOG

Method	SafeJdbcSetter.setSafeJdbcDbUrl()		
Description	Setting the URL for the SAFEJDBC_CATALOG database.		
Arguments	String	Surl	Full URL for accessing the SAFEJDBC_CATALOG database.
Returns	Nothing.		

Example:

Setting the URL for the SafeJDBC database using MySQL:

```
// Set SafeJdbc's own catalog
sjSetter.setSafeJdbcDbUrl("jdbc:mysql://localhost:3306/"
    + "safejdbc_catalog?user=safelogic&password=my_password");
```

Method	SafeJdbcSetter.setSafeJdbcDbUrl()		
Description	Sets the URL for the SAFEJDBC_CATALOG database		
Arguments	String	sUrl	Full URL for accessing the SAFEJDBC_CATALOG database
	String	sUser	User name for accessing the database
	String	sPassword	Password for accessing the database
Returns	Nothing		

Example:

```
// Set SafeJdbc's own catalog
sjSetter.setSafeJdbcDbUrl("jdbc:mysql://localhost:3306/safejdbc_catalog",
    "safelogic",
    "safejdbc_catalog_secret_password");
```

5.2.4 Defining the Logfile

SafeJDBC can trace/log all SQL statements in an ASCII file:

Method	SafeJdbcSetter.setSafeJdbcLogfile()		
Description	Logfile configuration		
Arguments	String	sLogfile	Name of logfile. If the name does not contain a directory, the logfile will be created in System.getProperty("user.dir")
Returns	Nothing		

Example:

Logging SQL statements in the file safejdbc.log:

```
// Set SafeJdbc's logfile
sjSetter.setSafeJdbcLogfile("safejdbc.log");
```

5.2.5 Loading the SafeJDBC driver

The driver is loaded using the usual pair of instructions for loading a JDBC driver:

```
Class.forName("com.safejdbc.api.Driver").newInstance();
Connection connection = DriverManager.getConnection(sDbUrl);
```

Where:

- **SdbUrl** is the URL for the application database.

5.3 SafeJdbcSetter - API for configuring cryptographic elements

5.3.1 Generalities

The cryptographic elements to be configured are:

- The cryptography provider to be used.
- The symmetrical encryption **algorithm** to be used, from a preset list.
- The value of the **secret key** to be used for symmetrical encryption. This secret key will be used for all encryption and decryption operations. The value of the key must be kept secret and protected from the outside world.

SafeJDBC applies the algorithm with the associated key to the data to be encrypted or decrypted. The encryption is described as symmetrical, because the same key is used to encrypt and decrypt the data (or SQL rows).

5.3.2 Providers

SafeJDBC can define the encryption **Provider**, which is the bottom encryption layer used:

Provider	Comment	SafeJDBC constant (package <code>com.safejdbc.api</code>)
Sun JCE	see http://java.sun.com/products/jce/	<code>Provider.SUNJCE</code>
Cryptix	see http://www.cryptix.org/	<code>Provider.CRYPTIX</code> (default Provider)

Notes:

- Use of the Cryptix Provider does not require any additional configuration, and is recommended, as the source code is available and can be examined.
- Use of the Sun JCE Provider requires downloading the "JCE Unlimited Strength Jurisdiction Policy Files".
See <http://java.sun.com/products/jce/> for more information.

The algorithms include the mode and the padding scheme. Here is the list of algorithms available within SafeJDBC, together with their associated SafeJDBC constants:

Algorithm	Mode	Padding	SafeJDBC constant (package com.safejdbc.api)
Blowfish	CBC	None	Algo.BLOWFISH_CBC
		PKCS#5	Algo.BLOWFISH_CBC_PKCS5
	CFB	None	Algo.BLOWFISH_CFB (default algorithm)
CAST	ECB	None	Algo.CAST_128_ECB
	CBC	None	Algo.CAST_128_CBC
		PKCS#5	Algo.CAST_128_CBC_PKCS5
	CFB	None	Algo.CAST_128_CFB
IDEA ¹	CBC	None	Algo.IDEA_CBC
		PKCS#5	Algo.IDEA_CBC_PKCS5
	CFB	None	Algo.IDEA_CFB

Notes:

- The encryption key always consists of **128 bits**.
- The default algorithm is **Algo.BLOWFISH_CFB**.

5.3.3 setAlgorithm() – Setting the algorithm to use

Method	safeJdbcSetter.setAlgorithm()		
Description	Sets the symmetrical encryption algorithm to use.		
Arguments	String	sAlgorithm	Symmetrical encryption algorithm to use, including the mode and padding scheme.
Returns	Nothing.		

Note:

- It is not essential to call this method. The default combination of algorithm/mode/padding is Algo.BLOWFISH_CFB.

Example:

```
// Set SafeJdbc's algorithm to use
sjSetter.setAlgorithm(Algo.CAST_128_CBC_PKCS5);
```

¹ IDEA is patented by ASCOMM and requires a special licence from SafeLogic enabling its use.

5.3.4 setKey() – Setting the secret encryption key

The **setKey()** API enables the secret key to be loaded into memory in the JVM.

There are two ways of defining an encryption key in SafeJDBC:

1. The encryption key is retrieved via a Java **java.security.Key** object value from a program external to SafeJDBC. (For example, SafeLogic’s SafeAPI toolkit provides a means of generating and storing secret keys securely – see <http://www.safelogic.com/safeapi>.)
2. The encryption key is derived directly from a **passphrase** using an MD5 hashing function.

Method	SafeJdbcSetter.setKey()		
Description	Loads a symmetrical key into SafeJDBC in the Java Key format.		
Arguments	String	sUserId	The SafeJDBC user’s identifier.
	Key	key	128-bit symmetrical key in the Java Key format.
Returns	Nothing.		

Notes:

- **sUserId** is an arbitrary identifier. Its purpose is to “link” the triplet (user, algorithm, key) to secure subsequent uses. The triplet is stored by SafeJDBC on its first use. From the second use onwards, SafeJDBC verifies that the three elements (user, algorithm, key) correspond to the stored values. This principle makes it possible to avoid decryption using algorithms, keys or passphrases which do not match those used for the first encryption that was carried out.
- The value of **key** must be protected and kept secret to ensure the security of SafeJDBC.

Example:

```
// Set SafeJdbc's secret Key from an external program/store
Key key = getKeyFromSomeStore(key_reference);
sjSetter.setKey("safejdbc", key);
```

Method	SafeJdbcSetter.setKey()		
Description	Loads a symmetrical key derived from a passphrase into SafeJDBC in the Java Key format.		
Arguments	String	sUserId	The SafeJDBC user's identifier.
	Char []	caPassphrase	Secret passphrase used to generate the secret key in the Java Key format.
Returns	Nothing.		

Example:

```
// Set SafeJdbc's secret Key from a passphrase
char[] caPassphrase= {'s', 'a', 'f', 'e', '*', 'l', 'o', 'g', 'i', 'c', ':', '!'};
sjSetter.setKey("safejdbc", caPassphrase);
```

Note:

- The encryption key is regenerated dynamically from the passphrase each time.
- The value of **caPassphrase** must be protected and kept secret to ensure the security of SafeJDBC.

5.4 Calling SafeJDBC in pure JDBC Driver mode

5.4.1 Principle

SafeJDBC can be called similarly to any other driver, without programming SafeJdbcSetter parameters in Java. The parameters are contained in an ASCII file that must be in the CLASSPATH.

The default name for the parameter file is "**con_safejdbc.ini**". The SAFEJDBC_SETTER_INI environment variable (or System Property) can be used to define a different name when launching the JVM:

```
Java -DSAFEJDBC_SETTER_INI=my_safejdbc_ini_name.ini MyProgram
```

The SafeJDBC driver is then called normally, with the name of the driver being "**com.safejdbc.api.Driver**":

```
// Load SafeJDBC as a pure Driver & get a Cipher Connection to the database
String sDriver = "com.safejdbc.api.Driver";
Class.forName(sDriver).newInstance();
Connection connection = DriverManager.getConnection(sDbUrl, prop);
```

5.4.2 Configuring con_safejdbc.ini

Each "set" method of the SafeJdbcSetter class has one or more parameters to be defined in con_safejdbc.ini, in the format:

```
# comment line
PARAMETER_NAME = parameter_value
```

It is not necessary to define the parameters that have default values defined in the "set" methods of SafeJdbcSetter. Here is the list of parameters in con_safejdbc.ini:

Parameter name	Corresponding method & role in SafeJdbcSetter	Definition required
DRIVER	Name of native driver with wrapper. SetWrappedDriverName()	Yes
JDBC_URL	URL for the SAFEJDBC_CATALOG database. SetSafeJdbcDbUrl()	Yes
USER	SQL user for accessing SAFEJDBC_CATALOG. SetSafeJdbcDbUrl()	Yes
PASSWORD	SQL password for accessing SAFEJDBC_CATALOG. SetSafeJdbcDbUrl()	Yes
SAFEJDBC_USERID	SafeJDBC userid for the owner of the key or the passphrase. SetKey()	Yes
SAFEJDBC_KEY	Encryption key in hexadecimal format. SetKey()	Yes (one of the two)
SAFEJDBC_PASSPHRASE	Encryption passphrase. SetKey()	
SAFEJDBC_LOGFILE	Logfile. SetSafeJdbcLogFile()	No
SAFEJDBC_PROVIDER	Encryption Provider. SetProvider()	No
SAFEJDBC_ALGORITHM	Encryption algorithm. SetAlgorithm()	No

Notes:

- The file con_safejdbc.ini is located automatically by SafeJDBC, which searches for it in this order:
 - 1) In the directory corresponding to the environment variable `user.home`.
 - 2) In the directories defined in `CLASSPATH`.

If several files exist, SafeJDBC retrieves the contents of the first one accessed.

5.5 Verifying the secure state of a Connection

A Cipher Connection allows data to be encrypted and decrypted. It is an instance of the class **com.safejdbc.java.sql.CipherConnection**

The `instanceof` Java operator may be used to check that SafeJDBC succeeded in loading a “real” Cipher Connection:

```
import com.safejdbc.java.sql.CipherConnection;

// Etc...

if (connection instanceof CipherConnection)
{
    System.out.println("Good! This is a Cipher Connection.");
}
else
{
    System.out.println("Bad! This is *not* a Cipher Connection.");
}
```

5.6 How to get the wrapped «native» Connection

It is still possible to use the “clear” or “native” non-cipher Connection to directly access the application database. The **getWrappedConnection()** method of **CipherConnection** allows this operation.

```
// Try to get back the native/wrapped Connection
// from the Cipher Connection
if (connection instanceof CipherConnection)
{
    Connection conWrapped = null;

    // getWrapped will get the native/wrapped Connection
    // to application database
    conWrapped = ((CipherConnection) connection).getWrappedConnection();

    ...
}
```

Direct usage of wrapped Connection allows access to non-encrypted tables and columns to be speeded up. This should be the preferred means of accessing normal tables.

5.7 CipherConnectionFactory – Api for creating a "Connection"

It may be necessary to create a secure Connection without accessing the "Driver" class. This is the case with J2EE application servers.

CipherConnectionFactory is used to create a secure connection (type **CipherConnection**) without loading a driver, by using an encapsulated Connection that is native to the application database.

Since each CipherConnection encapsulates a native Connection, this process allows advanced mechanisms to be used to establish pools of secure Connections.

Note that it is much better and easier to manage connection pooling using the **DataSource** mechanism, such as in the Tomcat 5.0 implementation. The **SafeDataSourceFactory** class is designed for this purpose and is documented in chapter: **5.8 Using DataSources with J2EE Application Servers**

A CipherConnection Secure Connection is constructed out of two Connections:

1. One Connection to the application database.
2. A second Connection to the database of the SafeJDBC Catalog.

Prior to calling CipherConnectionFactory, some settings must be done using **SafeJdbcSetter** class or using an ini file:

1. Setting all parameters so that SafeJDBC can get a Connection to the SafeJDBC Catalog.
2. Set the SafeJDBC Userid & Key.

The following example uses the parameters defined in the file con_safejdbc.ini:

```
// This is the application database url
String sDbUrl = "jdbc:mysql://localhost:3306/sj_clients";

// These are the application database username & password
Properties propsApp = new Properties();
propsApp.put("user", "safelogic");
propsApp.put("password", "safelogic*$");

// Load the native Driver and get a normal Connection to the application
// database.
// Note: this could be done using the DataSource mechanism in a J2EE
// environment.
Class.forName("org.gjt.mm.mysql.Driver").newInstance();

// Get a Connection to the application database
Connection conApplication = DriverManager.getConnection(sDbUrl, propsApp);

// This is the SafeJdbc Catalog database url
String sSafeJdbcDbUrl = "jdbc:mysql://localhost:3306/safejdbc_catalog";

Properties propsSjdbc = new Properties();
propsSjdbc.put("user", "safelogic");
propsSjdbc.put("password", "safelogic*$");

// Get a Connection to the SafeJdbc Catalog database
Connection conSafeJdbcCatalog = DriverManager.getConnection(sSafeJdbcDbUrl,
propsSjdbc);

// Get the Cipher Connection for encryption of statements and result
Connection connection = CipherConnectionFactory.getInstance(conApplication,
conSafeJdbcCatalog);
```

5.8 Using DataSources with J2EE Application Servers

5.8.1 Generalities

Modern J2EE Application Server integrate advanced connection pooling techniques.

These advanced techniques are implemented through the standardized interfaces classes of the **javax.sql** package. Connections are accessed through the creation of **DataSource** objects and using the JNDI “lookup” mechanisms.

The `DataSource.getConnection()` method allows then to easily retrieve a Connection from the pool.

This is an example of how to get a Connection to the application database. It uses a **DataSource** identified by it's JNDI name «`jdbc/application` »:

```
// Get a Connection to application Database.
//
InitialContext initCtx = new InitialContext();

// Get a DataSource for connection to application database
DataSource ds
    = (DataSource) initCtx.lookup("java:/comp/env/jdbc/application");

// Get the Connection from the DataSource.
// Connection is extracted from the Connection Pool:
Connection connection = ds.getConnection() ;

// Etc...
```

5.8.2 The SafeDataSourceFactory JavaBean

SafeJDBC includes a JavaBean Factory for DataSources: **com.safejdbc.javax.sql.SafeDataSourceFactory**.

SafeDataSourceFactory allows the previous program to establish without any source code update a Cipher Connection to the application database.

SafeDataSourceFactory acts as a wrapper. It wraps transparently two DataSources:

- The DataSource to use to access the application database.
- The DataSource to use to access the SafeJDBC Catalog database.

This class must not be called directly by a user program.

This class will be called/created automatically by any EJB compatible modern application server or web server : Caucho Resin, JBoss, JOnAS, Orion, Sun Application Server, Tomcat, WebLogic, WebSphere, etc.

SafeDataSourceFactory use three parameters :

- `base_jndi_context`:
Base JNDI `Context` name for DataSources name lookup. (Defaults to `"java:/comp/env/"`)
- `ds_wrapped_lookup_name`:
Application DataSource lookup name in JNDI syntax naming.
- `ds_safejdbc_catalog_lookup_name`:
SafeJdbc Catalog DataSource lookup name in JNDI syntax naming.

To establish a CipherConnection Connection : you just need to connect the **SafeDataSourceFactory** with the « `env/jdbc/application` » DataSource.

Following is an example with Tomcat 5.0.

5.9 Real world example: Secure DataSources with Tomcat 5.0

In Tomcat 5.0 (and in all modern J2EE Server), DataSources allow use of connection pool.

To configure DataSources in Tomcat 5.0:

- Define a new DataSource in `server.xml`. The definition includes the connections pool parameters.
- The DataSource name is retrieved in the Java program through the JNDI Lookup mechanism. It uses the namespace `"java:/comp/env/jdbc/"`.
- The DataSource is created by instantiation of a "factory" class of name **`org.apache.commons.dbcp.BasicDataSourceFactory`**.

There are three steps to configure a Secure DataSource that manage a pool of CipherConnections:

5.9.1 Step 1: define the application database DataSource

This step allows to define the name of Resource which corresponds to DataSource of access to the application database.

This is an example to insert in `server.xml`:

```
<!--Step 1 : this is the normal connection pool to application database -->
<Resource name="jdbc/application" auth="Container"
          type="javax.sql.DataSource"/>

<ResourceParams name="jdbc/application_wrapped">
  <parameter>
    <name>factory</name>
    <value>org.apache.commons.dbcp.BasicDataSourceFactory</value>
  </parameter>

  <parameter>
    <name>maxActive</name>
    <value>100</value>
  </parameter>

  <parameter>
    <name>maxIdle</name>
    <value>30</value>
  </parameter>

  <parameter>
    <name>maxWait</name>
    <value>10000</value>
  </parameter>

  <!-- MySQL sj_clients username and password for dB connections -->
  <parameter>
    <name>username</name>
    <value>safelogic</value>
  </parameter>

  <parameter>
    <name>password</name>
    <value>safelogic*${</value>
  </parameter>

  <!-- Class name for the official MySQL Connector/J driver -->
  <parameter>
    <name>driverClassName</name>
    <value>com.mysql.jdbc.Driver</value>
  </parameter>

  <!-- The sj_clients example database -->
  <parameter>
    <name>url</name>
<value>jdbc:mysql://localhost:3306/sj_clients?autoReconnect=true</value>
  </parameter>
</ResourceParams>
```

5.9.2 Step 2 : Define the SafeJDBC Catalog DataSource

One then defines in `server.xml` DataSource of access to the database Catalogue SafeJDBC (`safejdbc_catalog`):

```
<!-- Step 2: Add the jdbc/safejdbc_catalog Resource
This is the connection pool to safejdbc catalog database -->

<Resource name="jdbc/safejdbc_catalog" auth="Container"
type="javax.sql.DataSource"/>

  <ResourceParams name="jdbc/safejdbc_catalog">
    <parameter>
      <name>factory</name>
      <value>org.apache.commons.dbcp.BasicDataSourceFactory</value>
    </parameter>

    <parameter>
      <name>maxActive</name>
      <value>100</value>
    </parameter>

    <parameter>
      <name>maxIdle</name>
      <value>30</value>
    </parameter>

    <parameter>
      <name>maxWait</name>
      <value>10000</value>
    </parameter>

    <!-- MySQL safejdbc_catalog dB username and password for dB -->
    <parameter>
      <name>username</name>
      <value>safelogic</value>
    </parameter>

    <parameter>
      <name>password</name>
      <value>safelogic*${</value>
    </parameter>

    <!-- Class name for the official MySQL Connector/J driver -->
    <parameter>
      <name>driverClassName</name>
      <value>com.mysql.jdbc.Driver</value>
    </parameter>

    <!-- The SafeJDBC Catalog database -->
    <parameter>
      <name>url</name>

<value>jdbc:mysql://localhost:3306/safejdbc_catalog?autoReconnect=true</val
ue>
    </parameter>
  </ResourceParams>
```

5.9.3 Step 3 : Define the Secure SafeDataSource

One defines to finish secure DataSource which wraps both previous DataSources: application database DataSource and SafeJDBC Catalog DataSource.

This DataSource will make it possible to create a DataSource which will be an instance of `com.safejdbc.javax.sql.SafeDataSource`, exactly according to the principle of secure/encrypted Connection which is an instance of `CipherConnection`:

```
<!--
Step 3: Add the new encrypted jdbc/application Resource.

This is the Cipher Connection pool.
Is will use the two previous connection pools defined by their
Resource/lookup Name:
    - 1) jdbc/application_wrapped
    - 2) jdbc/safejdbc_catalog
-->

<Resource name="jdbc/application" auth="Container"
type="javax.sql.DataSource"/>

    <ResourceParams name="jdbc/application">

        <!-- We are using a special SafeJDBC factory -->
        <parameter>
            <name>factory</name>
            <value>com.safejdbc.javax.sql.SafeDataSourceFactory</value>
        </parameter>

        <parameter>
            <name>ds_wrapped_lookup_name</name>
            <value>jdbc/application_wrapped</value>
        </parameter>

        <parameter>
            <name>ds_safejdbc_catalog_lookup_name</name>
            <value>jdbc/safejdbc_catalog</value>
        </parameter>

    </ResourceParams>
```

We may then complete the source code in **5.8.1 Generalities** to check that we extract from the pool a secure CipherConnection:

```
// Get a Connection to application Database.
//
InitialContext initCtx = new InitialContext();

// Get a DataSource for connection to application database
DataSource ds
    = (DataSource) initCtx.lookup("java:/comp/env/jdbc/application");

// Get the Connection from the DataSource.
// Connection is extracted from the Connection Pool:
Connection connection = ds.getConnection() ;

if (connection instanceof CipherConnection)
{
    System.out.println("OK! Cipher Connection extracted from the pool!");
}

// Etc...
```

Connection made safe thus obtained are managed by completely using the mechanism of "Connection Pooling" implemented by Tomcat 5.0.

5.10 ColumnsCipher - API for selecting the columns to encrypt/decrypt

5.10.1 Generalities

The API for selecting columns to encrypt/decrypt enables you to:

- Define a list of columns for SafeJDBC to encrypt (or decrypt).
- Set the output stream for displaying the execution of the class.
- Launch the execution of SQL UPDATE instructions corresponding to the list of selected columns, in order to encrypt or decrypt their values.

The **com.safejdbc.api.ColumnsCipher** class is used to set up the secure Java connection.

The SafeJDBC driver must be loaded in the calling class **before** the calls to **ColumnsCipher**.

5.10.2 getInstance () – Creating an instance of ColumnsCipher

Method	ColumnsCipher.getInstance()		
Description	Returns an instance of the ColumnsCipher class.		
Arguments	Connection	conCipher	Connection created by the SafeJDBC driver.
Returns	An instance of the ColumnsCipher class.		

Example:

```
ColumnsCipher columnsCipher = ColumnsCipher.getInstance(cipherConnection);
```

The following import declaration must be made:

```
import com.safejdbc.api.ColumnsCipher;
```

5.10.3 setOutputStream() – Configuring the output stream

This setting defines the stream to which output should be directed during execution:

Method	ColumnsCipher.setOutputStream()		
Description	Defines an OutputStream to which information produced during the execution of the class should be sent.		
Arguments	OutputStream	outputStream	OutputStream to be used to display messages during execution.
Returns	Nothing.		

Example:

Output to the console:

```
columnsCipher.getOutputStream(System.out);
```

5.10.4 setFileOutputStream() – Setting up an output stream to a file

Method	ColumnsCipher.setFileOutputStream()		
Description	Defines a file to be the output stream, i.e. to receive messages during the execution of the class.		
Arguments	String	sFile	Name of the output file to be used to receive messages.
Returns	Nothing		

Example:

```
columnsCipher.setFileOutputStream("c:\\temp\\cipher.output.txt");
```

5.10.5 addColumn() – Adding a column

Columns to be encrypted are added using the **addColumn()** method, which supports two different syntaxes to specify the table name:

Method	ColumnsCipher.addColumn()		
Description	Adds a column to be encrypted/decrypted to the list.		
Arguments	String	sColumn	Name of the column, using the syntax "table.column".
Returns	Nothing.		

Example:

```
ColumnsCipher.addColumn("clients.cli_lastname");
ColumnsCipher.addColumn("clients.cli_firstname");
```

Method	ColumnsCipher.addColumn()		
Description	Adds a column to be encrypted/decrypted to the list.		
Arguments	String	sTable	Name of the table.
Arguments	String	sColumn	Name of the column.
Returns	Nothing.		

Example:

```
ColumnsCipher.addColumn("clients", "cli_lastname");
ColumnsCipher.addColumn("clients", "cli_firstname");
```

5.10.6 addColumnsFromFile() – Adding columns from an “.ini” file

Method	ColumnsCipher.addColumnsFromFile()		
Description	Adds a list of columns from an “.ini” file.		
Arguments	String	sIniFile	Name of the file, without the path, containing the columns to add, using the syntax “table.column”.
Returns	Nothing.		

Notes:

The “.ini” file must contain one column name per line, using the syntax “**table.column**”. The # character is used at the beginning of a line to denote a comment.

Here is an example of a valid file:

```
# cipher_columns.ini
# Register all the cipher columns of a database

# keep secret clients personal info!
clients.cli_lastname
clients.cli_firstname
clients.cli_email
clients.cli_phone

# keep secret clients credit card info!

credit_cards.cb_type
credit_cards.cb_number
credit_cards.cb_expiration_date

# keep secret sales info!

sales.cli_lastname
sales.cli_firstname
sales.sa_order
sales.sa_cli_remarks

#end
```

5.10.7 defineJoinColumn() – Defining encrypted joins

defineJoinColumn() is used to specify joins between encrypted columns.

A join between two encrypted columns is defined in pairs by specifying:

- The **Join** column.
- The **Reference** column

With rules for avoiding errors (non existence, circular references, etc.)

1. The two columns must first have been defined using an **addColumn()** or **addColumnFromFile()** function.
2. A Join column cannot be defined as a Reference column as well.
3. A Reference column cannot be defined as a Join column as well.

defineJoinColumn() supports two different syntaxes to specify the names of the Join and Reference tables:

Syntax 1

Method	ColumnsCipher.defineJoinColumn()		
Description	Defining an encrypted Join column.		
Arguments	String	sJoinColumn	Name of the Join column, using the syntax "table.column".
	String	sRefColumn	Name of the Reference column, using the syntax "table.column".
Return	None.		

Example:

```
columnsCipher.defineJoinColumn("sales.cli_lastname",
                               "clients.cli_lastname");
columnsCipher.defineJoinColumn("products.cli_lastname",
                               "clients.cli_lastname");
```

Syntax 2

Method	ColumnsCipher.defineJoinColumn()		
Description	Defining an encrypted Join column.		
Arguments	String	sTableJoin	Name of the Join table.
	String	sJoinColumn	Name of the Join column.
	String	sTableRef	Name of the Reference table.
	String	sRefColumn	Name of the Reference column.
Return	None.		

Example:

```
columnsCipher.defineJoinColumn("sales", "cli_lastname",
                               "clients", "cli_lastname");
columnsCipher.defineJoinColumn("products", "cli_lastname",
                               "clients", "cli_lastname");
```

5.10.8 defineJoinColumnsFromFile() – Defining Join columns from an “ini” file

Method	ColumnsCipher.defineJoinColumnsFromFile()		
Description	Defining a list of Join columns from an “ini” file.		
Arguments	String	sIniFile	Name of the file, without the path, containing the Join and Reference columns in the following format: Table1.colonne_jointure = Table2.colonne_ref With the “=” sign acting as a separator.
Return	None.		

Notes:

- The “.ini” file must contain one column name per line, using the syntax “**table1.column_join = table2.column_ref**”.
- The # character is used at the beginning of a line to denote a comment.
- The Join/Reference columns can be specified in the same file as the encrypted columns.

Here is a valid example, added at the end of the previous ini file for defining encrypted columns:

```
# cipher_columns.ini
# Register all the cipher columns of a database
# and add Join and Reference columns

# keep secret clients personal info!
clients.cli_lastname
clients.cli_firstname
clients.cli_email
clients.cli_phone

# keep secret clients credit cards info!

credit_cards.cb_type
credit_cards.cb_number
credit_cards.cb_expiration_date

# keep secret sales info!

sales.cli_lastname
sales.cli_firstname
sales.sa_order
sales.sa_cli_remarks

# define Join and Reference columns:

sales.cli_lastname = clients.cli_lastname
sales.cli_firstname = clients.cli_firstname

#end
```

5.10.9 execute() – Running the encrypting or decrypting process

CipherColumns.execute() runs the encrypting or decrypting process, with two different stages:

- 1) **Definition:** The tables of the **safejdbc_catalog** meta_database are updated in accordance with the previously specified columns using the **addColumn()** and **defineJoinColumn()** functions.
- 2) **Switching of values:** If the application database contains values, the values to be encrypted (or decrypted) in the selected columns are updated by automatically generating and running SQL instructions of the type:

```
UPDATE TABLE SET COLUMN = NEW_VALUE WHERE COLUMN = OLD_VALUE
```

The **definition** phase must be performed **systematically** before SafeJDBC is used. For this reason, the **execute()** function must be run even if the application database is empty.

Method	ColumnsCipher.execute()		
Description	Running the encrypting/decrypting process of the specific columns using addColumn()		
Arguments	int	nOpMode	Procedure. Encrypting or decrypting: <ul style="list-style-type: none"> • ColumnsCipher.ENCRYPT_MODE • ColumnsCipher.DECRYPT_MODE
Return	None.		

Notes

- It is recommended to save the application database *and* **safejdbc_catalog** before running each **execute()**.
- The operations are always reversible, i.e. the **execute(ColumnsCipher.ENCRYPT_MODE)** and **execute(ColumnsCipher.DECRYPT_MODE)** instructions can be run alternately.
- The same column encrypting or decrypting mode cannot be specified twice in succession.
- SafeJDBC includes safety features which block an incorrect attempt to perform a “double encryption” or “double decryption”. An error message is then generated on the console.

Example:

```
columnsCipher.execute(ColumnsCipher.ENCRYPT_MODE);
```

5.11 ColumnsCipherMain batch file

The Java program **com.safejdbc.bat.ColumnsCipherMain** contains an implementation of the **ColumnsCipherMain** API with parameters.

The program is self-documenting and provides the following help when it is called with no parameters:

```
c:\Test>java com.safejdbc.bat.ColumnsCipherMain
Usage:
java com.safejdbc.bat.ColumnsCipherMain <parms>

parms: <mode> <con_application.ini> <con_safejdbc.ini> <cipher_columns.ini>
       <passphrase>
with:
- mode           : (e)ncrypt | (d)ecrypt
- con_application.ini: ini file with JDBC connection parms
                   to application database.
- con_safejdbc.ini  : ini file with JDBC connection parms
                   to SafeJDBC database.
- cipher_columns.ini : ini file containing columns to encrypt/decrypt.
- passphrase       : the passphrase for crypto operations.
notes:
- Pass raw ini file names, without directory name.
- The ini files must be located in the CLASSPATH.

c:\Test>
```

Here are some examples of SQL configuration “.ini” files:

```
# con_application.ini
#
# ini file for com.safejdbc.bat.ColumnsCipherMain batch.
# contains connections parameters to application database.

# JDBC driver Class
DRIVER=org.gjt.mm.mysql.Driver

# DB URI root
DB_URI_ROOT=jdbc:mysql://localhost:3306/sj_clients?

# Login & password to use for database connection
USER=safelogic
PASSWORD=safelogic*$

#end
```

```
# con_safejdbc.ini
#
# ini file for com.safejdbc.bat.ColumnsCipherMain batch
# contains connections parameters to the SafeJDBC catalog

# JDBC driver Class
DRIVER=org.gjt.mm.mysql.Driver

# URL to SafeJDBC database
DB_URI_ROOT=jdbc:mysql://localhost:3306/safejdbc_catalog?

# SafeJDBC database user & passwords
USER=safelogic
PASSWORD=safelogic*$

#end
```

5.12 How to define encrypted columns in a VIEW

SafeJDBC v2.00 allows encrypted VIEW fields to be defined using **addColumnFromFile()** and **defineJoinColumnFromFile()**.

The steps are:

1. Create the VIEW using **CREATE VIEW** (without *any* **WHERE** clause on the encrypted field) before doing anything else.
2. Add each encrypted field in the VIEW to the .ini file.
3. Add each encrypted field in the VIEW as if it were a JOIN column.
4. **ColumnsCipher.execute()**
5. Only If you want to add a simple WHERE clause on the encrypted field:
 - DROP the VIEW.
 - Recreate the VIEW with the WHERE CLAUSE using SafeJDBC, namely using a Java Program that will call/use SafeJDBC as its Driver.

Example:

We want to create the following (encrypted) VIEW with an encrypted WHERE clause:

```
create view myview as select cli_lastname, cli_firstname
    from clients where cli_lastname = 'SMITH';
```

Step 1

We stop all running instances of SafeJDBC (J2EE programs & server, etc).

We launch an interactive SQL environment and enter the command:

```
/* Note that there is no WHERE clause */
create view myview as select cli_lastname, cli_firstname
from clients;
```

Step 2 & Step 3

We add the encrypted VIEW fields to the .ini file

```
# cipher_columns.ini
# Register all the cipher columns of a database
# and add Join and Reference columns
# add VIEW myview on CLIENTS table

# keep secret clients personal info!
clients.cli_lastname
clients.cli_firstname
clients.cli_email
clients.cli_phone

# keep secret myview!
myview.cli_lastname
myview.cli_firstname

# keep secret clients credit cards info!

credit_cards.cb_type
credit_cards.cb_number
credit_cards.cb_expiration_date

# keep secret sales info!

sales.cli_lastname
sales.cli_firstname
sales.sa_order
sales.sa_cli_remarks

# define Join and Reference columns:

sales.cli_lastname = clients.cli_lastname
sales.cli_firstname = clients.cli_firstname

#define the View myview columns Reference columns
myview.cli_lastname = clients.cli_lastname
myview.cli_firstname = clients.cli_firstname

#end
```

Step 4

```
run ColumnsCipher.execute()
```

or

```
java com.safejdbc.bat.ColumnsCipherMain e
```

Step 5 (only if the view has a encrypted field in the WHERE clause)

In interactive SQL:

```
DROP VIEW myview;
```

Then recreate the VIEW with the WHERE clause using SafeJDBC as the JDBC Driver:

```
// Create a Secure Connection
...
if (connection instanceof CipherConnection)
{
    System.out.println("Good! This is a Cipher Connection.");
}
else
{
    System.out.println("Bad! This is *not* a Cipher Connection.");
    return;
}

String sStatement =
    "create view myview as select cli_lastname, cli_firstname "
    + "from clients where cli_lastname = 'Smith'";
Statement stat = connection.createStatement();
stat.execute(sStatement);

// End
```

6 SUPPORT

If you have any technical questions, please contact:

SafeLogic
27/29, rue Raffet
75016 Paris

Tel.: (33) (0)1 45 72 25 15

Fax: (33) (0)1 45 72 14 06

Web: <http://www.safelogic.com>
