MySQL Connector/J 5.1 Developer Guide

Abstract

This manual describes how to install, configure, and develop database applications using MySQL Connector/J 5.1, a JDBC driver for communicating with MySQL servers. It also contains information on earlier versions of Connector/J.

> **Support EOL for MySQL Connector/J 5.1**
>
> Per Oracle's Lifetime Support policy, as of Feb 9th, 2021, MySQL Connector/J 5.1 series is covered under Oracle Sustaining Support. Users are encouraged to upgrade to MySQL Connector/J 8.0 series.

For notes detailing the changes in each release of Connector/J 5.1, see MySQL Connector/J 5.1 Release Notes.

For legal information, see the Legal Notices.

For help with using MySQL, please visit the MySQL Forums, where you can discuss your issues with other MySQL users.

Document generated on: 2021-08-05 (revision: 70529)

Table of Contents

Preface and Legal Notices

This manual describes how to install, configure, and develop database applications using MySQL Connector/J 5.1, the JDBC driver for communicating with MySQL servers. It also contains information on earlier versions of Connector/J.

Legal Notices

unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

This documentation is NOT distributed under a GPL license. Use of this documentation is subject to the following terms:

You may create a printed copy of this documentation solely for your own personal use. Conversion to other formats is allowed as long as the actual content is not altered or edited in any way. You shall not publish or distribute this documentation in any form or on any media, except if you distribute the documentation in a manner similar to how Oracle disseminates it (that is, electronically for download on a Web site with the software) or on a CD-ROM or similar medium, provided however that the documentation is disseminated together with the software on the same medium. Any other use, such as any dissemination of printed copies or use of this documentation, in whole or in part, in another publication, requires the prior written consent from an authorized representative of Oracle. Oracle and/or its affiliates reserve any and all rights to this documentation not expressly granted above.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at
https://www.oracle.com/corporate/accessibility/.

Access to Oracle Support for Accessibility

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit
https://www.oracle.com/corporate/accessibility/learning-support.html#support-tab.

Chapter 1 Overview of MySQL Connector/J

MySQL provides connectivity for client applications developed in the Java programming language with MySQL Connector/J, a driver that implements the Java Database Connectivity (JDBC) API.

MySQL Connector/J is a JDBC Type 4 driver. Different versions are available that are compatible with the JDBC 3.0 and JDBC 4.x specifications (see Chapter 2, *Connector/J Versions, and the MySQL and Java Versions They Require*). The Type 4 designation means that the driver is a pure Java implementation of the MySQL protocol and does not rely on the MySQL client libraries.

For large-scale programs that use common design patterns of data access, consider using one of the popular persistence frameworks such as Hibernate, Spring's JDBC templates or MyBatis SQL Maps to reduce the amount of JDBC code for you to debug, tune, secure, and maintain.

Key Topics

- For installation instructions for Connector/J, see Chapter 3, *Connector/J Installation*.
- For help with connection strings, connection options, and setting up your connection through JDBC, see Section 5.3, “Configuration Properties for Connector/J”.
- For information on connection pooling, see Chapter 7, *Connection Pooling with Connector/J*.
- For information on multi-host connections, see Chapter 8, *Multi-Host Connections*.

Chapter 2 Connector/J Versions, and the MySQL and Java Versions They Require

Table of Contents

Two versions of MySQL Connector/J are available:

- Connector/J 5.1 is a Type 4 pure Java JDBC driver, which conforms to the JDBC 3.0, 4.0, 4.1, and 4.2 specifications. It provides compatibility with all the functionality of MySQL, including 5.6, 5.7 and 8.0. Connector/J 5.1 provides ease of development features, including auto-registration with the Driver Manager, standardized validity checks, categorized SQLExceptions, support for large update counts, support for local and offset date-time variants from the `java.time` package, support for JDBC-4.x XML processing, support for per connection client information, and support for the `NCHAR`, `NVARCHAR` and `NCLOB` data types.
- Connector/J 8.0 is a Type 4 pure Java JDBC 4.2 driver for the Java 8 platform. It provides compatibility with all the functionality of MySQL 5.6, 5.7, and 8.0. See MySQL Connector/J 8.0 Developer Guide for details.

> **Note**
>
> MySQL Connector/J 8.0 is highly recommended for use with MySQL Server 8.0, 5.7, and 5.6. Please upgrade to MySQL Connector/J 8.0.

The following table summarizes the Connector/J versions available, along with the compatibility information for different versions of JDBC, MySQL Server, and Java, as well as the support status for each of the Connector/J versions:

Table 2.1 Summary of Connector/J Versions

Connector/J version	JDBC version	MySQL Server version	JRE Required	JDK Required for Compilation	Status
5.1	3.0, 4.0, 4.1, 4.2	5.6[1], 5.7[1], 8.0[1]	JRE 5 or higher[1]	JDK 5.0 AND JDK 8.0 or higher[2, 3]	General availability
8.0	4.2	5.6, 5.7, 8.0	JRE 8 or higher	JDK 8.0 or higher[2]	General availability. Recommended version.

> **Notes**
>
> - [1] JRE 8 or higher is required for Connector/J 5.1 to connect to MySQL 5.6, 5.7, and 8.0 with SSL/TLS when using some cipher suites.
> - [2] A customized JSSE provider might be required to use some later TLS versions and cipher suites when connecting to MySQL servers. For example, because Oracle's Java 8 is shipped with a JSSE implementation that only supports

TLSv1.2 and lower, you need a customized JSSE implementation to use TLSv1.3 on Oracle's Java 8 platform.

- [3] If you are building Connector/J 5.1 from source code using the source distribution, you must have both JDK 8.0 or higher *AND* JDK 5.0 installed. It is also good to have JRE 6 for compiling Connector/J 5.1. See Section 3.4, “Installing from Source” for details.

This guide also covers earlier versions of Connector/J, with specific notes given where a setting applies to a specific version.

2.1 Connector/J Release Notes and Change History

For details of new features and bug fixes in each Connector/J release, see the MySQL Connector/J 5.1 Release Notes.

2.2 Java Versions Supported

See Chapter 2, *Connector/J Versions, and the MySQL and Java Versions They Require*.

Chapter 3 Connector/J Installation

Table of Contents

MySQL Connector/J is distributed as a `.zip` or `.tar.gz` archive, available for download from the Connector/J Download page. The archive contains the sources and the JAR archive named `mysql-connector-java-version-bin.jar`.

You can install the Connector/J package using either the binary or source distribution. The binary distribution provides the easiest method for installation; the source distribution lets you customize your installation further. With either solution, you manually add the Connector/J location to your Java `CLASSPATH`.

If you are upgrading from a previous version, read the upgrade information in Section 3.3, “Upgrading from an Older Version” before continuing.

Connector/J is also available as part of the Maven project. For more information and to download the Connector/J JAR files, see the Maven repository.

> **Important**
>
> You also need to install the Simple Logging Facade API to use the logging capabilities provided by the default implementation of `org.slf4j.Logger.Slf4JLogger` by Connector/J. That and other third-party libraries are required for building Connector/J from source (see the section for more information on the required libraries).

3.1 Installing Connector/J from a Binary Distribution

For the easiest method of installation, use the binary distribution of the Connector/J package. Extract the JAR archive from the tar/gzip or zip archive to a suitable location, then optionally make the information about the JAR archive available by changing your `CLASSPATH` (see Section 3.2, “Installing the Driver and Configuring the `CLASSPATH`”).

Use the appropriate graphical or command-line utility to extract the distribution (for example, WinZip for the .zip archive, and `tar` for the .tar.gz archive). Because there are potentially long file names in the distribution, we use the GNU tar archive format. Use GNU tar (or an application that understands the GNU tar archive format) to unpack the .tar.gz variant of the distribution.

3.2 Installing the Driver and Configuring the `CLASSPATH`

Once you have extracted the distribution archive, you can install the driver by placing `mysql-connector-java-version-bin.jar` in your classpath, either by adding the full path to it to your `CLASSPATH` environment variable, or by directly specifying it with the command line switch `-cp` when starting the JVM.

To use the driver with the JDBC `DriverManager`, use `com.mysql.jdbc.Driver` as the class that implements `java.sql.Driver`.

You can set the `CLASSPATH` environment variable under Unix, Linux, or macOS either locally for a user within their `.profile`, `.login` or other login file. You can also set it globally by editing the global `/etc/profile` file.

For example, add the Connector/J driver to your `CLASSPATH` using one of the following forms, depending on your command shell:

```
# Bourne-compatible shell (sh, ksh, bash, zsh):
shell> export CLASSPATH=/path/mysql-connector-java-ver-bin.jar:$CLASSPATH

# C shell (csh, tcsh):
shell> setenv CLASSPATH /path/mysql-connector-java-ver-bin.jar:$CLASSPATH
```

For Windows platforms, you set the environment variable through the System Control Panel.

To use MySQL Connector/J with an application server such as GlassFish, Tomcat, or JBoss, read your vendor's documentation for more information on how to configure third-party class libraries, as most application servers ignore the `CLASSPATH` environment variable. For configuration examples for some J2EE application servers, see Chapter 7, *Connection Pooling with Connector/J*, Section 8.2, “Configuring Load Balancing with Connector/J”, and Section 8.4, “Advanced Load-balancing and Failover Configuration”. However, the authoritative source for JDBC connection pool configuration information for your particular application server is the documentation for that application server.

If you are developing servlets or JSPs, and your application server is J2EE-compliant, you can put the driver's `.jar` file in the `WEB-INF/lib` subdirectory of your webapp, as this is a standard location for third party class libraries in J2EE web applications.

You can also use the `MysqlDataSource` or `MysqlConnectionPoolDataSource` classes in the `com.mysql.jdbc.jdbc2.optional` package, if your J2EE application server supports or requires them. Starting with Connector/J 5.0.0, the `javax.sql.XADataSource` interface is implemented using the `com.mysql.jdbc.jdbc2.optional.MysqlXADataSource` class, which supports XA distributed transactions when used in combination with MySQL server version 5.0 and later.

The various `MysqlDataSource` classes support the following parameters (through standard set mutators):

- `user`
- `password`
- `serverName` (see the previous section about failover hosts)
- `databaseName`
- `port`

3.3 Upgrading from an Older Version

This section has information for users who are upgrading from one version of Connector/J to another, or to a new version of the MySQL server that supports a more recent level of JDBC. A newer version of Connector/J might include changes to support new features, improve existing functionality, or comply with new standards.

3.3.1 Upgrading to MySQL Connector/J 5.1.x

- In Connector/J 5.0.x and earlier, the alias for a table in a `SELECT` statement is returned when accessing the result set metadata using `ResultSetMetaData.getColumnName()`. This behavior however is not JDBC compliant, and in Connector/J 5.1, this behavior has been changed so that the original table name, rather than the alias, is returned.

  The JDBC-compliant behavior is designed to let API users reconstruct the DML statement based on the metadata within `ResultSet` and `ResultSetMetaData`.

  You can get the alias for a column in a result set by calling `ResultSetMetaData.getColumnLabel()`. To use the old noncompliant behavior with `ResultSetMetaData.getColumnName()`, use the `useOldAliasMetadataBehavior` option and set the value to `true`.

  In Connector/J 5.0.x, the default value of `useOldAliasMetadataBehavior` was `true`, but in Connector/J 5.1 this was changed to a default value of `false`.

3.3.2 JDBC-Specific Issues When Upgrading to MySQL Server 4.1 or Newer

- *Using the UTF-8 Character Encoding* - Prior to MySQL server version 4.1, the UTF-8 character encoding was not supported by the server, however the JDBC driver could use it, allowing storage of multiple character sets in `latin1` tables on the server.

  Starting with MySQL-4.1, this functionality is deprecated. If you have applications that rely on this functionality, and can not upgrade them to use the official Unicode character support in MySQL server version 4.1 or newer, add the following property to your connection URL:

  ```
  useOldUTF8Behavior=true
  ```

- *Server-side Prepared Statements* - Connector/J 3.1 will automatically detect and use server-side prepared statements when they are available (MySQL server version 4.1.0 and newer). If your application encounters issues with server-side prepared statements, you can revert to the older client-side emulated prepared statement code that is still presently used for MySQL servers older than 4.1.0 with the following connection property:

  ```
  useServerPrepStmts=false
  ```

3.3.3 Upgrading from MySQL Connector/J 3.0 to 3.1

Connector/J 3.1 is designed to be backward-compatible with Connector/J 3.0 as much as possible. Major changes are isolated to new functionality exposed in MySQL-4.1 and newer, which includes Unicode character sets, server-side prepared statements, `SQLState` codes returned in error messages by the server and various performance enhancements that can be enabled or disabled using configuration properties.

- **Unicode Character Sets**: See the next section, as well as Character Sets, Collations, Unicode, for information on this MySQL feature. If you have something misconfigured, it will usually show up as an error with a message similar to `Illegal mix of collations`.

- **Server-side Prepared Statements**: Connector/J 3.1 will automatically detect and use server-side prepared statements when they are available (MySQL server version 4.1.0 and newer).

  Starting with version 3.1.7, the driver scans SQL you are preparing using all variants of `Connection.prepareStatement()` to determine if it is a supported type of statement to prepare on the server side, and if it is not supported by the server, it instead prepares it as a client-side emulated

prepared statement. You can disable this feature by passing `emulateUnsupportedPstmts=false` in your JDBC URL.

If your application encounters issues with server-side prepared statements, you can revert to the older client-side emulated prepared statement code that is still presently used for MySQL servers older than 4.1.0 with the connection property `useServerPrepStmts=false`.

- **Datetimes** with all-zero components (`0000-00-00 ...`): These values cannot be represented reliably in Java. Connector/J 3.0.x always converted them to `NULL` when being read from a ResultSet.

  Connector/J 3.1 throws an exception by default when these values are encountered, as this is the most correct behavior according to the JDBC and SQL standards. This behavior can be modified using the `zeroDateTimeBehavior` configuration property. The permissible values are:

  - `exception` (the default), which throws an SQLException with an SQLState of `S1009`.
  - `convertToNull`, which returns `NULL` instead of the date.
  - `round`, which rounds the date to the nearest closest value which is `0001-01-01`.

  Starting with Connector/J 3.1.7, `ResultSet.getString()` can be decoupled from this behavior using `noDatetimeStringSync=true` (the default value is `false`) so that you can retrieve the unaltered all-zero value as a String. Note that this also precludes using any time zone conversions, therefore the driver will not allow you to enable `noDatetimeStringSync` and `useTimezone` at the same time.

- **New SQLState Codes**: Connector/J 3.1 uses SQL:1999 SQLState codes returned by the MySQL server (if supported), which are different from the legacy X/Open state codes that Connector/J 3.0 uses. If connected to a MySQL server older than MySQL-4.1.0 (the oldest version to return SQLStates as part of the error code), the driver will use a built-in mapping. You can revert to the old mapping by using the configuration property `useSqlStateCodes=false`.

- **`ResultSet.getString()`**: Calling `ResultSet.getString()` on a `BLOB` column will now return the address of the `byte[]` array that represents it, instead of a `String` representation of the `BLOB`. `BLOB` values have no character set, so they cannot be converted to `java.lang.String`s without data loss or corruption.

  To store strings in MySQL with LOB behavior, use one of the `TEXT` types, which the driver will treat as a `java.sql.Clob`.

- **Debug builds**: Starting with Connector/J 3.1.8 a debug build of the driver in a file named `mysql-connector-java-`*`version`*`-bin-g.jar` is shipped alongside the normal binary jar file that is named `mysql-connector-java-`*`version`*`-bin.jar`.

  Starting with Connector/J 3.1.9, we do not ship the `.class` files unbundled, they are only available in the JAR archives that ship with the driver.

  Do not use the debug build of the driver unless instructed to do so when reporting a problem or bug, as it is not designed to be run in production environments, and will have adverse performance impact when used. The debug binary also depends on the Aspect/J runtime library, which is located in the `src/lib/aspectjrt.jar` file that comes with the Connector/J distribution.

3.4 Installing from Source

> **Caution**
>
> To just get MySQL Connector/J up and running on your system, install Connector/J using a standard binary release distribution. Instructions in this section is only for users who, for various reasons, want to compile Connector/J from source.

The requirements and steps for installing from source Connector/J 5.1.37 or later, 5.1.34 to 5.1.36, and 5.1.33 or earlier are different; check the section below that is relevant for the version you want.

Installing Connector/J 5.1.37 or later from source. To install MySQL Connector/J from its source tree on GitHub, you need to have the following software on your system:

- A Git client, to check out the sources from our GitHub repository (available from http://git-scm.com/downloads).
- Apache Ant version 1.8.2 or newer (available from http://ant.apache.org/).
- JDK 1.8.x *AND* JDK 1.5.x.
- JRE 1.6.x (optional)
- The following third-party libraries:
  - JUnit 4.12 (`junit-4.12.jar`, available from https://github.com/junit-team/junit/wiki/Download-and-Install).
  - C3P0 0.9.1 or newer (both `c3p0-0.9.1.x.jar` and `c3p0-0.9.1.x.src.zip`, available from https://sourceforge.net/projects/c3p0/).
  - JBoss common JDBC wrapper 3.2.3 or newer (`jboss-common-jdbc-wrapper-3.2.3.jar`, available from, for example, the Maven Central Repository at http://central.maven.org/maven2/jboss/jboss-common-jdbc-wrapper/).
  - Simple Logging Facade API 1.6.1 or newer (`slf4j-api-1.6.1.jar`, available from https://www.slf4j.org/download.html).
  - The required `.jar` files from the Hibernate ORM 4.1 or 4.2 Final release bundle, which is available at http://sourceforge.net/projects/hibernate/files/hibernate4/.

To check out and compile MySQL Connector/J, follow these steps:

1. Check out the code from the source code repository for MySQL Connector/J located on GitHub at https://github.com/mysql/mysql-connector-j; for the latest release of the Connector/J 5.1 series, use the following command:

   ```
   shell> git clone https://github.com/mysql/mysql-connector-j.git
   ```

   To check out a release other than the latest one, use the `--branch` option to specify the revision tag for it:

   ```
   shell> git clone --branch 5.1 https://github.com/mysql/mysql-connector-j.git
   ```

   Under the current directory, the commands create a `mysql-connector-j` subdirectory , which contains the code you want.

2. Make sure that you have both JDK 1.8.x *AND* JDK 1.5.x installed. You need both JDKs because besides supporting JDBC from 4.0 to 4.2, Connector/J 5.1 also supports JDBC 3.0, which is an older version and requires the older JDK 1.5.x.

3. Consider also having JRE 1.6.x installed. This is optional: if JRE 1.6.x is not available or not supplied to Ant with the property `com.mysql.jdbc.java6.rtjar`, the Java 8 bootstrap classes will be used. A warning will be returned, saying that the bootstrap class path was not set with the option to compile sources written for Java 6.

4. Place the required `junit.jar` file in a separate directory—for example, `/home/username/ant-extralibs`.

5. In the same directory for extra libraries described in the last step, create a directory named `hibernate4`, and put under it all the `.jar` files you can find under the `/lib/required/` folder in the Hibernate ORM 4 Final release bundle.

6. Change your current working directory to the `mysql-connector-j` directory created in step 1 above.

7. In the directory, create a file named `build.properties` to indicate to Ant the locations of the root directories for your JDK 1.8.x and JDK 1.5.x installations, the location of the `rt.jar` of your JRE 1.6.x (optional), and the location of the extra libraries. The file should contain the following property settings, with the "*`path_to_*`*" parts replaced by the appropriate filepaths:

   ```
   com.mysql.jdbc.jdk8=path_to_jdk_1.8
   com.mysql.jdbc.jdk5=path_to_jdk_1.5
   com.mysql.jdbc.java6.rtjar=path_to_rt.jar_under_jre_1.6/rt.jar
   com.mysql.jdbc.extra.libs=path_to_folder_for_extra_libraries
   ```

   Alternatively, you can set the values of those properties through the Ant `-D` options.

8. Issue the following command to compile the driver and create a `.jar` file for Connector/J:

   ```
   shell> ant dist
   ```

   This creates a `build` directory in the current directory, where all the build output goes. A directory is created under the `build` directory, whose name includes the version number of the release you are building. That directory contains the sources, the compiled `.class` files, and a `.jar` file for deployment. For more information and other possible targets, including those that create a fully packaged distribution, issue the following command:

   ```
   shell> ant -projecthelp
   ```

9. Install the newly created `.jar` file for the JDBC driver as you would install a binary `.jar` file you download from MySQL by following the instructions given in Section 3.2, "Installing the Driver and Configuring the CLASSPATH".

Note that a package containing both the binary and source code for Connector/J 5.1 can also be downloaded from the Connector/J Download page.

Installing Connector/J 5.1.34 to 5.1.36 from source. To install MySQL Connector/J 5.1.34 to 5.1.36 from the Connector/J source tree on GitHub, make sure that you have the following software on your system:

- A Git client, to check out the sources from our GitHub repository (available from http://git-scm.com/downloads).
- Apache Ant version 1.8.2 or newer (available from http://ant.apache.org/).
- JDK 1.6.x *AND* JDK 1.5.x.

- JUnit libraries (available from https://github.com/junit-team/junit/wiki/Download-and-Install).
- The required `.jar` files from the Hibernate ORM 4.1 or 4.2 Final release bundle, which is available at http://sourceforge.net/projects/hibernate/files/hibernate4/.

To check out and compile MySQL Connector/J, follow these steps:

1. Check out the code from the source code repository for MySQL Connector/J located on GitHub at https://github.com/mysql/mysql-connector-j, using the `--branch` option to specify the revision tag for release 5.1.*xx*:

   ```
   shell> git clone --branch 5.1.xx https://github.com/mysql/mysql-connector-j.git
   ```

   Under the current directory, the commands create a `mysql-connector-j` subdirectory , which contains the code you want.

2. Make sure that you have both JDK 1.6.x *AND* JDK 1.5.x installed. You need both JDKs because Connector/J 5.1 supports both JDBC 3.0 (which has existed prior to JDK 1.6.x) and JDBC 4.0.

3. Place the required `junit.jar` file in a separate directory—for example, `/home/username/ant-extralibs`.

4. In the same directory for extra libraries described in the last step, create a directory named `hibernate4`, and put under it all the `.jar` files you can find under the `/lib/required/` folder in the Hibernate ORM 4 Final release bundle.

5. Change your current working directory to the `mysql-connector-j` directory created in step 1 above.

6. In the directory, create a file named `build.properties` to indicate to Ant the locations of the root directories for your JDK 1.5.x and JDK 1.6.x installations, as well as the location of the extra libraries. The file should contain the following property settings, with the "*`path_to_*`*" parts replaced by the appropriate filepaths:

   ```
   com.mysql.jdbc.jdk5=path_to_jdk_1.5
   com.mysql.jdbc.jdk6=path_to_jdk_1.6
   com.mysql.jdbc.extra.libs=path_to_folder_for_extra_libraries
   ```

   Alternatively, you can set the values of those properties through the Ant `-D` options.

7. Issue the following command to compile the driver and create a `.jar` file for Connector/J:

   ```
   shell> ant dist
   ```

   This creates a `build` directory in the current directory, where all the build output goes. A directory is created under the `build` directory, whose name includes the version number of the release you are building. That directory contains the sources, the compiled `.class` files, and a `.jar` file for deployment. For more information and other possible targets, including those that create a fully packaged distribution, issue the following command:

   ```
   shell> ant -projecthelp
   ```

8. Install the newly created `.jar` file for the JDBC driver as you would install a binary `.jar` file you download from MySQL by following the instructions given in Section 3.2, "Installing the Driver and Configuring the CLASSPATH".

Installing Connector/J 5.1.33 or earlier from the source tree. To install MySQL Connector/J 5.1.33 or earlier from the Connector/J source tree on GitHub, make sure that you have the following software on your system:

- A Git client, to check out the source code from our GitHub repository (available from http://git-scm.com/downloads).
- Apache Ant version 1.7 or newer (available from http://ant.apache.org/).
- JDK 1.6.x *AND* JDK 1.5.x. Refer to Section 2.2, "Java Versions Supported" for the version of Java you need to build or run any Connector/J release.
- The Ant Contrib (version 1.03b is available from http://sourceforge.net/projects/ant-contrib/files/ant-contrib/1.0b3/) and JUnit (available from https://github.com/junit-team/junit/wiki/Download-and-Install) libraries.
- The required `.jar` files from the Hibernate ORM 4.1 or 4.2 Final release bundle, which is available at http://sourceforge.net/projects/hibernate/files/hibernate4/.

To check out and compile a specific branch of MySQL Connector/J, follow these steps:

1. Check out the code from the source code repository for MySQL Connector/J located on GitHub at https://github.com/mysql/mysql-connector-j, using the `--branch` option to specify the revision tag for release 5.1.*xx*:

   ```
   shell> git clone --branch 5.1.xx https://github.com/mysql/mysql-connector-j.git
   ```

   Under the current directory, the commands create a `mysql-connector-j` subdirectory , which contains the code you want.

2. To build Connector/J 5.1, make sure that you have both JDK 1.6.x *AND* JDK 1.5.x installed. You need both JDKs because Connector/J 5.1 supports both JDBC 3.0 (which has existed prior to JDK 1.6.x) and JDBC 4.0. Set your `JAVA_HOME` environment variable to the path to the JDK 1.5.x installation.

3. Place the required `ant-contrib.jar` file (in exactly that name, without the version number in it; rename the `jar` file if needed) and `junit.jar` file in a separate directory—for example, `/home/username/ant-extralibs`.

4. In the same directory for extra libraries described in the last step, create a directory named `hibernate4`, and put under it all the `.jar` files you can find under the `/lib/required/` folder in the Hibernate ORM 4 Final release bundle.

5. Change your current working directory to the `mysql-connector-j` directory created in step 1 above.

6. In the directory, create a file named `build.properties` to indicate to Ant the locations of the Javac and `rt.jar` of your JDK 1.6.x, as well as the location of the extra libraries. The file should contain the following property settings, with the "*`path_to_*`*" parts replaced by the appropriate filepaths:

   ```
   com.mysql.jdbc.java6.javac=path_to_javac_1.6/javac
   com.mysql.jdbc.java6.rtjar=path_to_rt.jar_under_jdk_1.6/rt.jar
   com.mysql.jdbc.extra.libs=path_to_folder_for_extra_libraries
   ```

   Alternatively, you can set the values of those properties through the Ant `-D` options.

7. Issue the following command to compile the driver and create a `.jar` file for Connector/J:

   ```
   shell> ant dist
   ```

   This creates a `build` directory in the current directory, where all the build output goes. A directory is created under the `build` directory, whose name includes the version number of the release you are building. That directory contains the sources, the compiled `.class` files, and a `.jar` file for deployment. For more information and other possible targets, including those that create a fully packaged distribution, issue the following command:

```
shell> ant -projecthelp
```

8. Install the newly created `.jar` file for the JDBC driver as you would install a binary `.jar` file you download from MySQL by following the instructions given in Section 3.2, “Installing the Driver and Configuring the CLASSPATH”.

3.5 Testing Connector/J

The Connector/J source code repository or packages that are shipped with source code include an extensive test suite, containing test cases that can be executed independently. The test cases are divided into the following categories:

- *Functional or unit tests:* Classes from the package `testsuite.simple`. Include test code for the main features of the Connector/J.
- *Performance tests:* Classes from the package `testsuite.perf`. Include test code to make measurements for the performance of Connector/J.
- *Regression tests:* Classes from the package `testsuite.regression`. Includes code for testing bug and regression fixes.

The bundled Ant build file contains targets like `test` and `test-multijvm`, which can facilitate the process of running the Connector/J tests; see the target descriptions in the build file for details. Besides the requirements for building Connector/J from the source code described in Section 3.4, “Installing from Source”, a number of the tests also require the File System Service Provider 1.2 for the Java Naming and Directory Interface (JNDI), available at http://www.oracle.com/technetwork/java/javasebusiness/downloads/java-archive-downloads-java-plat-419418.html)—place the jar files downloaded from there into the `lib` directory or in the directory pointed to by the property `com.mysql.jdbc.extra.libs`.

To run the test using Ant, in addition to the properties required for Section 3.4, “Installing from Source”, you must set the following properties in the `build.properties` file or through the Ant `-D` options:

- `com.mysql.jdbc.testsuite.url`: it specifies the JDBC URL for connection to a MySQL test server; see Section 5.3, “Configuration Properties for Connector/J”.
- `com.mysql.jdbc.testsuite.jvm`: the JVM to be used for the tests. If the property is set, the specified JVM will be used for all test cases except if it points to a Java 5 directory, in which case any test cases for JDBC 4.0 and later are run with the JVM supplied with the property `com.mysql.jdbc.jdk8` (for 5.1.36 and earlier, supplied with the property `com.mysql.jdbc.jdk6`). If the property is not set, the JVM supplied with `com.mysql.jdbc.jdk5` will be used to run test cases for JDBC 3.0 and the one supplied with `com.mysql.jdbc.jdk8` (for 5.1.36 and earlier, supplied with the property `com.mysql.jdbc.jdk6`) will be used to run test cases for JDBC 4.0 and later.

After setting these parameters, run the tests with Ant in the following ways:

- Building the `test` target with `ant test` runs all test cases by default on a single server instance. If you want to run a particular test case, put the test's fully qualified class names in the `test` variable; for example:

  ```
  shell > ant -Dtest=testsuite.simple.StringUtilsTest test
  ```

  You can also run individual tests in a test case by specifying the names of the corresponding methods in the `methods` variable, separating multiple methods by commas; for example:

  ```
  shell > ant -Dtest=testsuite.simple.StringUtilsTest -Dmethods=testIndexOfIgnoreCase,testGetBytes test
  ```

- Building the `test-multijvm` target with `ant test-multijvm` runs all the test cases using multiple JVMs of different versions on multiple server instances. For example, if you want to run the tests using a

Java 7 and a Java 8 JVM on three server instances with different configurations, you will need to use the following properties:

```
com.mysql.jdbc.testsuite.jvm.1=path_to_Java_7
com.mysql.jdbc.testsuite.jvm.2=path_to_Java_8
com.mysql.jdbc.testsuite.url.1=URL_to_1st_server
com.mysql.jdbc.testsuite.url.2=URL_to_2nd_server
com.mysql.jdbc.testsuite.url.3=URL_to_3rd_server
```

Unlike the target `test`, the target `test-multijvm` only recognizes the properties `com.mysql.jdbc.testsuite.jvm.N` and `com.mysql.jdbc.testsuite.url.N`, where `N` is a numeric suffice; the same properties without the suffices are ignored by `test-multijvm`. As with the target `test`, if any of the `com.mysql.jdbc.testsuite.jvm.N` settings points to Java 5, then Ant relies on the property `com.mysql.jdbc.jdk8` to run the tests specific to JDBC 4.0 and later.

You can choose to run individual test cases or specific tests by using the `test` or `methods` property, as explained in the last bullet for the target `test`. Each test is run once per possible combination of JVMs and server instances (that is, 6 times in total for in this example).

When a test for a certain JVM-server combination has failed, `test-multijvm` does not throw an error, but moves on to the next combination, until all tests for all combinations are finished.

While the test results are partially reported by the console, complete reports in HTML and XML formats are provided:

- For results of `test`: view the HTML report by opening `build/junit/unitregress/report/index.html`. XML version of the reports are located in the folder `build/junit/unitregress`.
- For results of `test-multijvm`: view the HTML report for each JVM-server combination by opening `build/junit/MySQLN.server_version/operating_system_version/jvm-version/unitregress/report/index.html`. XML version of the reports are located in the folder `build/junit/MySQLN.server_version/operating_system_version/jvm-version/unitregress`.

Chapter 4 Connector/J Examples

Examples of using Connector/J are located throughout this document. This section provides a summary and links to these examples.

- Example 6.1, “Connector/J: Obtaining a connection from the `DriverManager`”
- Example 6.2, “Connector/J: Using java.sql.Statement to execute a `SELECT` query”
- Example 6.3, “Connector/J: Calling Stored Procedures”
- Example 6.4, “Connector/J: Using `Connection.prepareCall()`”
- Example 6.5, “Connector/J: Registering output parameters”
- Example 6.6, “Connector/J: Setting `CallableStatement` input parameters”
- Example 6.7, “Connector/J: Retrieving results and output parameter values”
- Example 6.8, “Connector/J: Retrieving `AUTO_INCREMENT` column values using `Statement.getGeneratedKeys()`”
- Example 6.9, “Connector/J: Retrieving `AUTO_INCREMENT` column values using `SELECT LAST_INSERT_ID()`”
- Example 6.10, “Connector/J: Retrieving `AUTO_INCREMENT` column values in `Updatable ResultSets`”
- Example 7.1, “Connector/J: Using a connection pool with a J2EE application server”
- Example 15.1, “Connector/J: Example of transaction with retry logic”

Chapter 5 Connector/J (JDBC) Reference

Table of Contents

This section of the manual contains reference material for MySQL Connector/J.

5.1 Driver/Datasource Class Name

The name of the class that implements `java.sql.Driver` in MySQL Connector/J is `com.mysql.jdbc.Driver`.

The `org.gjt.mm.mysql.Driver` class name is also usable for backward compatibility with MM.MySQL, the predecessor of Connector/J. Use this class name when registering the driver, or when configuring a software to use MySQL Connector/J.

5.2 Connection URL Syntax

This section explains the syntax of the URLs for connecting to MySQL.

This is the generic format of the connection URL:

```
protocol//[hosts][/database][?properties]
```

The URL consists of the following parts:

> **Important**
>
> Any reserved characters for URLs (for example, `/`, `:`, `@`, `(`, `)`, `[`, `]`, `&`, `#`, `=`, `?`, and space) that appear in any part of the connection URL must be percent encoded.

protocol

There are three possible protocols for a connection:

- `jdbc:mysql:` is for ordinary and basic failover connections.
- `jdbc:mysql:loadbalance:` is for configuring load balancing. See Section 8.2, “Configuring Load Balancing with Connector/J” for details.
- `jdbc:mysql:replication:` is for configuring a replication setup. See Section 8.3, “Configuring Source/Replica Replication with Connector/J” for details.

hosts

Depending on the situation, the *host*s part may consist simply of a host name, or it can be a complex structure consisting of various elements like multiple host names, port numbers, host-specific properties, and user credentials.

- Single host:
  - Single-host connections without adding host-specific properties:
    - The *hosts* part is written in the format of *host*:*port*. This is an example of a simple single-host connection URL:

      ```
      jdbc:mysql://host1:33060/sakila
      ```

    - When *host* is not specified, the default value of `localhost` is used.
    - *port* is a standard port number, i.e., an integer between 1 and 65535. The default port number for an ordinary MySQL connection is 3306. If *port* is not specified, the default value is used.
  - Single-host connections adding host-specific properties:
    - The host is defined as a succession of *key*=*value* pairs. Keys are used to identify the host, the port, as well as any host-specific properties, and they are preceded by “`address=`”:

      ```
      address=(host=host_or_ip)(port=port)(key1=value1)(key2=value2)...(keyN=valueN)
      ```

      Here is a sample URL:

      ```
      jdbc:mysql://address=(host=myhost)(port=1111)(key1=value1)/db
      ```

      This is the mandatory format for iPv6 addresses, but it also supports the iPv4 addresses.
    - The host and the port are identified by the keys *host* and *port*. The description of the default values of *host* and *port* in Single host without host-specific properties [18] above also applies here.
    - Other keys that can be added include `user`, `password`, `protocol`, and so on. They override the global values set in the *properties* part of the URL. Limit the overrides to user, password, network timeouts, and statement and metadata cache sizes; the effects of other per-host overrides are not defined.
    - *key* is case-sensitive. Two keys differing in case only are considered conflicting, and there are no guarantees on which one will be used.
- Multiple hosts
  - Specify multiple hosts by listing them in a comma-separated list:

    ```
    host1,host2,...,hostN
    ```

Each host can be specified in any of the two ways described in Single host [18] above. Here are some examples:

```
jdbc:mysql://myhost1:1111,myhost2:2222/db
jdbc:mysql://address=(host=myhost1)(port=1111)(key1=value1),address=(host=myhost2)(port=2222)(key2=val
jdbc:mysql://myhost1:1111,address=(host=myhost2)(port=2222)(key2=value2)/db
```

- User credentials

  User credentials can be set outside of the connection URL—for example, as arguments when getting a connection from the `java.sql.DriverManager` (see Section 5.3, "Configuration Properties for Connector/J" for details). When set with the connection URL, use the keys `user` and `password` to specify credentials for each host:

  ```
  (user=sandy)(password=mypass)
  ```

  For example:

  ```
  jdbc:mysql://address=(host=myhost1)(port=1111)(user=sandy)(password=secret),address=(host=myhost2)(port=
  ```

  When multiple user credentials are specified, the one to the left takes precedence—that is, going from left to right in the connection string, the first one found that is applicable to a host is the one that is used.

database

The default database or catalog to open. If the database is not specified, the connection is made with no default database. In this case, either call the `setCatalog()` method on the `Connection` instance, or specify table names using the database name (that is, `SELECT` *`dbname.tablename.colname`* `FROM dbname.tablename...`) in your SQL statements. Opening a connection without specifying the database to use is, in general, only useful when building tools that work with multiple databases, such as GUI database managers.

> **Note**
>
> Always use the `Connection.setCatalog()` method to specify the desired database in JDBC applications, rather than the `USE` *`database`* statement.

properties

A succession of global properties applying to all hosts, preceded by "`?`" and written as *`key=value`* pairs separated by the symbol "`&`." Here are some examples:

```
jdbc:mysql://(host=myhost1,port=1111),(host=myhost2,port=2222)/db?key1=value1&key2=value2&key3=value3
```

The following are true for the key-value pairs:

- *`key`* and *`value`* are just strings. Proper type conversion and validation are performed internally in Connector/J.
- *`key`* is case-sensitive. Two keys differing in case only are considered conflicting, and it is uncertain which one will be used.
- Any host-specific values specified with key-value pairs as explained in Single host with host-specific properties [18] and Multiple hosts [18] above override the global values set here.

5.3 Configuration Properties for Connector/J

Configuration properties define how Connector/J will make a connection to a MySQL server. Unless otherwise noted, properties can be set for a `DataSource` object or for a `Connection` object.

Configuration properties can be set in one of the following ways:

- Using the `set*()` methods on MySQL implementations of `java.sql.DataSource` (which is the preferred method when using implementations of `java.sql.DataSource`):
  - `com.mysql.jdbc.jdbc2.optional.MysqlDataSource`
  - `com.mysql.jdbc.jdbc2.optional.MysqlConnectionPoolDataSource`
- As a key-value pair in the `java.util.Properties` instance passed to `DriverManager.getConnection()` or `Driver.connect()`
- As a JDBC URL parameter in the URL given to `java.sql.DriverManager.getConnection()`, `java.sql.Driver.connect()` or the MySQL implementations of the `javax.sql.DataSource setURL()` method. If you specify a configuration property in the URL without providing a value for it, nothing will be set; for example, adding `useServerPrepStmts` alone to the URL does not make Connector/J use server-side prepared statements; you need to add `useServerPrepStmts=true`.

  Note

  If the mechanism you use to configure a JDBC URL is XML-based, use the XML character literal `&` to separate configuration parameters, as the ampersand is a reserved character for XML.

The configuration properties are categorized into the following sections, in which details for each property are given.

5.3.1 Connection/Authentication

- `user`

  The user to connect as

Since Version	all versions

- `password`

  The password to use when connecting

Since Version	all versions

- `socketFactory`

  The name of the class that the driver should use for creating socket connections to the server. This class must implement the interface 'com.mysql.jdbc.SocketFactory' and have public no-args constructor.

Default Value	com.mysql.jdbc.StandardSocketFactory
Since Version	3.0.3

- `connectTimeout`

  Timeout for socket connect (in milliseconds), with 0 being no timeout. Only works on JDK-1.4 or newer. Defaults to '0'.

Default Value	0
Since Version	3.0.1

- `socketTimeout`

  Timeout (in milliseconds) on network socket operations (0, the default means no timeout).

Default Value	0
Since Version	3.0.1

- `connectionLifecycleInterceptors`

  A comma-delimited list of classes that implement "com.mysql.jdbc.ConnectionLifecycleInterceptor" that should notified of connection lifecycle events (creation, destruction, commit, rollback, setCatalog and setAutoCommit) and potentially alter the execution of these commands. ConnectionLifecycleInterceptors are "stackable", more than one interceptor may be specified via the configuration property as a comma-delimited list, with the interceptors executed in order from left to right.

Since Version	5.1.4

- `useConfigs`

  Load the comma-delimited list of configuration properties before parsing the URL or applying user-specified properties. These configurations are explained in the 'Configurations' of the documentation.

Since Version	3.1.5

- `authenticationPlugins`

  Comma-delimited list of classes that implement com.mysql.jdbc.AuthenticationPlugin and which will be used for authentication unless disabled by "disabledAuthenticationPlugins" property.

Since Version	5.1.19

- `defaultAuthenticationPlugin`

  Name of a class implementing com.mysql.jdbc.AuthenticationPlugin which will be used as the default authentication plugin (see below). It is an error to use a class which is not listed in "authenticationPlugins" nor it is one of the built-in plugins. It is an error to set as default a plugin which was disabled with "disabledAuthenticationPlugins" property. It is an error to set this value to null or the empty string (i.e. there must be at least a valid default authentication plugin specified for the connection, meeting all constraints listed above).

Default Value	com.mysql.jdbc.authentication.MysqlNativePasswordPlugin
Since Version	5.1.19

- `disabledAuthenticationPlugins`

  Comma-delimited list of classes implementing com.mysql.jdbc.AuthenticationPlugin or mechanisms, i.e. "mysql_native_password". The authentication plugins or mechanisms listed will not be used for authentication which will fail if it requires one of them. It is an error to disable the default authentication plugin (either the one named by "defaultAuthenticationPlugin" property or the hard-coded one if "defaultAuthenticationPlugin" property is not set).

Since Version	5.1.19

- `disconnectOnExpiredPasswords`

  If "disconnectOnExpiredPasswords" is set to "false" and password is expired then server enters "sandbox" mode and sends ERR(08001, ER_MUST_CHANGE_PASSWORD) for all commands that are not needed to set a new password until a new password is set.

Default Value	true
Since Version	5.1.23

- `interactiveClient`

  Set the CLIENT_INTERACTIVE flag, which tells MySQL to timeout connections based on INTERACTIVE_TIMEOUT instead of WAIT_TIMEOUT

Default Value	false
Since Version	3.1.0

- `localSocketAddress`

  Hostname or IP address given to explicitly configure the interface that the driver will bind the client side of the TCP/IP connection to when connecting.

Since Version	5.0.5

- `propertiesTransform`

  An implementation of com.mysql.jdbc.ConnectionPropertiesTransform that the driver will use to modify URL properties passed to the driver before attempting a connection

Since Version	3.1.4

- `useCompression`

  Use zlib compression when communicating with the server (true/false)? Defaults to 'false'.

Default Value	false
Since Version	3.0.17

5.3.2 Networking

- `socksProxyHost`

  Name or IP address of SOCKS host to connect through.

Since Version	5.1.34

- `socksProxyPort`

  Port of SOCKS server.

Default Value	1080

Since Version	5.1.34

- `maxAllowedPacket`

  Maximum allowed packet size to send to server. If not set, the value of system variable 'max_allowed_packet' will be used to initialize this upon connecting. This value will not take effect if set larger than the value of 'max_allowed_packet'. Also, due to an internal dependency with the property 'blobSendChunkSize', this setting has a minimum value of '8203' if 'useServerPrepStmts' is set to 'true'.

Default Value	-1
Since Version	5.1.8

- `tcpKeepAlive`

  If connecting using TCP/IP, should the driver set SO_KEEPALIVE?

Default Value	true
Since Version	5.0.7

- `tcpNoDelay`

  If connecting using TCP/IP, should the driver set SO_TCP_NODELAY (disabling the Nagle Algorithm)?

Default Value	true
Since Version	5.0.7

- `tcpRcvBuf`

  If connecting using TCP/IP, should the driver set SO_RCV_BUF to the given value? The default value of '0', means use the platform default value for this property)

Default Value	0
Since Version	5.0.7

- `tcpSndBuf`

  If connecting using TCP/IP, should the driver set SO_SND_BUF to the given value? The default value of '0', means use the platform default value for this property)

Default Value	0
Since Version	5.0.7

- `tcpTrafficClass`

  If connecting using TCP/IP, should the driver set traffic class or type-of-service fields ?See the documentation for java.net.Socket.setTrafficClass() for more information.

Default Value	0
Since Version	5.0.7

5.3.3 High Availability and Clustering

- `autoReconnect`

Should the driver try to re-establish stale and/or dead connections? If enabled the driver will throw an exception for a queries issued on a stale or dead connection, which belong to the current transaction, but will attempt reconnect before the next query issued on the connection in a new transaction. The use of this feature is not recommended, because it has side effects related to session state and data consistency when applications don't handle SQLExceptions properly, and is only designed to be used when you are unable to configure your application to handle SQLExceptions resulting from dead and stale connections properly. Alternatively, as a last option, investigate setting the MySQL server variable "wait_timeout" to a high value, rather than the default of 8 hours.

Default Value	false
Since Version	1.1

- `autoReconnectForPools`

  Use a reconnection strategy appropriate for connection pools (defaults to 'false')

Default Value	false
Since Version	3.1.3

- `failOverReadOnly`

  When failing over in autoReconnect mode, should the connection be set to 'read-only'?

Default Value	true
Since Version	3.0.12

- `maxReconnects`

  Maximum number of reconnects to attempt if autoReconnect is true, default is '3'.

Default Value	3
Since Version	1.1

- `reconnectAtTxEnd`

  If autoReconnect is set to true, should the driver attempt reconnections at the end of every transaction?

Default Value	false
Since Version	3.0.10

- `retriesAllDown`

  When using loadbalancing or failover, the number of times the driver should cycle through available hosts, attempting to connect. Between cycles, the driver will pause for 250ms if no servers are available.

Default Value	120
Since Version	5.1.6

- `initialTimeout`

  If autoReconnect is enabled, the initial time to wait between re-connect attempts (in seconds, defaults to '2').

Default Value	2
Since Version	1.1

- `roundRobinLoadBalance`

  When autoReconnect is enabled, and failoverReadonly is false, should we pick hosts to connect to on a round-robin basis?

Default Value	false
Since Version	3.1.2

- `queriesBeforeRetryMaster`

  Number of queries to issue before falling back to the primary host when failed over (when using multi-host failover). Whichever condition is met first, 'queriesBeforeRetryMaster' or 'secondsBeforeRetryMaster' will cause an attempt to be made to reconnect to the primary host. Setting both properties to 0 disables the automatic fall back to the primary host at transaction boundaries. Defaults to 50.

Default Value	50
Since Version	3.0.2

- `secondsBeforeRetryMaster`

  How long should the driver wait, when failed over, before attempting to reconnect to the primary host? Whichever condition is met first, 'queriesBeforeRetryMaster' or 'secondsBeforeRetryMaster' will cause an attempt to be made to reconnect to the source. Setting both properties to 0 disables the automatic fall back to the primary host at transaction boundaries. Time in seconds, defaults to 30

Default Value	30
Since Version	3.0.2

- `allowMasterDownConnections`

  By default, a replication-aware connection will fail to connect when configured source hosts are all unavailable at initial connection. Setting this property to 'true' allows to establish the initial connection, by failing over to the replica servers, in read-only state. It won't prevent subsequent failures when switching back to the source hosts i.e. by setting the replication connection to read/write state.

Default Value	false
Since Version	5.1.27

- `allowSlaveDownConnections`

  By default, a replication-aware connection will fail to connect when configured replica hosts are all unavailable at initial connection. Setting this property to 'true' allows to establish the initial connection. It won't prevent failures when switching to replicas i.e. by setting the replication connection to read-only state. The property 'readFromMasterWhenNoSlaves' should be used for this purpose.

Default Value	false
Since Version	5.1.38

- `readFromMasterWhenNoSlaves`

  Replication-aware connections distribute load by using the source hosts when in read/write state and by using the replica hosts when in read-only state. If, when setting the connection to read-only state, none of the replica hosts are available, an SQLExeception is thrown back. Setting this property to 'true' allows to fail over to the source hosts, while setting the connection state to read-only, when no replica hosts are available at switch instant.

Default Value	false
Since Version	5.1.38

- `replicationEnableJMX`

  Enables JMX-based management of load-balanced connection groups, including live addition/removal of hosts from load-balancing pool.

Default Value	false
Since Version	5.1.27

- `selfDestructOnPingMaxOperations`

  If set to a non-zero value, the driver will report close the connection and report failure when Connection.ping() or Connection.isValid(int) is called if the connection's count of commands sent to the server exceeds this value.

Default Value	0
Since Version	5.1.6

- `selfDestructOnPingSecondsLifetime`

  If set to a non-zero value, the driver will close the connection and report failure when Connection.ping() or Connection.isValid(int) is called if the connection's lifetime exceeds this value (in milliseconds).

Default Value	0
Since Version	5.1.6

- `replicationConnectionGroup`

  Logical group of replication connections within a classloader, used to manage different groups independently. If not specified, live management of replication connections is disabled.

Since Version	5.1.27

- `resourceId`

  A globally unique name that identifies the resource that this datasource or connection is connected to, used for XAResource.isSameRM() when the driver can't determine this value based on hostnames used in the URL

Since Version	5.0.1

5.3.4 Security

- `allowMultiQueries`

Allow the use of ';' to delimit multiple queries during one statement (true/false), defaults to 'false', and does not affect the addBatch() and executeBatch() methods, which instead rely on rewriteBatchedStatements.

Default Value	false
Since Version	3.1.1

- `useSSL`

  Use SSL when communicating with the server (true/false), default is 'true' when connecting to MySQL 5.5.45+, 5.6.26+, or 5.7.6+, otherwise default is 'false'

Default Value	false
Since Version	3.0.2

- `requireSSL`

  Require server support of SSL connection if useSSL=true? (defaults to 'false').

Default Value	false
Since Version	3.1.0

- `verifyServerCertificate`

  If "useSSL" is set to "true", should the driver verify the server's certificate? When using this feature, the keystore parameters should be specified by the "clientCertificateKeyStore*" properties, rather than system properties. Default is 'false' when connecting to MySQL 5.5.45+, 5.6.26+, or 5.7.6+ and "useSSL" was not explicitly set to "true". Otherwise default is 'true'

Default Value	true
Since Version	5.1.6

- `clientCertificateKeyStoreUrl`

  URL to the client certificate KeyStore (if not specified, use defaults)

Since Version	5.1.0

- `clientCertificateKeyStoreType`

  KeyStore type for client certificates (NULL or empty means use the default, which is "JKS". Standard keystore types supported by the JVM are "JKS" and "PKCS12", your environment may have more available depending on what security products are installed and available to the JVM.

Default Value	JKS
Since Version	5.1.0

- `clientCertificateKeyStorePassword`

  Password for the client certificates KeyStore

Since Version	5.1.0

- `trustCertificateKeyStoreUrl`

  URL to the trusted root certificate KeyStore (if not specified, use defaults)

Since Version	5.1.0

- `trustCertificateKeyStoreType`

  KeyStore type for trusted root certificates (NULL or empty means use the default, which is "JKS". Standard keystore types supported by the JVM are "JKS" and "PKCS12", your environment may have more available depending on what security products are installed and available to the JVM.

Default Value	JKS
Since Version	5.1.0

- `trustCertificateKeyStorePassword`

  Password for the trusted root certificates KeyStore

Since Version	5.1.0

- `enabledSSLCipherSuites`

  If "useSSL" is set to "true", overrides the cipher suites enabled for use on the underlying SSL sockets. This may be required when using external JSSE providers or to specify cipher suites compatible with both MySQL server and used JVM.

Since Version	5.1.35

- `enabledTLSProtocols`

  If "useSSL" is set to "true", overrides the TLS protocols enabled for use on the underlying SSL sockets. This may be used to restrict connections to specific TLS versions.

Since Version	5.1.44

- `allowLoadLocalInfile`

  Should the driver allow use of 'LOAD DATA LOCAL INFILE...'?

Default Value	false
Since Version	3.0.3

- `allowUrlInLocalInfile`

  Should the driver allow URLs in 'LOAD DATA LOCAL INFILE' statements?

Default Value	false
Since Version	3.1.4

- `allowPublicKeyRetrieval`

  Allows special handshake roundtrip to get server RSA public key directly from server.

Default Value	false
Since Version	5.1.31

- `paranoid`

  Take measures to prevent exposure sensitive information in error messages and clear data structures holding sensitive data when possible? (defaults to 'false')

Default Value	false
Since Version	3.0.1

- `passwordCharacterEncoding`

  What character encoding is used for passwords? Leaving this set to the default value (null), uses the value set in "characterEncoding" if there is one, otherwise uses UTF-8 as default encoding. If the password contains non-ASCII characters, the password encoding must match what server encoding was set to when the password was created. For passwords in other character encodings, the encoding will have to be specified with this property (or with "characterEncoding"), as it's not possible for the driver to auto-detect this.

Since Version	5.1.7

- `serverRSAPublicKeyFile`

  File path to the server RSA public key file for sha256_password authentication. If not specified, the public key will be retrieved from the server.

Since Version	5.1.31

5.3.5 Performance Extensions

- `callableStmtCacheSize`

  If 'cacheCallableStmts' is enabled, how many callable statements should be cached?

Default Value	100
Since Version	3.1.2

- `metadataCacheSize`

  The number of queries to cache ResultSetMetadata for if cacheResultSetMetaData is set to 'true' (default 50)

Default Value	50
Since Version	3.1.1

- `useLocalSessionState`

  Should the driver refer to the internal values of autocommit and transaction isolation that are set by Connection.setAutoCommit() and Connection.setTransactionIsolation() and transaction state as

maintained by the protocol, rather than querying the database or blindly sending commands to the database for commit() or rollback() method calls?

Default Value	false
Since Version	3.1.7

- `useLocalTransactionState`

  Should the driver use the in-transaction state provided by the MySQL protocol to determine if a commit() or rollback() should actually be sent to the database?

Default Value	false
Since Version	5.1.7

- `prepStmtCacheSize`

  If prepared statement caching is enabled, how many prepared statements should be cached?

Default Value	25
Since Version	3.0.10

- `prepStmtCacheSqlLimit`

  If prepared statement caching is enabled, what's the largest SQL the driver will cache the parsing for?

Default Value	256
Since Version	3.0.10

- `parseInfoCacheFactory`

  Name of a class implementing com.mysql.jdbc.CacheAdapterFactory, which will be used to create caches for the parsed representation of client-side prepared statements.

Default Value	com.mysql.jdbc.PerConnectionLRUFactory
Since Version	5.1.1

- `serverConfigCacheFactory`

  Name of a class implementing com.mysql.jdbc.CacheAdapterFactory<String, Map<String, String>>, which will be used to create caches for MySQL server configuration values

Default Value	com.mysql.jdbc.PerVmServerConfigCacheFactory
Since Version	5.1.1

- `alwaysSendSetIsolation`

  Should the driver always communicate with the database when Connection.setTransactionIsolation() is called? If set to false, the driver will only communicate with the database when the requested transaction isolation is different than the whichever is newer, the last value that was set via Connection.setTransactionIsolation(), or the value that was read from the server when the

connection was established. Note that useLocalSessionState=true will force the same behavior as alwaysSendSetIsolation=false, regardless of how alwaysSendSetIsolation is set.

Default Value	true
Since Version	3.1.7

- `maintainTimeStats`

  Should the driver maintain various internal timers to enable idle time calculations as well as more verbose error messages when the connection to the server fails? Setting this property to false removes at least two calls to System.getCurrentTimeMillis() per query.

Default Value	true
Since Version	3.1.9

- `useCursorFetch`

  If connected to MySQL > 5.0.2, and setFetchSize() > 0 on a statement, should that statement use cursor-based fetching to retrieve rows?

Default Value	false
Since Version	5.0.0

- `blobSendChunkSize`

  Chunk size to use when sending BLOB/CLOBs via ServerPreparedStatements. Note that this value cannot exceed the value of "maxAllowedPacket" and, if that is the case, then this value will be corrected automatically.

Default Value	1048576
Since Version	3.1.9

- `cacheCallableStmts`

  Should the driver cache the parsing stage of CallableStatements

Default Value	false
Since Version	3.1.2

- `cachePrepStmts`

  Should the driver cache the parsing stage of PreparedStatements of client-side prepared statements, the "check" for suitability of server-side prepared and server-side prepared statements themselves?

Default Value	false
Since Version	3.0.10

- `cacheResultSetMetadata`

  Should the driver cache ResultSetMetaData for Statements and PreparedStatements? (Req. JDK-1.4+, true/false, default 'false')

Default Value	false

Since Version	3.1.1

- `cacheServerConfiguration`

  Should the driver cache the results of 'SHOW VARIABLES' and 'SHOW COLLATION' on a per-URL basis?

Default Value	false
Since Version	3.1.5

- `defaultFetchSize`

  The driver will call setFetchSize(n) with this value on all newly-created Statements

Default Value	0
Since Version	3.1.9

- `dontCheckOnDuplicateKeyUpdateInSQL`

  Stops checking if every INSERT statement contains the "ON DUPLICATE KEY UPDATE" clause. As a side effect, obtaining the statement's generated keys information will return a list where normally it wouldn't. Also be aware that, in this case, the list of generated keys returned may not be accurate. The effect of this property is canceled if set simultaneously with 'rewriteBatchedStatements=true'.

Default Value	false
Since Version	5.1.32

- `dontTrackOpenResources`

  The JDBC specification requires the driver to automatically track and close resources, however if your application doesn't do a good job of explicitly calling close() on statements or result sets, this can cause memory leakage. Setting this property to true relaxes this constraint, and can be more memory efficient for some applications. Also the automatic closing of the Statement and current ResultSet in Statement.closeOnCompletion() and Statement.getMoreResults ([Statement.CLOSE_CURRENT_RESULT | Statement.CLOSE_ALL_RESULTS]), respectively, ceases to happen. This property automatically sets holdResultsOpenOverStatementClose=true.

Default Value	false
Since Version	3.1.7

- `dynamicCalendars`

  Should the driver retrieve the default calendar when required, or cache it per connection/session?

Default Value	false
Since Version	3.1.5

- `elideSetAutoCommits`

  If using MySQL-4.1 or newer, should the driver only issue 'set autocommit=n' queries when the server's state doesn't match the requested state by Connection.setAutoCommit(boolean)?

Default Value	false

Since Version	3.1.3

- `enableEscapeProcessing`

  Sets the default escape processing behavior for Statement objects. The method Statement.setEscapeProcessing() can be used to specify the escape processing behavior for an individual Statement object. Default escape processing behavior in prepared statements must be defined with the property 'processEscapeCodesForPrepStmts'.

Default Value	true
Since Version	5.1.37

- `enableQueryTimeouts`

  When enabled, query timeouts set via Statement.setQueryTimeout() use a shared java.util.Timer instance for scheduling. Even if the timeout doesn't expire before the query is processed, there will be memory used by the TimerTask for the given timeout which won't be reclaimed until the time the timeout would have expired if it hadn't been cancelled by the driver. High-load environments might want to consider disabling this functionality.

Default Value	true
Since Version	5.0.6

- `holdResultsOpenOverStatementClose`

  Should the driver close result sets on Statement.close() as required by the JDBC specification?

Default Value	false
Since Version	3.1.7

- `largeRowSizeThreshold`

  What size result set row should the JDBC driver consider "large", and thus use a more memory-efficient way of representing the row internally?

Default Value	2048
Since Version	5.1.1

- `loadBalanceStrategy`

  If using a load-balanced connection to connect to SQL nodes in a MySQL Cluster/NDB configuration (by using the URL prefix "jdbc:mysql:loadbalance://"), which load balancing algorithm should the driver use: (1) "random" - the driver will pick a random host for each request. This tends to work better than round-robin, as the randomness will somewhat account for spreading loads where requests vary in response time, while round-robin can sometimes lead to overloaded nodes if there are variations in response times across the workload. (2) "bestResponseTime" - the driver will route the request to the host that had the best response time for the previous transaction. (3) "serverAffinity" - the driver initially attempts to enforce server affinity while still respecting and benefiting from the fault tolerance aspects of the load-balancing implementation. The server affinity ordered list is provided using the property 'serverAffinityOrder'. If none of the servers listed in the affinity list is responsive, the driver then refers to the "random" strategy to proceed with choosing the next server.

Default Value	random

Since Version	5.0.6

- `locatorFetchBufferSize`

  If 'emulateLocators' is configured to 'true', what size buffer should be used when fetching BLOB data for getBinaryInputStream?

Default Value	1048576
Since Version	3.2.1

- `readOnlyPropagatesToServer`

  Should the driver issue appropriate statements to implicitly set the transaction access mode on server side when Connection.setReadOnly() is called? Setting this property to 'true' enables InnoDB read-only potential optimizations but also requires an extra roundtrip to set the right transaction state. Even if this property is set to 'false', the driver will do its best effort to prevent the execution of database-state-changing queries. Requires minimum of MySQL 5.6.

Default Value	true
Since Version	5.1.35

- `rewriteBatchedStatements`

  Should the driver use multiqueries (irregardless of the setting of "allowMultiQueries") as well as rewriting of prepared statements for INSERT into multi-value inserts when executeBatch() is called? Notice that this has the potential for SQL injection if using plain java.sql.Statements and your code doesn't sanitize input correctly. Notice that for prepared statements, if you don't specify stream lengths when using PreparedStatement.set*Stream(), the driver won't be able to determine the optimum number of parameters per batch and you might receive an error from the driver that the resultant packet is too large. Statement.getGeneratedKeys() for these rewritten statements only works when the entire batch includes INSERT statements. Please be aware using rewriteBatchedStatements=true with INSERT .. ON DUPLICATE KEY UPDATE that for rewritten statement server returns only one value as sum of all affected (or found) rows in batch and it isn't possible to map it correctly to initial statements; in this case driver returns 0 as a result of each batch statement if total count was 0, and the Statement.SUCCESS_NO_INFO as a result of each batch statement if total count was > 0.

Default Value	false
Since Version	3.1.13

- `serverAffinityOrder`

  A comma-separated list containing the host/port pairs that are to be used in load-balancing "serverAffinity" strategy. Only the subset of the hosts enumerated in the main hosts section in this URL will be used and they must be identical in case and type, i.e., can't use an IP address in one place and the corresponding host name in the other.

Since Version	5.1.43

- `useDirectRowUnpack`

  Use newer result set row unpacking code that skips a copy from network buffers to a MySQL packet instance and instead reads directly into the result set row data buffers.

Default Value	true
Since Version	5.1.1

- `useDynamicCharsetInfo`

  Should the driver use a per-connection cache of character set information queried from the server when necessary, or use a built-in static mapping that is more efficient, but isn't aware of custom character sets or character sets implemented after the release of the JDBC driver?

Default Value	true
Since Version	5.0.6

- `useFastDateParsing`

  Use internal String->Date/Time/Timestamp conversion routines to avoid excessive object creation? This is part of the legacy date-time code, thus the property has an effect only when "useLegacyDatetimeCode=true."

Default Value	true
Since Version	5.0.5

- `useFastIntParsing`

  Use internal String->Integer conversion routines to avoid excessive object creation?

Default Value	true
Since Version	3.1.4

- `useJvmCharsetConverters`

  Always use the character encoding routines built into the JVM, rather than using lookup tables for single-byte character sets?

Default Value	false
Since Version	5.0.1

- `useReadAheadInput`

  Use newer, optimized non-blocking, buffered input stream when reading from the server?

Default Value	true
Since Version	3.1.5

5.3.6 Debugging/Profiling

- `logger`

The name of a class that implements "com.mysql.jdbc.log.Log" that will be used to log messages to. (default is "com.mysql.jdbc.log.StandardLogger", which logs to STDERR)

Default Value	com.mysql.jdbc.log.StandardLogger
Since Version	3.1.1

- `gatherPerfMetrics`

  Should the driver gather performance metrics, and report them via the configured logger every 'reportMetricsIntervalMillis' milliseconds?

Default Value	false
Since Version	3.1.2

- `profileSQL`

  Trace queries and their execution/fetch times to the configured 'profilerEventHandler'

Default Value	false
Since Version	3.1.0

- `profileSql`

  Deprecated, use 'profileSQL' instead. Trace queries and their execution/fetch times on STDERR (true/false) defaults to 'false'

Since Version	2.0.14

- `reportMetricsIntervalMillis`

  If 'gatherPerfMetrics' is enabled, how often should they be logged (in ms)?

Default Value	30000
Since Version	3.1.2

- `maxQuerySizeToLog`

  Controls the maximum length of the part of a query that will get logged when profiling or tracing

Default Value	2048
Since Version	3.1.3

- `packetDebugBufferSize`

  The maximum number of packets to retain when 'enablePacketDebug' is true

Default Value	20
Since Version	3.1.3

- `slowQueryThresholdMillis`

  If 'logSlowQueries' is enabled, how long should a query take (in ms) before it is logged as slow?

Default Value	2000
Since Version	3.1.2

- `slowQueryThresholdNanos`

  If 'logSlowQueries' is enabled, 'useNanosForElapsedTime' is set to true, and this property is set to a non-zero value, the driver will use this threshold (in nanosecond units) to determine if a query was slow.

Default Value	0
Since Version	5.0.7

- `useUsageAdvisor`

  Should the driver issue 'usage' warnings advising proper and efficient usage of JDBC and MySQL Connector/J to the 'profilerEventHandler'?

Default Value	false
Since Version	3.1.1

- `autoGenerateTestcaseScript`

  Should the driver dump the SQL it is executing, including server-side prepared statements to STDERR?

Default Value	false
Since Version	3.1.9

- `autoSlowLog`

  Instead of using slowQueryThreshold* to determine if a query is slow enough to be logged, maintain statistics that allow the driver to determine queries that are outside the 99th percentile?

Default Value	true
Since Version	5.1.4

- `clientInfoProvider`

  The name of a class that implements the com.mysql.jdbc.JDBC4ClientInfoProvider interface in order to support JDBC-4.0's Connection.get/setClientInfo() methods

Default Value	com.mysql.jdbc.JDBC4CommentClientInfoProvider
Since Version	5.1.0

- `dumpMetadataOnColumnNotFound`

  Should the driver dump the field-level metadata of a result set into the exception message when ResultSet.findColumn() fails?

Default Value	false
Since Version	3.1.13

- `dumpQueriesOnException`

  Should the driver dump the contents of the query sent to the server in the message for SQLExceptions?

Default Value	false
Since Version	3.1.3

- `enablePacketDebug`

  When enabled, a ring-buffer of 'packetDebugBufferSize' packets will be kept, and dumped when exceptions are thrown in key areas in the driver's code

Default Value	false
Since Version	3.1.3

- `explainSlowQueries`

  If 'logSlowQueries' is enabled, should the driver automatically issue an 'EXPLAIN' on the server and send the results to the configured logger at a WARN level?

Default Value	false
Since Version	3.1.2

- `includeInnodbStatusInDeadlockExceptions`

  Include the output of "SHOW ENGINE INNODB STATUS" in exception messages when deadlock exceptions are detected?

Default Value	false
Since Version	5.0.7

- `includeThreadDumpInDeadlockExceptions`

  Include a current Java thread dump in exception messages when deadlock exceptions are detected?

Default Value	false
Since Version	5.1.15

- `includeThreadNamesAsStatementComment`

  Include the name of the current thread as a comment visible in "SHOW PROCESSLIST", or in Innodb deadlock dumps, useful in correlation with "includeInnodbStatusInDeadlockExceptions=true" and "includeThreadDumpInDeadlockExceptions=true".

Default Value	false
Since Version	5.1.15

- `logSlowQueries`

  Should queries that take longer than 'slowQueryThresholdMillis' or detected by the 'autoSlowLog' monitoring be reported to the registered 'profilerEventHandler'?

Default Value	false

Since Version	3.1.2

- `logXaCommands`

  Should the driver log XA commands sent by MysqlXaConnection to the server, at the DEBUG level of logging?

Default Value	false
Since Version	5.0.5

- `profilerEventHandler`

  Name of a class that implements the interface com.mysql.jdbc.profiler.ProfilerEventHandler that will be used to handle profiling/tracing events.

Default Value	com.mysql.jdbc.profiler.LoggingProfilerEventHandler
Since Version	5.1.6

- `resultSetSizeThreshold`

  If 'useUsageAdvisor' is true, how many rows should a result set contain before the driver warns that it is suspiciously large?

Default Value	100
Since Version	5.0.5

- `traceProtocol`

  Should the network protocol be logged at the TRACE level?

Default Value	false
Since Version	3.1.2

- `useNanosForElapsedTime`

  For profiling/debugging functionality that measures elapsed time, should the driver try to use nanoseconds resolution if available (JDK >= 1.5)?

Default Value	false
Since Version	5.0.7

5.3.7 Miscellaneous

- `useUnicode`

  Should the driver use Unicode character encodings when handling strings? Should only be used when the driver can't determine the character set mapping, or you are trying to 'force' the driver to use a character set that MySQL either doesn't natively support (such as UTF-8), true/false, defaults to 'true'

Default Value	true
Since Version	1.1g

- `characterEncoding`

If 'useUnicode' is set to true, what character encoding should the driver use when dealing with strings? (defaults is to 'autodetect')

Since Version	1.1g

- `characterSetResults`

  Character set to tell the server to return results as.

Since Version	3.0.13

- `connectionAttributes`

  A comma-delimited list of user-defined key:value pairs (in addition to standard MySQL-defined key:value pairs) to be passed to MySQL Server for display as connection attributes in the PERFORMANCE_SCHEMA.SESSION_CONNECT_ATTRS table. Example usage: connectionAttributes=key1:value1,key2:value2 This functionality is available for use with MySQL Server version 5.6 or later only. Earlier versions of MySQL Server do not support connection attributes, causing this configuration option to be ignored. Setting connectionAttributes=none will cause connection attribute processing to be bypassed, for situations where Connection creation/initialization speed is critical.

Since Version	5.1.25

- `connectionCollation`

  If set, tells the server to use this collation in SET NAMES charset COLLATE connectionCollation. Also overrides the characterEncoding with those corresponding to character set of this collation.

Since Version	3.0.13

- `useBlobToStoreUTF8OutsideBMP`

  Tells the driver to treat [MEDIUM/LONG]BLOB columns as [LONG]VARCHAR columns holding text encoded in UTF-8 that has characters outside the BMP (4-byte encodings), which MySQL server can't handle natively.

Default Value	false
Since Version	5.1.3

- `utf8OutsideBmpExcludedColumnNamePattern`

  When "useBlobToStoreUTF8OutsideBMP" is set to "true", column names matching the given regex will still be treated as BLOBs unless they match the regex specified for "utf8OutsideBmpIncludedColumnNamePattern". The regex must follow the patterns used for the java.util.regex package.

Since Version	5.1.3

- `utf8OutsideBmpIncludedColumnNamePattern`

  Used to specify exclusion rules to "utf8OutsideBmpExcludedColumnNamePattern". The regex must follow the patterns used for the java.util.regex package.

Since Version	5.1.3

- `loadBalanceEnableJMX`

  Enables JMX-based management of load-balanced connection groups, including live addition/removal of hosts from load-balancing pool.

Default Value	false
Since Version	5.1.13

- `loadBalanceHostRemovalGracePeriod`

  Sets the grace period to wait for a host being removed from a load-balanced connection, to be released when it is currently the active host.

Default Value	15000
Since Version	5.1.39

- `sessionVariables`

  A comma or semicolon separated list of name=value pairs to be sent as SET [SESSION] ... to the server when the driver connects.

Since Version	3.1.8

- `useColumnNamesInFindColumn`

  Prior to JDBC-4.0, the JDBC specification had a bug related to what could be given as a "column name" to ResultSet methods like findColumn(), or getters that took a String property. JDBC-4.0 clarified "column name" to mean the label, as given in an "AS" clause and returned by ResultSetMetaData.getColumnLabel(), and if no AS clause, the column name. Setting this property to "true" will give behavior that is congruent to JDBC-3.0 and earlier versions of the JDBC specification, but which because of the specification bug could give unexpected results. This property is preferred over "useOldAliasMetadataBehavior" unless you need the specific behavior that it provides with respect to ResultSetMetadata.

Default Value	false
Since Version	5.1.7

- `allowNanAndInf`

  Should the driver allow NaN or +/- INF values in PreparedStatement.setDouble()?

Default Value	false
Since Version	3.1.5

- `autoClosePStmtStreams`

  Should the driver automatically call .close() on streams/readers passed as arguments via set*() methods?

Default Value	false
Since Version	3.1.12

- `autoDeserialize`

Should the driver automatically detect and de-serialize objects stored in BLOB fields?

Default Value	false
Since Version	3.1.5

- `blobsAreStrings`

  Should the driver always treat BLOBs as Strings - specifically to work around dubious metadata returned by the server for GROUP BY clauses?

Default Value	false
Since Version	5.0.8

- `cacheDefaultTimezone`

  Caches client's default time zone. This results in better performance when dealing with time zone conversions in Date and Time data types, however it won't be aware of time zone changes if they happen at runtime.

Default Value	true
Since Version	5.1.35

- `capitalizeTypeNames`

  Capitalize type names in DatabaseMetaData? (usually only useful when using WebObjects, true/false, defaults to 'false')

Default Value	true
Since Version	2.0.7

- `clobCharacterEncoding`

  The character encoding to use for sending and retrieving TEXT, MEDIUMTEXT and LONGTEXT values instead of the configured connection characterEncoding

Since Version	5.0.0

- `clobberStreamingResults`

  This will cause a 'streaming' ResultSet to be automatically closed, and any outstanding data still streaming from the server to be discarded if another query is executed before all the data has been read from the server.

Default Value	false
Since Version	3.0.9

- `compensateOnDuplicateKeyUpdateCounts`

  Should the driver compensate for the update counts of "ON DUPLICATE KEY" INSERT statements (2 = 1, 0 = 1) when using prepared statements?

Default Value	false

Since Version	5.1.7

- `continueBatchOnError`

  Should the driver continue processing batch commands if one statement fails. The JDBC spec allows either way (defaults to 'true').

Default Value	true
Since Version	3.0.3

- `createDatabaseIfNotExist`

  Creates the database given in the URL if it doesn't yet exist. Assumes the configured user has permissions to create databases.

Default Value	false
Since Version	3.1.9

- `detectCustomCollations`

  Should the driver detect custom charsets/collations installed on server (true/false, defaults to 'false'). If this option set to 'true' driver gets actual charsets/collations from server each time connection establishes. This could slow down connection initialization significantly.

Default Value	false
Since Version	5.1.29

- `emptyStringsConvertToZero`

  Should the driver allow conversions from empty string fields to numeric values of '0'?

Default Value	true
Since Version	3.1.8

- `emulateLocators`

  Should the driver emulate java.sql.Blobs with locators? With this feature enabled, the driver will delay loading the actual Blob data until the one of the retrieval methods (getInputStream(), getBytes(), and so forth) on the blob data stream has been accessed. For this to work, you must use a column alias with the value of the column to the actual name of the Blob. The feature also has the following restrictions: The SELECT that created the result set must reference only one table, the table must have a primary key; the SELECT must alias the original blob column name, specified as a string, to an alternate name; the SELECT must cover all columns that make up the primary key.

Default Value	false
Since Version	3.1.0

- `emulateUnsupportedPstmts`

  Should the driver detect prepared statements that are not supported by the server, and replace them with client-side emulated versions?

Default Value	true

Since Version	3.1.7

- `exceptionInterceptors`

  Comma-delimited list of classes that implement com.mysql.jdbc.ExceptionInterceptor. These classes will be instantiated one per Connection instance, and all SQLExceptions thrown by the driver will be allowed to be intercepted by these interceptors, in a chained fashion, with the first class listed as the head of the chain.

Since Version	5.1.8

- `functionsNeverReturnBlobs`

  Should the driver always treat data from functions returning BLOBs as Strings - specifically to work around dubious metadata returned by the server for GROUP BY clauses?

Default Value	false
Since Version	5.0.8

- `generateSimpleParameterMetadata`

  Should the driver generate simplified parameter metadata for PreparedStatements when no metadata is available either because the server couldn't support preparing the statement, or server-side prepared statements are disabled?

Default Value	false
Since Version	5.0.5

- `getProceduresReturnsFunctions`

  Pre-JDBC4 DatabaseMetaData API has only the getProcedures() and getProcedureColumns() methods, so they return metadata info for both stored procedures and functions. JDBC4 was extended with the getFunctions() and getFunctionColumns() methods and the expected behaviours of previous methods are not well defined. For JDBC4 and higher, default 'true' value of the option means that calls of DatabaseMetaData.getProcedures() and DatabaseMetaData.getProcedureColumns() return metadata for both procedures and functions as before, keeping backward compatibility. Setting this property to 'false' decouples Connector/J from its pre-JDBC4 behaviours for DatabaseMetaData.getProcedures() and DatabaseMetaData.getProcedureColumns(), forcing them to return metadata for procedures only.

Default Value	true
Since Version	5.1.26

- `ignoreNonTxTables`

  Ignore non-transactional table warning for rollback? (defaults to 'false').

Default Value	false
Since Version	3.0.9

- `jdbcCompliantTruncation`

  Should the driver throw java.sql.DataTruncation exceptions when data is truncated as is required by the JDBC specification when connected to a server that supports warnings (MySQL 4.1.0 and newer)? This property has no effect if the server sql-mode includes STRICT_TRANS_TABLES.

Default Value	true
Since Version	3.1.2

- `loadBalanceAutoCommitStatementRegex`

  When load-balancing is enabled for auto-commit statements (via loadBalanceAutoCommitStatementThreshold), the statement counter will only increment when the SQL matches the regular expression. By default, every statement issued matches.

Since Version	5.1.15

- `loadBalanceAutoCommitStatementThreshold`

  When auto-commit is enabled, the number of statements which should be executed before triggering load-balancing to rebalance. Default value of 0 causes load-balanced connections to only rebalance when exceptions are encountered, or auto-commit is disabled and transactions are explicitly committed or rolled back.

Default Value	0
Since Version	5.1.15

- `loadBalanceBlacklistTimeout`

  Time in milliseconds between checks of servers which are unavailable, by controlling how long a server lives in the global blocklist.

Default Value	0
Since Version	5.1.0

- `loadBalanceConnectionGroup`

  Logical group of load-balanced connections within a classloader, used to manage different groups independently. If not specified, live management of load-balanced connections is disabled.

Since Version	5.1.13

- `loadBalanceExceptionChecker`

  Fully-qualified class name of custom exception checker. The class must implement com.mysql.jdbc.LoadBalanceExceptionChecker interface, and is used to inspect SQLExceptions and determine whether they should trigger fail-over to another host in a load-balanced deployment.

Default Value	com.mysql.jdbc.StandardLoadBalanceExceptionChecker
Since Version	5.1.13

- `loadBalancePingTimeout`

  Time in milliseconds to wait for ping response from each of load-balanced physical connections when using load-balanced Connection.

Default Value	0
Since Version	5.1.13

- `loadBalanceSQLExceptionSubclassFailover`

  Comma-delimited list of classes/interfaces used by default load-balanced exception checker to determine whether a given SQLException should trigger failover. The comparison is done using Class.isInstance(SQLException) using the thrown SQLException.

Since Version	5.1.13

- `loadBalanceSQLStateFailover`

  Comma-delimited list of SQLState codes used by default load-balanced exception checker to determine whether a given SQLException should trigger failover. The SQLState of a given SQLException is evaluated to determine whether it begins with any value in the comma-delimited list.

Since Version	5.1.13

- `loadBalanceValidateConnectionOnSwapServer`

  Should the load-balanced Connection explicitly check whether the connection is live when swapping to a new physical connection at commit/rollback?

Default Value	false
Since Version	5.1.13

- `maxRows`

  The maximum number of rows to return (0, the default means return all rows).

Default Value	-1
Since Version	all versions

- `netTimeoutForStreamingResults`

  What value should the driver automatically set the server setting 'net_write_timeout' to when the streaming result sets feature is in use? (value has unit of seconds, the value '0' means the driver will not try and adjust this value)

Default Value	600
Since Version	5.1.0

- `noAccessToProcedureBodies`

  When determining procedure parameter types for CallableStatements, and the connected user can't access procedure bodies through "SHOW CREATE PROCEDURE" or select on mysql.proc should the driver instead create basic metadata (all parameters reported as IN VARCHARs, but allowing registerOutParameter() to be called on them anyway) instead of throwing an exception?

Default Value	false
Since Version	5.0.3

- `noDatetimeStringSync`

  Don't ensure that ResultSet.getDatetimeType().toString().equals(ResultSet.getString())

Default Value	false
Since Version	3.1.7

- `noTimezoneConversionForDateType`

  Don't convert DATE values using the server time zone if 'useTimezone'='true' or 'useLegacyDatetimeCode'='false'

Default Value	true
Since Version	5.1.35

- `noTimezoneConversionForTimeType`

  Don't convert TIME values using the server time zone if 'useTimezone'='true'

Default Value	false
Since Version	5.0.0

- `nullCatalogMeansCurrent`

  When DatabaseMetadataMethods ask for a 'catalog' parameter, does the value null mean use the current catalog? (this is not JDBC-compliant, but follows legacy behavior from earlier versions of the driver)

Default Value	true
Since Version	3.1.8

- `nullNamePatternMatchesAll`

  Should DatabaseMetaData methods that accept *pattern parameters treat null the same as '%' (this is not JDBC-compliant, however older versions of the driver accepted this departure from the specification)

Default Value	true
Since Version	3.1.8

- `overrideSupportsIntegrityEnhancementFacility`

  Should the driver return "true" for DatabaseMetaData.supportsIntegrityEnhancementFacility() even if the database doesn't support it to workaround applications that require this method to return "true" to signal support of foreign keys, even though the SQL specification states that this facility contains much more than just foreign key support (one such application being OpenOffice)?

Default Value	false
Since Version	3.1.12

- `padCharsWithSpace`

  If a result set column has the CHAR type and the value does not fill the amount of characters specified in the DDL for the column, should the driver pad the remaining characters with space (for ANSI compliance)?

Default Value	false
Since Version	5.0.6

- `pedantic`

  Follow the JDBC spec to the letter.

Default Value	false
Since Version	3.0.0

- `pinGlobalTxToPhysicalConnection`

  When using XAConnections, should the driver ensure that operations on a given XID are always routed to the same physical connection? This allows the XAConnection to support "XA START ... JOIN" after "XA END" has been called

Default Value	false
Since Version	5.0.1

- `populateInsertRowWithDefaultValues`

  When using ResultSets that are CONCUR_UPDATABLE, should the driver pre-populate the "insert" row with default values from the DDL for the table used in the query so those values are immediately available for ResultSet accessors? This functionality requires a call to the database for metadata each time a result set of this type is created. If disabled (the default), the default values will be populated by the an internal call to refreshRow() which pulls back default values and/or values changed by triggers.

Default Value	false
Since Version	5.0.5

- `processEscapeCodesForPrepStmts`

  Should the driver process escape codes in queries that are prepared? Default escape processing behavior in non-prepared statements must be defined with the property 'enableEscapeProcessing'.

Default Value	true
Since Version	3.1.12

- `queryTimeoutKillsConnection`

  If the timeout given in Statement.setQueryTimeout() expires, should the driver forcibly abort the Connection instead of attempting to abort the query?

Default Value	false
Since Version	5.1.9

- `relaxAutoCommit`

If the version of MySQL the driver connects to does not support transactions, still allow calls to commit(), rollback() and setAutoCommit() (true/false, defaults to 'false')?

Default Value	false
Since Version	2.0.13

- `retainStatementAfterResultSetClose`

  Should the driver retain the Statement reference in a ResultSet after ResultSet.close() has been called. This is not JDBC-compliant after JDBC-4.0.

Default Value	false
Since Version	3.1.11

- `rollbackOnPooledClose`

  Should the driver issue a rollback() when the logical connection in a pool is closed?

Default Value	true
Since Version	3.0.15

- `runningCTS13`

  Enables workarounds for bugs in Sun's JDBC compliance testsuite version 1.3

Default Value	false
Since Version	3.1.7

- `sendFractionalSeconds`

  Send fractional part from TIMESTAMP seconds. If set to false, the nanoseconds value of TIMESTAMP values will be truncated before sending any data to the server. This option applies only to prepared statements, callable statements or updatable result sets.

Default Value	true
Since Version	5.1.37

- `serverTimezone`

  Override detection/mapping of time zone. Used when time zone from server doesn't map to Java time zone

Since Version	3.0.2

- `statementInterceptors`

  A comma-delimited list of classes that implement "com.mysql.jdbc.StatementInterceptor" that should be placed "in between" query execution to influence the results. StatementInterceptors are "chainable", the results returned by the "current" interceptor will be passed on to the next in in the chain, from left-to-right order, as specified in this property.

Since Version	5.1.1

- `strictFloatingPoint`

  Used only in older versions of compliance test

Default Value	false
Since Version	3.0.0

- `strictUpdates`

  Should the driver perform strict checking (all primary keys selected) of updatable result sets (true, false, defaults to 'true')?

Default Value	true
Since Version	3.0.4

- `tinyInt1isBit`

  Should the driver treat the datatype TINYINT(1) as the BIT type (because the server silently converts BIT -> TINYINT(1) when creating tables)?

Default Value	true
Since Version	3.0.16

- `transformedBitIsBoolean`

  If the driver converts TINYINT(1) to a different type, should it use BOOLEAN instead of BIT for future compatibility with MySQL-5.0, as MySQL-5.0 has a BIT type?

Default Value	false
Since Version	3.1.9

- `treatUtilDateAsTimestamp`

  Should the driver treat java.util.Date as a TIMESTAMP for the purposes of PreparedStatement.setObject()?

Default Value	true
Since Version	5.0.5

- `ultraDevHack`

  Create PreparedStatements for prepareCall() when required, because UltraDev is broken and issues a prepareCall() for _all_ statements? (true/false, defaults to 'false')

Default Value	false
Since Version	2.0.3

- `useAffectedRows`

  Don't set the CLIENT_FOUND_ROWS flag when connecting to the server (not JDBC-compliant, will break most applications that rely on "found" rows vs. "affected rows" for DML statements), but does cause "correct" update counts from "INSERT ... ON DUPLICATE KEY UPDATE" statements to be returned by the server.

Default Value	false
Since Version	5.1.7

- `useGmtMillisForDatetimes`

  Convert between session time zone and GMT before creating Date and Timestamp instances (value of 'false' leads to legacy behavior, 'true' leads to more JDBC-compliant behavior)? This is part of the legacy date-time code, thus the property has an effect only when "useLegacyDatetimeCode=true."

Default Value	false
Since Version	3.1.12

- `useHostsInPrivileges`

  Add '@hostname' to users in DatabaseMetaData.getColumn/TablePrivileges() (true/false), defaults to 'true'.

Default Value	true
Since Version	3.0.2

- `useInformationSchema`

  When connected to MySQL-5.0.7 or newer, should the driver use the INFORMATION_SCHEMA to derive information used by DatabaseMetaData?

Default Value	false
Since Version	5.0.0

- `useJDBCCompliantTimezoneShift`

  Should the driver use JDBC-compliant rules when converting TIME/TIMESTAMP/DATETIME values' time zone information for those JDBC arguments which take a java.util.Calendar argument? This is part of the legacy date-time code, thus the property has an effect only when "useLegacyDatetimeCode=true."

Default Value	false
Since Version	5.0.0

- `useLegacyDatetimeCode`

  Use code for DATE/TIME/DATETIME/TIMESTAMP handling in result sets and statements that consistently handles time zone conversions from client to server and back again, or use the legacy code for these datatypes that has been in the driver for backwards-compatibility? Setting this property to 'false' voids the effects of "useTimezone," "useJDBCCompliantTimezoneShift," "useGmtMillisForDatetimes," and "useFastDateParsing."

Default Value	true
Since Version	5.1.6

- `useOldAliasMetadataBehavior`

  Should the driver use the legacy behavior for "AS" clauses on columns and tables, and only return aliases (if any) for ResultSetMetaData.getColumnName() or ResultSetMetaData.getTableName() rather than the original column/table name? In 5.0.x, the default value was true.

Default Value	false
Since Version	5.0.4

- `useOldUTF8Behavior`

  Use the UTF-8 behavior the driver did when communicating with 4.0 and older servers

Default Value	false
Since Version	3.1.6

- `useOnlyServerErrorMessages`

  Don't prepend 'standard' SQLState error messages to error messages returned by the server.

Default Value	true
Since Version	3.0.15

- `useSSPSCompatibleTimezoneShift`

  If migrating from an environment that was using server-side prepared statements, and the configuration property "useJDBCCompliantTimeZoneShift" set to "true", use compatible behavior when not using server-side prepared statements when sending TIMESTAMP values to the MySQL server.

Default Value	false
Since Version	5.0.5

- `useServerPrepStmts`

  Use server-side prepared statements if the server supports them?

Default Value	false
Since Version	3.1.0

- `useSqlStateCodes`

  Use SQL Standard state codes instead of 'legacy' X/Open/SQL state codes (true/false), default is 'true'

Default Value	true
Since Version	3.1.3

- `useStreamLengthsInPrepStmts`

  Honor stream length parameter in PreparedStatement/ResultSet.setXXXStream() method calls (true/false, defaults to 'true')?

Default Value	true
Since Version	3.0.2

- `useTimezone`

  Convert time/date types between client and server time zones (true/false, defaults to 'false')? This is part of the legacy date-time code, thus the property has an effect only when "useLegacyDatetimeCode=true."

Default Value	false
Since Version	3.0.2

- `useUnbufferedInput`

  Don't use BufferedInputStream for reading data from the server

Default Value	true
Since Version	3.0.11

- `yearIsDateType`

  Should the JDBC driver treat the MySQL type "YEAR" as a java.sql.Date, or as a SHORT?

Default Value	true
Since Version	3.1.9

- `zeroDateTimeBehavior`

  What should happen when the driver encounters DATETIME values that are composed entirely of zeros (used by MySQL to represent invalid dates)? Valid values are "exception", "round" and "convertToNull".

Default Value	exception
Since Version	3.1.4

5.3.8 The useConfigs Option and the Configuration Property Files

The `useConfigs` connection option is convenient shorthand for specifying combinations of options for particular scenarios. The argument values you can use with this option correspond to the names of `.properties` files within the Connector/J `mysql-connector-java-`*`version`*`-bin.jar` JAR file. For example, the Connector/J 5.1.9 driver includes the following configuration properties files:

```
$ unzip mysql-connector-java-5.1.19-bin.jar '*/configs/*'
Archive:  mysql-connector-java-5.1.19-bin.jar
   creating: com/mysql/jdbc/configs/
  inflating: com/mysql/jdbc/configs/3-0-Compat.properties
  inflating: com/mysql/jdbc/configs/5-0-Compat.properties
  inflating: com/mysql/jdbc/configs/clusterBase.properties
  inflating: com/mysql/jdbc/configs/coldFusion.properties
  inflating: com/mysql/jdbc/configs/fullDebug.properties
  inflating: com/mysql/jdbc/configs/maxPerformance.properties
  inflating: com/mysql/jdbc/configs/solarisMaxPerformance.properties
```

To specify one of these combinations of options, specify `useConfigs=3-0-Compat`, `useConfigs=maxPerformance`, and so on. The following sections show the options that are part of each `useConfigs` setting. For the details of why each one is included, see the comments in the `.properties` files.

3-0-Compat

```
emptyStringsConvertToZero=true
```

```
jdbcCompliantTruncation=false
noDatetimeStringSync=true
nullCatalogMeansCurrent=true
nullNamePatternMatchesAll=true
transformedBitIsBoolean=false
dontTrackOpenResources=true
zeroDateTimeBehavior=convertToNull
useServerPrepStmts=false
autoClosePStmtStreams=true
processEscapeCodesForPrepStmts=false
useFastDateParsing=false
populateInsertRowWithDefaultValues=false
useDirectRowUnpack=false
```

5-0-Compat

```
useDirectRowUnpack=false
```

clusterBase

```
autoReconnect=true
failOverReadOnly=false
roundRobinLoadBalance=true
```

coldFusion

```
useDynamicCharsetInfo=false
alwaysSendSetIsolation=false
useLocalSessionState=true
autoReconnect=true
```

fullDebug

```
profileSQL=true
gatherPerfMetrics=true
useUsageAdvisor=true
logSlowQueries=true
explainSlowQueries=true
```

maxPerformance

```
cachePrepStmts=true
cacheCallableStmts=true
cacheServerConfiguration=true
useLocalSessionState=true
elideSetAutoCommits=true
alwaysSendSetIsolation=false
enableQueryTimeouts=false
```

solarisMaxPerformance

```
useUnbufferedInput=false
useReadAheadInput=false
maintainTimeStats=false
```

5.4 JDBC API Implementation Notes

MySQL Connector/J, as a rigorous implementation of the JDBC API, passes all of the tests in the publicly available version of Oracle's JDBC compliance test suite. The JDBC specification is flexible on how certain functionality should be implemented. This section gives details on an interface-by-interface level about implementation decisions that might affect how you code applications with MySQL Connector/J.

- **BLOB**

Starting with Connector/J version 3.1.0, you can emulate BLOBs with locators by adding the property `emulateLocators=true` to your JDBC URL. Using this method, the driver will delay loading the actual BLOB data until you retrieve the other data and then use retrieval methods (`getInputStream()`, `getBytes()`, and so forth) on the BLOB data stream.

You must use a column alias with the value of the column to the actual name of the BLOB, for example:

```
SELECT id, 'data' as blob_data from blobtable
```

You must also follow these rules:

- The `SELECT` must reference only one table. The table must have a primary key.
- The `SELECT` must alias the original BLOB column name, specified as a string, to an alternate name.
- The `SELECT` must cover all columns that make up the primary key.

The BLOB implementation does not allow in-place modification (they are copies, as reported by the `DatabaseMetaData.locatorsUpdateCopies()` method). Because of this, use the corresponding `PreparedStatement.setBlob()` or `ResultSet.updateBlob()` (in the case of updatable result sets) methods to save changes back to the database.

- **CallableStatement**

  Starting with Connector/J 3.1.1, stored procedures are supported when connecting to MySQL version 5.0 or newer using the `CallableStatement` interface. Currently, the `getParameterMetaData()` method of `CallableStatement` is not supported.

- **Connection**

  Unlike the pre-Connector/J JDBC driver (`MM.MySQL`), the `isClosed()` method does not ping the server to determine if it is available. In accordance with the JDBC specification, it only returns true if `closed()` has been called on the connection. If you need to determine if the connection is still valid, issue a simple query, such as `SELECT 1`. The driver will throw an exception if the connection is no longer valid.

- **DatabaseMetaData**

  Foreign key information (`getImportedKeys()`/`getExportedKeys()` and `getCrossReference()`) is only available from `InnoDB` tables. The driver uses `SHOW CREATE TABLE` to retrieve this information, so if any other storage engines add support for foreign keys, the driver would transparently support them as well.

- **PreparedStatement**

  Two variants of prepared statements are implemented by Connector/J, the client-side and the server-side prepared statements. Client-side prepared statements are used by default because early MySQL versions did not support the prepared statement feature or had problems with its implementation. Server-side prepared statements and binary-encoded result sets are used when the server supports them. To enable usage of server-side prepared statements, set `useServerPrepStmts=true`.

  Be careful when using a server-side prepared statement with **large** parameters that are set using `setBinaryStream()`, `setAsciiStream()`, `setUnicodeStream()`, `setCharacterStream()`, `setNCharacterStream()`, `setBlob()`, `setClob()`, or `setNCLob()`. To re-execute the statement

with any large parameter changed to a nonlarge parameter, call `clearParameters()` and set all parameters again. The reason for this is as follows:

- During both server-side prepared statements and client-side emulation, large data is exchanged only when `PreparedStatement.execute()` is called.
- Once that has been done, the stream used to read the data on the client side is closed (as per the JDBC spec), and cannot be read from again.
- If a parameter changes from large to nonlarge, the driver must reset the server-side state of the prepared statement to allow the parameter that is being changed to take the place of the prior large value. This removes all of the large data that has already been sent to the server, thus requiring the data to be re-sent, using the `setBinaryStream()`, `setAsciiStream()`, `setUnicodeStream()`, `setCharacterStream()`, `setNCharacterStream()`, `setBlob()`, `setClob()`, or `setNCLob()` method.

Consequently, to change the type of a parameter to a nonlarge one, you must call `clearParameters()` and set all parameters of the prepared statement again before it can be re-executed.

- **ResultSet**

  By default, ResultSets are completely retrieved and stored in memory. In most cases this is the most efficient way to operate and, due to the design of the MySQL network protocol, is easier to implement. If you are working with ResultSets that have a large number of rows or large values and cannot allocate heap space in your JVM for the memory required, you can tell the driver to stream the results back one row at a time.

  To enable this functionality, create a `Statement` instance in the following manner:

```
stmt = conn.createStatement(java.sql.ResultSet.TYPE_FORWARD_ONLY,
              java.sql.ResultSet.CONCUR_READ_ONLY);
stmt.setFetchSize(Integer.MIN_VALUE);
```

  The combination of a forward-only, read-only result set, with a fetch size of `Integer.MIN_VALUE` serves as a signal to the driver to stream result sets row-by-row. After this, any result sets created with the statement will be retrieved row-by-row.

  There are some caveats with this approach. You must read all of the rows in the result set (or close it) before you can issue any other queries on the connection, or an exception will be thrown.

  The earliest the locks these statements hold can be released (whether they be `MyISAM` table-level locks or row-level locks in some other storage engine such as `InnoDB`) is when the statement completes.

  If the statement is within scope of a transaction, then locks are released when the transaction completes (which implies that the statement needs to complete first). As with most other databases, statements are not complete until all the results pending on the statement are read or the active result set for the statement is closed.

  Therefore, if using streaming results, process them as quickly as possible if you want to maintain concurrent access to the tables referenced by the statement producing the result set.

  Another alternative is to use cursor-based streaming to retrieve a set number of rows each time. This can be done by setting the connection property `useCursorFetch` to true, and then calling `setFetchSize(int)` with `int` being the desired number of rows to be fetched each time:

```
conn = DriverManager.getConnection("jdbc:mysql://localhost/?useCursorFetch=true", "user", "s3cr3t");
```

```
stmt = conn.createStatement();
stmt.setFetchSize(100);
rs = stmt.executeQuery("SELECT * FROM your_table_here");
```

- **ResultSetMetaData**

  The `isAutoIncrement()` method only works when using MySQL servers 4.0 and newer.

- **Statement**

  When using versions of the JDBC driver earlier than 3.2.1, and connected to server versions earlier than 5.0.3, the `setFetchSize()` method has no effect, other than to toggle result set streaming as described above.

  Connector/J 5.0.0 and later include support for both `Statement.cancel()` and `Statement.setQueryTimeout()`. Both require MySQL 5.0.0 or newer server, and require a separate connection to issue the `KILL QUERY` statement. In the case of `setQueryTimeout()`, the implementation creates an additional thread to handle the timeout functionality.

  > **Note**
  >
  > Failures to cancel the statement for `setQueryTimeout()` may manifest themselves as `RuntimeException` rather than failing silently, as there is currently no way to unblock the thread that is executing the query being cancelled due to timeout expiration and have it throw the exception instead.

  > **Note**
  >
  > The MySQL statement `KILL QUERY` (which is what the driver uses to implement `Statement.cancel()`) is non-deterministic; thus, avoid the use of `Statement.cancel()` if possible. If no query is in process, the next query issued will be killed by the server. This race condition is guarded against as of Connector/J 5.1.18.

  MySQL does not support SQL cursors, and the JDBC driver doesn't emulate them, so `setCursorName()` has no effect.

  Connector/J 5.1.3 and later include two additional methods:

  - `setLocalInfileInputStream()` sets an `InputStream` instance that will be used to send data to the MySQL server for a `LOAD DATA LOCAL INFILE` statement rather than a `FileInputStream` or `URLInputStream` that represents the path given as an argument to the statement.

    This stream will be read to completion upon execution of a `LOAD DATA LOCAL INFILE` statement, and will automatically be closed by the driver, so it needs to be reset before each call to `execute*()` that would cause the MySQL server to request data to fulfill the request for `LOAD DATA LOCAL INFILE`.

    If this value is set to `NULL`, the driver will revert to using a `FileInputStream` or `URLInputStream` as required.

  - `getLocalInfileInputStream()` returns the `InputStream` instance that will be used to send data in response to a `LOAD DATA LOCAL INFILE` statement.

    This method returns `NULL` if no such stream has been set using `setLocalInfileInputStream()`.

5.5 Java, JDBC and MySQL Types

MySQL Connector/J is flexible in the way it handles conversions between MySQL data types and Java data types.

In general, any MySQL data type can be converted to a `java.lang.String`, and any numeric type can be converted to any of the Java numeric types, although round-off, overflow, or loss of precision may occur.

> **Note**
>
> All `TEXT` types return `Types.LONGVARCHAR` with different `getPrecision()` values (65535, 255, 16777215, and 2147483647 respectively) with `getColumnType()` returning `-1`. This behavior is intentional even though `TINYTEXT` does not fall, regarding to its size, within the `LONGVARCHAR` category. This is to avoid different handling inside the same base type. And `getColumnType()` returns `-1` because the internal server handling is of type `TEXT`, which is similar to `BLOB`.
>
> Also note that `getColumnTypeName()` will return `VARCHAR` even though `getColumnType()` returns `Types.LONGVARCHAR`, because `VARCHAR` is the designated column database-specific name for this type.

Starting with Connector/J 3.1.0, the JDBC driver issues warnings or throws `DataTruncation` exceptions as is required by the JDBC specification unless the connection was configured not to do so by using the property `jdbcCompliantTruncation` and setting it to `false`.

The conversions that are always guaranteed to work are listed in the following table. The first column lists one or more MySQL data types, and the second column lists one or more Java types to which the MySQL types can be converted.

Table 5.1 Possible Conversions Between MySQL and Java Data Types

These MySQL Data Types	Can always be converted to these Java types
`CHAR, VARCHAR, BLOB, TEXT, ENUM, and SET`	`java.lang.String, java.io.InputStream, java.io.Reader, java.sql.Blob, java.sql.Clob`
`FLOAT, REAL, DOUBLE PRECISION, NUMERIC, DECIMAL, TINYINT, SMALLINT, MEDIUMINT, INTEGER, BIGINT`	`java.lang.String, java.lang.Short, java.lang.Integer, java.lang.Long, java.lang.Double, java.math.BigDecimal`
`DATE, TIME, DATETIME, TIMESTAMP`	`java.lang.String, java.sql.Date, java.sql.Timestamp`

> **Note**
>
> Round-off, overflow or loss of precision may occur if you choose a Java numeric data type that has less precision or capacity than the MySQL data type you are converting to/from.

The `ResultSet.getObject()` method uses the type conversions between MySQL and Java types, following the JDBC specification where appropriate. The values returned by `ResultSetMetaData.GetColumnTypeName()`and `ResultSetMetaData.GetColumnClassName()` are shown in the table below. For more information on the JDBC types, see the reference on the java.sql.Types class.

Table 5.2 MySQL Types and Return Values for ResultSetMetaData.GetColumnTypeName()and ResultSetMetaData.GetColumnClassName()

MySQL Type Name	Return value of `GetColumnTypeName`	Return value of `GetColumnClassName`
`BIT(1)` (new in MySQL-5.0)	`BIT`	`java.lang.Boolean`
`BIT( > 1)` (new in MySQL-5.0)	`BIT`	`byte[]`
`TINYINT`	`TINYINT`	`java.lang.Boolean` if the configuration property `tinyInt1isBit` is set to `true` (the default) and the storage size is 1, or `java.lang.Integer` if not.
`BOOL`, `BOOLEAN`	`TINYINT`	See `TINYINT`, above as these are aliases for `TINYINT(1)`, currently.
`SMALLINT[(M)] [UNSIGNED]`	`SMALLINT [UNSIGNED]`	`java.lang.Integer` (regardless of whether it is `UNSIGNED` or not)
`MEDIUMINT[(M)] [UNSIGNED]`	`MEDIUMINT [UNSIGNED]`	`java.lang.Integer` (regardless of whether it is `UNSIGNED` or not)
`INT,INTEGER[(M)] [UNSIGNED]`	`INTEGER [UNSIGNED]`	`java.lang.Integer`, if `UNSIGNED` `java.lang.Long`
`BIGINT[(M)] [UNSIGNED]`	`BIGINT [UNSIGNED]`	`java.lang.Long`, if UNSIGNED `java.math.BigInteger`
`FLOAT[(M,D)]`	`FLOAT`	`java.lang.Float`
`DOUBLE[(M,B)]`	`DOUBLE`	`java.lang.Double`
`DECIMAL[(M[,D])]`	`DECIMAL`	`java.math.BigDecimal`
`DATE`	`DATE`	`java.sql.Date`
`DATETIME`	`DATETIME`	`java.sql.Timestamp`
`TIMESTAMP[(M)]`	`TIMESTAMP`	`java.sql.Timestamp`
`TIME`	`TIME`	`java.sql.Time`
`YEAR[(2|4)]`	`YEAR`	If `yearIsDateType` configuration property is set to `false`, then the returned object type is `java.sql.Short`. If set to `true` (the default), then the returned object is of type `java.sql.Date` with the date set to January 1st, at midnight.
`CHAR(M)`	`CHAR`	`java.lang.String` (unless the character set for the column is `BINARY`, then `byte[]` is returned.
`VARCHAR(M) [BINARY]`	`VARCHAR`	`java.lang.String` (unless the character set for the column is `BINARY`, then `byte[]` is returned.
`BINARY(M)`	`BINARY`	`byte[]`
`VARBINARY(M)`	`VARBINARY`	`byte[]`
`TINYBLOB`	`TINYBLOB`	`byte[]`
`TINYTEXT`	`VARCHAR`	`java.lang.String`
`BLOB`	`BLOB`	`byte[]`
`TEXT`	`VARCHAR`	`java.lang.String`

MySQL Type Name	Return value of `GetColumnTypeName`	Return value of `GetColumnClassName`
`MEDIUMBLOB`	`MEDIUMBLOB`	`byte[]`
`MEDIUMTEXT`	`VARCHAR`	`java.lang.String`
`LONGBLOB`	`LONGBLOB`	`byte[]`
`LONGTEXT`	`VARCHAR`	`java.lang.String`
`ENUM('value1','value2',..)`	`CHAR`	`java.lang.String`
`SET('value1','value2',..)`	`CHAR`	`java.lang.String`

5.6 Using Character Sets and Unicode

All strings sent from the JDBC driver to the server are converted automatically from native Java Unicode form to the client character encoding, including all queries sent using `Statement.execute()`, `Statement.executeUpdate()`, `Statement.executeQuery()` as well as all `PreparedStatement` and `CallableStatement` parameters with the exclusion of parameters set using `setBytes()`, `setBinaryStream()`, `setAsciiStream()`, `setUnicodeStream()`, and `setBlob()`.

Number of Encodings Per Connection

Connector/J supports a single character encoding between client and server, and any number of character encodings for data returned by the server to the client in `ResultSets`.

Setting the Character Encoding

The character encoding between client and server is automatically detected upon connection (provided that the Connector/J connection properties `characterEncoding` and `connectionCollation` are not set). You specify the encoding on the server using the system variable `character_set_server` (for more information, see Server Character Set and Collation). The driver automatically uses the encoding specified by the server. For example, to use the 4-byte UTF-8 character set with Connector/J, configure the MySQL server with `character_set_server=utf8mb4`, and leave `characterEncoding` and `connectionCollation` out of the Connector/J connection string. Connector/J will then autodetect the UTF-8 setting.

To override the automatically detected encoding on the client side, use the `characterEncoding` property in the connection URL to the server. Use Java-style names when specifying character encodings. The following table lists MySQL character set names and their corresponding Java-style names:

Table 5.3 MySQL to Java Encoding Name Translations

MySQL Character Set Name	Java-Style Character Encoding Name
`ascii`	`US-ASCII`
`big5`	`Big5`
`gbk`	`GBK`
`sjis`	`SJIS (or Cp932 or MS932 for MySQL Server < 4.1.11)`
`cp932`	`Cp932 or MS932 (MySQL Server > 4.1.11)`
`gb2312`	`EUC_CN`
`ujis`	`EUC_JP`
`euckr`	`EUC_KR`

MySQL Character Set Name	Java-Style Character Encoding Name
latin1	Cp1252
latin2	ISO8859_2
greek	ISO8859_7
hebrew	ISO8859_8
cp866	Cp866
tis620	TIS620
cp1250	Cp1250
cp1251	Cp1251
cp1257	Cp1257
macroman	MacRoman
macce	MacCentralEurope
For 5.1.46 and earlier: utf8 *For 5.1.47 and later*: utf8mb4	UTF-8
ucs2	UnicodeBig

Notes

For Connector/J 5.1.46 and earlier: In order to use the utf8mb4 character set for the connection, the server MUST be configured with character_set_server=utf8mb4; if that is not the case, when UTF-8 is used for characterEncoding in the connection string, it will map to the MySQL character set name utf8, which is an alias for utf8mb3.

For Connector/J 5.1.47 and later:

- When UTF-8 is used for characterEncoding in the connection string, it maps to the MySQL character set name utf8mb4.
- If the connection option connectionCollation is also set alongside characterEncoding and is incompatible with it, characterEncoding will be overridden with the encoding corresponding to connectionCollation.
- Because there is no Java-style character set name for utfmb3 that you can use with the connection option charaterEncoding, the only way to use utf8mb3 as your connection character set is to use a utf8mb3 collation (for example, utf8_general_ci) for the connection option connectionCollation, which forces a utf8mb3 character set to be used, as explained in the last bullet.

Warning

Do not issue the query SET NAMES with Connector/J, as the driver will not detect that the character set has been changed by the query, and will continue to use the character set configured when the connection was first set up.

5.7 Connecting Securely Using SSL

Connector/J can encrypt all data communicated between the JDBC driver and the server (except for the initial handshake) using SSL. There is a performance penalty for enabling connection encryption, the

severity of which depends on multiple factors including (but not limited to) the size of the query, the amount of data returned, the server hardware, the SSL library used, the network bandwidth, and so on.

The system works through two Java keystore files: one file contains the certificate information for the server (`truststore` in the examples below), and another contains the keys and certificate for the client (`keystore` in the examples below). All Java keystore files are protected by the password supplied to the `keytool` when you created the files. You need the file names and the associated passwords to create an SSL connection.

For SSL support to work, you must have the following:

- A MySQL server that supports SSL, and compiled and configured to do so. For more information, see Using Encrypted Connections and Configuring SSL Library Support.
- A signed client certificate, if using mutual (two-way) authentication.

By default, Connector/J establishes secure connections with the MySQL servers. Note that MySQL servers 5.7 and 8.0, when compiled with OpenSSL, can automatically generate missing SSL files at startup and configure the SSL connection accordingly.

As long as the server is correctly configured to use SSL, there is no need to configure anything on the Connector/J client to use encrypted connections (the exception is when Connector/J is connecting to very old server versions like 5.6.25 and earlier or 5.7.5 and earlier, in which case the client must set the connection property `useSSL=true` in order to use encrypted connections). The client can demand SSL to be used by setting the connection property `requireSSL=true`; the connection then fails if the server is not configured to use SSL. Without `requireSSL=true`, the connection just falls back to non-encrypted mode if the server is not configured to use SSL.

For additional security, you can setup the client for a one-way (server or client) or two-way (server and client) SSL authentication, allowing the client or the server to authenticate each other's identity.

Setting up Server Authentication

Server authentication via server certificate verification is enabled when the Connector/J connection property `verifyServerCertificate` is `true` (which is the default setting when `useSSL=true`).

> **Note**
>
> Standard Java SSL sockets do not support hostname verification, thus Connector/J does not support it. Host authentication is by certificates only.

Certificates signed by a trusted CA. When `verifyServerCertificate=true` (which is the default when `useSSL` is true), if no additional configurations are made regarding server authentication, Java verifies the server certificate using its default trusted CA certificates, usually from `$JAVA_HOME/lib/security/cacerts`.

Using self-signed certificates. It is pretty common though for MySQL server certificates to be self signed or signed by a self-signed CA certificate; the auto-generated certificates and keys created by the MySQL server are based on the latter—that is, the server generates all required keys and a self-signed CA certificate that is used to sign a server and a client certificate. The server then configures itself to use the CA certificate and the server certificate. Although the client certificate file is placed in the same directory, it is not used by the server.

To verify the server certificate, Connector/J needs to be able to read the certificate that signed it, that is, the server certificate that signed itself or the self-signed CA certificate. This can be accomplished by either importing the certificate (`ca.pem` or any other certificate) into the Java default truststore (although tampering the default truststore is not recommended) or by importing it into a custom Java truststore

file and configuring the Connector/J driver accordingly. Use Java's keytool (typically located in the `bin` subdirectory of your JDK or JRE installation) to import the server certificates:

```
shell> keytool -importcert -alias MySQLCACert -file ca.pem \
    -keystore truststore -storepass mypassword
```

Supply the proper arguments for the command options. If the truststore file does not already exist, a new one will be created; otherwise the certificate will be added to the existing file. Interaction with `keytool` looks like this:

```
Owner: CN=MySQL_Server_5.7.17_Auto_Generated_CA_Certificate
Issuer: CN=MySQL_Server_5.7.17_Auto_Generated_CA_Certificate
Serial number: 1
Valid from: Thu Feb 16 11:42:43 EST 2017 until: Sun Feb 14 11:42:43 EST 2027
Certificate fingerprints:
  MD5:  18:87:97:37:EA:CB:0B:5A:24:AB:27:76:45:A4:78:C1
  SHA1: 2B:0D:D9:69:2C:99:BF:1E:2A:25:4E:8D:2D:38:B8:70:66:47:FA:ED
  SHA256: C3:29:67:1B:E5:37:06:F7:A9:93:DF:C7:B3:27:5E:09:C7:FD:EE:2D:18:86:F4:9C:40:D8:26:CB:DA:95:A0:24
Signature algorithm name: SHA256withRSA
Subject Public Key Algorithm: 2048-bit RSA key
Version: 1
Trust this certificate? [no]:  yes
Certificate was added to keystore
```

The output of the command shows all details about the imported certificate. Make sure you remember the password you have supplied. Also, be mindful that the password will have to be written as plain text in your Connector/J configuration file or application source code.

The next step is to configure Java or Connector/J to read the truststore you just created or modified. This can be done by using one of the following three methods:

- Using the Java command line arguments:

  ```
  -Djavax.net.ssl.trustStore=path_to_truststore_file
  -Djavax.net.ssl.trustStorePassword=mypassword
  ```

- Setting the system properties directly in the client code:

  ```
  System.setProperty("javax.net.ssl.trustStore","path_to_truststore_file");
  System.setProperty("javax.net.ssl.trustStorePassword","mypassword");
  ```

- Setting the Connector/J connection properties:

  ```
  trustCertificateKeyStoreUrl=file:path_to_truststore_file
  trustCertificateKeyStorePassword=mypassword
  ```

Notice that when used together, the connection properties override the values set by the other two methods. Also, whatever values set with connection properties are used in that connection only, while values set using the system-wide values are used for all connections (unless overridden by the connection properties).

With the above setup and the connection property `verifyServerCertificate=true`, all connections established are going to be SSL-encrypted, with the server being authenticated in the SSL handshake process, and the client can now safely trust the server it is connecting to.

Setting up Client Authentication

The server may want to authenticate a client and require the client to provide an SSL certificate to it, which it verifies against its known certificate authorities or performs additional checks on the client identity if needed (see CREATE USER SSL/TLS Options for details). In that case, Connector/J needs to have access to the client certificate, so it can be sent to the server while establishing new database connections. This is done using the Java keystore files.

To allow client authentication, the client connecting to the server must have its own set of keys and an SSL certificate. The client certificate must be signed so that the server can verify it. While you can have the client certificates signed by official certificate authorities, it is more common to use an intermediate, private, CA certificate to sign client certificates. Such an intermediate CA certificate may be self-signed or signed by a trusted root CA. The requirement is that the server knows a CA certificate that is capable of validating the client certificate.

Some MySQL server builds are able to generate SSL keys and certificates for communication encryption, including a certificate and a private key (contained in the `client-cert.pem` and `client-key.pem` files), which can be used by any client. This SSL certificate is already signed by the self-signed CA certificate `ca.pem`, which the server may have already been configured to use.

If you do not want to use the client keys and certificate files generated by the server, you can also generate new ones using the procedures described in Creating SSL and RSA Certificates and Keys. Notice that, according to the setup of the server, you may have to reuse the already existing CA certificate the server is configured to work with to sign the new client certificate, instead of creating a new one.

Once you have the client private key and certificate files you want to use, you need to import them into a Java keystore so that they can be used by the Java SSL library and Connector/J. The following instructions explain how to create the keystore file:

- Convert the client key and certificate files to a PKCS #12 archive:

```
shell> openssl pkcs12 -export -in client-cert.pem -inkey client-key.pem \
  -name "mysqlclient" -passout pass:mypassword -out client-keystore.p12
```

- Import the client key and certificate into a Java keystore:

```
shell> keytool -importkeystore -srckeystore client-keystore.p12 -srcstoretype pkcs12 \
 -srcstorepass mypassword -destkeystore keystore -deststoretype JKS -deststorepass mypassword
```

  Supply the proper arguments for the command options. If the keystore file does not already exist, a new one will be created; otherwise the certificate will be added to the existing file. Output by `keytool` looks like this:

```
Entry for alias mysqlclient successfully imported.
Import command completed:  1 entries successfully imported, 0 entries failed or cancelled
```

  Make sure you remember the password you have chosen. Also, be mindful that the password will have to be written as plain text in your Connector/J configuration file or application source code.

After the step, you can delete the PKCS #12 archive (*`client-keystore.p12`* in the example).

The next step is to configure Java or Connector/J so that it reads the truststore you just created or modified. This can be done by using one of the following three methods:

- Using the Java command line arguments:

```
-Djavax.net.ssl.keyStore=path_to_keystore_file
-Djavax.net.ssl.keyStorePassword=mypassword
```

- Setting the system properties directly in the client code:

```
System.setProperty("javax.net.ssl.keyStore","path_to_keystore_file");
System.setProperty("javax.net.ssl.keyStorePassword","mypassword");
```

- Through Connector/J connection properties:

```
clientCertificateKeyStoreUrl=file:path_to_truststore_file
clientCertificateKeyStorePassword=mypassword
```

Notice that when used together, the connection properties override the values set by the other two methods. Also, whatever values set with connection properties are used in that connection only, while values set using the system-wide values are used for all connections (unless overridden by the connection properties).

With the above setups , all connections established are going to be SSL-encrypted with the client being authenticated in the SSL handshake process, and the server can now safely trust the client that is requesting a connection to it.

Setting up 2-Way Authentication

Apply the steps outlined in both Setting up Server Authentication and Setting up Client Authentication to set up a mutual, two-way authentication process in which the server and the client authenticate each other before establishing a connection.

Although the typical setup described above uses the same CA certificate in both ends for mutual authentication, it does not have to be the case. The only requirements are that the CA certificate configured in the server must be able to validate the client certificate and the CA certificate imported into the client truststore must be able to validate the server certificate; the two CA certificates used on the two ends can be distinct.

Debugging an SSL Connection

JSSE provides debugging information to `stdout` when you set the system property `-Djavax.net.debug=all`. Java then tells you what keystores and truststores are being used, as well as what is going on during the SSL handshake and certificate exchange. That will be helpful when you are trying to debug a failed SSL connection.

5.8 Connecting Using Unix Domain Sockets

Connector/J does not natively support connections to MySQL Servers with Unix domain sockets. However, there is provision for using 3rd-party libraries that supply the function via a pluggable socket factory. Such a custom factory should implement the legacy `com.mysql.jdbc.SocketFactory` interface of Connector/J. Follow these requirements when you use such a custom socket factory for Unix sockets :

- The MySQL Server must be configured with the system variable `--socket`, which must contain the file path of the Unix socket file.
- The fully-qualified class name of the custom factory should be passed to Connector/J via the connection property `socketFactory`. For example, with the junixsocket library, set:

  ```
  socketFactory=org.newsclub.net.mysql.AFUNIXDatabaseSocketFactory
  ```

  You might also need to pass other parameters to the custom factory as connection properties. For example, for the junixsocket library, provide the file path of the socket file with the property `junixsocket.file`:

  ```
  junixsocket.file=path_to_socket_file
  ```

5.9 Connecting Using Named Pipes

> **Important**
>
> *For MySQL 8.0.14 and later, 5.7.25 and later, and 5.6.43 and later*, minimal permissions on named pipes are granted to clients that use them to connect to the server. Connector/J, however, can only use named pipes when granted full access

> on them. As a workaround, the MySQL Server that Connector/J wants to connect to must be started with the system variable `named_pipe_full_access_group`, which specifies a Windows local group containing the user by which the client application JVM (and thus Connector/J) is being executed; see the description for `named_pipe_full_access_group` for more details.

Connector/J also supports access to MySQL using named pipes on Windows platforms with the `NamedPipeSocketFactory` as a plugin-sockets factory. If you do not use a `namedPipePath` property, the default of `'\\.\pipe\MySQL'` is used. If you use the `NamedPipeSocketFactory`, the host name and port number values in the JDBC URL are ignored. To enable this feature, set the `socketFactory` property:

```
socketFactory=com.mysql.cj.protocol.NamedPipeSocketFactory
```

Set this property, as well as the path of the named pipe, with the following connection URL:

```
jdbc:mysql:///test?socketFactory=com.mysql.cj.protocol.NamedPipeSocketFactory&namedPipePath=\\.\pipe\MySQL57
```

To create your own socket factories, follow the sample code in `com.mysql.cj.protocol.NamedPipeSocketFactory` or `com.mysql.cj.protocol.StandardSocketFactory`.

An alternate approach is to use the following two properties in connection URLs for establishing named pipe connections on Windows platforms:

- `(protocol=pipe)` for named pipes (default value for the property is `tcp`).
- `(path=`*`path_to_pipe`*`)` for path of named pipes. Default value for the path is `\\.\pipe\MySQL`.

The “address-equals” or “key-value” form of host specification (see Single host [18] for details) greatly simplifies the URL for a named pipe connection on Windows. For example, to use the default named pipe of “`\\.\pipe\MySQL`,” just specify:

```
jdbc:mysql://address=(protocol=pipe)/test
```

To use the custom named pipe of “`\\.\pipe\MySQL57`” :

```
jdbc:mysql://address=(protocol=pipe)(path=\\.\pipe\MySQL57)/test
```

With `(protocol=pipe)`, the `NamedPipeSocketFactory` is automatically selected.

Named pipes only work when connecting to a MySQL server on the same physical machine where the JDBC driver is running. In simple performance tests, named pipe access is between 30%-50% faster than the standard TCP/IP access. However, this varies per system, and named pipes are slower than TCP/IP in many Windows configurations.

5.10 Connecting Using PAM Authentication

Java applications using Connector/J 5.1.21 and higher can connect to MySQL servers that use the pluggable authentication module (PAM) authentication scheme.

For PAM authentication to work, you must have the following:

- A MySQL server that supports PAM authentication. See PAM Pluggable Authentication for more information. Connector/J implements the same cleartext authentication method as in Client-Side Cleartext Pluggable Authentication.
- SSL capability, as explained in Section 5.7, “Connecting Securely Using SSL”. Because the PAM authentication scheme sends the original password to the server, the connection to the server must be encrypted.

PAM authentication support is enabled by default in Connector/J 5.1.21 and up, so no extra configuration is needed.

To disable the PAM authentication feature, specify `mysql_clear_password` (the method) or `com.mysql.jdbc.authentication.MysqlClearPasswordPlugin` (the class name) in the comma-separated list of arguments for the `disabledAuthenticationPlugins` connection option. See Section 5.3, “Configuration Properties for Connector/J” for details about that connection option.

5.11 Source/Replica Using Replication with ReplicationConnection

See Section 8.3, “Configuring Source/Replica Replication with Connector/J” for details on the topic.

5.12 Mapping MySQL Error Numbers to JDBC SQLState Codes

The table below provides a mapping of the MySQL error numbers to JDBC `SQLState` values.

Table 5.4 Mapping of MySQL Error Numbers to SQLStates

MySQL Error Number	MySQL Error Name	Legacy (X/Open) SQLState	SQL Standard SQLState
1022	ER_DUP_KEY	23000	23000
1037	ER_OUTOFMEMORY	S1001	HY001
1038	ER_OUT_OF_SORTMEMORY	S1001	HY001
1040	ER_CON_COUNT_ERROR	08004	08004
1042	ER_BAD_HOST_ERROR	08004	08S01
1043	ER_HANDSHAKE_ERROR	08004	08S01
1044	ER_DBACCESS_DENIED_ERROR	42000	42000
1045	ER_ACCESS_DENIED_ERROR	28000	28000
1046	ER_NO_DB_ERROR	3D000	3D000
1047	ER_UNKNOWN_COM_ERROR	08S01	08S01
1048	ER_BAD_NULL_ERROR	23000	23000
1049	ER_BAD_DB_ERROR	42000	42000
1050	ER_TABLE_EXISTS_ERROR	42S01	42S01
1051	ER_BAD_TABLE_ERROR	42S02	42S02
1052	ER_NON_UNIQ_ERROR	23000	23000
1053	ER_SERVER_SHUTDOWN	08S01	08S01
1054	ER_BAD_FIELD_ERROR	S0022	42S22
1055	ER_WRONG_FIELD_WITH_GROUP	S1009	42000
1056	ER_WRONG_GROUP_FIELD	S1009	42000
1057	ER_WRONG_SUM_SELECT	S1009	42000
1058	ER_WRONG_VALUE_COUNT	21S01	21S01
1059	ER_TOO_LONG_IDENT	S1009	42000
1060	ER_DUP_FIELDNAME	S1009	42S21
1061	ER_DUP_KEYNAME	S1009	42000

MySQL Error Number	MySQL Error Name	Legacy (X/Open) SQLState	SQL Standard SQLState
1062	ER_DUP_ENTRY	S1009	23000
1063	ER_WRONG_FIELD_SPEC	S1009	42000
1064	ER_PARSE_ERROR	42000	42000
1065	ER_EMPTY_QUERY	42000	42000
1066	ER_NONUNIQ_TABLE	S1009	42000
1067	ER_INVALID_DEFAULT	S1009	42000
1068	ER_MULTIPLE_PRI_KEY	S1009	42000
1069	ER_TOO_MANY_KEYS	S1009	42000
1070	ER_TOO_MANY_KEY_PARTS	S1009	42000
1071	ER_TOO_LONG_KEY	S1009	42000
1072	ER_KEY_COLUMN_DOES_NOT_EXITS	S1009	42000
1073	ER_BLOB_USED_AS_KEY	S1009	42000
1074	ER_TOO_BIG_FIELDLENGTH	S1009	42000
1075	ER_WRONG_AUTO_KEY	S1009	42000
1080	ER_FORCING_CLOSE	08S01	08S01
1081	ER_IPSOCK_ERROR	08S01	08S01
1082	ER_NO_SUCH_INDEX	S1009	42S12
1083	ER_WRONG_FIELD_TERMINATORS	S1009	42000
1084	ER_BLOBS_AND_NO_TERMINATED	S1009	42000
1090	ER_CANT_REMOVE_ALL_FIELDS	42000	42000
1091	ER_CANT_DROP_FIELD_OR_KEY	42000	42000
1101	ER_BLOB_CANT_HAVE_DEFAULT	42000	42000
1102	ER_WRONG_DB_NAME	42000	42000
1103	ER_WRONG_TABLE_NAME	42000	42000
1104	ER_TOO_BIG_SELECT	42000	42000
1106	ER_UNKNOWN_PROCEDURE	42000	42000
1107	ER_WRONG_PARAMCOUNT_TO_PROCEDURE	42000	42000
1109	ER_UNKNOWN_TABLE	42S02	42S02
1110	ER_FIELD_SPECIFIED_TWICE	42000	42000
1112	ER_UNSUPPORTED_EXTENSION	42000	42000
1113	ER_TABLE_MUST_HAVE_COLUMNS	42000	42000
1115	ER_UNKNOWN_CHARACTER_SET	42000	42000
1118	ER_TOO_BIG_ROWSIZE	42000	42000
1120	ER_WRONG_OUTER_JOIN	42000	42000
1121	ER_NULL_COLUMN_IN_INDEX	42000	42000
1129	ER_HOST_IS_BLOCKED	08004	HY000

MySQL Error Number	MySQL Error Name	Legacy (X/Open) SQLState	SQL Standard SQLState
1130	ER_HOST_NOT_PRIVILEGED	08004	HY000
1131	ER_PASSWORD_ANONYMOUS_USER	42000	42000
1132	ER_PASSWORD_NOT_ALLOWED	42000	42000
1133	ER_PASSWORD_NO_MATCH	42000	42000
1136	ER_WRONG_VALUE_COUNT_ON_ROW	21S01	21S01
1138	ER_INVALID_USE_OF_NULL	S1000	42000
1139	ER_REGEXP_ERROR	42000	42000
1140	ER_MIX_OF_GROUP_FUNC_AND_FIELDS	42000	42000
1141	ER_NONEXISTING_GRANT	42000	42000
1142	ER_TABLEACCESS_DENIED_ERROR	42000	42000
1143	ER_COLUMNACCESS_DENIED_ERROR	42000	42000
1144	ER_ILLEGAL_GRANT_FOR_TABLE	42000	42000
1145	ER_GRANT_WRONG_HOST_OR_USER	42000	42000
1146	ER_NO_SUCH_TABLE	42S02	42S02
1147	ER_NONEXISTING_TABLE_GRANT	42000	42000
1148	ER_NOT_ALLOWED_COMMAND	42000	42000
1149	ER_SYNTAX_ERROR	42000	42000
1152	ER_ABORTING_CONNECTION	08S01	08S01
1153	ER_NET_PACKET_TOO_LARGE	08S01	08S01
1154	ER_NET_READ_ERROR_FROM_PIPE	08S01	08S01
1155	ER_NET_FCNTL_ERROR	08S01	08S01
1156	ER_NET_PACKETS_OUT_OF_ORDER	08S01	08S01
1157	ER_NET_UNCOMPRESS_ERROR	08S01	08S01
1158	ER_NET_READ_ERROR	08S01	08S01
1159	ER_NET_READ_INTERRUPTED	08S01	08S01
1160	ER_NET_ERROR_ON_WRITE	08S01	08S01
1161	ER_NET_WRITE_INTERRUPTED	08S01	08S01
1162	ER_TOO_LONG_STRING	42000	42000
1163	ER_TABLE_CANT_HANDLE_BLOB	42000	42000
1164	ER_TABLE_CANT_HANDLE_AUTO_INCREMENT	42000	42000
1166	ER_WRONG_COLUMN_NAME	42000	42000
1167	ER_WRONG_KEY_COLUMN	42000	42000
1169	ER_DUP_UNIQUE	23000	23000
1170	ER_BLOB_KEY_WITHOUT_LENGTH	42000	42000
1171	ER_PRIMARY_CANT_HAVE_NULL	42000	42000
1172	ER_TOO_MANY_ROWS	42000	42000

MySQL Error Number	MySQL Error Name	Legacy (X/Open) SQLState	SQL Standard SQLState
1173	ER_REQUIRES_PRIMARY_KEY	42000	42000
1176	ER_KEY_DOES_NOT_EXITS	42000	42000
1177	ER_CHECK_NO_SUCH_TABLE	42000	42000
1178	ER_CHECK_NOT_IMPLEMENTED	42000	42000
1179	ER_CANT_DO_THIS_DURING_AN_TRANSACTION	25000	25000
1184	ER_NEW_ABORTING_CONNECTION	08S01	08S01
1189	ER_MASTER_NET_READ	08S01	08S01
1190	ER_MASTER_NET_WRITE	08S01	08S01
1203	ER_TOO_MANY_USER_CONNECTIONS	42000	42000
1205	ER_LOCK_WAIT_TIMEOUT	40001	40001
1207	ER_READ_ONLY_TRANSACTION	25000	25000
1211	ER_NO_PERMISSION_TO_CREATE_USER	42000	42000
1213	ER_LOCK_DEADLOCK	40001	40001
1216	ER_NO_REFERENCED_ROW	23000	23000
1217	ER_ROW_IS_REFERENCED	23000	23000
1218	ER_CONNECT_TO_MASTER	08S01	08S01
1222	ER_WRONG_NUMBER_OF_COLUMNS_IN_SELECT	21000	21000
1226	ER_USER_LIMIT_REACHED	42000	42000
1227	ER_SPECIFIC_ACCESS_DENIED_ERROR	42000	42000
1230	ER_NO_DEFAULT	42000	42000
1231	ER_WRONG_VALUE_FOR_VAR	42000	42000
1232	ER_WRONG_TYPE_FOR_VAR	42000	42000
1234	ER_CANT_USE_OPTION_HERE	42000	42000
1235	ER_NOT_SUPPORTED_YET	42000	42000
1239	ER_WRONG_FK_DEF	42000	42000
1241	ER_OPERAND_COLUMNS	21000	21000
1242	ER_SUBQUERY_NO_1_ROW	21000	21000
1247	ER_ILLEGAL_REFERENCE	42S22	42S22
1248	ER_DERIVED_MUST_HAVE_ALIAS	42000	42000
1249	ER_SELECT_REDUCED	01000	01000
1250	ER_TABLENAME_NOT_ALLOWED_HERE	42000	42000
1251	ER_NOT_SUPPORTED_AUTH_MODE	08004	08004
1252	ER_SPATIAL_CANT_HAVE_NULL	42000	42000
1253	ER_COLLATION_CHARSET_MISMATCH	42000	42000
1261	ER_WARN_TOO_FEW_RECORDS	01000	01000
1262	ER_WARN_TOO_MANY_RECORDS	01000	01000

MySQL Error Number	MySQL Error Name	Legacy (X/Open) SQLState	SQL Standard SQLState
1263	ER_WARN_NULL_TO_NOTNULL	S1000	01000
1264	ER_WARN_DATA_OUT_OF_RANGE	01000	01000
1265	ER_WARN_DATA_TRUNCATED	01000	01000
1280	ER_WRONG_NAME_FOR_INDEX	42000	42000
1281	ER_WRONG_NAME_FOR_CATALOG	42000	42000
1286	ER_UNKNOWN_STORAGE_ENGINE	42000	42000
1292	ER_TRUNCATED_WRONG_VALUE	22007	22007
1303	ER_SP_NO_RECURSIVE_CREATE	S1000	2F003
1304	ER_SP_ALREADY_EXISTS	42000	42000
1305	ER_SP_DOES_NOT_EXIST	42000	42000
1308	ER_SP_LILABEL_MISMATCH	42000	42000
1309	ER_SP_LABEL_REDEFINE	42000	42000
1310	ER_SP_LABEL_MISMATCH	42000	42000
1311	ER_SP_UNINIT_VAR	01000	01000
1312	ER_SP_BADSELECT	0A000	0A000
1313	ER_SP_BADRETURN	42000	42000
1314	ER_SP_BADSTATEMENT	0A000	0A000
1315	ER_UPDATE_LOG_DEPRECATED_IGNORED	42000	42000
1316	ER_UPDATE_LOG_DEPRECATED_TRANSLATED	42000	42000
1317	ER_QUERY_INTERRUPTED	S1000	70100
1318	ER_SP_WRONG_NO_OF_ARGS	42000	42000
1319	ER_SP_COND_MISMATCH	42000	42000
1320	ER_SP_NORETURN	42000	42000
1321	ER_SP_NORETURNEND	S1000	2F005
1322	ER_SP_BAD_CURSOR_QUERY	42000	42000
1323	ER_SP_BAD_CURSOR_SELECT	42000	42000
1324	ER_SP_CURSOR_MISMATCH	42000	42000
1325	ER_SP_CURSOR_ALREADY_OPEN	24000	24000
1326	ER_SP_CURSOR_NOT_OPEN	24000	24000
1327	ER_SP_UNDECLARED_VAR	42000	42000
1329	ER_SP_FETCH_NO_DATA	S1000	02000
1330	ER_SP_DUP_PARAM	42000	42000
1331	ER_SP_DUP_VAR	42000	42000
1332	ER_SP_DUP_COND	42000	42000
1333	ER_SP_DUP_CURS	42000	42000
1335	ER_SP_SUBSELECT_NYI	0A000	0A000

MySQL Error Number	MySQL Error Name	Legacy (X/Open) SQLState	SQL Standard SQLState
1336	ER_STMT_NOT_ALLOWED_IN_SF_OR_TRG	0A000	0A000
1337	ER_SP_VARCOND_AFTER_CURSHNDLR	42000	42000
1338	ER_SP_CURSOR_AFTER_HANDLER	42000	42000
1339	ER_SP_CASE_NOT_FOUND	S1000	20000
1365	ER_DIVISION_BY_ZERO	22012	22012
1367	ER_ILLEGAL_VALUE_FOR_TYPE	22007	22007
1370	ER_PROCACCESS_DENIED_ERROR	42000	42000
1397	ER_XAER_NOTA	S1000	XAE04
1398	ER_XAER_INVAL	S1000	XAE05
1399	ER_XAER_RMFAIL	S1000	XAE07
1400	ER_XAER_OUTSIDE	S1000	XAE09
1401	ER_XA_RMERR	S1000	XAE03
1402	ER_XA_RBROLLBACK	S1000	XA100
1403	ER_NONEXISTING_PROC_GRANT	42000	42000
1406	ER_DATA_TOO_LONG	22001	22001
1407	ER_SP_BAD_SQLSTATE	42000	42000
1410	ER_CANT_CREATE_USER_WITH_GRANT	42000	42000
1413	ER_SP_DUP_HANDLER	42000	42000
1414	ER_SP_NOT_VAR_ARG	42000	42000
1415	ER_SP_NO_RETSET	0A000	0A000
1416	ER_CANT_CREATE_GEOMETRY_OBJECT	22003	22003
1425	ER_TOO_BIG_SCALE	42000	42000
1426	ER_TOO_BIG_PRECISION	42000	42000
1427	ER_M_BIGGER_THAN_D	42000	42000
1437	ER_TOO_LONG_BODY	42000	42000
1439	ER_TOO_BIG_DISPLAYWIDTH	42000	42000
1440	ER_XAER_DUPID	S1000	XAE08
1441	ER_DATETIME_FUNCTION_OVERFLOW	22008	22008
1451	ER_ROW_IS_REFERENCED_2	23000	23000
1452	ER_NO_REFERENCED_ROW_2	23000	23000
1453	ER_SP_BAD_VAR_SHADOW	42000	42000
1458	ER_SP_WRONG_NAME	42000	42000
1460	ER_SP_NO_AGGREGATE	42000	42000
1461	ER_MAX_PREPARED_STMT_COUNT_REACHED	42000	42000
1463	ER_NON_GROUPING_FIELD_USED	42000	42000
1557	ER_FOREIGN_DUPLICATE_KEY	23000	23000

MySQL Error Number	MySQL Error Name	Legacy (X/Open) SQLState	SQL Standard SQLState
1568	ER_CANT_CHANGE_TX_ISOLATION	S1000	25001
1582	ER_WRONG_PARAMCOUNT_TO_NATIVE_FCT	42000	42000
1583	ER_WRONG_PARAMETERS_TO_NATIVE_FCT	42000	42000
1584	ER_WRONG_PARAMETERS_TO_STORED_FCT	42000	42000
1586	ER_DUP_ENTRY_WITH_KEY_NAME	23000	23000
1613	ER_XA_RBTIMEOUT	S1000	XA106
1614	ER_XA_RBDEADLOCK	S1000	XA102
1630	ER_FUNC_INEXISTENT_NAME_COLLISION	42000	42000
1641	ER_DUP_SIGNAL_SET	42000	42000
1642	ER_SIGNAL_WARN	01000	01000
1643	ER_SIGNAL_NOT_FOUND	S1000	02000
1645	ER_RESIGNAL_WITHOUT_ACTIVE_HANDLER	S1000	0K000
1687	ER_SPATIAL_MUST_HAVE_GEOM_COL	42000	42000
1690	ER_DATA_OUT_OF_RANGE	22003	22003
1698	ER_ACCESS_DENIED_NO_PASSWORD_ERROR	28000	28000
1701	ER_TRUNCATE_ILLEGAL_FK	42000	42000
1758	ER_DA_INVALID_CONDITION_NUMBER	35000	35000
1761	ER_FOREIGN_DUPLICATE_KEY_WITH_CHILD_INFO	23000	23000
1762	ER_FOREIGN_DUPLICATE_KEY_WITHOUT_CHILD_INFO	23000	23000
1792	ER_CANT_EXECUTE_IN_READ_ONLY_TRANSACTION	S1000	25006
1845	ER_ALTER_OPERATION_NOT_SUPPORTED	0A000	0A000
1846	ER_ALTER_OPERATION_NOT_SUPPORTED_REASON	0A000	0A000
1859	ER_DUP_UNKNOWN_IN_INDEX	23000	23000
1873	ER_ACCESS_DENIED_CHANGE_USER_ERROR	28000	28000
1887	ER_GET_STACKED_DA_WITHOUT_ACTIVE_HANDLER	S1000	0Z002
1903	ER_INVALID_ARGUMENT_FOR_LOGARITHM	S1000	2201E

Chapter 6 JDBC Concepts

Table of Contents

This section provides some general JDBC background.

6.1 Connecting to MySQL Using the JDBC `DriverManager` Interface

When you are using JDBC outside of an application server, the `DriverManager` class manages the establishment of connections.

Specify to the `DriverManager` which JDBC drivers to try to make Connections with. The easiest way to do this is to use `Class.forName()` on the class that implements the `java.sql.Driver` interface. With MySQL Connector/J, the name of this class is `com.mysql.jdbc.Driver`. With this method, you could use an external configuration file to supply the driver class name and driver parameters to use when connecting to a database.

The following section of Java code shows how you might register MySQL Connector/J from the `main()` method of your application. If testing this code, first read the installation section at Chapter 3, *Connector/J Installation*, to make sure you have connector installed correctly and the `CLASSPATH` set up. Also, ensure that MySQL is configured to accept external TCP/IP connections.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

// Notice, do not import com.mysql.jdbc.*
// or you will have problems!

public class LoadDriver {
    public static void main(String[] args) {
        try {
            // The newInstance() call is a work around for some
            // broken Java implementations

            Class.forName("com.mysql.jdbc.Driver").newInstance();
        } catch (Exception ex) {
            // handle the error
        }
    }
}
```

After the driver has been registered with the `DriverManager`, you can obtain a `Connection` instance that is connected to a particular database by calling `DriverManager.getConnection()`:

Example 6.1 Connector/J: Obtaining a connection from the `DriverManager`

If you have not already done so, please review the portion of Section 6.1, “Connecting to MySQL Using the JDBC `DriverManager` Interface” above before working with the example below.

This example shows how you can obtain a Connection instance from the DriverManager. There are a few different signatures for the getConnection() method. Consult the API documentation that comes with your JDK for more specific information on how to use them.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

Connection conn = null;
...
try {
    conn =
       DriverManager.getConnection("jdbc:mysql://localhost/test?" +
                                   "user=minty&password=greatsqldb");

    // Do something with the Connection

   ...
} catch (SQLException ex) {
    // handle any errors
    System.out.println("SQLException: " + ex.getMessage());
    System.out.println("SQLState: " + ex.getSQLState());
    System.out.println("VendorError: " + ex.getErrorCode());
}
```

Once a Connection is established, it can be used to create Statement and PreparedStatement objects, as well as retrieve metadata about the database. This is explained in the following sections.

6.2 Using JDBC Statement Objects to Execute SQL

Statement objects allow you to execute basic SQL queries and retrieve the results through the ResultSet class, which is described later.

To create a Statement instance, you call the createStatement() method on the Connection object you have retrieved using one of the DriverManager.getConnection() or DataSource.getConnection() methods described earlier.

Once you have a Statement instance, you can execute a SELECT query by calling the executeQuery(String) method with the SQL you want to use.

To update data in the database, use the executeUpdate(String SQL) method. This method returns the number of rows matched by the update statement, not the number of rows that were modified.

If you do not know ahead of time whether the SQL statement will be a SELECT or an UPDATE/INSERT, then you can use the execute(String SQL) method. This method will return true if the SQL query was a SELECT, or false if it was an UPDATE, INSERT, or DELETE statement. If the statement was a SELECT query, you can retrieve the results by calling the getResultSet() method. If the statement was an UPDATE, INSERT, or DELETE statement, you can retrieve the affected rows count by calling getUpdateCount() on the Statement instance.

Example 6.2 Connector/J: Using java.sql.Statement to execute a SELECT query

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.ResultSet;

// assume that conn is an already created JDBC connection (see previous examples)
```

```
Statement stmt = null;
ResultSet rs = null;

try {
    stmt = conn.createStatement();
    rs = stmt.executeQuery("SELECT foo FROM bar");

    // or alternatively, if you don't know ahead of time that
    // the query will be a SELECT...

    if (stmt.execute("SELECT foo FROM bar")) {
        rs = stmt.getResultSet();
    }

    // Now do something with the ResultSet ....
}
catch (SQLException ex){
    // handle any errors
    System.out.println("SQLException: " + ex.getMessage());
    System.out.println("SQLState: " + ex.getSQLState());
    System.out.println("VendorError: " + ex.getErrorCode());
}
finally {
    // it is a good idea to release
    // resources in a finally{} block
    // in reverse-order of their creation
    // if they are no-longer needed

    if (rs != null) {
        try {
            rs.close();
        } catch (SQLException sqlEx) { } // ignore

        rs = null;
    }

    if (stmt != null) {
        try {
            stmt.close();
        } catch (SQLException sqlEx) { } // ignore

        stmt = null;
    }
}
```

6.3 Using JDBC CallableStatements to Execute Stored Procedures

Starting with MySQL server version 5.0 when used with Connector/J 3.1.1 or newer, the `java.sql.CallableStatement` interface is fully implemented with the exception of the `getParameterMetaData()` method.

For more information on MySQL stored procedures, please refer to Using Stored Routines.

Connector/J exposes stored procedure functionality through JDBC's `CallableStatement` interface.

> **Note**
>
> Current versions of MySQL server do not return enough information for the JDBC driver to provide result set metadata for callable statements. This means that when using `CallableStatement`, `ResultSetMetaData` may return `NULL`.

The following example shows a stored procedure that returns the value of inOutParam incremented by 1, and the string passed in using inputParam as a ResultSet:

Example 6.3 Connector/J: Calling Stored Procedures

```
CREATE PROCEDURE demoSp(IN inputParam VARCHAR(255), \
                        INOUT inOutParam INT)
BEGIN
    DECLARE z INT;
    SET z = inOutParam + 1;
    SET inOutParam = z;

    SELECT inputParam;

    SELECT CONCAT('zyxw', inputParam);
END
```

To use the demoSp procedure with Connector/J, follow these steps:

1. Prepare the callable statement by using Connection.prepareCall().

   Notice that you have to use JDBC escape syntax, and that the parentheses surrounding the parameter placeholders are not optional:

   Example 6.4 Connector/J: Using Connection.prepareCall()

   ```
   import java.sql.CallableStatement;

   ...

       //
       // Prepare a call to the stored procedure 'demoSp'
       // with two parameters
       //
       // Notice the use of JDBC-escape syntax ({call ...})
       //

       CallableStatement cStmt = conn.prepareCall("{call demoSp(?, ?)}");



       cStmt.setString(1, "abcdefg");
   ```

   > **Note**
   >
   > Connection.prepareCall() is an expensive method, due to the metadata retrieval that the driver performs to support output parameters. For performance reasons, minimize unnecessary calls to Connection.prepareCall() by reusing CallableStatement instances in your code.

2. Register the output parameters (if any exist)

   To retrieve the values of output parameters (parameters specified as OUT or INOUT when you created the stored procedure), JDBC requires that they be specified before statement execution using the various registerOutputParameter() methods in the CallableStatement interface:

   Example 6.5 Connector/J: Registering output parameters

   ```
   import java.sql.Types;
   ...
   //
   // Connector/J supports both named and indexed
   // output parameters. You can register output
   ```

```
// parameters using either method, as well
// as retrieve output parameters using either
// method, regardless of what method was
// used to register them.
//
// The following examples show how to use
// the various methods of registering
// output parameters (you should of course
// use only one registration per parameter).
//

//
// Registers the second parameter as output, and
// uses the type 'INTEGER' for values returned from
// getObject()
//

cStmt.registerOutParameter(2, Types.INTEGER);

//
// Registers the named parameter 'inOutParam', and
// uses the type 'INTEGER' for values returned from
// getObject()
//

cStmt.registerOutParameter("inOutParam", Types.INTEGER);
...
```

3. Set the input parameters (if any exist)

   Input and in/out parameters are set as for PreparedStatement objects. However, CallableStatement also supports setting parameters by name:

   Example 6.6 Connector/J: Setting CallableStatement input parameters

```
...

    //
    // Set a parameter by index
    //

    cStmt.setString(1, "abcdefg");

    //
    // Alternatively, set a parameter using
    // the parameter name
    //

    cStmt.setString("inputParam", "abcdefg");

    //
    // Set the 'in/out' parameter using an index
    //

    cStmt.setInt(2, 1);

    //
    // Alternatively, set the 'in/out' parameter
    // by name
    //

    cStmt.setInt("inOutParam", 1);

...
```

4. Execute the CallableStatement, and retrieve any result sets or output parameters.

Although CallableStatement supports calling any of the Statement execute methods (executeUpdate(), executeQuery() or execute()), the most flexible method to call is execute(), as you do not need to know ahead of time if the stored procedure returns result sets:

Example 6.7 Connector/J: Retrieving results and output parameter values

```
...

    boolean hadResults = cStmt.execute();

    //
    // Process all returned result sets
    //

    while (hadResults) {
        ResultSet rs = cStmt.getResultSet();

        // process result set
        ...

        hadResults = cStmt.getMoreResults();
    }

    //
    // Retrieve output parameters
    //
    // Connector/J supports both index-based and
    // name-based retrieval
    //

    int outputValue = cStmt.getInt(2); // index-based

    outputValue = cStmt.getInt("inOutParam"); // name-based

...
```

6.4 Retrieving AUTO_INCREMENT Column Values through JDBC

Before version 3.0 of the JDBC API, there was no standard way of retrieving key values from databases that supported auto increment or identity columns. With older JDBC drivers for MySQL, you could always use a MySQL-specific method on the Statement interface, or issue the query SELECT LAST_INSERT_ID() after issuing an INSERT to a table that had an AUTO_INCREMENT key. Using the MySQL-specific method call isn't portable, and issuing a SELECT to get the AUTO_INCREMENT key's value requires another round-trip to the database, which isn't as efficient as possible. The following code snippets demonstrate the three different ways to retrieve AUTO_INCREMENT values. First, we demonstrate the use of the new JDBC 3.0 method getGeneratedKeys() which is now the preferred method to use if you need to retrieve AUTO_INCREMENT keys and have access to JDBC 3.0. The second example shows how you can retrieve the same value using a standard SELECT LAST_INSERT_ID() query. The final example shows how updatable result sets can retrieve the AUTO_INCREMENT value when using the insertRow() method.

Example 6.8 Connector/J: Retrieving AUTO_INCREMENT column values using Statement.getGeneratedKeys()

```
Statement stmt = null;
ResultSet rs = null;

try {

    //
    // Create a Statement instance that we can use for
```

```
    // 'normal' result sets assuming you have a
    // Connection 'conn' to a MySQL database already
    // available

    stmt = conn.createStatement();

    //
    // Issue the DDL queries for the table for this example
    //

    stmt.executeUpdate("DROP TABLE IF EXISTS autoIncTutorial");
    stmt.executeUpdate(
            "CREATE TABLE autoIncTutorial ("
            + "priKey INT NOT NULL AUTO_INCREMENT, "
            + "dataField VARCHAR(64), PRIMARY KEY (priKey))");

    //
    // Insert one row that will generate an AUTO INCREMENT
    // key in the 'priKey' field
    //

    stmt.executeUpdate(
            "INSERT INTO autoIncTutorial (dataField) "
            + "values ('Can I Get the Auto Increment Field?')",
            Statement.RETURN_GENERATED_KEYS);

    //
    // Example of using Statement.getGeneratedKeys()
    // to retrieve the value of an auto-increment
    // value
    //

    int autoIncKeyFromApi = -1;

    rs = stmt.getGeneratedKeys();

    if (rs.next()) {
        autoIncKeyFromApi = rs.getInt(1);
    } else {

        // throw an exception from here
    }

    System.out.println("Key returned from getGeneratedKeys():"
        + autoIncKeyFromApi);
} finally {

    if (rs != null) {
        try {
            rs.close();
        } catch (SQLException ex) {
            // ignore
        }
    }

    if (stmt != null) {
        try {
            stmt.close();
        } catch (SQLException ex) {
            // ignore
        }
    }
}
```

Example 6.9 Connector/J: Retrieving AUTO_INCREMENT column values using SELECT LAST_INSERT_ID()

```
Statement stmt = null;
ResultSet rs = null;

try {

    //
    // Create a Statement instance that we can use for
    // 'normal' result sets.

    stmt = conn.createStatement();

    //
    // Issue the DDL queries for the table for this example
    //

    stmt.executeUpdate("DROP TABLE IF EXISTS autoIncTutorial");
    stmt.executeUpdate(
            "CREATE TABLE autoIncTutorial ("
            + "priKey INT NOT NULL AUTO_INCREMENT, "
            + "dataField VARCHAR(64), PRIMARY KEY (priKey))");

    //
    // Insert one row that will generate an AUTO INCREMENT
    // key in the 'priKey' field
    //

    stmt.executeUpdate(
            "INSERT INTO autoIncTutorial (dataField) "
            + "values ('Can I Get the Auto Increment Field?')");

    //
    // Use the MySQL LAST_INSERT_ID()
    // function to do the same thing as getGeneratedKeys()
    //

    int autoIncKeyFromFunc = -1;
    rs = stmt.executeQuery("SELECT LAST_INSERT_ID()");

    if (rs.next()) {
        autoIncKeyFromFunc = rs.getInt(1);
    } else {
        // throw an exception from here
    }

    System.out.println("Key returned from " +
                       "'SELECT LAST_INSERT_ID()': " +
                       autoIncKeyFromFunc);

} finally {

    if (rs != null) {
        try {
            rs.close();
        } catch (SQLException ex) {
            // ignore
        }
    }

    if (stmt != null) {
        try {
            stmt.close();
        } catch (SQLException ex) {
            // ignore
        }
```

```
    }
}
```

Example 6.10 Connector/J: Retrieving AUTO_INCREMENT column values in Updatable ResultSets

```
Statement stmt = null;
ResultSet rs = null;

try {

    //
    // Create a Statement instance that we can use for
    // 'normal' result sets as well as an 'updatable'
    // one, assuming you have a Connection 'conn' to
    // a MySQL database already available
    //

    stmt = conn.createStatement(java.sql.ResultSet.TYPE_FORWARD_ONLY,
                                java.sql.ResultSet.CONCUR_UPDATABLE);

    //
    // Issue the DDL queries for the table for this example
    //

    stmt.executeUpdate("DROP TABLE IF EXISTS autoIncTutorial");
    stmt.executeUpdate(
            "CREATE TABLE autoIncTutorial ("
            + "priKey INT NOT NULL AUTO_INCREMENT, "
            + "dataField VARCHAR(64), PRIMARY KEY (priKey))");

    //
    // Example of retrieving an AUTO INCREMENT key
    // from an updatable result set
    //

    rs = stmt.executeQuery("SELECT priKey, dataField "
       + "FROM autoIncTutorial");

    rs.moveToInsertRow();

    rs.updateString("dataField", "AUTO INCREMENT here?");
    rs.insertRow();

    //
    // the driver adds rows at the end
    //

    rs.last();

    //
    // We should now be on the row we just inserted
    //

    int autoIncKeyFromRS = rs.getInt("priKey");

    System.out.println("Key returned for inserted row: "
        + autoIncKeyFromRS);

} finally {

    if (rs != null) {
        try {
            rs.close();
        } catch (SQLException ex) {
            // ignore
        }
    }
```

```
    if (stmt != null) {
        try {
            stmt.close();
        } catch (SQLException ex) {
            // ignore
        }
    }
}
```

Running the preceding example code should produce the following output:

```
Key returned from getGeneratedKeys(): 1
Key returned from SELECT LAST_INSERT_ID(): 1
Key returned for inserted row: 1
```

At times, it can be tricky to use the `SELECT LAST_INSERT_ID()` query, as that function's value is scoped to a connection. So, if some other query happens on the same connection, the value is overwritten. On the other hand, the `getGeneratedKeys()` method is scoped by the `Statement` instance, so it can be used even if other queries happen on the same connection, but not on the same `Statement` instance.

Chapter 7 Connection Pooling with Connector/J

Connection pooling is a technique of creating and managing a pool of connections that are ready for use by any thread that needs them. Connection pooling can greatly increase the performance of your Java application, while reducing overall resource usage.

How Connection Pooling Works

Most applications only need a thread to have access to a JDBC connection when they are actively processing a transaction, which often takes only milliseconds to complete. When not processing a transaction, the connection sits idle. Connection pooling enables the idle connection to be used by some other thread to do useful work.

In practice, when a thread needs to do work against a MySQL or other database with JDBC, it requests a connection from the pool. When the thread is finished using the connection, it returns it to the pool, so that it can be used by any other threads.

When the connection is loaned out from the pool, it is used exclusively by the thread that requested it. From a programming point of view, it is the same as if your thread called `DriverManager.getConnection()` every time it needed a JDBC connection. With connection pooling, your thread may end up using either a new connection or an already-existing connection.

Benefits of Connection Pooling

The main benefits to connection pooling are:

- Reduced connection creation time.

  Although this is not usually an issue with the quick connection setup that MySQL offers compared to other databases, creating new JDBC connections still incurs networking and JDBC driver overhead that will be avoided if connections are recycled.

- Simplified programming model.

  When using connection pooling, each individual thread can act as though it has created its own JDBC connection, allowing you to use straightforward JDBC programming techniques.

- Controlled resource usage.

  If you create a new connection every time a thread needs one rather than using connection pooling, your application's resource usage can be wasteful, and it could lead to unpredictable behaviors for your application when it is under a heavy load.

Using Connection Pooling with Connector/J

The concept of connection pooling in JDBC has been standardized through the JDBC 2.0 Optional interfaces, and all major application servers have implementations of these APIs that work with MySQL Connector/J.

Generally, you configure a connection pool in your application server configuration files, and access it through the Java Naming and Directory Interface (JNDI). The following code shows how you might use a connection pool from an application deployed in a J2EE application server:

Example 7.1 Connector/J: Using a connection pool with a J2EE application server

```
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;
```

```
import javax.naming.InitialContext;
import javax.sql.DataSource;


public class MyServletJspOrEjb {

    public void doSomething() throws Exception {
        /*
         * Create a JNDI Initial context to be able to
         *  lookup  the DataSource
         *
         * In production-level code, this should be cached as
         * an instance or static variable, as it can
         * be quite expensive to create a JNDI context.
         *
         * Note: This code only works when you are using servlets
         * or EJBs in a J2EE application server. If you are
         * using connection pooling in standalone Java code, you
         * will have to create/configure datasources using whatever
         * mechanisms your particular connection pooling library
         * provides.
         */

        InitialContext ctx = new InitialContext();

         /*
          * Lookup the DataSource, which will be backed by a pool
          * that the application server provides. DataSource instances
          * are also a good candidate for caching as an instance
          * variable, as JNDI lookups can be expensive as well.
          */

        DataSource ds =
          (DataSource)ctx.lookup("java:comp/env/jdbc/MySQLDB");

        /*
         * The following code is what would actually be in your
         * Servlet, JSP or EJB 'service' method...where you need
         * to work with a JDBC connection.
         */

        Connection conn = null;
        Statement stmt = null;

        try {
            conn = ds.getConnection();

            /*
             * Now, use normal JDBC programming to work with
             * MySQL, making sure to close each resource when you're
             * finished with it, which permits the connection pool
             * resources to be recovered as quickly as possible
             */

            stmt = conn.createStatement();
            stmt.execute("SOME SQL QUERY");

            stmt.close();
            stmt = null;

            conn.close();
            conn = null;
        } finally {
            /*
             * close any jdbc instances here that weren't
             * explicitly closed during normal code path, so
```

```
             * that we don't 'leak' resources...
             */

            if (stmt != null) {
                try {
                    stmt.close();
                } catch (sqlexception sqlex) {
                    // ignore, as we can't do anything about it here
                }

                stmt = null;
            }

            if (conn != null) {
                try {
                    conn.close();
                } catch (sqlexception sqlex) {
                    // ignore, as we can't do anything about it here
                }

                conn = null;
            }
        }
    }
}
```

As shown in the example above, after obtaining the JNDI `InitialContext`, and looking up the `DataSource`, the rest of the code follows familiar JDBC conventions.

When using connection pooling, always make sure that connections, and anything created by them (such as statements or result sets) are closed. This rule applies no matter what happens in your code (exceptions, flow-of-control, and so forth). When these objects are closed, they can be re-used; otherwise, they will be stranded, which means that the MySQL server resources they represent (such as buffers, locks, or sockets) are tied up for some time, or in the worst case can be tied up forever.

Sizing the Connection Pool

Each connection to MySQL has overhead (memory, CPU, context switches, and so forth) on both the client and server side. Every connection limits how many resources there are available to your application as well as the MySQL server. Many of these resources will be used whether or not the connection is actually doing any useful work! Connection pools can be tuned to maximize performance, while keeping resource utilization below the point where your application will start to fail rather than just run slower.

The optimal size for the connection pool depends on anticipated load and average database transaction time. In practice, the optimal connection pool size can be smaller than you might expect. If you take Oracle's Java Petstore blueprint application for example, a connection pool of 15-20 connections can serve a relatively moderate load (600 concurrent users) using MySQL and Tomcat with acceptable response times.

To correctly size a connection pool for your application, create load test scripts with tools such as Apache JMeter or The Grinder, and load test your application.

An easy way to determine a starting point is to configure your connection pool's maximum number of connections to be unbounded, run a load test, and measure the largest amount of concurrently used connections. You can then work backward from there to determine what values of minimum and maximum pooled connections give the best performance for your particular application.

Validating Connections

MySQL Connector/J can validate the connection by executing a lightweight ping against a server. In the case of load-balanced connections, this is performed against all active pooled internal connections that are

retained. This is beneficial to Java applications using connection pools, as the pool can use this feature to validate connections. Depending on your connection pool and configuration, this validation can be carried out at different times:

1. Before the pool returns a connection to the application.
2. When the application returns a connection to the pool.
3. During periodic checks of idle connections.

To use this feature, specify a validation query in your connection pool that starts with `/* ping */`. Note that the syntax must be exactly as specified. This will cause the driver send a ping to the server and return a dummy lightweight result set. When using a `ReplicationConnection` or `LoadBalancedConnection`, the ping will be sent across all active connections.

It is critical that the syntax be specified correctly. The syntax needs to be exact for reasons of efficiency, as this test is done for every statement that is executed:

```
protected static final String PING_MARKER = "/* ping */";
...
if (sql.charAt(0) == '/') {
if (sql.startsWith(PING_MARKER)) {
doPingInstead();
...
```

None of the following snippets will work, because the ping syntax is sensitive to whitespace, capitalization, and placement:

```
sql = "/* PING */ SELECT 1";
sql = "SELECT 1 /* ping*/";
sql = "/*ping*/ SELECT 1";
sql = " /* ping */ SELECT 1";
sql = "/*to ping or not to ping*/ SELECT 1";
```

All of the previous statements will issue a normal `SELECT` statement and will **not** be transformed into the lightweight ping. Further, for load-balanced connections, the statement will be executed against one connection in the internal pool, rather than validating each underlying physical connection. This results in the non-active physical connections assuming a stale state, and they may die. If Connector/J then re-balances, it might select a dead connection, resulting in an exception being passed to the application. To help prevent this, you can use `loadBalanceValidateConnectionOnSwapServer` to validate the connection before use.

If your Connector/J deployment uses a connection pool that allows you to specify a validation query, take advantage of it, but ensure that the query starts *exactly* with `/* ping */`. This is particularly important if you are using the load-balancing or replication-aware features of Connector/J, as it will help keep alive connections which otherwise will go stale and die, causing problems later.

Chapter 8 Multi-Host Connections

Table of Contents

The following sections discuss a number of topics that involve multi-host connections, namely, server load-balancing, failover, and replication.

Developers should know the following things about multi-host connections that are managed through Connector/J:

- Each multi-host connection is a wrapper of the underlying physical connections.
- Each of the underlying physical connections has its own session. Sessions cannot be tracked, shared, or copied, given the MySQL architecture.
- Every switch between physical connections means a switch between sessions.
- Within a transaction boundary, there are no switches between physical connections. Beyond a transaction boundary, there is no guarantee that a switch does not occur.

> **Note**
>
> If an application reuses session-scope data (for example, variables, SSPs) beyond a transaction boundary, failures are possible, as a switch between the physical connections (which is also a switch between sessions) might occur. Therefore, the application should re-prepare the session data and also restart the last transaction in case of an exception, or it should re-prepare session data for each new transaction if it does not want to deal with exception handling.

8.1 Configuring Server Failover

MySQL Connector/J supports server failover. A failover happens when connection-related errors occur for an underlying, active connection. The connection errors are, by default, propagated to the client, which has to handle them by, for example, recreating the working objects (`Statement`, `ResultSet`, etc.) and restarting the processes. Sometimes, the driver might eventually fall back to the original host automatically before the client application continues to run, in which case the host switch is transparent and the client application will not even notice it.

A connection using failover support works just like a standard connection: the client does not experience any disruptions in the failover process. This means the client can rely on the same connection instance even if two successive statements might be executed on two different physical hosts. However, this does not mean the client does not have to deal with the exception that triggered the server switch.

The failover is configured at the initial setup stage of the server connection by the connection URL (see explanations for its format here):

```
jdbc:mysql://[primary host][:port],[secondary host 1][:port][,[secondary host 2][:port]]...[/[database]]»
[?propertyName1=propertyValue1[&propertyName2=propertyValue2]...]
```

The host list in the connection URL comprises of two types of hosts, the primary and the secondary. When starting a new connection, the driver always tries to connect to the primary host first and, if required, fails over to the secondary hosts on the list sequentially when communication problems are experienced. Even if the initial connection to the primary host fails and the driver gets connected to a secondary host, the primary host never loses its special status: for example, it can be configured with an access mode distinct from those of the secondary hosts, and it can be put on a higher priority when a host is to be picked during a failover process.

The failover support is configured by the following connection properties (their functions are explained in the paragraphs below):

- `failOverReadOnly`
- `secondsBeforeRetryMaster`
- `queriesBeforeRetryMaster`
- `retriesAllDown`
- `autoReconnect`
- `autoReconnectForPools`

Configuring Connection Access Mode

As with any standard connection, the initial connection to the primary host is in read/write mode. However, if the driver fails to establish the initial connection to the primary host and it automatically switches to the next host on the list, the access mode now depends on the value of the property `failOverReadOnly`, which is “true” by default. The same happens if the driver is initially connected to the primary host and, because of some connection failure, it fails over to a secondary host. Every time the connection falls back to the primary host, its access mode will be read/write, irrespective of whether or not the primary host has been connected to before. The connection access mode can be changed any time at runtime by calling the method `Connection.setReadOnly(boolean)`, which partially overrides the property `failOverReadOnly`. When `failOverReadOnly=false` and the access mode is explicitly set to either true or false, it becomes the mode for every connection after a host switch, no matter what host type are we connected to; but, if `failOverReadOnly=true`, changing the access mode to read/write is only possible if the driver is connecting to the primary host; however, even if the access mode cannot be changed for the current connection, the driver remembers the client's last intention and, when falling back to the primary host, that is the mode that will be used. For an illustration, see the following successions of events with a two-host connection.

- Sequence A, with `failOverReadOnly=true`:
  1. Connects to primary host in read/write mode
  2. Sets `Connection.setReadOnly(true)`; primary host now in read-only mode
  3. Failover event; connects to secondary host in read-only mode
  4. Sets `Connection.setReadOnly(false)`; secondary host remains in read-only mode
  5. Falls back to primary host; connection now in read/write mode
- Sequence B, with `failOverReadOnly=false`

1. Connects to primary host in read/write mode
2. Sets `Connection.setReadOnly(true)`; primary host now in read-only mode
3. Failover event; connects to secondary host in read-only mode
4. Set `Connection.setReadOnly(false)`; connection to secondary host switches to read/write mode
5. Falls back to primary host; connection now in read/write mode

The difference between the two scenarios is in step 4: the access mode for the secondary host in sequence A does not change at that step, but the driver remembers and uses the set mode when falling back to the primary host, which would be read-only otherwise; but in sequence B, the access mode for the secondary host changes immediately.

Configuring Fallback to Primary Host

As already mentioned, the primary host is special in the failover arrangement when it comes to the host's access mode. Additionally, the driver tries to fall back to the primary host as soon as possible by default, even if no communication exception occurs. Two properties, `secondsBeforeRetryMaster` and `queriesBeforeRetryMaster`, determine when the driver is ready to retry a reconnection to the primary host (the `Master` in the property names stands for the primary host of our connection URL, which is not necessarily a source host in a replication setup; the naming was maintained for back compatibility with Connector/J versions prior to 5.1.35):

- `secondsBeforeRetryMaster` determines how much time the driver waits before trying to fall back to the primary host
- `queriesBeforeRetryMaster` determines the number of queries that are executed before the driver tries to fall back to the primary host. Note that for the driver, each call to a `Statement.execute*()` method increments the query execution counter; therefore, when calls are made to `Statement.executeBatch()` or if `allowMultiQueries` or `rewriteBatchStatements` are enabled, the driver may not have an accurate count of the actual number of queries executed on the server. Also, the driver calls the `Statement.execute*()` methods internally in several occasions. All these mean you can only use `queriesBeforeRetryMaster` only as a coarse specification for when to fall back to the primary host.

In general, an attempt to fallback to the primary host is made when at least one of the conditions specified by the two properties is met, and the attempt always takes place at transaction boundaries. However, if auto-commit is turned off, the check happens only when the method `Connection.commit()` or `Connection.rollback()` is called. The automatic fallback to the primary host can be turned off by setting simultaneously `secondsBeforeRetryMaster` and `queriesBeforeRetryMaster` to “0”. Setting only one of the properties to “0” only disables one part of the check.

Configuring Reconnection Attempts

When establishing a new connection or when a failover event occurs, the driver tries to connect successively to the next candidate on the host list. When the end of the list has been reached, it restarts all over again from the beginning of the list; however, the primary host is skipped over, if (a) NOT all the secondary hosts have already been tested at least once, AND (b) the fallback conditions defined by `secondsBeforeRetryMaster` and `queriesBeforeRetryMaster` are not yet fulfilled. Each run-through of the whole host list, (which is not necessarily completed at the end of the host list) counts as a single connection attempt. The driver tries as many connection attempts as specified by the value of the property `retriesAllDown`.

Seamless Reconnection

Although not recommended, you can make the driver perform failovers without invalidating the active `Statement` or `ResultSet` instances by setting either the parameter `autoReconnect` or `autoReconnectForPools` to `true`. This allows the client to continue using the same object instances after a failover event, without taking any exceptional measures. This, however, may lead to unexpected results: for example, if the driver is connected to the primary host with read/write access mode and it fails-over to a secondary host in real-only mode, further attempts to issue data-changing queries will result in errors, and the client will not be aware of that. This limitation is particularly relevant when using data streaming: after the failover, the `ResultSet` looks to be alright, but the underlying connection may have changed already, and no backing cursor is available anymore.

8.2 Configuring Load Balancing with Connector/J

Connector/J has long provided an effective means to distribute read/write load across multiple MySQL server instances for Cluster or source-source replication deployments. Starting with Connector/J 5.1.3, you can now dynamically configure load-balanced connections, with no service outage. In-process transactions are not lost, and no application exceptions are generated if any application is trying to use that particular server instance.

The load balancing is configured at the initial setup stage of the server connection by the following connection URL, which has a similar format as the general URL for MySQL connection, but a specialized scheme:

```
jdbc:mysql:loadbalance://[host1][:port],[host2][:port][,[host3][:port]]...[/[database]] »
[?propertyName1=propertyValue1[&propertyName2=propertyValue2]...]
```

There are two configuration properties associated with this functionality:

- `loadBalanceConnectionGroup` – This provides the ability to group connections from different sources. This allows you to manage these JDBC sources within a single class loader in any combination you choose. If they use the same configuration, and you want to manage them as a logical single group, give them the same name. This is the key property for management: if you do not define a name (string) for `loadBalanceConnectionGroup`, you cannot manage the connections. All load-balanced connections sharing the same `loadBalanceConnectionGroup` value, regardless of how the application creates them, will be managed together.

- `loadBalanceEnableJMX` – The ability to manage the connections is exposed when you define a `loadBalanceConnectionGroup`; but if you want to manage this externally, enable JMX by setting this property to `true`. This enables a JMX implementation, which exposes the management and monitoring operations of a connection group. Further, start your application with the `-Dcom.sun.management.jmxremote` JVM flag. You can then perform connect and perform operations using a JMX client such as `jconsole`.

Once a connection has been made using the correct connection properties, a number of monitoring properties are available:

- Current active host count.
- Current active physical connection count.
- Current active logical connection count.
- Total logical connections created.
- Total transaction count.

The following management operations can also be performed:

- Add host.
- Remove host.

The JMX interface, `com.mysql.jdbc.jmx.LoadBalanceConnectionGroupManagerMBean`, has the following methods:

- `int getActiveHostCount(String group);`
- `int getTotalHostCount(String group);`
- `long getTotalLogicalConnectionCount(String group);`
- `long getActiveLogicalConnectionCount(String group);`
- `long getActivePhysicalConnectionCount(String group);`
- `long getTotalPhysicalConnectionCount(String group);`
- `long getTotalTransactionCount(String group);`
- `void removeHost(String group, String host) throws SQLException;`
- `void stopNewConnectionsToHost(String group, String host) throws SQLException;`
- `void addHost(String group, String host, boolean forExisting);`
- `String getActiveHostsList(String group);`
- `String getRegisteredConnectionGroups();`

The `getRegisteredConnectionGroups()` method returns the names of all connection groups defined in that class loader.

You can test this setup with the following code:

```
public class Test {

    private static String URL = "jdbc:mysql:loadbalance://" +
        "localhost:3306,localhost:3310/test?" +
        "loadBalanceConnectionGroup=first&loadBalanceEnableJMX=true";

    public static void main(String[] args) throws Exception {
        new Thread(new Repeater()).start();
        new Thread(new Repeater()).start();
        new Thread(new Repeater()).start();
    }

    static Connection getNewConnection() throws SQLException, ClassNotFoundException {
        Class.forName("com.mysql.jdbc.Driver");
        return DriverManager.getConnection(URL, "root", "");
    }

    static void executeSimpleTransaction(Connection c, int conn, int trans){
        try {
            c.setAutoCommit(false);
            Statement s = c.createStatement();
            s.executeQuery("SELECT SLEEP(1) /* Connection: " + conn + ", transaction: " + trans + " */");
            c.commit();
```

```
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    public static class Repeater implements Runnable {
        public void run() {
            for(int i=0; i < 100; i++){
                try {
                    Connection c = getNewConnection();
                    for(int j=0; j < 10; j++){
                        executeSimpleTransaction(c, i, j);
                        Thread.sleep(Math.round(100 * Math.random()));
                    }
                    c.close();
                    Thread.sleep(100);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

After compiling, the application can be started with the `-Dcom.sun.management.jmxremote` flag, to enable remote management. `jconsole` can then be started. The `Test` main class will be listed by `jconsole`. Select this and click **Connect**. You can then navigate to the `com.mysql.jdbc.jmx.LoadBalanceConnectionGroupManager` bean. At this point, you can click on various operations and examine the returned result.

If you now had an additional instance of MySQL running on port 3309, you could ensure that Connector/J starts using it by using the `addHost()`, which is exposed in `jconsole`. Note that these operations can be performed dynamically without having to stop the application running.

For further information on the combination of load balancing and failover, see Section 8.4, “Advanced Load-balancing and Failover Configuration”.

8.3 Configuring Source/Replica Replication with Connector/J

This section describe a number of features of Connector/J's support for replication-aware deployments.

The replication is configured at the initial setup stage of the server connection by the connection URL, which has a similar format as the general URL for MySQL connection, but a specialized scheme:

```
jdbc:mysql:replication://[source host][:port],[replica host 1][:port][,[replica host 2][:port]]...[/[database]
[?propertyName1=propertyValue1[&propertyName2=propertyValue2]...]
```

Users may specify the property `allowMasterDownConnections=true` to allow `Connection` objects to be created even though no source hosts are reachable. Such `Connection` objects report they are read-only, and `isMasterConnection()` returns false for them. The `Connection` tests for available source hosts when `Connection.setReadOnly(false)` is called, throwing an SQLException if it cannot establish a connection to a source, or switching to a source connection if the host is available.

For Connector/J 5.1.38 and later, users may specify the property `allowSlavesDownConnections=true` to allow `Connection` objects to be created even though no replica hosts are reachable. A `Connection` then, at runtime, tests for available replica hosts when `Connection.setReadOnly(true)` is called (see explanation for the method below), throwing an SQLException if it cannot establish a connection to a replica, unless the property `readFromMasterWhenNoSlaves` is set to be “true” (see below for a description of the property).

Scaling out Read Load by Distributing Read Traffic to Replicas

Connector/J 3.1.7 and higher includes a variant of the driver that will automatically send queries to a read/write source, or a failover or round-robin loadbalanced set of replicas based on the state of `Connection.getReadOnly()`.

An application signals that it wants a transaction to be read-only by calling `Connection.setReadOnly(true)`. The replication-aware connection will use one of the replica connections, which are load-balanced per replica host using a round-robin scheme. A given connection is sticky to a replica until a transaction boundary command (a commit or rollback) is issued, or until the replica is removed from service. For Connector/J 5.1.38 and later, after calling `Connection.setReadOnly(true)`, if you want to allow connection to a source when no replicas are available, set the property `readFromMasterWhenNoSlaves` to “true.” Notice that the source host will be used in read-only state in those cases, as if it is a replica host. Also notice that setting `readFromMasterWhenNoSlaves=true` might result in an extra load for the source host in a transparent manner.

If you have a write transaction, or if you have a read that is time-sensitive (remember, replication in MySQL is asynchronous), set the connection to be not read-only, by calling `Connection.setReadOnly(false)` and the driver will ensure that further calls are sent to the source MySQL server. The driver takes care of propagating the current state of autocommit, isolation level, and catalog between all of the connections that it uses to accomplish this load balancing functionality.

To enable this functionality, use the `com.mysql.jdbc.ReplicationDriver` class when configuring your application server's connection pool or when creating an instance of a JDBC driver for your standalone application. Because it accepts the same URL format as the standard MySQL JDBC driver, `ReplicationDriver` does not currently work with `java.sql.DriverManager`-based connection creation unless it is the only MySQL JDBC driver registered with the `DriverManager` .

Here is a short example of how `ReplicationDriver` might be used in a standalone application:

```
import java.sql.Connection;
import java.sql.ResultSet;
import java.util.Properties;

import com.mysql.jdbc.ReplicationDriver;

public class ReplicationDriverDemo {

  public static void main(String[] args) throws Exception {
    ReplicationDriver driver = new ReplicationDriver();

    Properties props = new Properties();

    // We want this for failover on the replicas
    props.put("autoReconnect", "true");

    // We want to load balance between the replicas
    props.put("roundRobinLoadBalance", "true");

    props.put("user", "foo");
    props.put("password", "password");

    //
    // Looks like a normal MySQL JDBC url, with a
    // comma-separated list of hosts, the first
    // being the 'source', the rest being any number
    // of replicas that the driver will load balance against
    //
```

```
    Connection conn =
        driver.connect("jdbc:mysql:replication://source,replica1,replica2,replica3/test",
            props);

    //
    // Perform read/write work on the source
    // by setting the read-only flag to "false"
    //

    conn.setReadOnly(false);
    conn.setAutoCommit(false);
    conn.createStatement().executeUpdate("UPDATE some_table ....");
    conn.commit();

    //
    // Now, do a query from a replica, the driver automatically picks one
    // from the list
    //

    conn.setReadOnly(true);

    ResultSet rs =
      conn.createStatement().executeQuery("SELECT a,b FROM alt_table");

     .......
  }
}
```

Consider using the Load Balancing JDBC Pool (`lbpool`) tool, which provides a wrapper around the standard JDBC driver and enables you to use DB connection pools that includes checks for system failures and uneven load distribution. For more information, see Load Balancing JDBC Driver for MySQL (mysql-lbpool).

Support for Multiple-Source Replication Topographies

Since Connector/J 5.1.27, multi-source replication topographies are supported.

The connection URL for replication discussed earlier (i.e., in the format of `jdbc:mysql:replication://source,replica1,replica2,replica3/test`) assumes that the first (and only the first) host is the source. Supporting deployments with an arbitrary number of sources and replicas requires the URL syntax for specifying the hosts and the properties for specific hosts (which is discussed here) and the use of the property `type=[master|slave]`; for example:

```
jdbc:mysql:replication://address=(type=master)(host=source1host),address=(type=master)(host=source2host),addre
```

Connector/J uses a load-balanced connection internally for management of the source connections, which means that `ReplicationConnection`, when configured to use multiple sources, exposes the same options to balance load across source hosts as described in Section 8.2, “Configuring Load Balancing with Connector/J”.

Live Reconfiguration of Replication Topography

Since Connector/J 5.1.28, live management of replication host (single or multi-source) topographies is also supported. This enables users to promote replicas for Java applications without requiring an application restart.

The replication hosts are most effectively managed in the context of a replication connection group. A ReplicationConnectionGroup class represents a logical grouping of connections which can be managed together. There may be one or more such replication connection groups in a given Java class loader (there

can be an application with two different JDBC resources needing to be managed independently). This key class exposes host management methods for replication connections, and `ReplicationConnection` objects register themselves with the appropriate `ReplicationConnectionGroup` if a value for the new `replicationConnectionGroup` property is specified. The `ReplicationConnectionGroup` object tracks these connections until they are closed, and it is used to manipulate the hosts associated with these connections.

Some important methods related to host management include:

- `getMasterHosts()`: Returns a collection of strings representing the hosts configured as sources
- `getSlaveHosts()`: Returns a collection of strings representing the hosts configured as replicas
- `addSlaveHost(String host)`: Adds new host to pool of possible replica hosts for selection at start of new read-only workload
- `promoteSlaveToMaster(String host)`: Removes the host from the pool of potential replicas for future read-only processes (existing read-only process is allowed to continue to completion) and adds the host to the pool of potential source hosts
- `removeSlaveHost(String host, boolean closeGently)`: Removes the host (host name match must be exact) from the list of configured replicas; if `closeGently` is false, existing connections which have this host as currently active will be closed hardly (application should expect exceptions)
- `removeMasterHost(String host, boolean closeGently)`: Same as `removeSlaveHost()`, but removes the host from the list of configured sources

Some useful management metrics include:

- `getConnectionCountWithHostAsSlave(String host)`: Returns the number of ReplicationConnection objects that have the given host configured as a possible replica
- `getConnectionCountWithHostAsMaster(String host)`: Returns the number of ReplicationConnection objects that have the given host configured as a possible source
- `getNumberOfSlavesAdded()`: Returns the number of times a replica host has been dynamically added to the group pool
- `getNumberOfSlavesRemoved()`: Returns the number of times a replica host has been dynamically removed from the group pool
- `getNumberOfSlavePromotions()`: Returns the number of times a replica host has been promoted to a source
- `getTotalConnectionCount()`: Returns the number of ReplicationConnection objects which have been registered with this group
- `getActiveConnectionCount()`: Returns the number of ReplicationConnection objects currently being managed by this group

ReplicationConnectionGroupManager

`com.mysql.jdbc.ReplicationConnectionGroupManager` provides access to the replication connection groups, together with some utility methods.

- `getConnectionGroup(String groupName)`: Returns the `ReplicationConnectionGroup` object matching the groupName provided

The other methods in `ReplicationConnectionGroupManager` mirror those of `ReplicationConnectionGroup`, except that the first argument is a String group name. These methods will operate on all matching ReplicationConnectionGroups, which are helpful for removing a server from service and have it decommissioned across all possible `ReplicationConnectionGroups`.

These methods might be useful for in-JVM management of replication hosts if an application triggers topography changes. For managing host configurations from outside the JVM, JMX can be used.

Using JMX for Managing Replication Hosts

When Connector/J is started with `replicationEnableJMX=true` and a value set for the property `replicationConnectionGroup`, a JMX MBean will be registered, allowing manipulation of replication hosts by a JMX client. The MBean interface is defined in `com.mysql.jdbc.jmx.ReplicationGroupManagerMBean`, and leverages the `ReplicationConnectionGroupManager` static methods:

```
public abstract void addSlaveHost(String groupFilter, String host) throws SQLException;
public abstract void removeSlaveHost(String groupFilter, String host) throws SQLException;
public abstract void promoteSlaveToMaster(String groupFilter, String host) throws SQLException;
public abstract void removeMasterHost(String groupFilter, String host) throws SQLException;
public abstract String getMasterHostsList(String group);
public abstract String getSlaveHostsList(String group);
public abstract String getRegisteredConnectionGroups();
public abstract int getActiveMasterHostCount(String group);
public abstract int getActiveSlaveHostCount(String group);
public abstract int getSlavePromotionCount(String group);
public abstract long getTotalLogicalConnectionCount(String group);
public abstract long getActiveLogicalConnectionCount(String group);
```

8.4 Advanced Load-balancing and Failover Configuration

Connector/J provides a useful load-balancing implementation for MySQL Cluster or multi-source deployments, as explained in Section 8.2, “Configuring Load Balancing with Connector/J” and Support for Multiple-Source Replication Topographies. As of Connector/J 5.1.12, this same implementation is used for balancing load between read-only replicas with `ReplicationDriver`.

When trying to balance workload between multiple servers, the driver has to determine when it is safe to swap servers, doing so in the middle of a transaction, for example, could cause problems. It is important not to lose state information. For this reason, Connector/J will only try to pick a new server when one of the following happens:

1. At transaction boundaries (transactions are explicitly committed or rolled back).
2. A communication exception (SQL State starting with "08") is encountered.
3. When a `SQLException` matches conditions defined by user, using the extension points defined by the `loadBalanceSQLStateFailover`, `loadBalanceSQLExceptionSubclassFailover` or `loadBalanceExceptionChecker` properties.

The third condition revolves around three new properties introduced with Connector/J 5.1.13. It allows you to control which `SQLException`s trigger failover.

- `loadBalanceExceptionChecker` - The `loadBalanceExceptionChecker` property is really the key. This takes a fully-qualified class name which implements the new `com.mysql.jdbc.LoadBalanceExceptionChecker` interface. This interface is very simple, and you only need to implement the following method:

  ```
  public boolean shouldExceptionTriggerFailover(SQLException ex)
  ```

A `SQLException` is passed in, and a boolean returned. A value of `true` triggers a failover, `false` does not.

You can use this to implement your own custom logic. An example where this might be useful is when dealing with transient errors when using MySQL Cluster, where certain buffers may become overloaded. The following code snippet illustrates this:

```
public class NdbLoadBalanceExceptionChecker
 extends StandardLoadBalanceExceptionChecker {

 public boolean shouldExceptionTriggerFailover(SQLException ex) {
  return super.shouldExceptionTriggerFailover(ex)
    ||  checkNdbException(ex);
 }

 private boolean checkNdbException(SQLException ex){
 // Have to parse the message since most NDB errors
 // are mapped to the same DEMC.
  return (ex.getMessage().startsWith("Lock wait timeout exceeded") ||
  (ex.getMessage().startsWith("Got temporary error")
  && ex.getMessage().endsWith("from NDB")));
 }
}
```

The code above extends `com.mysql.jdbc.StandardLoadBalanceExceptionChecker`, which is the default implementation. There are a few convenient shortcuts built into this, for those who want to have some level of control using properties, without writing Java code. This default implementation uses the two remaining properties: `loadBalanceSQLStateFailover` and `loadBalanceSQLExceptionSubclassFailover`.

- `loadBalanceSQLStateFailover` - allows you to define a comma-delimited list of `SQLState` code prefixes, against which a `SQLException` is compared. If the prefix matches, failover is triggered. So, for example, the following would trigger a failover if a given `SQLException` starts with "00", or is "12345":

  ```
  loadBalanceSQLStateFailover=00,12345
  ```

- `loadBalanceSQLExceptionSubclassFailover` - can be used in conjunction with `loadBalanceSQLStateFailover` or on its own. If you want certain subclasses of `SQLException` to trigger failover, simply provide a comma-delimited list of fully-qualified class or interface names to check against. For example, if you want all `SQLTransientConnectionExceptions` to trigger failover, you would specify:

  ```
  loadBalanceSQLExceptionSubclassFailover=java.sql.SQLTransientConnectionException
  ```

While the three failover conditions enumerated earlier suit most situations, if `autocommit` is enabled, Connector/J never re-balances, and continues using the same physical connection. This can be problematic, particularly when load-balancing is being used to distribute read-only load across multiple replicas. However, Connector/J can be configured to re-balance after a certain number of statements are executed, when `autocommit` is enabled. This functionality is dependent upon the following properties:

- `loadBalanceAutoCommitStatementThreshold` – defines the number of matching statements which will trigger the driver to potentially swap physical server connections. The default value, 0, retains the behavior that connections with `autocommit` enabled are never balanced.

- `loadBalanceAutoCommitStatementRegex` – the regular expression against which statements must match. The default value, blank, matches all statements. So, for example, using the following properties will cause Connector/J to re-balance after every third statement that contains the string “test”:

```
loadBalanceAutoCommitStatementThreshold=3
loadBalanceAutoCommitStatementRegex=.*test.*
```

`loadBalanceAutoCommitStatementRegex` can prove useful in a number of situations. Your application may use temporary tables, server-side session state variables, or connection state, where letting the driver arbitrarily swap physical connections before processing is complete could cause data loss or other problems. This allows you to identify a trigger statement that is only executed when it is safe to swap physical connections.

Chapter 9 Using the Connector/J Interceptor Classes

An interceptor is a software design pattern that provides a transparent way to extend or modify some aspect of a program, similar to a user exit. No recompiling is required. With Connector/J, the interceptors are enabled and disabled by updating the connection string to refer to different sets of interceptor classes that you instantiate.

The connection properties that control the interceptors are explained in Section 5.3, “Configuration Properties for Connector/J”:

- `connectionLifecycleInterceptors`, where you specify the fully qualified names of classes that implement the `com.mysql.jdbc.ConnectionLifecycleInterceptor` interface. In these kinds of interceptor classes, you might log events such as rollbacks, measure the time between transaction start and end, or count events such as calls to `setAutoCommit()`.

- `exceptionInterceptors`, where you specify the fully qualified names of classes that implement the `com.mysql.jdbc.ExceptionInterceptor` interface. In these kinds of interceptor classes, you might add extra diagnostic information to exceptions that can have multiple causes or indicate a problem with server settings. Because `exceptionInterceptors` classes are only called when handling a `SQLException` thrown from Connector/J code, they can be used even in production deployments without substantial performance overhead.

- `statementInterceptors`, where you specify the fully qualified names of classes that implement the `com.mysql.jdbc.StatementInterceptorV2` interface. In these kinds of interceptor classes, you might change or augment the processing done by certain kinds of statements, such as automatically checking for queried data in a `memcached` server, rewriting slow queries, logging information about statement execution, or route requests to remote servers.

Chapter 10 Using Logging Frameworks with SLF4J

Besides its default logger `com.mysql.cj.log.StandardLogger`, which logs to `stderr`, Connector/J supports the SLF4J logging facade, allowing end users of applications using Connector/J to plug in logging frameworks of their own choices at deployment time. Popular logging frameworks such as `java.util.logging`, `logback`, and `log4j` are supported by SLF4J. Follow these requirements to use a logging framework with SLF4J and Connector/J:

- In the development environment:
  - Install on your system `slf4j-api-x.y.z.jar` (available at https://www.slf4j.org/download.html) and add it to the Java classpath.
  - In the code of your application, obtain an `SLF4JLogger` as a `Log` instantiated within a `MysqlConnection`, and then use the `Log` instance for your logging.
- On the deployment system:
  - Install on your system `slf4j-api-x.y.z.jar` and add it to the Java classpath
  - Install on your system the SLF4J binding for the logging framework of your choice and add it to your Java classpath. SLF4J bindings are available at, for example, https://www.slf4j.org/manual.html#swapping.

    Note

    Do not put more than one SLF4J binding in you Java classpath. Switch from one logging framework to another by removing a binding and adding a new one to the classpath.
  - Install the logging framework of your choice on your system and add it to the Java classpath.
  - Configure the logging framework of your choice. This often consists of setting up appenders or handlers for log messages using a configuration file; see your logging framework's documentation for details.
  - When connecting the application to the MySQL Server, set the Connector/J connection property `logger` to `Slf4JLogger`.

The log category name used by Connector/J with SLF4J is `MySQL`. See the SLF4J user manual for more details about using SLF4J, including discussions on Maven dependency and bindings. Here is a sample code for using SLF4J with Connector/J:

```
import java.sql.DriverManager;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import com.mysql.jdbc.MySQLConnection;
import com.mysql.jdbc.log.Log;

public class JDBCDemo {

  public static void main(String[] args) {

    Connection conn = null;
    Statement statement = null;
    ResultSet resultSet = null;
    Log logger = null;
```

```
  try {
     // Database parameters
     String url = "jdbc:mysql://myexample.com:3306/pets?logger=Slf4JLogger&explainSlowQueries=true";
     String user = "user";
     String password = "password";
     // create a connection to the database
     conn = DriverManager.getConnection(url, user, password);
     logger = ((MySQLConnection)conn).getLog();
  }
  catch (SQLException e) {
     System.err.println(e.getMessage());
     System.exit(1);
   }

  try {
     statement = conn.createStatement();
     resultSet = statement.executeQuery("SELECT * FROM pets.dogs");
     while(resultSet.next()){
       System.out.printf("%d\t%s\t%s\t %4$ty.%4$tm.%4$td \n",
       resultSet.getInt(1),
       resultSet.getString(2),
       resultSet.getString(3),
       resultSet.getDate(4));
     }
  }
  catch(SQLException e) {
     logger.logWarn("Warning: Select failed!");
  }

}

}
```

If you want to use, for example, Log4j 1.2.17 as your logging framework when running this program, use its binding to SLF4J: put `slf4j-api-1.7.28.jar` (SLF4J API module), `slf4j-log4j12-1.7.28.jar` (SLF4J's binding for Log4J 1.2), and `log4j-1.2.17.jar` (Log4J library) in your Java classpath.

Here is output of the program when the SELECT statement failed:

```
[2019-09-05 12:06:19,624] WARN     0[main] - WARN MySQL - Warning: Select failed!
```

Chapter 11 Using Connector/J with Tomcat

The following instructions are based on the instructions for Tomcat-5.x, available at http://tomcat.apache.org/tomcat-5.5-doc/jndi-datasource-examples-howto.html which is current at the time this document was written.

First, install the `.jar` file that comes with Connector/J in `$CATALINA_HOME/common/lib` so that it is available to all applications installed in the container.

Next, configure the JNDI DataSource by adding a declaration resource to `$CATALINA_HOME/conf/server.xml` in the context that defines your web application:

```
<Context ....>

...

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

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

  <parameter>
    <name>maxActive</name>
    <value>10</value>
  </parameter>

  <parameter>
    <name>maxIdle</name>
    <value>5</value>
  </parameter>

  <parameter>
    <name>validationQuery</name>
    <value>SELECT 1</value>
  </parameter>

  <parameter>
    <name>testOnBorrow</name>
    <value>true</value>
  </parameter>

  <parameter>
    <name>testWhileIdle</name>
    <value>true</value>
  </parameter>

  <parameter>
    <name>timeBetweenEvictionRunsMillis</name>
    <value>10000</value>
  </parameter>

  <parameter>
    <name>minEvictableIdleTimeMillis</name>
    <value>60000</value>
  </parameter>

  <parameter>
   <name>username</name>
   <value>someuser</value>
```

```
    </parameter>

    <parameter>
     <name>password</name>
     <value>somepass</value>
    </parameter>

    <parameter>
       <name>driverClassName</name>
       <value>com.mysql.jdbc.Driver</value>
    </parameter>

    <parameter>
      <name>url</name>
      <value>jdbc:mysql://localhost:3306/test</value>
    </parameter>

  </ResourceParams>
</Context>
```

Note that Connector/J 5.1.3 introduced a facility whereby, rather than use a `validationQuery` value of `SELECT 1`, it is possible to use `validationQuery` with a value set to `/* ping */`. This sends a ping to the server which then returns a fake result set. This is a lighter weight solution. It also has the advantage that if using `ReplicationConnection` or `LoadBalancedConnection` type connections, the ping will be sent across all active connections. The following XML snippet illustrates how to select this option:

```
<parameter>
 <name>validationQuery</name>
 <value>/* ping */</value>
</parameter>
```

Note that `/* ping */` has to be specified exactly.

In general, follow the installation instructions that come with your version of Tomcat, as the way you configure datasources in Tomcat changes from time to time, and if you use the wrong syntax in your XML file, you will most likely end up with an exception similar to the following:

```
Error: java.sql.SQLException: Cannot load JDBC driver class 'null ' SQL
state: null
```

Note that the auto-loading of drivers having the `META-INF/service/java.sql.Driver` class in JDBC 4.0 and above causes an improper undeployment of the Connector/J driver in Tomcat on Windows. Namely, the Connector/J jar remains locked. This is an initialization problem that is not related to the driver. The possible workarounds, if viable, are as follows: use "`antiResourceLocking=true`" as a Tomcat Context attribute, or remove the `META-INF/` directory.

Chapter 12 Using Connector/J with JBoss

> **Note**
>
> JBoss AS is no longer maintained by its owner. Oracle does *not* provide support for JBoss AS. The Connector/J integration for JBoss is provided "as-is" and may stop functioning or be removed at anytime without notice.

These instructions cover JBoss-4.x. To make the JDBC driver classes available to the application server, put the JBoss common JDBC wrapper JAR archive (available from, for example, the Maven Central Repository at http://central.maven.org/maven2/jboss/jboss-common-jdbc-wrapper/) into the `lib` directory for your server configuration (which is usually called `default`). Then, in the same configuration directory, in the subdirectory named `deploy`, create a datasource configuration file that ends with `-ds.xml`, which tells JBoss to deploy this file as a JDBC Datasource. The file should have the following contents:

```
<datasources>
    <local-tx-datasource>

        <jndi-name>MySQLDB</jndi-name>
        <connection-url>jdbc:mysql://localhost:3306/dbname</connection-url>
        <driver-class>com.mysql.jdbc.Driver</driver-class>
        <user-name>user</user-name>
        <password>pass</password>

        <min-pool-size>5</min-pool-size>

        <max-pool-size>20</max-pool-size>

        <idle-timeout-minutes>5</idle-timeout-minutes>

        <exception-sorter-class-name>
  com.mysql.jdbc.integration.jboss.ExtendedMysqlExceptionSorter
        </exception-sorter-class-name>
        <valid-connection-checker-class-name>
  com.mysql.jdbc.integration.jboss.MysqlValidConnectionChecker
        </valid-connection-checker-class-name>

    </local-tx-datasource>
</datasources>
```

Chapter 13 Using Connector/J with Spring

Table of Contents

The Spring Framework is a Java-based application framework designed for assisting in application design by providing a way to configure components. The technique used by Spring is a well known design pattern called Dependency Injection (see Inversion of Control Containers and the Dependency Injection pattern). This article will focus on Java-oriented access to MySQL databases with Spring 2.0. For those wondering, there is a .NET port of Spring appropriately named Spring.NET.

Spring is not only a system for configuring components, but also includes support for aspect oriented programming (AOP). This is one of the main benefits and the foundation for Spring's resource and transaction management. Spring also provides utilities for integrating resource management with JDBC and Hibernate.

For the examples in this section the MySQL world sample database will be used. The first task is to set up a MySQL data source through Spring. Components within Spring use the “bean” terminology. For example, to configure a connection to a MySQL server supporting the world sample database, you might use:

```
<util:map id="dbProps">
    <entry key="db.driver" value="com.mysql.jdbc.Driver"/>
    <entry key="db.jdbcurl" value="jdbc:mysql://localhost/world"/>
    <entry key="db.username" value="myuser"/>
    <entry key="db.password" value="mypass"/>
</util:map>
```

In the above example, we are assigning values to properties that will be used in the configuration. For the datasource configuration:

```
<bean id="dataSource"
       class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="${db.driver}"/>
    <property name="url" value="${db.jdbcurl}"/>
    <property name="username" value="${db.username}"/>
    <property name="password" value="${db.password}"/>
</bean>
```

The placeholders are used to provide values for properties of this bean. This means that you can specify all the properties of the configuration in one place instead of entering the values for each property on each bean. We do, however, need one more bean to pull this all together. The last bean is responsible for actually replacing the placeholders with the property values.

```
<bean
 class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="properties" ref="dbProps"/>
</bean>
```

Now that we have our MySQL data source configured and ready to go, we write some Java code to access it. The example below will retrieve three random cities and their corresponding country using the data source we configured with Spring.

```
// Create a new application context. this processes the Spring config
ApplicationContext ctx =
    new ClassPathXmlApplicationContext("ex1appContext.xml");
// Retrieve the data source from the application context
    DataSource ds = (DataSource) ctx.getBean("dataSource");
// Open a database connection using Spring's DataSourceUtils
Connection c = DataSourceUtils.getConnection(ds);
try {
    // retrieve a list of three random cities
    PreparedStatement ps = c.prepareStatement(
        "select City.Name as 'City', Country.Name as 'Country' " +
        "from City inner join Country on City.CountryCode = Country.Code " +
        "order by rand() limit 3");
    ResultSet rs = ps.executeQuery();
    while(rs.next()) {
        String city = rs.getString("City");
        String country = rs.getString("Country");
        System.out.printf("The city %s is in %s%n", city, country);
    }
} catch (SQLException ex) {
    // something has failed and we print a stack trace to analyse the error
    ex.printStackTrace();
    // ignore failure closing connection
    try { c.close(); } catch (SQLException e) { }
} finally {
    // properly release our connection
    DataSourceUtils.releaseConnection(c, ds);
}
```

This is very similar to normal JDBC access to MySQL with the main difference being that we are using DataSourceUtils instead of the DriverManager to create the connection.

While it may seem like a small difference, the implications are somewhat far reaching. Spring manages this resource in a way similar to a container managed data source in a J2EE application server. When a connection is opened, it can be subsequently accessed in other parts of the code if it is synchronized with a transaction. This makes it possible to treat different parts of your application as transactional instead of passing around a database connection.

13.1 Using `JdbcTemplate`

Spring makes extensive use of the Template method design pattern (see Template Method Pattern). Our immediate focus will be on the `JdbcTemplate` and related classes, specifically `NamedParameterJdbcTemplate`. The template classes handle obtaining and releasing a connection for data access when one is needed.

The next example shows how to use `NamedParameterJdbcTemplate` inside of a DAO (Data Access Object) class to retrieve a random city given a country code.

```
public class Ex2JdbcDao {
     /**
     * Data source reference which will be provided by Spring.
     */
     private DataSource dataSource;

     /**
     * Our query to find a random city given a country code. Notice
     * the ":country" parameter toward the end. This is called a
     * named parameter.
     */
```

```
    private String queryString = "select Name from City " +
       "where CountryCode = :country order by rand() limit 1";

    /**
    * Retrieve a random city using Spring JDBC access classes.
    */
    public String getRandomCityByCountryCode(String cntryCode) {
        // A template that permits using queries with named parameters
        NamedParameterJdbcTemplate template =
        new NamedParameterJdbcTemplate(dataSource);
        // A java.util.Map is used to provide values for the parameters
        Map params = new HashMap();
        params.put("country", cntryCode);
        // We query for an Object and specify what class we are expecting
        return (String)template.queryForObject(queryString, params, String.class);
    }

   /**
   * A JavaBean setter-style method to allow Spring to inject the data source.
   * @param dataSource
   */
   public void setDataSource(DataSource dataSource) {
       this.dataSource = dataSource;
   }
}
```

The focus in the above code is on the `getRandomCityByCountryCode()` method. We pass a country code and use the `NamedParameterJdbcTemplate` to query for a city. The country code is placed in a Map with the key "country", which is the parameter is named in the SQL query.

To access this code, you need to configure it with Spring by providing a reference to the data source.

```
<bean id="dao" class="code.Ex2JdbcDao">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

At this point, we can just grab a reference to the DAO from Spring and call `getRandomCityByCountryCode()`.

```
    // Create the application context
    ApplicationContext ctx =
    new ClassPathXmlApplicationContext("ex2appContext.xml");
    // Obtain a reference to our DAO
    Ex2JdbcDao dao = (Ex2JdbcDao) ctx.getBean("dao");

    String countryCode = "USA";

    // Find a few random cities in the US
    for(int i = 0; i < 4; ++i)
        System.out.printf("A random city in %s is %s%n", countryCode,
            dao.getRandomCityByCountryCode(countryCode));
```

This example shows how to use Spring's JDBC classes to completely abstract away the use of traditional JDBC classes including `Connection` and `PreparedStatement`.

13.2 Transactional JDBC Access

You might be wondering how we can add transactions into our code if we do not deal directly with the JDBC classes. Spring provides a transaction management package that not only replaces JDBC transaction management, but also enables declarative transaction management (configuration instead of code).

To use transactional database access, we will need to change the storage engine of the tables in the world database. The downloaded script explicitly creates MyISAM tables which do not support transactional semantics. The InnoDB storage engine does support transactions and this is what we will be using. We can change the storage engine with the following statements.

```
ALTER TABLE City ENGINE=InnoDB;
ALTER TABLE Country ENGINE=InnoDB;
ALTER TABLE CountryLanguage ENGINE=InnoDB;
```

A good programming practice emphasized by Spring is separating interfaces and implementations. What this means is that we can create a Java interface and only use the operations on this interface without any internal knowledge of what the actual implementation is. We will let Spring manage the implementation and with this it will manage the transactions for our implementation.

First you create a simple interface:

```
public interface Ex3Dao {
    Integer createCity(String name, String countryCode,
    String district, Integer population);
}
```

This interface contains one method that will create a new city record in the database and return the id of the new record. Next you need to create an implementation of this interface.

```
public class Ex3DaoImpl implements Ex3Dao {
    protected DataSource dataSource;
    protected SqlUpdate updateQuery;
    protected SqlFunction idQuery;

    public Integer createCity(String name, String countryCode,
        String district, Integer population) {
            updateQuery.update(new Object[] { name, countryCode,
                   district, population });
            return getLastId();
        }

    protected Integer getLastId() {
        return idQuery.run();
    }
}
```

You can see that we only operate on abstract query objects here and do not deal directly with the JDBC API. Also, this is the complete implementation. All of our transaction management will be dealt with in the configuration. To get the configuration started, we need to create the DAO.

```
<bean id="dao" class="code.Ex3DaoImpl">
    <property name="dataSource" ref="dataSource"/>
    <property name="updateQuery">...</property>
    <property name="idQuery">...</property>
</bean>
```

Now you need to set up the transaction configuration. The first thing you must do is create transaction manager to manage the data source and a specification of what transaction properties are required for the dao methods.

```
<bean id="transactionManager"
  class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

```
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <tx:attributes>
        <tx:method name="*"/>
    </tx:attributes>
</tx:advice>
```

The preceding code creates a transaction manager that handles transactions for the data source provided to it. The `txAdvice` uses this transaction manager and the attributes specify to create a transaction for all methods. Finally you need to apply this advice with an AOP pointcut.

```
<aop:config>
    <aop:pointcut id="daoMethods"
        expression="execution(* code.Ex3Dao.*(..))"/>
     <aop:advisor advice-ref="txAdvice" pointcut-ref="daoMethods"/>
</aop:config>
```

This basically says that all methods called on the `Ex3Dao` interface will be wrapped in a transaction. To make use of this, you only have to retrieve the `dao` from the application context and call a method on the `dao` instance.

```
Ex3Dao dao = (Ex3Dao) ctx.getBean("dao");
Integer id = dao.createCity(name,  countryCode, district, pop);
```

We can verify from this that there is no transaction management happening in our Java code and it is all configured with Spring. This is a very powerful notion and regarded as one of the most beneficial features of Spring.

13.3 Connection Pooling with Spring

In many situations, such as web applications, there will be a large number of small database transactions. When this is the case, it usually makes sense to create a pool of database connections available for web requests as needed. Although MySQL does not spawn an extra process when a connection is made, there is still a small amount of overhead to create and set up the connection. Pooling of connections also alleviates problems such as collecting large amounts of sockets in the `TIME_WAIT` state.

Setting up pooling of MySQL connections with Spring is as simple as changing the data source configuration in the application context. There are a number of configurations that we can use. The first example is based on the Jakarta Commons DBCP library. The example below replaces the source configuration that was based on `DriverManagerDataSource` with DBCP's BasicDataSource.

```
<bean id="dataSource" destroy-method="close"
  class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName" value="${db.driver}"/>
    <property name="url" value="${db.jdbcurl}"/>
    <property name="username" value="${db.username}"/>
    <property name="password" value="${db.password}"/>
    <property name="initialSize" value="3"/>
</bean>
```

The configuration of the two solutions is very similar. The difference is that DBCP will pool connections to the database instead of creating a new connection every time one is requested. We have also set a parameter here called `initialSize`. This tells DBCP that we want three connections in the pool when it is created.

Another way to configure connection pooling is to configure a data source in our J2EE application server. Using JBoss as an example, you can set up the MySQL connection pool by creating a file called `mysql-`

`local-ds.xml` and placing it in the server/default/deploy directory in JBoss. Once we have this setup, we can use JNDI to look it up. With Spring, this lookup is very simple. The data source configuration looks like this.

```
<jee:jndi-lookup id="dataSource" jndi-name="java:MySQL_DS"/>
```

Chapter 14 Using Connector/J with GlassFish

Table of Contents

This section explains how to use MySQL Connector/J with GlassFish ™ Server Open Source Edition 3.0.1. GlassFish can be downloaded from the GlassFish website.

Once GlassFish is installed, make sure it can access MySQL Connector/J. To do this, copy the MySQL Connector/J `jar` file to the *`domain-dir`*`/lib` directory. For example, copy `mysql-connector-java-5.1.30-bin.jar` to `C:\`*`glassfish-install-path`*`\domains\`*`domain-name`*`\lib`. Restart the GlassFish Application Server. For more information, see “Integrating the JDBC Driver” in *GlassFish Server Open Source Edition Administration Guide*, available at GlassFish Server Documentation.

You are now ready to create JDBC Connection Pools and JDBC Resources.

Creating a Connection Pool

1. In the GlassFish Administration Console, using the navigation tree navigate to **Resources**, **JDBC**, **Connection Pools**.
2. In the **JDBC Connection Pools** frame click **New**. You will enter a two step wizard.
3. In the **Name** field under **General Settings** enter the name for the connection pool, for example enter **`MySQLConnPool`**.
4. In the **Resource Type** field, select `javax.sql.DataSource` from the drop-down listbox.
5. In the **Database Vendor** field, select `MySQL` from the drop-down listbox. Click **Next** to go to the next page of the wizard.
6. You can accept the default settings for General Settings, Pool Settings and Transactions for this example. Scroll down to Additional Properties.
7. In Additional Properties you will need to ensure the following properties are set:
   - **ServerName** - The server to connect to. For local testing this will be `localhost`.
   - **User** - The user name with which to connect to MySQL.
   - **Password** - The corresponding password for the user.
   - **DatabaseName** - The database to connect to, for example the sample MySQL database `World`.
8. Click **Finish** to exit the wizard. You will be taken to the **JDBC Connection Pools** page where all current connection pools, including the one you just created, will be displayed.
9. In the **JDBC Connection Pools** frame click on the connection pool you just created. Here, you can review and edit information about the connection pool. Because Connector/J does not support optimized validation queries, go to the **Advanced** tab, and under Connection Validation, configure the following settings:

- **Connection Validation** - select **Required**.
- **Validation Method** - select **table** from the drop-down menu.
- **Table Name** - enter `DUAL`.

10. To test your connection pool click the **Ping** button at the top of the frame. A message will be displayed confirming correct operation or otherwise. If an error message is received recheck the previous steps, and ensure that MySQL Connector/J has been correctly copied into the previously specified location.

Now that you have created a connection pool you will also need to create a JDBC Resource (data source) for use by your application.

Creating a JDBC Resource

Your Java application will usually reference a data source object to establish a connection with the database. This needs to be created first using the following procedure.

- Using the navigation tree in the GlassFish Administration Console, navigate to **Resources**, **JDBC**, **JDBC Resources**. A list of resources will be displayed in the **JDBC Resources** frame.
- Click **New**. The **New JDBC Resource** frame will be displayed.
- In the **JNDI Name** field, enter the JNDI name that will be used to access this resource, for example enter **`jdbc/MySQLDataSource`**.
- In the **Pool Name** field, select a connection pool you want this resource to use from the drop-down listbox.
- Optionally, you can enter a description into the **Description** field.
- Additional properties can be added if required.
- Click **OK** to create the new JDBC resource. The **JDBC Resources** frame will list all available JDBC Resources.

14.1 A Simple JSP Application with GlassFish, Connector/J and MySQL

This section shows how to deploy a simple JSP application on GlassFish, that connects to a MySQL database.

This example assumes you have already set up a suitable Connection Pool and JDBC Resource, as explained in the preceding sections. It is also assumed you have a sample database installed, such as `world`.

The main application code, `index.jsp` is presented here:

```
<%@ page import="java.sql.*, javax.sql.*, java.io.*, javax.naming.*" %>
<html>
<head><title>Hello world from JSP</title></head>
<body>
<%
  InitialContext ctx;
  DataSource ds;
  Connection conn;
  Statement stmt;
```

```
  ResultSet rs;

  try {
    ctx = new InitialContext();
    ds = (DataSource) ctx.lookup("java:comp/env/jdbc/MySQLDataSource");
    //ds = (DataSource) ctx.lookup("jdbc/MySQLDataSource");
    conn = ds.getConnection();
    stmt = conn.createStatement();
    rs = stmt.executeQuery("SELECT * FROM Country");

    while(rs.next()) {
%>
    <h3>Name: <%= rs.getString("Name") %></h3>
    <h3>Population: <%= rs.getString("Population") %></h3>
<%
    }
  }
  catch (SQLException se) {
%>
    <%= se.getMessage() %>
<%
  }
  catch (NamingException ne) {
%>
    <%= ne.getMessage() %>
<%
  }
%>
</body>
</html>
```

In addition two XML files are required: `web.xml`, and `sun-web.xml`. There may be other files present, such as classes and images. These files are organized into the directory structure as follows:

```
index.jsp
WEB-INF
   |
   - web.xml
   - sun-web.xml
```

The code for `web.xml` is:

```
<?xml version="1.0" encoding="utf-8"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee" xmlns:xsi="http://www.w3.org/2001/XMLSchema
  <display-name>HelloWebApp</display-name>
  <distributable/>
  <resource-ref>
    <res-ref-name>jdbc/MySQLDataSource</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
    <res-sharing-scope>Shareable</res-sharing-scope>
  </resource-ref>
</web-app>
```

The code for `sun-web.xml` is:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE sun-web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Application Server 8.1 Servlet 2.4//EN" "http
<sun-web-app>
  <context-root>HelloWebApp</context-root>
  <resource-ref>
    <res-ref-name>jdbc/MySQLDataSource</res-ref-name>
```

```
    <jndi-name>jdbc/MySQLDataSource</jndi-name>
  </resource-ref>
</sun-web-app>
```

These XML files illustrate a very important aspect of running JDBC applications on GlassFish. On GlassFish it is important to map the string specified for a JDBC resource to its JNDI name, as set up in the GlassFish administration console. In this example, the JNDI name for the JDBC resource, as specified in the GlassFish Administration console when creating the JDBC Resource, was `jdbc/MySQLDataSource`. This must be mapped to the name given in the application. In this example the name specified in the application, `jdbc/MySQLDataSource`, and the JNDI name, happen to be the same, but this does not necessarily have to be the case. Note that the XML element <res-ref-name> is used to specify the name as used in the application source code, and this is mapped to the JNDI name specified using the <jndi-name> element, in the file `sun-web.xml`. The resource also has to be created in the `web.xml` file, although the mapping of the resource to a JNDI name takes place in the `sun-web.xml` file.

If you do not have this mapping set up correctly in the XML files you will not be able to lookup the data source using a JNDI lookup string such as:

```
ds = (DataSource) ctx.lookup("java:comp/env/jdbc/MySQLDataSource");
```

You will still be able to access the data source directly using:

```
ds = (DataSource) ctx.lookup("jdbc/MySQLDataSource");
```

With the source files in place, in the correct directory structure, you are ready to deploy the application:

1. In the navigation tree, navigate to **Applications** - the **Applications** frame will be displayed. Click **Deploy**.

2. You can now deploy an application packaged into a single WAR file from a remote client, or you can choose a packaged file or directory that is locally accessible to the server. If you are simply testing an application locally you can simply point GlassFish at the directory that contains your application, without needing to package the application into a WAR file.

3. Now select the application type from the **Type** drop-down listbox, which in this example is `Web application`.

4. Click OK.

Now, when you navigate to the **Applications** frame, you will have the option to **Launch**, **Redeploy**, or **Restart** your application. You can test your application by clicking **Launch**. The application will connection to the MySQL database and display the Name and Population of countries in the `Country` table.

14.2 A Simple Servlet with GlassFish, Connector/J and MySQL

This section describes a simple servlet that can be used in the GlassFish environment to access a MySQL database. As with the previous section, this example assumes the sample database `world` is installed.

The project is set up with the following directory structure:

```
index.html
WEB-INF
   |
   - web.xml
   - sun-web.xml
   - classes
        |
        - HelloWebServlet.java
```

```
        - HelloWebServlet.class
```

The code for the servlet, located in `HelloWebServlet.java`, is as follows:

```
import javax.servlet.http.*;
import javax.servlet.*;
import java.io.*;
import java.sql.*;
import javax.sql.*;
import javax.naming.*;

public class HelloWebServlet extends HttpServlet {

  InitialContext ctx = null;
  DataSource ds = null;
  Connection conn = null;
  PreparedStatement ps = null;
  ResultSet rs = null;

  String sql = "SELECT Name, Population FROM Country WHERE Name=?";

  public void init () throws ServletException {
    try {
      ctx = new InitialContext();
      ds = (DataSource) ctx.lookup("java:comp/env/jdbc/MySQLDataSource");
      conn = ds.getConnection();
      ps = conn.prepareStatement(sql);
    }
    catch (SQLException se) {
      System.out.println("SQLException: "+se.getMessage());
    }
    catch (NamingException ne) {
      System.out.println("NamingException: "+ne.getMessage());
    }
  }

  public void destroy () {
    try {
      if (rs != null)
        rs.close();
      if (ps != null)
        ps.close();
      if (conn != null)
        conn.close();
      if (ctx != null)
        ctx.close();
    }
    catch (SQLException se) {
      System.out.println("SQLException: "+se.getMessage());
    }
    catch (NamingException ne) {
      System.out.println("NamingException: "+ne.getMessage());
    }
  }

  public void doPost(HttpServletRequest req, HttpServletResponse resp){
    try {
      String country_name = req.getParameter("country_name");
      resp.setContentType("text/html");
      PrintWriter writer = resp.getWriter();
      writer.println("<html><body>");
      writer.println("<p>Country: "+country_name+"</p>");
      ps.setString(1, country_name);
      rs = ps.executeQuery();
      if (!rs.next()){
        writer.println("<p>Country does not exist!</p>");
```

```
      }
      else {
        rs.beforeFirst();
        while(rs.next()) {
          writer.println("<p>Name: "+rs.getString("Name")+"</p>");
          writer.println("<p>Population: "+rs.getString("Population")+"</p>");
        }
      }
      writer.println("</body></html>");
      writer.close();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }

  public void doGet(HttpServletRequest req, HttpServletResponse resp){
    try {
      resp.setContentType("text/html");
      PrintWriter writer = resp.getWriter();
      writer.println("<html><body>");
      writer.println("<p>Hello from servlet doGet()</p>");
      writer.println("</body></html>");
      writer.close();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

In the preceding code a basic `doGet()` method is implemented, but is not used in the example. The code to establish the connection with the database is as shown in the previous example, Section 14.1, "A Simple JSP Application with GlassFish, Connector/J and MySQL", and is most conveniently located in the servlet `init()` method. The corresponding freeing of resources is located in the destroy method. The main functionality of the servlet is located in the `doPost()` method. If the user enters into the input form a country name that can be located in the database, the population of the country is returned. The code is invoked using a POST action associated with the input form. The form is defined in the file `index.html`:

```
<html>
  <head><title>HelloWebServlet</title></head>

  <body>
    <h1>HelloWebServlet</h1>

    <p>Please enter country name:</p>

    <form action="HelloWebServlet" method="POST">
      <input type="text" name="country_name" length="50" />
      <input type="submit" value="Submit" />
    </form>

  </body>
</html>
```

The XML files `web.xml` and `sun-web.xml` are as for the example in the preceding section, Section 14.1, "A Simple JSP Application with GlassFish, Connector/J and MySQL", no additional changes are required.

When compiling the Java source code, you will need to specify the path to the file `javaee.jar`. On Windows, this can be done as follows:

```
shell> javac -classpath c:\glassfishv3\glassfish\lib\javaee.jar HelloWebServlet.java
```

Once the code is correctly located within its directory structure, and compiled, the application can be deployed in GlassFish. This is done in exactly the same way as described in the preceding section, Section 14.1, “A Simple JSP Application with GlassFish, Connector/J and MySQL”.

Once deployed the application can be launched from within the GlassFish Administration Console. Enter a country name such as “England”, and the application will return “Country does not exist!”. Enter “France”, and the application will return a population of 59225700.

Chapter 15 Troubleshooting Connector/J Applications

This section explains the symptoms and resolutions for the most commonly encountered issues with applications using MySQL Connector/J.

Questions

- 15.1: When I try to connect to the database with MySQL Connector/J, I get the following exception:

  ```
  SQLException: Server configuration denies access to data source
  SQLState: 08001
  VendorError: 0
  ```

  What is going on? I can connect just fine with the MySQL command-line client.

- 15.2: My application throws an SQLException 'No Suitable Driver'. Why is this happening?

- 15.3: I'm trying to use MySQL Connector/J in an applet or application and I get an exception similar to:

  ```
  SQLException: Cannot connect to MySQL server on host:3306.
  Is there a MySQL server running on the machine/port you
  are trying to connect to?

  (java.security.AccessControlException)
  SQLState: 08S01
  VendorError: 0
  ```

- 15.4: I have a servlet/application that works fine for a day, and then stops working overnight

- 15.5: I'm trying to use JDBC 2.0 updatable result sets, and I get an exception saying my result set is not updatable.

- 15.6: I cannot connect to the MySQL server using Connector/J, and I'm sure the connection parameters are correct.

- 15.7: My application is deployed through JBoss and I am using transactions to handle the statements on the MySQL database. Under heavy loads, I am getting an error and stack trace, but these only occur after a fixed period of heavy activity.

- 15.8: When using `gcj`, a `java.io.CharConversionException` exception is raised when working with certain character sequences.

- 15.9: Updating a table that contains a primary key that is either `FLOAT` or compound primary key that uses `FLOAT` fails to update the table and raises an exception.

- 15.10: You get an `ER_NET_PACKET_TOO_LARGE` exception, even though the binary blob size you want to insert using JDBC is safely below the `max_allowed_packet` size.

- 15.11: What should you do if you receive error messages similar to the following: “Communications link failure – Last packet sent to the server was X ms ago”?

- 15.12: Why does Connector/J not reconnect to MySQL and re-issue the statement after a communication failure, instead of throwing an Exception, even though I use the `autoReconnect` connection string option?

- 15.13: How can I use 3-byte UTF8 with Connector/J?

- 15.14: How can I use 4-byte UTF8 (`utf8mb4`) with Connector/J?

- 15.15: Using `useServerPrepStmts=false` and certain character encodings can lead to corruption when inserting BLOBs. How can this be avoided?

Questions and Answers

15.1: When I try to connect to the database with MySQL Connector/J, I get the following exception:

```
SQLException: Server configuration denies access to data source
SQLState: 08001
VendorError: 0
```

What is going on? I can connect just fine with the MySQL command-line client.

MySQL Connector/J must use TCP/IP sockets to connect to MySQL, as Java does not support Unix Domain Sockets. Therefore, when MySQL Connector/J connects to MySQL, the security manager in MySQL server will use its grant tables to determine whether the connection is permitted.

You must add the necessary security credentials to the MySQL server for this to happen, using the `GRANT` statement to your MySQL Server. See GRANT Statement, for more information.

> **Note**
>
> Testing your connectivity with the `mysql` command-line client will not work unless you add the "host" flag, and use something other than `localhost` for the host. The `mysql` command-line client will use Unix domain sockets if you use the special host name `localhost`. If you are testing connectivity to `localhost`, use `127.0.0.1` as the host name instead.

> **Warning**
>
> Changing privileges and permissions improperly in MySQL can potentially cause your server installation to not have optimal security properties.

15.2: My application throws an SQLException 'No Suitable Driver'. Why is this happening?

There are three possible causes for this error:

- The Connector/J driver is not in your `CLASSPATH`, see Chapter 3, *Connector/J Installation*.
- The format of your connection URL is incorrect, or you are referencing the wrong JDBC driver.
- When using DriverManager, the `jdbc.drivers` system property has not been populated with the location of the Connector/J driver.

15.3: I'm trying to use MySQL Connector/J in an applet or application and I get an exception similar to:

```
SQLException: Cannot connect to MySQL server on host:3306.
Is there a MySQL server running on the machine/port you
are trying to connect to?

(java.security.AccessControlException)
SQLState: 08S01
VendorError: 0
```

Either you're running an Applet, your MySQL server has been installed with the `skip_networking` system variable enabled, or your MySQL server has a firewall sitting in front of it.

Applets can only make network connections back to the machine that runs the web server that served the .class files for the applet. This means that MySQL must run on the same machine (or you must have some sort of port re-direction) for this to work. This also means that you will not be able to test applets from your local file system, you must always deploy them to a web server.

MySQL Connector/J can only communicate with MySQL using TCP/IP, as Java does not support Unix domain sockets. TCP/IP communication with MySQL might be affected if MySQL was started with the `skip_networking` system variable enabled, or if it is firewalled.

If MySQL has been started with `skip_networking` enabled (the Debian Linux package of MySQL server does this for example), you need to comment it out in the file `/etc/mysql/my.cnf` or `/etc/my.cnf`. Of course your `my.cnf` file might also exist in the `data` directory of your MySQL server, or anywhere else (depending on how MySQL was compiled for your system). Binaries created by us always look in `/etc/my.cnf` and *`datadir`*`/my.cnf`. If your MySQL server has been firewalled, you will need to have the firewall configured to allow TCP/IP connections from the host where your Java code is running to the MySQL server on the port that MySQL is listening to (by default, 3306).

15.4: I have a servlet/application that works fine for a day, and then stops working overnight

MySQL closes connections after 8 hours of inactivity. You either need to use a connection pool that handles stale connections or use the `autoReconnect` parameter (see Section 5.3, “Configuration Properties for Connector/J”).

Also, catch `SQLExceptions` in your application and deal with them, rather than propagating them all the way until your application exits. This is just good programming practice. MySQL Connector/J will set the `SQLState` (see `java.sql.SQLException.getSQLState()` in your API docs) to `08S01` when it encounters network-connectivity issues during the processing of a query. Attempt to reconnect to MySQL at this point.

The following (simplistic) example shows what code that can handle these exceptions might look like:

Example 15.1 Connector/J: Example of transaction with retry logic

```
public void doBusinessOp() throws SQLException {
    Connection conn = null;
    Statement stmt = null;
    ResultSet rs = null;

    //
    // How many times do you want to retry the transaction
    // (or at least _getting_ a connection)?
    //
    int retryCount = 5;

    boolean transactionCompleted = false;

    do {
        try {
            conn = getConnection(); // assume getting this from a
                                    // javax.sql.DataSource, or the
                                    // java.sql.DriverManager

            conn.setAutoCommit(false);

            //
            // Okay, at this point, the 'retry-ability' of the
            // transaction really depends on your application logic,
            // whether or not you're using autocommit (in this case
            // not), and whether you're using transactional storage
            // engines
            //
            // For this example, we'll assume that it's _not_ safe
            // to retry the entire transaction, so we set retry
            // count to 0 at this point
            //
            // If you were using exclusively transaction-safe tables,
            // or your application could recover from a connection going
            // bad in the middle of an operation, then you would not
```

```
            // touch 'retryCount' here, and just let the loop repeat
            // until retryCount == 0.
            //
            retryCount = 0;

            stmt = conn.createStatement();

            String query = "SELECT foo FROM bar ORDER BY baz";

            rs = stmt.executeQuery(query);

            while (rs.next()) {
            }

            rs.close();
            rs = null;

            stmt.close();
            stmt = null;

            conn.commit();
            conn.close();
            conn = null;

            transactionCompleted = true;
        } catch (SQLException sqlEx) {

            //
            // The two SQL states that are 'retry-able' are 08S01
            // for a communications error, and 40001 for deadlock.
            //
            // Only retry if the error was due to a stale connection,
            // communications problem or deadlock
            //

            String sqlState = sqlEx.getSQLState();

            if ("08S01".equals(sqlState) || "40001".equals(sqlState)) {
                retryCount -= 1;
            } else {
                retryCount = 0;
            }
        } finally {
            if (rs != null) {
                try {
                    rs.close();
                } catch (SQLException sqlEx) {
                    // You'd probably want to log this...
                }
            }

            if (stmt != null) {
                try {
                    stmt.close();
                } catch (SQLException sqlEx) {
                    // You'd probably want to log this as well...
                }
            }

            if (conn != null) {
                try {
                    //
                    // If we got here, and conn is not null, the
                    // transaction should be rolled back, as not
                    // all work has been done

                    try {
```

```
                    conn.rollback();
                } finally {
                    conn.close();
                }
            } catch (SQLException sqlEx) {
                //
                // If we got an exception here, something
                // pretty serious is going on, so we better
                // pass it up the stack, rather than just
                // logging it...

                throw sqlEx;
            }
        }
    }
} while (!transactionCompleted && (retryCount > 0));
}
```

> **Note**
>
> Use of the `autoReconnect` option is not recommended because there is no safe method of reconnecting to the MySQL server without risking some corruption of the connection state or database state information. Instead, use a connection pool, which will enable your application to connect to the MySQL server using an available connection from the pool. The `autoReconnect` facility is deprecated, and may be removed in a future release.

15.5: I'm trying to use JDBC 2.0 updatable result sets, and I get an exception saying my result set is not updatable.

Because MySQL does not have row identifiers, MySQL Connector/J can only update result sets that have come from queries on tables that have at least one primary key, the query must select every primary key column, and the query can only span one table (that is, no joins). This is outlined in the JDBC specification.

Note that this issue only occurs when using updatable result sets, and is caused because Connector/J is unable to guarantee that it can identify the correct rows within the result set to be updated without having a unique reference to each row. There is no requirement to have a unique field on a table if you are using `UPDATE` or `DELETE` statements on a table where you can individually specify the criteria to be matched using a `WHERE` clause.

15.6: I cannot connect to the MySQL server using Connector/J, and I'm sure the connection parameters are correct.

Make sure that the `skip_networking` system variable has not been enabled on your server. Connector/J must be able to communicate with your server over TCP/IP; named sockets are not supported. Also ensure that you are not filtering connections through a firewall or other network security system. For more information, see Can't connect to [local] MySQL server.

15.7: My application is deployed through JBoss and I am using transactions to handle the statements on the MySQL database. Under heavy loads, I am getting an error and stack trace, but these only occur after a fixed period of heavy activity.

This is a JBoss, not Connector/J, issue and is connected to the use of transactions. Under heavy loads the time taken for transactions to complete can increase, and the error is caused because you have exceeded the predefined timeout.

You can increase the timeout value by setting the `TransactionTimeout` attribute to the `TransactionManagerService` within the `/conf/jboss-service.xml` file (pre-4.0.3) or `/deploy/jta-service.xml` for JBoss 4.0.3 or later. See TransactionTimeout within the JBoss wiki for more information.

15.8: When using `gcj`, a `java.io.CharConversionException` exception is raised when working with certain character sequences.

This is a known issue with `gcj` which raises an exception when it reaches an unknown character or one it cannot convert. Add `useJvmCharsetConverters=true` to your connection string to force character conversion outside of the `gcj` libraries, or try a different JDK.

15.9: Updating a table that contains a primary key that is either `FLOAT` or compound primary key that uses `FLOAT` fails to update the table and raises an exception.

Connector/J adds conditions to the `WHERE` clause during an `UPDATE` to check the old values of the primary key. If there is no match, then Connector/J considers this a failure condition and raises an exception.

The problem is that rounding differences between supplied values and the values stored in the database may mean that the values never match, and hence the update fails. The issue will affect all queries, not just those from Connector/J.

To prevent this issue, use a primary key that does not use `FLOAT`. If you have to use a floating point column in your primary key, use `DOUBLE` or `DECIMAL` types in place of `FLOAT`.

15.10: You get an `ER_NET_PACKET_TOO_LARGE` exception, even though the binary blob size you want to insert using JDBC is safely below the `max_allowed_packet` size.

This is because the `hexEscapeBlock()` method in `com.mysql.jdbc.PreparedStatement.streamToBytes()` may almost double the size of your data.

15.11: What should you do if you receive error messages similar to the following: "Communications link failure – Last packet sent to the server was X ms ago"?

Generally speaking, this error suggests that the network connection has been closed. There can be several root causes:

- Firewalls or routers may clamp down on idle connections (the MySQL client/server protocol does not ping).
- The MySQL Server may be closing idle connections that exceed the `wait_timeout` or `interactive_timeout` threshold.

To help troubleshoot these issues, the following tips can be used. If a recent (5.1.13+) version of Connector/J is used, you will see an improved level of information compared to earlier versions. Older versions simply display the last time a packet was sent to the server, which is frequently 0 ms ago. This is of limited use, as it may be that a packet was just sent, while a packet from the server has not been received for several hours. Knowing the period of time since Connector/J last received a packet from the server is useful information, so if this is not displayed in your exception message, it is recommended that you update Connector/J.

Further, if the time a packet was last sent/received exceeds the `wait_timeout` or `interactive_timeout` threshold, this is noted in the exception message.

Although network connections can be volatile, the following can be helpful in avoiding problems:

- Ensure connections are valid when used from the connection pool. Use a query that starts with `/* ping */` to execute a lightweight ping instead of full query. Note, the syntax of the ping needs to be exactly as specified here.
- Minimize the duration a connection object is left idle while other application logic is executed.

- Explicitly validate the connection before using it if the connection has been left idle for an extended period of time.
- Ensure that `wait_timeout` and `interactive_timeout` are set sufficiently high.
- Ensure that `tcpKeepalive` is enabled.
- Ensure that any configurable firewall or router timeout settings allow for the maximum expected connection idle time.

> **Note**
>
> Do not expect to be able to reuse a connection without problems, if it has being lying idle for a period. If a connection is to be reused after being idle for any length of time, ensure that you explicitly test it before reusing it.

15.12: Why does Connector/J not reconnect to MySQL and re-issue the statement after a communication failure, instead of throwing an Exception, even though I use the `autoReconnect` connection string option?

There are several reasons for this. The first is transactional integrity. The MySQL Reference Manual states that “there is no safe method of reconnecting to the MySQL server without risking some corruption of the connection state or database state information”. Consider the following series of statements for example:

```
conn.createStatement().execute(
  "UPDATE checking_account SET balance = balance - 1000.00 WHERE customer='Smith'");
conn.createStatement().execute(
  "UPDATE savings_account SET balance = balance + 1000.00 WHERE customer='Smith'");
conn.commit();
```

Consider the case where the connection to the server fails after the `UPDATE` to `checking_account`. If no exception is thrown, and the application never learns about the problem, it will continue executing. However, the server did not commit the first transaction in this case, so that will get rolled back. But execution continues with the next transaction, and increases the `savings_account` balance by 1000. The application did not receive an exception, so it continued regardless, eventually committing the second transaction, as the commit only applies to the changes made in the new connection. Rather than a transfer taking place, a deposit was made in this example.

Note that running with `autocommit` enabled does not solve this problem. When Connector/J encounters a communication problem, there is no means to determine whether the server processed the currently executing statement or not. The following theoretical states are equally possible:

- The server never received the statement, and therefore no related processing occurred on the server.
- The server received the statement, executed it in full, but the response was not received by the client.

If you are running with `autocommit` enabled, it is not possible to guarantee the state of data on the server when a communication exception is encountered. The statement may have reached the server, or it may not. All you know is that communication failed at some point, before the client received confirmation (or data) from the server. This does not only affect `autocommit` statements though. If the communication problem occurred during `Connection.commit()`, the question arises of whether the transaction was committed on the server before the communication failed, or whether the server received the commit request at all.

The second reason for the generation of exceptions is that transaction-scoped contextual data may be vulnerable, for example:

- Temporary tables.
- User-defined variables.
- Server-side prepared statements.

These items are lost when a connection fails, and if the connection silently reconnects without generating an exception, this could be detrimental to the correct execution of your application.

In summary, communication errors generate conditions that may well be unsafe for Connector/J to simply ignore by silently reconnecting. It is necessary for the application to be notified. It is then for the application developer to decide how to proceed in the event of connection errors and failures.

15.13: How can I use 3-byte UTF8 with Connector/J?

For 5.1.46 and earlier: To use 3-byte UTF8 with Connector/J set `characterEncoding=utf8` and set `useUnicode=true` in the connection string.

For 5.1.47 and later: Because there is no Java-style character set name for `utfmb3` that you can use with the connection option `charaterEncoding`, the only way to use `utf8mb3` as your connection character set is to use a `utf8mb3` collation (for example, `utf8_general_ci`) for the connection option `connectionCollation`, which forces a `utf8mb3` character set to be used. See Section 5.6, “Using Character Sets and Unicode” for details.

15.14: How can I use 4-byte UTF8 (`utf8mb4`) with Connector/J?

To use 4-byte UTF8 with Connector/J configure the MySQL server with `character_set_server=utf8mb4`. Connector/J will then use that setting, if `characterEncoding` and `connectionCollation` have not been set in the connection string. This is equivalent to autodetection of the character set. See Section 5.6, “Using Character Sets and Unicode” for details. *For 5.1.47 and later:* You can use `characterEncoding=UTF-8` to use `utf8mb4`, even if `character_set_server` on the server has been set to something else.

15.15: Using `useServerPrepStmts=false` and certain character encodings can lead to corruption when inserting BLOBs. How can this be avoided?

When using certain character encodings, such as SJIS, CP932, and BIG5, it is possible that BLOB data contains characters that can be interpreted as control characters, for example, backslash, '\'. This can lead to corrupted data when inserting BLOBs into the database. There are two things that need to be done to avoid this:

1. Set the connection string option `useServerPrepStmts` to `true`.
2. Set `SQL_MODE` to `NO_BACKSLASH_ESCAPES`.

Chapter 16 Known Issues and Limitations

The following are some known issues and limitations for MySQL Connector/J 5.1:

- When Connector/J retrieves timestamps for a daylight saving time (DST) switch day using the `getTimeStamp()` method on the result set, some of the returned values might be wrong. The errors can be avoided by using the following connection options when connecting to a database:

```
useTimezone=true
useLegacyDatetimeCode=false
serverTimezone=UTC
```

- Since release 5.1.41, the functionality of the property `elideSetAutoCommits` has been disabled due to Bug# 66884. Any value given for the property is ignored by Connector/J.

- MySQL Server uses a proleptic Gregorian calendar internally. However, Connector/J uses `java.sql.Date`, which is non-proleptic. Therefore, when setting and retrieving dates that were before the Julian-Gregorian cutover (October 15, 1582) using the `PreparedStatement` methods, always supply explicitly a proleptic Gregorian calendar to the `setDate()` and `getDate()` methods, in order to avoid possible errors with dates stored to and calculated by the server.

- *For MySQL 8.0.14 and later, 5.7.25 and later, and 5.6.43 and later*: To use Windows named pipes for connections, the MySQL Server that Connector/J wants to connect to must be started with the system variable `named_pipe_full_access_group`; see Section 5.9, “Connecting Using Named Pipes” for details.

Chapter 17 Connector/J Support

Table of Contents

17.1 Connector/J Community Support

You can join the `#connectors` channel in the MySQL Community Slack workspace, where you can get help directly from MySQL developers and other users.

17.2 How to Report Connector/J Bugs or Problems

The normal place to report bugs is http://bugs.mysql.com/, which is the address for our bugs database. This database is public, and can be browsed and searched by anyone. If you log in to the system, you will also be able to enter new reports.

If you find a sensitive security bug in MySQL Server, please let us know immediately by sending an email message to `<secalert_us@oracle.com>`. Exception: Support customers should report all problems, including security bugs, to Oracle Support at http://support.oracle.com/.

Writing a good bug report takes patience, but doing it right the first time saves time both for us and for yourself. A good bug report, containing a full test case for the bug, makes it very likely that we will fix the bug in the next release.

This section will help you write your report correctly so that you do not waste your time doing things that may not help us much or at all.

If you have a repeatable bug report, please report it to the bugs database at http://bugs.mysql.com/. Any bug that we are able to repeat has a high chance of being fixed in the next MySQL release.

To report other problems, you can use one of the MySQL mailing lists.

Remember that it is possible for us to respond to a message containing too much information, but not to one containing too little. People often omit facts because they think they know the cause of a problem and assume that some details do not matter.

A good principle is this: If you are in doubt about stating something, state it. It is faster and less troublesome to write a couple more lines in your report than to wait longer for the answer if we must ask you to provide information that was missing from the initial report.

The most common errors made in bug reports are (a) not including the version number of Connector/J or MySQL used, and (b) not fully describing the platform on which Connector/J is installed (including the JVM version, and the platform type and version number that MySQL itself is installed on).

This is highly relevant information, and in 99 cases out of 100, the bug report is useless without it. Very often we get questions like, “Why doesn't this work for me?” Then we find that the feature requested wasn't implemented in that MySQL version, or that a bug described in a report has already been fixed in newer MySQL versions.

Sometimes the error is platform-dependent; in such cases, it is next to impossible for us to fix anything without knowing the operating system and the version number of the platform.

If at all possible, create a repeatable, standalone testcase that doesn't involve any third-party classes.

To streamline this process, we ship a base class for testcases with Connector/J, named '`com.mysql.jdbc.util.BaseBugReport`'. To create a testcase for Connector/J using this class, create your own class that inherits from `com.mysql.jdbc.util.BaseBugReport` and override the methods `setUp()`, `tearDown()` and `runTest()`.

In the `setUp()` method, create code that creates your tables, and populates them with any data needed to demonstrate the bug.

In the `runTest()` method, create code that demonstrates the bug using the tables and data you created in the `setUp` method.

In the `tearDown()` method, drop any tables you created in the `setUp()` method.

In any of the above three methods, use one of the variants of the `getConnection()` method to create a JDBC connection to MySQL:

- `getConnection()` - Provides a connection to the JDBC URL specified in `getUrl()`. If a connection already exists, that connection is returned, otherwise a new connection is created.
- `getNewConnection()` - Use this if you need to get a new connection for your bug report (that is, there is more than one connection involved).
- `getConnection(String url)` - Returns a connection using the given URL.
- `getConnection(String url, Properties props)` - Returns a connection using the given URL and properties.

If you need to use a JDBC URL that is different from 'jdbc:mysql:///test', override the method `getUrl()` as well.

Use the `assertTrue(boolean expression)` and `assertTrue(String failureMessage, boolean expression)` methods to create conditions that must be met in your testcase demonstrating the behavior you are expecting (vs. the behavior you are observing, which is why you are most likely filing a bug report).

Finally, create a `main()` method that creates a new instance of your testcase, and calls the `run` method:

```
public static void main(String[] args) throws Exception {
      new MyBugReport().run();
 }
```

Once you have finished your testcase, and have verified that it demonstrates the bug you are reporting, upload it with your bug report to http://bugs.mysql.com/.

Index

F

G

H

I

J

K

L

M

U

V

Y

Z