Abstract

This manual describes how to install, configure, and develop database applications using MySQL Connector/J, the JDBC driver for communicating with MySQL servers.

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

For legal information, see the Legal Notices.

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

For additional documentation on MySQL products, including translations of the documentation into other languages, and downloadable versions in variety of formats, including HTML and PDF formats, see the MySQL Documentation Library.

Document generated on: 2016-01-20 (revision: 46162)
# Table of Contents

Preface and Legal Notices ............................................................................................................ v  
1 Overview of MySQL Connector/J .......................................................................................... 1  
2 Connector/J Versions ............................................................................................................... 3  
   2.1 Connector/J Release Notes and Change History .................................................................. 3  
   2.2 Java Versions Supported .................................................................................................... 4  
3 Connector/J Installation .......................................................................................................... 5  
   3.1 Installing Connector/J from a Binary Distribution .............................................................. 5  
   3.2 Installing the Driver and Configuring the CLASSPATH ...................................................... 5  
   3.3 Upgrading from an Older Version ....................................................................................... 6  
      3.3.1 Upgrading to MySQL Connector/J 5.1.x ........................................................................ 7  
      3.3.2 JDBC-Specific Issues When Upgrading to MySQL Server 4.1 or Newer .................... 7  
      3.3.3 Upgrading from MySQL Connector/J 3.0 to 3.1 ............................................................ 7  
   3.4 Installing from Source ....................................................................................................... 9  
   3.5 Testing Connector/J ......................................................................................................... 13  
4 Connector/J Examples ........................................................................................................... 15  
5 Connector/J (JDBC) Reference .............................................................................................. 17  
   5.1 Driver/DataSource Class Names, URL Syntax and Configuration Properties for Connector/J  
      5.1.1 Properties Files for the useConfig option ...................................................................... 17  
   5.2 JDBC API Implementation Notes ...................................................................................... 51  
   5.3 Java, JDBC and MySQL Types .......................................................................................... 56  
   5.4 Using Character Sets and Unicode ................................................................................. 58  
   5.5 Connecting Securely Using SSL ....................................................................................... 59  
   5.6 Connecting Using PAM Authentication ............................................................................ 62  
   5.7 Using Master/Slave Replication with ReplicationConnection ........................................ 62  
   5.8 Mapping MySQL Error Numbers to JDBC SQLState Codes ............................................ 62  
6 JDBC Concepts ....................................................................................................................... 71  
   6.1 Connecting to MySQL Using the JDBC DriverManager Interface ..................................... 71  
   6.2 Using JDBC Statement Objects to Execute SQL ............................................................... 72  
   6.3 Using JDBC CallableStatements to Execute Stored Procedures .................................... 73  
   6.4 Retrieving AUTO_INCREMENT Column Values through JDBC ...................................... 76  
7 Connection Pooling with Connector/J ................................................................................... 81  
8 Multi-Host Connections ........................................................................................................ 85  
   8.1 Configuring Server Failover ............................................................................................... 85  
   8.2 Configuring Load Balancing with Connector/J ................................................................. 88  
   8.3 Configuring Master/Slave Replication with Connector/J ................................................ 90  
   8.4 Advanced Load-balancing and Failover Configuration .................................................... 94  
9 Using the Connector/J Interceptor Classes .......................................................................... 97  
10 Using Connector/J with Tomcat ............................................................................................ 99  
11 Using Connector/J with JBoss ............................................................................................. 101  
12 Using Connector/J with Spring ............................................................................................ 103  
   12.1 Using JdbcTemplate ........................................................................................................ 104  
   12.2 Transactional JDBC Access ............................................................................................. 106  
   12.3 Connection Pooling with Spring ..................................................................................... 107  
13 Using Connector/J with GlassFish ..................................................................................... 109  
   13.1 A Simple JSP Application with GlassFish, Connector/J and MySQL .............................. 110  
   13.2 A Simple Servlet with GlassFish, Connector/J and MySQL ............................................ 112  
14 Using Connector/J with MySQL Fabric .............................................................................. 117  
15 Troubleshooting Connector/J Applications ......................................................................... 119  
16 Known Issues and Limitations ............................................................................................. 129  
17 Connector/J Support ............................................................................................................ 131
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>17.1 Connector/J Community Support</td>
<td>131</td>
</tr>
<tr>
<td>17.2 How to Report Connector/J Bugs or Problems</td>
<td>131</td>
</tr>
<tr>
<td>A Licenses for Third-Party Components</td>
<td>133</td>
</tr>
<tr>
<td>A.1 Ant-Contrib License</td>
<td>133</td>
</tr>
<tr>
<td>A.2 c3p0 JDBC Library License</td>
<td>134</td>
</tr>
<tr>
<td>A.3 GNU Lesser General Public License Version 2.1, February 1999</td>
<td>134</td>
</tr>
<tr>
<td>A.4 jboss-common-jdbc-wrapper.jar License</td>
<td>142</td>
</tr>
<tr>
<td>A.5 NanoXML License</td>
<td>142</td>
</tr>
<tr>
<td>A.6 rox.jar License</td>
<td>143</td>
</tr>
<tr>
<td>A.7 Simple Logging Facade for Java (SLF4J) License</td>
<td>143</td>
</tr>
<tr>
<td>A.8 Unicode Data Files</td>
<td>144</td>
</tr>
</tbody>
</table>
Preface and Legal Notices

This manual describes how to install, configure, and develop database applications using MySQL Connector/J, the JDBC driver for communicating with MySQL servers.

Legal Notices

Copyright © 1998, 2016, Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services 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.

Documentation Accessibility
For information about Oracle’s commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

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.
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). 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 Ibatis SQL Maps to reduce the amount of JDBC code for you to debug, tune, secure, and maintain.

Key Topics

• For help with connection strings, connection options, and setting up your connection through JDBC, see Section 5.1, “Driver/Datasource Class Names, URL Syntax and Configuration Properties for Connector/J”.
Chapter 2 Connector/J Versions

Table of Contents

2.1 Connector/J Release Notes and Change History ................................................................. 3
2.2 Java Versions Supported ........................................................................................................ 4

There are currently four versions of MySQL Connector/J 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 4.1, 5.0, 5.1, 5.5, 5.6, and 5.7. 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 \texttt{java.time} package, support for JDBC-4.x XML processing, support for per connection client information, and support for the \texttt{NCHAR}, \texttt{NVARCHAR} and \texttt{NCLOB} data types. This release also includes all bug fixes up to and including Connector/J 5.0.6.

• Connector/J 5.0 provides support for all the functionality offered by Connector/J 3.1 and includes distributed transaction (XA) support.

• Connector/J 3.1 was designed for connectivity to MySQL 4.1 and MySQL 5.0 servers and provides support for all the functionality in MySQL 5.0 except distributed transaction (XA) support.

• Connector/J 3.0 provides core functionality and was designed for connectivity to MySQL 3.x or MySQL 4.1 servers, although it provides basic compatibility with later versions of MySQL. Connector/J 3.0 does not support server-side prepared statements, and does not support any of the features in versions of MySQL later than 4.1.

The following table summarizes the Connector/J versions available, along with the details of JDBC driver type, what version of the JDBC API it supports, what versions of MySQL Server it works with, and whether it is currently supported or not:

Table 2.1 Summary of Connector/J Versions

<table>
<thead>
<tr>
<th>Connector/J version</th>
<th>Driver Type</th>
<th>JDBC version</th>
<th>MySQL Server version</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>4</td>
<td>3.0, 4.0, 4.1, 4.2</td>
<td>4.1, 5.0, 5.1, 5.5, 5.6, 5.7</td>
<td>Recommended version</td>
</tr>
<tr>
<td>5.0</td>
<td>4</td>
<td>3.0</td>
<td>4.1, 5.0</td>
<td>Released version</td>
</tr>
<tr>
<td>3.1</td>
<td>4</td>
<td>3.0</td>
<td>4.1, 5.0</td>
<td>Obsolete</td>
</tr>
<tr>
<td>3.0</td>
<td>4</td>
<td>3.0</td>
<td>3.x, 4.1</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

The current recommended version for Connector/J is 5.1. This guide covers all four connector versions, with specific notes given where a setting applies to a specific option.

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 Release Notes.
2.2 Java Versions Supported

The following table summarizes what version of Java RTE is required to use Connector/J with Java applications, and what version of JDK is required to build Connector/J source code:

<table>
<thead>
<tr>
<th>Connector/J version</th>
<th>Java RTE required</th>
<th>JDK required (to build source code)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>1.5.x, 1.6.x, 1.7.x, 1.8.x</td>
<td>1.6.x and 1.5.x</td>
</tr>
<tr>
<td>5.0</td>
<td>1.3.x, 1.4.x, 1.5.x, 1.6.x</td>
<td>1.4.2, 1.5.x, 1.6.x</td>
</tr>
<tr>
<td>3.1</td>
<td>1.2.x, 1.3.x, 1.4.x, 1.5.x, 1.6.x</td>
<td>1.4.2, 1.5.x, 1.6.x</td>
</tr>
<tr>
<td>3.0</td>
<td>1.2.x, 1.3.x, 1.4.x, 1.5.x, 1.6.x</td>
<td>1.4.2, 1.5.x, 1.6.x</td>
</tr>
</tbody>
</table>

If you are building Connector/J from source code using the source distribution (see Section 3.4, “Installing from Source”), you must use JDK 1.4.2 or newer to compile the package for Connector/J 5.0 or earlier. For Connector/J 5.1, you must have both JDK-1.6.x AND JDK-1.5.x installed to be able to build the source code.

Java 1.7 support requires Connector/J 5.1.21 and higher.

Several JDBC 4.1 methods were implemented for the first time in Connector/J 5.1.21.

Because of the implementation of java.sql.Savepoint, Connector/J 3.1.0 and newer will not run on a Java runtime older than 1.4 unless the class verifier is turned off (by setting the -Xverify:none option to the Java runtime). This is because the class verifier will try to load the class definition for java.sql.Savepoint even though it is not accessed by the driver unless you actually use savepoint functionality.

Caching functionality provided by Connector/J 3.1.0 or newer is also not available on JVMs older than 1.4.x, as it relies on java.util.LinkedHashMap, which was first available in JDK-1.4.0.

MySQL Connector/J does not support JDK-1.1.x or JDK-1.0.x.
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.

### 3.1 Installing Connector/J from a Binary Distribution

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

MySQL Connector/J is distributed as a .zip or .tar.gz archive containing the sources, the class files, and the JAR archive named `mysql-connector-java-version-bin.jar`.

Starting with Connector/J 3.1.9, the .class files that constitute the JAR files are only included as part of the driver JAR file.

Starting with Connector/J 3.1.8, the archive also includes a debug build of the driver in a file named `mysql-connector-java-version-bin-g.jar`. Do not use the debug build of the driver unless instructed to do so when reporting a problem or a 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.

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 OS X 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 cannot 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 Set Support, 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
Upgrading from MySQL Connector/J 3.0 to 3.1

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 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.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 1.8.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.rt.jar`, 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 filepath:

```java
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 found at Connector/J 5.1 Download.

**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.
- 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:

```properties
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/).
Installing from Source

- 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 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:
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.
- **Fabric tests**: Classes from the package testsuite.fabric. Includes the code to test Fabric-specific features. These tests require the setting of some special properties that are not documented here. Consult the code or the Fabric-related targets in the bundled Ant build file, build.xml.
- **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 5.1, “Driver/Datasource Class Names, URL Syntax and 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
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 suffix; the same properties without the suffixes 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 6.3, “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

5.1 Driver/Datasource Class Names, URL Syntax and Configuration Properties for Connector/J .......... 17
  5.1.1 Properties Files for the useConfigs Option ............................................................ 51
5.2 JDBC API Implementation Notes .................................................................................... 52
5.3 Java, JDBC and MySQL Types ....................................................................................... 56
5.4 Using Character Sets and Unicode ................................................................................ 58
5.5 Connecting Securely Using SSL ..................................................................................... 59
5.6 Connecting Using PAM Authentication .......................................................................... 62
5.7 Using Master/Slave Replication with ReplicationConnection ....................................... 62
5.8 Mapping MySQL Error Numbers to JDBC SQLState Codes ........................................... 62

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

5.1 Driver/Datasource Class Names, URL Syntax and Configuration Properties for Connector/J

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.

JDBC URL Format

The general format for a JDBC URL for connecting to a MySQL server is as follows, with items in square brackets ([ ]) being optional:

jdbc:mysql://[host1][:port1][,[host2][:port2]]...[/[database]] ? [propertyName1=propertyValue1&propertyName2=propertyValue2]...

Here is a simple example for a connection URL:

jdbc:mysql://localhost:3306/sakila?profileSQL=true

Supply multiple hosts for a server failover setup (see Chapter 8, Multi-Host Connections for details):

* Connection URL for a server failover setup:
jdbc:mysql://primaryhost,secondaryhost1,secondaryhost2/test

There are specialized URL schemes for configuring Connector/J's multi-host functions like load balancing and replication; here are some examples (see Chapter 8, Multi-Host Connections for details):

* Connection URL for load balancing:
jdbc:mysql:loadbalance://localhost:3306,localhost:3310/sakila

* Connection URL for server replication:
jdbc:mysql:replication://master,slave1,slave2,slave3/test
Host and Port

If no hosts are not specified, the host name defaults to 127.0.0.1. If the port for a host is not specified, it defaults to 3306, the default port number for MySQL servers.

Initial Database for Connection

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 fully specify table names using the database name (that is, SELECT dbname.tablename.colname FROM dbname.tablename...) in your SQL. Opening a connection without specifying the database to use is generally 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.

IPv6 Connections

For IPv6 connections, use this alternative syntax to specify hosts in the URL (the same syntax can also be used for IPv4 connections):

```
jdbc:mysql://address=(key1=value)[(key2=value)]...,[address=(key3=value)[(key4=value)]...[/[database]]»
```

Supported keys include:

- (protocol=tcp), or (protocol=pipe) for named pipes on Windows.
- (path=path_to_pipe) for named pipes.
- (host=hostname) for TCP connections.
- (port=port_number) for TCP connections.

For example:

```
jdbc:mysql://address=(protocol=tcp)(host=localhost)(port=3306)/db
```

Keys other than the four mentioned above are treated as host-specific configuration properties, which allow per-host overrides of any configuration property set for multi-host connections (that is, when using failover, load balancing, or replication). For example:

```java
# IPv6 Connection URL for a server failover setup:
jdbc:mysql//address=(protocol=tcp)(host=primaryhost)(port=3306),»
address=(protocol=tcp)(host=secondaryhost1)(port=3310)(user=test2)/test
```

```
# IPv6 Connection URL for load balancing:
jdbc:mysql:loadbalance://address=(protocol=tcp)(host=localhost)(port=3306)(user=test1),»
address=(protocol=tcp)(host=localhost)(port=3310)(user=test2)/sakila
```

```
# IPv6 Connection URL for server replication:
jdbc:mysql:replication://address=(protocol=tcp)(host=master)(port=3306)(user=test1),»
address=(protocol=tcp)(host=slave1)(port=3310)(user=test2)/test
```

Limit the overrides to user, password, network timeouts, and statement and metadata cache sizes; the effects of other per-host overrides are not defined.
The ways to set the other configuration properties are the same for IPv6 and IPv4 URLs; see Setting Configuration Properties.

**Setting Configuration Properties**

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 properties are listed in the following tables.

### Connection/Authentication

<table>
<thead>
<tr>
<th>Properties and Descriptions</th>
<th>user</th>
<th>password</th>
<th>socketFactory</th>
<th>connectTimeout</th>
</tr>
</thead>
<tbody>
<tr>
<td>The user to connect as</td>
<td></td>
<td>The password to use when connecting</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>Since version: all versions</td>
<td></td>
<td>Since version: all versions</td>
<td>Default: <code>com.mysql.jdbc.StandardSocketFactory</code></td>
<td>Since version: 3.0.3</td>
</tr>
</tbody>
</table>

Since version: all versions
### Properties and Descriptions

**socketTimeout**

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

Default: 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: 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
### Setting Configuration Properties

#### Properties and Descriptions

**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: 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: 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: false

Since version: 3.0.17

### Networking

#### Properties and Descriptions

**socksProxyHost**

Name or IP address of SOCKS host to connect through.

Since version: 5.1.34

**socksProxyPort**

Port of SOCKS server.
### Properties and Descriptions

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Since version</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Default</strong>: 1080</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>maxAllowedPacket</strong></td>
<td>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 &quot;blobSendChunkSize&quot;, this setting has a minimum value of &quot;8203&quot; if &quot;useServerPrepStmts&quot; is set to &quot;true&quot;.</td>
<td>-1</td>
</tr>
<tr>
<td><strong>tcpKeepAlive</strong></td>
<td>If connecting using TCP/IP, should the driver set SO_KEEPALIVE?</td>
<td>true</td>
</tr>
<tr>
<td><strong>tcpNoDelay</strong></td>
<td>If connecting using TCP/IP, should the driver set SO_TCP_NODELAY (disabling the Nagle Algorithm)?</td>
<td>true</td>
</tr>
<tr>
<td><strong>tcpRcvBuf</strong></td>
<td>If connecting using TCP/IP, should the driver set SO_RCVBUF to the given value? The default value of '0', means use the platform default value for this property)</td>
<td>0</td>
</tr>
<tr>
<td><strong>tcpSndBuf</strong></td>
<td>If connecting using TCP/IP, should the driver set SO_SNDBUF to the given value? The default value of '0', means use the platform default value for this property)</td>
<td>0</td>
</tr>
<tr>
<td><strong>tcpTrafficClass</strong></td>
<td>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.</td>
<td>0</td>
</tr>
</tbody>
</table>

High Availability and Clustering.
### Properties and Descriptions

#### 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 SQLErrors properly, and is only designed to be used when you are unable to configure your application to handle SQLErrors 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: false

Since version: 1.1

#### autoReconnectForPools

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

Default: false

Since version: 3.1.3

#### failOverReadOnly

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

Default: true

Since version: 3.0.12

#### maxReconnects

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

Default: 3

Since version: 1.1

#### reconnectAtTxEnd

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

Default: 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: 120

Since version: 5.1.6

#### initialTimeout
### Properties and Descriptions

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

Default: 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: 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: 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 master. Setting both properties to 0 disables the automatic fall back to the primary host at transaction boundaries. Time in seconds, defaults to 30

Default: 30

Since version: 3.0.2

**allowMasterDownConnections**

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

Default: false

Since version: 5.1.27

**allowSlaveDownConnections**

By default, a replication-aware connection will fail to connect when configured slave 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 slaves i.e. by setting the replication connection to read-only state. The property 'readFromMasterWhenNoSlaves' should be used for this purpose.

Default: false
### Properties and Descriptions

**readFromMasterWhenNoSlaves**

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

Default: 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: 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: 0

Since version: 5.1.6

**selfDestructOnPingSecondsLifetime**

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 lifetime exceeds this value.

Default: 0

Since version: 5.1.6

**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

### 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 rewriteBatchStatements.
### Properties and Descriptions

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Since version</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>useSSL</strong></td>
<td>false</td>
<td>3.1.1</td>
</tr>
</tbody>
</table>

**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: false

Since version: 3.0.2

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Since version</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>requireSSL</strong></td>
<td>false</td>
<td>3.1.0</td>
</tr>
</tbody>
</table>

**requireSSL**

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

Default: false

Since version: 3.1.0

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Since version</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>verifyServerCertificate</strong></td>
<td>false</td>
<td>5.1.6</td>
</tr>
</tbody>
</table>

**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: true

Since version: 5.1.6

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Since version</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>clientCertificateKeyStoreUrl</strong></td>
<td></td>
<td>5.1.0</td>
</tr>
</tbody>
</table>

**clientCertificateKeyStoreUrl**

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

Since version: 5.1.0

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Since version</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>clientCertificateKeyStoreType</strong></td>
<td>JKS</td>
<td>5.1.0</td>
</tr>
</tbody>
</table>

**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: JKS

Since version: 5.1.0

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Since version</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>clientCertificateKeyStorePassword</strong></td>
<td></td>
<td>5.1.0</td>
</tr>
</tbody>
</table>

**clientCertificateKeyStorePassword**

Password for the client certificates KeyStore

Since version: 5.1.0

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Since version</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>trustCertificateKeyStoreUrl</strong></td>
<td></td>
<td>5.1.0</td>
</tr>
</tbody>
</table>

**trustCertificateKeyStoreUrl**

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

Since version: 5.1.0
### Properties and Descriptions

#### 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: 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

#### allowLoadLocalInfile

Should the driver allow use of 'LOAD DATA LOCAL INFILE...' (defaults to 'true').

Default: true

Since version: 3.0.3

#### allowUrlInLocalInfile

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

Default: false

Since version: 3.1.4

#### allowPublicKeyRetrieval

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

Default: 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: false

Since version: 3.0.1

#### passwordCharacterEncoding
### Properties and Descriptions

**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

### Performance Extensions.

#### Properties and Descriptions

**callableStmtCacheSize**

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

Default: 100

Since version: 3.1.2

**metadataCacheSize**

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

Default: 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: 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: false

Since version: 5.1.7

**prepStmtCacheSize**
### Properties and Descriptions

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

Default: 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: 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: `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: `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: 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.currentTimeMillis()` per query.

Default: 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?
<table>
<thead>
<tr>
<th>Properties and Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default: false</td>
</tr>
<tr>
<td>Since version: 5.0.0</td>
</tr>
</tbody>
</table>

**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: 1048576

Since version: 3.1.9

**cacheCallableStmts**

Should the driver cache the parsing stage of CallableStatements

Default: 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: 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: 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: false

Since version: 3.1.5

**defaultFetchSize**

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

Default: 0

Since version: 3.1.9

**dontCheckOnDuplicateKeyUpdateInSQL**
## Properties and Descriptions

### 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: false

Since version: 3.1.7

### dynamicCalendars

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

Default: 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: 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: 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.
<table>
<thead>
<tr>
<th>Properties and Descriptions</th>
<th>Default: true</th>
<th>Since version: 5.0.6</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>holdResultsOpenOverStatementClose</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Should the driver close result sets on Statement.close() as required by the JDBC specification?</td>
<td>Default: false</td>
<td>Since version: 3.1.7</td>
</tr>
<tr>
<td><strong>largeRowSizeThreshold</strong></td>
<td>What size result set row should the JDBC driver consider &quot;large&quot;, and thus use a more memory-efficient way of representing the row internally?</td>
<td>Default: 2048</td>
</tr>
<tr>
<td><strong>loadBalanceStrategy</strong></td>
<td>If using a load-balanced connection to connect to SQL nodes in a MySQL Cluster/NDB configuration (by using the URL prefix &quot;jdbc:mysql:loadbalance://&quot;), which load balancing algorithm should the driver use: (1) &quot;random&quot; - 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) &quot;bestResponseTime&quot; - the driver will route the request to the host that had the best response time for the previous transaction.</td>
<td>Default: random</td>
</tr>
<tr>
<td><strong>locatorFetchBufferSize</strong></td>
<td>If 'emulateLocators' is configured to 'true', what size buffer should be used when fetching BLOB data for getBinaryInputStream?</td>
<td>Default: 1048576</td>
</tr>
<tr>
<td><strong>readOnlyPropagatesToServer</strong></td>
<td>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.</td>
<td>Default: true</td>
</tr>
<tr>
<td><strong>rewriteBatchedStatements</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Properties and Descriptions

#### 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, server-side prepared statements can not currently take advantage of this rewrite option, and that 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: false
Since version: 3.1.13

---

#### 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: 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: 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: true
Since version: 5.0.5

---

#### useFastIntParsing

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

Default: true
Since version: 3.1.4

---

#### useJvmCharsetConverters
## Properties and Descriptions

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

Default: false

Since version: 5.0.1

### UseReadAheadInput

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

Default: true

Since version: 3.1.5

### Debugging/Profiling.

#### Properties and Descriptions

**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: 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: false

Since version: 3.1.2

**profileSQL**

Trace queries and their execution/fetch times to the configured logger (true/false) defaults to 'false'

Default: 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: 30000

Since version: 3.1.2
<table>
<thead>
<tr>
<th>Properties and Descriptions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>maxQuerySizeToLog</strong></td>
<td>Controls the maximum length/size of a query that will get logged when profiling or tracing</td>
</tr>
<tr>
<td>Default: 2048</td>
<td></td>
</tr>
<tr>
<td>Since version: 3.1.3</td>
<td></td>
</tr>
<tr>
<td><strong>packetDebugBufferSize</strong></td>
<td>The maximum number of packets to retain when 'enablePacketDebug' is true</td>
</tr>
<tr>
<td>Default: 20</td>
<td></td>
</tr>
<tr>
<td>Since version: 3.1.3</td>
<td></td>
</tr>
<tr>
<td><strong>slowQueryThresholdMillis</strong></td>
<td>If 'logSlowQueries' is enabled, how long should a query (in ms) before it is logged as 'slow'?</td>
</tr>
<tr>
<td>Default: 2000</td>
<td></td>
</tr>
<tr>
<td>Since version: 3.1.2</td>
<td></td>
</tr>
<tr>
<td><strong>slowQueryThresholdNanos</strong></td>
<td>If '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.</td>
</tr>
<tr>
<td>Default: 0</td>
<td></td>
</tr>
<tr>
<td>Since version: 5.0.7</td>
<td></td>
</tr>
<tr>
<td><strong>useUsageAdvisor</strong></td>
<td>Should the driver issue 'usage' warnings advising proper and efficient usage of JDBC and MySQL Connector/J to the log (true/false, defaults to 'false')?</td>
</tr>
<tr>
<td>Default: false</td>
<td></td>
</tr>
<tr>
<td>Since version: 3.1.1</td>
<td></td>
</tr>
<tr>
<td><strong>autoGenerateTestcaseScript</strong></td>
<td>Should the driver dump the SQL it is executing, including server-side prepared statements to STDERR?</td>
</tr>
<tr>
<td>Default: false</td>
<td></td>
</tr>
<tr>
<td>Since version: 3.1.9</td>
<td></td>
</tr>
<tr>
<td><strong>autoSlowLog</strong></td>
<td>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?</td>
</tr>
<tr>
<td>Default: true</td>
<td></td>
</tr>
<tr>
<td>Since version: 5.1.4</td>
<td></td>
</tr>
<tr>
<td><strong>clientInfoProvider</strong></td>
<td></td>
</tr>
</tbody>
</table>
### Properties and Descriptions

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: `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: 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: 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: 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 log at a WARNING level?

Default: false

Since version: 3.1.2

**includeInnodbStatusInDeadlockExceptions**

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

Default: false

Since version: 5.0.7

**includeThreadDumpInDeadlockExceptions**

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

Default: false

Since version: 5.1.15

**includeThreadNamesAsStatementComment**
### Properties and Descriptions

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: false

Since version: 5.1.15

#### logSlowQueries

Should queries that take longer than 'slowQueryThresholdMillis' be logged?

Default: 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: 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: com.mysql.jdbc.profiler.LoggingProfilerEventHandler

Since version: 5.1.6

#### resultSetSizeThreshold

If the usage advisor is enabled, how many rows should a result set contain before the driver warns that it is suspiciously large?

Default: 100

Since version: 5.0.5

#### traceProtocol

Should trace-level network protocol be logged?

Default: 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: false

Since version: 5.0.7
## Miscellaneous.

### Properties and Descriptions

#### 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: 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 via 'set collation_connection'

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: 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.
### Properties and Descriptions

*Since version: 5.1.3*

<table>
<thead>
<tr>
<th>Property Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>utf8OutsideBmpIncludedColumnNamePattern</strong></td>
<td>Used to specify exclusion rules to &quot;utf8OutsideBmpExcludedColumnNamePattern&quot;. The regex must follow the patterns used for the java.util.regex package. <em>Since version: 5.1.3</em></td>
</tr>
<tr>
<td><strong>loadBalanceEnableJMX</strong></td>
<td>Enables JMX-based management of load-balanced connection groups, including live addition/removal of hosts from load-balancing pool. <em>Default: false</em></td>
</tr>
<tr>
<td><strong>sessionVariables</strong></td>
<td>A comma-separated list of name/value pairs to be sent as SET SESSION ... to the server when the driver connects. <em>Since version: 3.1.8</em></td>
</tr>
<tr>
<td><strong>useColumnNamesInFindColumn</strong></td>
<td>Prior to JDBC-4.0, the JDBC specification had a bug related to what could be given as a &quot;column name&quot; to ResultSet methods like findColumn(), or getters that took a String property. JDBC-4.0 clarified &quot;column name&quot; to mean the label, as given in an &quot;AS&quot; clause and returned by ResultSetMetaData.getColumnLabel(), and if no AS clause, the column name. Setting this property to &quot;true&quot; 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 &quot;useOldAliasMetadataBehavior&quot; unless you need the specific behavior that it provides with respect to ResultSetMetadata. <em>Default: false</em></td>
</tr>
<tr>
<td><strong>allowNanAndInf</strong></td>
<td>Should the driver allow NaN or +/- INF values in PreparedStatement.setDouble()? <em>Default: false</em></td>
</tr>
<tr>
<td><strong>autoClosePStmtStreams</strong></td>
<td>Should the driver automatically call .close() on streams/readers passed as arguments via set*() methods? <em>Default: false</em></td>
</tr>
<tr>
<td><strong>autoDeserialize</strong></td>
<td>Should the driver automatically detect and de-serialize objects stored in BLOB fields? <em>Default: false</em></td>
</tr>
</tbody>
</table>
### Properties and Descriptions

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Default</th>
<th>Since version</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>blobsAreStrings</strong></td>
<td>Should the driver always treat BLOBs as Strings - specifically to work around dubious metadata returned by the server for GROUP BY clauses?</td>
<td>false</td>
<td>3.1.5</td>
</tr>
<tr>
<td><strong>cacheDefaultTimezone</strong></td>
<td>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.</td>
<td>true</td>
<td>5.1.35</td>
</tr>
<tr>
<td><strong>capitalizeTypeNames</strong></td>
<td>Capitalize type names in DatabaseMeta? (usually only useful when using WebObjects, true/false, defaults to ‘false’)</td>
<td>true</td>
<td>2.0.7</td>
</tr>
<tr>
<td><strong>clobCharacterEncoding</strong></td>
<td>The character encoding to use for sending and retrieving TEXT, MEDIUMTEXT and LONGTEXT values instead of the configured connection characterEncoding</td>
<td></td>
<td>5.0.0</td>
</tr>
<tr>
<td><strong>clobberStreamingResults</strong></td>
<td>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.</td>
<td>false</td>
<td>3.0.9</td>
</tr>
<tr>
<td><strong>compensateOnDuplicateKeyUpdateCounts</strong></td>
<td>Should the driver compensate for the update counts of &quot;ON DUPLICATE KEY&quot; INSERT statements (2 = 1, 0 = 1) when using prepared statements?</td>
<td>false</td>
<td>5.1.7</td>
</tr>
<tr>
<td><strong>continueBatchOnError</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Setting Configuration Properties

<table>
<thead>
<tr>
<th>Properties and Descriptions</th>
<th>Should the driver continue processing batch commands if one statement fails. The JDBC spec allows either way (defaults to 'true').</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Default</strong></td>
<td>true</td>
</tr>
<tr>
<td><strong>Since version</strong></td>
<td>3.0.3</td>
</tr>
<tr>
<td><strong>createDatabaseIfNotExist</strong></td>
<td>Creates the database given in the URL if it doesn't yet exist. Assumes the configured user has permissions to create databases.</td>
</tr>
<tr>
<td><strong>Default</strong></td>
<td>false</td>
</tr>
<tr>
<td><strong>Since version</strong></td>
<td>3.1.9</td>
</tr>
<tr>
<td><strong>detectCustomCollations</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Default</strong></td>
<td>false</td>
</tr>
<tr>
<td><strong>Since version</strong></td>
<td>5.1.29</td>
</tr>
<tr>
<td><strong>emptyStringsConvertToZero</strong></td>
<td>Should the driver allow conversions from empty string fields to numeric values of '0'?</td>
</tr>
<tr>
<td><strong>Default</strong></td>
<td>true</td>
</tr>
<tr>
<td><strong>Since version</strong></td>
<td>3.1.8</td>
</tr>
<tr>
<td><strong>emulateLocators</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Default</strong></td>
<td>false</td>
</tr>
<tr>
<td><strong>Since version</strong></td>
<td>3.1.0</td>
</tr>
<tr>
<td><strong>emulateUnsupportedPstmts</strong></td>
<td>Should the driver detect prepared statements that are not supported by the server, and replace them with client-side emulated versions?</td>
</tr>
<tr>
<td><strong>Default</strong></td>
<td>true</td>
</tr>
<tr>
<td><strong>Since version</strong></td>
<td>3.1.7</td>
</tr>
<tr>
<td><strong>exceptionInterceptors</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Properties and Descriptions</strong></td>
<td></td>
</tr>
<tr>
<td>--------------------------------</td>
<td></td>
</tr>
<tr>
<td><strong>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.</strong></td>
<td></td>
</tr>
<tr>
<td>Since version: 5.1.8</td>
<td></td>
</tr>
<tr>
<td><strong>functionsNeverReturnBlobs</strong></td>
<td></td>
</tr>
<tr>
<td>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?</td>
<td></td>
</tr>
<tr>
<td>Default: false</td>
<td></td>
</tr>
<tr>
<td>Since version: 5.0.8</td>
<td></td>
</tr>
<tr>
<td><strong>generateSimpleParameterMetadata</strong></td>
<td></td>
</tr>
<tr>
<td>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?</td>
<td></td>
</tr>
<tr>
<td>Default: false</td>
<td></td>
</tr>
<tr>
<td>Since version: 5.0.5</td>
<td></td>
</tr>
<tr>
<td><strong>getProceduresReturnsFunctions</strong></td>
<td></td>
</tr>
<tr>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>Default: true</td>
<td></td>
</tr>
<tr>
<td>Since version: 5.1.26</td>
<td></td>
</tr>
<tr>
<td><strong>ignoreNonTxTables</strong></td>
<td></td>
</tr>
<tr>
<td>Ignore non-transactional table warning for rollback? (defaults to 'false').</td>
<td></td>
</tr>
<tr>
<td>Default: false</td>
<td></td>
</tr>
<tr>
<td>Since version: 3.0.9</td>
<td></td>
</tr>
<tr>
<td><strong>jdbcCompliantTruncation</strong></td>
<td></td>
</tr>
<tr>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>Default: true</td>
<td></td>
</tr>
<tr>
<td>Since version: 3.1.2</td>
<td></td>
</tr>
<tr>
<td>Properties and Descriptions</td>
<td></td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td><strong>loadBalanceAutoCommitStatementRegex</strong></td>
<td></td>
</tr>
<tr>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>Since version: 5.1.15</td>
<td></td>
</tr>
<tr>
<td><strong>loadBalanceAutoCommitStatementThreshold</strong></td>
<td></td>
</tr>
<tr>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>Default: 0</td>
<td></td>
</tr>
<tr>
<td>Since version: 5.1.15</td>
<td></td>
</tr>
<tr>
<td><strong>loadBalanceBlacklistTimeout</strong></td>
<td></td>
</tr>
<tr>
<td>Time in milliseconds between checks of servers which are unavailable, by controlling how long a server lives in the global blacklist.</td>
<td></td>
</tr>
<tr>
<td>Default: 0</td>
<td></td>
</tr>
<tr>
<td>Since version: 5.1.0</td>
<td></td>
</tr>
<tr>
<td><strong>loadBalanceConnectionGroup</strong></td>
<td></td>
</tr>
<tr>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>Since version: 5.1.13</td>
<td></td>
</tr>
<tr>
<td><strong>loadBalanceExceptionChecker</strong></td>
<td></td>
</tr>
<tr>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>Default: com.mysql.jdbc.StandardLoadBalanceExceptionChecker</td>
<td></td>
</tr>
<tr>
<td>Since version: 5.1.13</td>
<td></td>
</tr>
<tr>
<td><strong>loadBalancePingTimeout</strong></td>
<td></td>
</tr>
<tr>
<td>Time in milliseconds to wait for ping response from each of load-balanced physical connections when using load-balanced Connection.</td>
<td></td>
</tr>
<tr>
<td>Default: 0</td>
<td></td>
</tr>
<tr>
<td>Since version: 5.1.13</td>
<td></td>
</tr>
<tr>
<td><strong>loadBalanceSQLExceptionSubclassFailover</strong></td>
<td></td>
</tr>
<tr>
<td>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.instanceOf(SQLException) using the thrown SQLException.</td>
<td></td>
</tr>
</tbody>
</table>
### Properties and Descriptions

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Since version: 5.1.13</strong></td>
<td></td>
</tr>
<tr>
<td><strong>loadBalanceSQLStateFailover</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Since version: 5.1.13</strong></td>
<td></td>
</tr>
<tr>
<td><strong>loadBalanceValidateConnectionOnSwapServer</strong></td>
<td>Should the load-balanced Connection explicitly check whether the connection is live when swapping to a new physical connection at commit/rollback?</td>
</tr>
<tr>
<td><strong>Default:</strong> false</td>
<td></td>
</tr>
<tr>
<td><strong>Since version: 5.1.13</strong></td>
<td></td>
</tr>
<tr>
<td><strong>maxRows</strong></td>
<td>The maximum number of rows to return (0, the default means return all rows).</td>
</tr>
<tr>
<td><strong>Default:</strong> -1</td>
<td></td>
</tr>
<tr>
<td><strong>Since version: all versions</strong></td>
<td></td>
</tr>
<tr>
<td><strong>netTimeoutForStreamingResults</strong></td>
<td>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)</td>
</tr>
<tr>
<td><strong>Default:</strong> 600</td>
<td></td>
</tr>
<tr>
<td><strong>Since version: 5.1.0</strong></td>
<td></td>
</tr>
<tr>
<td><strong>noAccessToProcedureBodies</strong></td>
<td>When determining procedure parameter types for CallableStatements, and the connected user can't access procedure bodies through &quot;SHOW CREATE PROCEDURE&quot; 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?</td>
</tr>
<tr>
<td><strong>Default:</strong> false</td>
<td></td>
</tr>
<tr>
<td><strong>Since version: 5.0.3</strong></td>
<td></td>
</tr>
<tr>
<td><strong>noDatetimeStringSync</strong></td>
<td>Don't ensure that ResultSet.getDatatype().toString().equals(ResultSet.getString())</td>
</tr>
<tr>
<td><strong>Default:</strong> false</td>
<td></td>
</tr>
<tr>
<td><strong>Since version: 3.1.7</strong></td>
<td></td>
</tr>
<tr>
<td><strong>noTimezoneConversionForDateType</strong></td>
<td></td>
</tr>
</tbody>
</table>
### Properties and Descriptions

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Default</th>
<th>Since version:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>noTimezoneConversionForTimeType</code></td>
<td>Don't convert TIME values using the server time zone if 'useTimezone'='true'</td>
<td>false</td>
<td>5.0.0</td>
</tr>
<tr>
<td><code>nullCatalogMeansCurrent</code></td>
<td>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)</td>
<td>true</td>
<td>3.1.8</td>
</tr>
<tr>
<td><code>nullNamePatternMatchesAll</code></td>
<td>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)</td>
<td>true</td>
<td>3.1.8</td>
</tr>
<tr>
<td><code>overrideSupportsIntegrityEnhancementFacility</code></td>
<td>Should the driver return &quot;true&quot; for DatabaseMetaData.supportsIntegrityEnhancementFacility() even if the database doesn't support it to workaround applications that require this method to return &quot;true&quot; 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)?</td>
<td>false</td>
<td>3.1.12</td>
</tr>
<tr>
<td><code>padCharsWithSpace</code></td>
<td>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)?</td>
<td>false</td>
<td>5.0.6</td>
</tr>
<tr>
<td><code>pedantic</code></td>
<td>Follow the JDBC spec to the letter.</td>
<td>false</td>
<td></td>
</tr>
</tbody>
</table>
## Properties and Descriptions

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
<th>Default</th>
<th>Since version:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pinGlobalTxToPhysicalConnection</code></td>
<td>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 &quot;XA START ... JOIN&quot; after &quot;XA END&quot; has been called.</td>
<td>false</td>
<td>3.0.0</td>
</tr>
<tr>
<td><code>populateInsertRowWithDefaultValues</code></td>
<td>When using ResultSets that are CONCUR_UPDATABLE, should the driver pre-populate the &quot;insert&quot; 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.</td>
<td>false</td>
<td>5.0.1</td>
</tr>
<tr>
<td><code>processEscapeCodesForPrepStmts</code></td>
<td>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'.</td>
<td>true</td>
<td>3.1.12</td>
</tr>
<tr>
<td><code>queryTimeoutKillsConnection</code></td>
<td>If the timeout given in Statement.setQueryTimeout() expires, should the driver forcibly abort the Connection instead of attempting to abort the query?</td>
<td>false</td>
<td>5.1.9</td>
</tr>
<tr>
<td><code>relaxAutoCommit</code></td>
<td>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')?</td>
<td>false</td>
<td>2.0.13</td>
</tr>
<tr>
<td><code>retainStatementAfterResultSetClose</code></td>
<td>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.</td>
<td>false</td>
<td>3.1.11</td>
</tr>
</tbody>
</table>
### Properties and Descriptions

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Default</th>
<th>Since version</th>
</tr>
</thead>
<tbody>
<tr>
<td>rollbackOnPooledClose</td>
<td>Should the driver issue a rollback() when the logical connection in a pool is closed?</td>
<td>true</td>
<td>3.0.15</td>
</tr>
<tr>
<td>runningCTS13</td>
<td>Enables workarounds for bugs in Sun's JDBC compliance testsuite version 1.3</td>
<td>false</td>
<td>3.1.7</td>
</tr>
<tr>
<td>sendFractionalSeconds</td>
<td>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.</td>
<td>true</td>
<td>5.1.37</td>
</tr>
<tr>
<td>serverTimezone</td>
<td>Override detection/mapping of time zone. Used when time zone from server doesn't map to Java time zone</td>
<td></td>
<td>3.0.2</td>
</tr>
<tr>
<td>statementInterceptors</td>
<td>A comma-delimited list of classes that implement &quot;com.mysql.jdbc.StatementInterceptor&quot; that should be placed &quot;in between&quot; query execution to influence the results. StatementInterceptors are &quot;chainable&quot;, the results returned by the &quot;current&quot; interceptor will be passed on to the next in in the chain, from left-to-right order, as specified in this property.</td>
<td></td>
<td>5.1.1</td>
</tr>
<tr>
<td>strictFloatingPoint</td>
<td>Used only in older versions of compliance test</td>
<td>false</td>
<td>3.0.0</td>
</tr>
<tr>
<td>strictUpdates</td>
<td>Should the driver do strict checking (all primary keys selected) of updatable result sets (true, false, defaults to 'true')?</td>
<td>true</td>
<td>3.0.4</td>
</tr>
</tbody>
</table>
### Properties and Descriptions

**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: 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: false

Since version: 3.1.9

**treatUtilDateAsTimestamp**

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

Default: 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: 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: 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: false

Since version: 3.1.12

**useHostsInPrivileges**
### Properties and Descriptions

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

Default: 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: 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 "useLegacyDatatimeCode=true."

Default: false
Since version: 5.0.0

**useLegacyDatatimeCode**

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: 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: false
Since version: 5.0.4

**useOldUTF8Behavior**

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

Default: false
Since version: 3.1.6

**useOnlyServerErrorMessages**

Don't prepend 'standard' SQLState error messages to error messages returned by the server.
<table>
<thead>
<tr>
<th>Properties and Descriptions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Default: true</td>
<td>Since version: 3.0.15</td>
</tr>
</tbody>
</table>

**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: false

Since version: 5.0.5

**useServerPrepStmts**

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

Default: 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: true

Since version: 3.1.3

**useStreamLengthsInPrepStmts**

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

Default: 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: false

Since version: 3.0.2

**useUnbufferedInput**

Don't use BufferedInputStream for reading data from the server

Default: 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?
Properties Files for the useConfigs Option

<table>
<thead>
<tr>
<th>Properties and Descriptions</th>
<th>Default: true</th>
</tr>
</thead>
<tbody>
<tr>
<td>Since version: 3.1.9</td>
<td></td>
</tr>
</tbody>
</table>

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: exception

Since version: 3.1.4

Connector/J also supports access to MySQL using named pipes on Windows platforms with the NamedPipeSocketFactory as a plugin-socket 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:

```java
socketFactory=com.mysql.jdbc.NamedPipeSocketFactory
```

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.

To create your own socket factories, follow the example code in com.mysql.jdbc.NamedPipeSocketFactory, or com.mysql.jdbc.StandardSocketFactory.

5.1.1 Properties Files for the useConfigs Option

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:

```bash
$ 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

```properties
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

5.2 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:

  ```sql
  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.

- **CLOB**

  The CLOB implementation does not allow in-place modification (they are copies, as reported by the `DatabaseMetaData.locatorsUpdateCopies()` method). Because of this, use the `PreparedStatement.setClob()` method to save changes back to the database. The JDBC API does not have a `ResultSet.updateClob()` method.

- **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**
PreparedStatements are implemented by the driver, as MySQL does not have a prepared statement feature. Because of this, the driver does not implement `getParameterMetaData()` or `getMetaData()` as it would require the driver to have a complete SQL parser in the client.

Starting with version 3.1.0 MySQL Connector/J, server-side prepared statements and binary-encoded result sets are used when the server supports them.

Take care when using a server-side prepared statement with large parameters that are set using `setBinaryStream()`, `setAsciiStream()`, `setUnicodeStream()`, `setBlob()`, or `setClob()`. 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()`, `setBlob()` or `setClob()` 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:

  ```java
callb = conn.createStatement(java.sql.ResultSet.TYPE_FORWARD_ONLY,
    java.sql.ResultSet.CONCUR_READ_ONLY);
callb.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
JDBC API Implementation Notes

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.

• 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().
5.3 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 Connection Properties - Miscellaneous

<table>
<thead>
<tr>
<th>These MySQL Data Types</th>
<th>Can always be converted to these Java types</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DATE, TIME, DATETIME, TIMESTAMP</code></td>
<td><code>java.lang.String, java.sql.Date, java.sql.Timestamp</code></td>
</tr>
</tbody>
</table>

**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 value returned by `ResultSetMetaData.getColumnClassName()` is also shown below. For more information on the JDBC types, see the reference on the `java.sql.Types` class.
### Table 5.2 MySQL Types to Java Types for ResultSet.getObject()

<table>
<thead>
<tr>
<th>MySQL Type Name</th>
<th>Return value of GetColumnClassName</th>
<th>Returned as Java Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT(1) (new in MySQL-5.0)</td>
<td>BIT</td>
<td>java.lang.Boolean</td>
</tr>
<tr>
<td>BIT( &gt; 1) (new in MySQL-5.0)</td>
<td>BIT</td>
<td>byte[]</td>
</tr>
<tr>
<td>TINYINT</td>
<td>TINYINT</td>
<td>java.lang.Boolean if the configuration property tinyIntlisBit is set to true (the default) and the storage size is 1, or java.lang.Integer if not.</td>
</tr>
<tr>
<td>BOOL, BOOLEAN</td>
<td>TINYINT</td>
<td>See TINYINT, above as these are aliases for TINYINT(1), currently.</td>
</tr>
<tr>
<td>SMALLINT[(M)] [UNSIGNED]</td>
<td>SMALLINT [UNSIGNED]</td>
<td>java.lang.Integer (regardless if UNSIGNED or not)</td>
</tr>
<tr>
<td>MEDIUMINT[(M)] [UNSIGNED]</td>
<td>MEDIUMINT [UNSIGNED]</td>
<td>java.lang.Integer, if UNSIGNED java.lang.Long (C/J 3.1 and earlier), or java.lang.Integer for C/J 5.0 and later</td>
</tr>
<tr>
<td>FLOAT[(M,D)]</td>
<td>FLOAT</td>
<td>java.lang.Float</td>
</tr>
<tr>
<td>DOUBLE[(M,B)]</td>
<td>DOUBLE</td>
<td>java.lang.Double</td>
</tr>
<tr>
<td>DECIMAL[(M[,D])]</td>
<td>DECIMAL</td>
<td>java.math.BigDecimal</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
<td>java.sql.Date</td>
</tr>
<tr>
<td>DATETIME</td>
<td>DATETIME</td>
<td>java.sql.Timestamp</td>
</tr>
<tr>
<td>TIMESTAMP[(M)]</td>
<td>TIMESTAMP</td>
<td>java.sql.Timestamp</td>
</tr>
<tr>
<td>TIME</td>
<td>TIME</td>
<td>java.sql.Time</td>
</tr>
<tr>
<td>YEAR[(2</td>
<td>4)]</td>
<td>YEAR</td>
</tr>
<tr>
<td>CHAR(M)</td>
<td>CHAR</td>
<td>java.lang.String (unless the character set for the column is BINARY, then byte[] is returned.</td>
</tr>
<tr>
<td>VARCHAR(M) [BINARY]</td>
<td>VARCHAR</td>
<td>java.lang.String (unless the character set for the column is BINARY, then byte[] is returned.</td>
</tr>
<tr>
<td>BINARY(M)</td>
<td>BINARY</td>
<td>byte[]</td>
</tr>
<tr>
<td>VARBINARY(M)</td>
<td>VARBINARY</td>
<td>byte[]</td>
</tr>
<tr>
<td>TINYBLOB</td>
<td>TINYBLOB</td>
<td>byte[]</td>
</tr>
<tr>
<td>TINYTEXT</td>
<td>VARCHAR</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>BLOB</td>
<td>BLOB</td>
<td>byte[]</td>
</tr>
<tr>
<td>TEXT</td>
<td>VARCHAR</td>
<td>java.lang.String</td>
</tr>
</tbody>
</table>
Using Character Sets and Unicode

<table>
<thead>
<tr>
<th>MySQL Type Name</th>
<th>Return value of getColumnClassName</th>
<th>Returned as Java Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEDIUMBLOB</td>
<td>MEDIUMBLOB</td>
<td>byte[]</td>
</tr>
<tr>
<td>MEDIUMTEXT</td>
<td>VARCHAR</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>LONGBLOB</td>
<td>LONGBLOB</td>
<td>byte[]</td>
</tr>
<tr>
<td>LONGTEXT</td>
<td>VARCHAR</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>ENUM('value1','value2',...)</td>
<td>CHAR</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>SET('value1','value2',...)</td>
<td>CHAR</td>
<td>java.lang.String</td>
</tr>
</tbody>
</table>

5.4 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.executeQuery()`, `Statement.executeUpdate()`, and `PreparedStatement` and `CallableStatement` parameters with the exclusion of parameters set using `setBytes()`, `setBinaryStream()`, `setAsciiStream()`, `setUnicodeStream()` and `setBlob()`.

Number of Encodings Per Connection

In MySQL Server 4.1 and higher, 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.

Prior to MySQL Server 4.1, Connector/J supported a single character encoding per connection, which could either be automatically detected from the server configuration, or could be configured by the user through the `useUnicode` and `characterEncoding` properties.

Setting the Character Encoding

The character encoding between client and server is automatically detected upon connection. You specify the encoding on the server using the `character_set_server` for server versions 4.1.0 and newer, and `character_set` system variable for server versions older than 4.1.0. The driver automatically uses the encoding specified by the server. For more information, see Server Character Set and Collation.

For example, to use 4-byte UTF-8 character sets with Connector/J, configure the MySQL server with `character_set_server=utf8mb4`, and leave `characterEncoding` 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 URL used to connect to the server.

To allow multiple character sets to be sent from the client, use the UTF-8 encoding, either by configuring `utf8` as the default server character set, or by configuring the JDBC driver to use UTF-8 through the `characterEncoding` property.

When specifying character encodings on the client side, use Java-style names. The following table lists MySQL character set names and the corresponding Java-style names:

**Table 5.3 MySQL to Java Encoding Name Translations**

<table>
<thead>
<tr>
<th>MySQL Character Set Name</th>
<th>Java-Style Character Encoding Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ascii</td>
<td>US-ASCII</td>
</tr>
<tr>
<td>big5</td>
<td>Big5</td>
</tr>
</tbody>
</table>
5.5 Connecting Securely Using SSL

SSL in MySQL Connector/J encrypts all data (other than the initial handshake) between the JDBC driver and the server. There is a performance penalty for enabling SSL, 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.

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

- A JDK that includes JSSE (Java Secure Sockets Extension), like JDK-1.4.1 or newer. SSL does not currently work with a JDK that you can add JSSE to, like JDK-1.2.x or JDK-1.3.x due to the following JSSE bug: http://bugs.java.com/bugdatabase/view_bug.do?bug_id=4273544
- A MySQL server that supports SSL and has been compiled and configured to do so, which is MySQL 4.0.4 or later. For more information, see Building MySQL with SSL Support.
- A client certificate (covered later in this section)

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

You will first need to import the MySQL server CA Certificate into a Java truststore. A sample MySQL server CA Certificate is located in the SSL subdirectory of the MySQL source distribution. This is what SSL will use to determine if you are communicating with a secure MySQL server. Alternatively, use the CA Certificate that you have generated or been provided with by your SSL provider.

To use Java's keytool to create a truststore in the current directory, and import the server's CA certificate (cacert.pem), you can do the following (assuming that keytool is in your path. The keytool is typically located in the bin subdirectory of your JDK or JRE):

```
shell> keytool -import -alias mysqlServerCACert -file cacert.pem -keystore truststore
```

Enter the password when prompted for the keystore file. Interaction with keytool looks like this:

```
Enter keystore password: *********
Owner: EMAILADDRESS=walrus@example.com, CN=Walrus,
       O=My Company, L=Orenburg, ST=Some-State, C=RU
Issuer: EMAILADDRESS=walrus@example.com, CN=Walrus,
       O=My Company, L=Orenburg, ST=Some-State, C=RU
Serial number: 0
Certificate fingerprints:
Trust this certificate? [no]: yes
Certificate was added to keystore
```

You then have two options: either import the client certificate that matches the CA certificate you just imported, or create a new client certificate.

Importing an existing certificate requires the certificate to be in DER format. You can use openssl to convert an existing certificate into the new format. For example:

```
shell> openssl x509 -outform DER -in client-cert.pem -out client.cert
```

Now import the converted certificate into your keystore using keytool:

```
shell> keytool -import -file client.cert -keystore keystore -alias mysqlClientCertificate
```

To generate your own client certificate, use keytool to create a suitable certificate and add it to the keystore file:

```
shell> keytool -genkey -keyalg rsa -alias mysqlClientCertificate -keystore keystore
```

Keytool will prompt you for the following information, and create a keystore named keystore in the current directory.

Respond with information that is appropriate for your situation:
Connecting Securely Using SSL

Enter keystore password: *******
What is your first and last name?
[Unknown]: Matthews
What is the name of your organizational unit?
[Unknown]: Software Development
What is the name of your organization?
[Unknown]: My Company
What is the name of your City or Locality?
[Unknown]: Flossmoor
What is the name of your State or Province?
[Unknown]: IL
What is the two-letter country code for this unit?
[Unknown]: US
Is <CN=Matthews, OU=Software Development, O=My Company, L=Flossmoor, ST=IL, C=US> correct?
[no]: y
Enter key password for <mysqlClientCertificate>
(RETURN if same as keystore password):

Finally, to get JSSE to use the keystore and truststore that you have generated, you need to set the following system properties when you start your JVM, replacing path_to_keystore_file with the full path to the keystore file you created, path_to_truststore_file with the path to the truststore file you created, and using the appropriate password values for each property. You can do this either on the command line:

-Djavax.net.ssl.keyStore=path_to_keystore_file
-Djavax.net.ssl.keyStorePassword=password
-Djavax.net.ssl.trustStore=path_to_truststore_file
-Djavax.net.ssl.trustStorePassword=password

Or you can set the values directly within the application:

System.setProperty("javax.net.ssl.keyStore","path_to_keystore_file");
System.setProperty("javax.net.ssl.keyStorePassword","password");
System.setProperty("javax.net.ssl.trustStore","path_to_truststore_file");
System.setProperty("javax.net.ssl.trustStorePassword","password");

You will also need to set useSSL to true in your connection parameters for MySQL Connector/J, either by adding useSSL=true to your URL, or by setting the property useSSL to true in the java.util.Properties instance you pass to DriverManager.getConnection().

You can test that SSL is working by turning on JSSE debugging (as detailed below), and look for the following key events:

*** ClientHello, v3.1
RandomCookie:  GMT: 1018531834 bytes = { 199, 148, 180, 215, 74, 12, »
54, 244, 0, 168, 55, 103, 215, 64, 16, 138, 225, 190, 132, 153, 2, »
217, 219, 239, 202, 19, 121, 78 }
Session ID: {}
Cipher Suites: { 0, 5, 0, 4, 0, 9, 0, 10, 0, 18, 0, 19, 0, 3, 0, 17 }
Compression Methods: { 0 }
***
[write] MD5 and SHA1 hashes:  len = 59
0000: 01 00 00 37 03 01 3D B6 90 FA C7 94 B4 D7 4A 0C ...?..=........J.
0010: 36 F4 00 A8 37 67 D7 40 10 8A E1 BE 84 99 02 D9 6...?g.@.......}
0020: DB EF CA 13 79 4E 00 00 10 00 05 00 04 00 09 00 ...yN............
0030: 8A 00 12 00 13 00 03 00 11 01 00 .............
main, WRITE: SSL v3.1 Handshake, length = 59
main, READ: SSL v3.1 Handshake, length = 74
Connecting Using PAM Authentication

5.6 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: a commercial distribution of MySQL 5.5.16 or higher. See The PAM Authentication Plugin for more information. Connector/J implements the same cleartext authentication method as in The Cleartext Client-Side Authentication Plugin.

- SSL capability, as explained in Section 5.5, “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 5.1, “Driver/Datasource Class Names, URL Syntax and Configuration Properties for Connector/J” for details about that connection option.

5.7 Using Master/Slave Replication with ReplicationConnection

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

5.8 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

<table>
<thead>
<tr>
<th>MySQL Error Number</th>
<th>MySQL Error Name</th>
<th>Legacy (X/Open) SQLState</th>
<th>SQL Standard SQLState</th>
</tr>
</thead>
<tbody>
<tr>
<td>1022</td>
<td>ER_DUP_KEY</td>
<td>23000</td>
<td>23000</td>
</tr>
<tr>
<td>1037</td>
<td>ER_OUTOFMEMORY</td>
<td>S1001</td>
<td>HY001</td>
</tr>
<tr>
<td>1038</td>
<td>ER_OUT_OF_SORTMEMORY</td>
<td>S1001</td>
<td>HY001</td>
</tr>
<tr>
<td>1040</td>
<td>ER_CON_COUNT_ERROR</td>
<td>08004</td>
<td>08004</td>
</tr>
<tr>
<td>1042</td>
<td>ER_BAD_HOST_ERROR</td>
<td>08004</td>
<td>08S01</td>
</tr>
<tr>
<td>1043</td>
<td>ER_HANDSHAKE_ERROR</td>
<td>08004</td>
<td>08S01</td>
</tr>
<tr>
<td>1044</td>
<td>ER_DBACCESS_DENIED_ERROR</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1045</td>
<td>ER_ACCESS_DENIED_ERROR</td>
<td>28000</td>
<td>28000</td>
</tr>
<tr>
<td>1046</td>
<td>ER_NO_DB_ERROR</td>
<td>3D000</td>
<td>3D000</td>
</tr>
<tr>
<td>1047</td>
<td>ER_UNKNOWN_COM_ERROR</td>
<td>08S01</td>
<td>08S01</td>
</tr>
<tr>
<td>1048</td>
<td>ER_BAD_NULL_ERROR</td>
<td>23000</td>
<td>23000</td>
</tr>
<tr>
<td>1049</td>
<td>ER_BAD_DB_ERROR</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1050</td>
<td>ER_TABLE_EXISTS_ERROR</td>
<td>42S01</td>
<td>42S01</td>
</tr>
<tr>
<td>1051</td>
<td>ER_BAD_TABLE_ERROR</td>
<td>42S02</td>
<td>42S02</td>
</tr>
<tr>
<td>1052</td>
<td>ER_NON_UNIQ_ERROR</td>
<td>23000</td>
<td>23000</td>
</tr>
<tr>
<td>1053</td>
<td>ER_SERVER_SHUTDOWN</td>
<td>08S01</td>
<td>08S01</td>
</tr>
<tr>
<td>1054</td>
<td>ER_BAD_FIELD_ERROR</td>
<td>S0022</td>
<td>42S22</td>
</tr>
<tr>
<td>1055</td>
<td>ER_WRONG_FIELD_WITH_GROUP</td>
<td>S1009</td>
<td>42000</td>
</tr>
<tr>
<td>1056</td>
<td>ER_WRONG_GROUP_FIELD</td>
<td>S1009</td>
<td>42000</td>
</tr>
<tr>
<td>1057</td>
<td>ER_WRONG_SUM_SELECT</td>
<td>S1009</td>
<td>42000</td>
</tr>
<tr>
<td>1058</td>
<td>ER_WRONG_VALUE_COUNT</td>
<td>21S01</td>
<td>21S01</td>
</tr>
<tr>
<td>1059</td>
<td>ER_TOO_LONG_IDENT</td>
<td>S1009</td>
<td>42000</td>
</tr>
<tr>
<td>1060</td>
<td>ER_DUP_FIELDNAME</td>
<td>S1009</td>
<td>42S21</td>
</tr>
<tr>
<td>1061</td>
<td>ER_DUP_KEYNAME</td>
<td>S1009</td>
<td>42000</td>
</tr>
<tr>
<td>1062</td>
<td>ER_DUP_ENTRY</td>
<td>S1009</td>
<td>23000</td>
</tr>
<tr>
<td>1063</td>
<td>ER_WRONG_FIELD_SPEC</td>
<td>S1009</td>
<td>42000</td>
</tr>
<tr>
<td>1064</td>
<td>ER_PARSE_ERROR</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1065</td>
<td>ER_EMPTY_QUERY</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1066</td>
<td>ER_NONUNIQ_TABLE</td>
<td>S1009</td>
<td>42000</td>
</tr>
<tr>
<td>1067</td>
<td>ER_INVALID_DEFAULT</td>
<td>S1009</td>
<td>42000</td>
</tr>
<tr>
<td>1068</td>
<td>ER_MULTIPLE_PRI_KEY</td>
<td>S1009</td>
<td>42000</td>
</tr>
<tr>
<td>1069</td>
<td>ER_TOO_MANY_KEYS</td>
<td>S1009</td>
<td>42000</td>
</tr>
<tr>
<td>1070</td>
<td>ER_TOO_MANY_KEY_PARTS</td>
<td>S1009</td>
<td>42000</td>
</tr>
<tr>
<td>1071</td>
<td>ER_TOO_LONG_KEY</td>
<td>S1009</td>
<td>42000</td>
</tr>
<tr>
<td>1072</td>
<td>ER_KEY_COLUMN_DOES_NOT_EXITS</td>
<td>S1009</td>
<td>42000</td>
</tr>
<tr>
<td>MySQL Error Number</td>
<td>MySQL Error Name</td>
<td>Legacy (X/Open) SQLState</td>
<td>Standard SQLState</td>
</tr>
<tr>
<td>--------------------</td>
<td>-----------------------------------------</td>
<td>--------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>1073</td>
<td>ER_BLOB_USED_AS_KEY</td>
<td>S1009</td>
<td>42000</td>
</tr>
<tr>
<td>1074</td>
<td>ER_TOO_BIG_FIELDLENGTH</td>
<td>S1009</td>
<td>42000</td>
</tr>
<tr>
<td>1075</td>
<td>ER_WRONG_AUTO_KEY</td>
<td>S1009</td>
<td>42000</td>
</tr>
<tr>
<td>1080</td>
<td>ER_FORCING_CLOSE</td>
<td>08S01</td>
<td>08S01</td>
</tr>
<tr>
<td>1081</td>
<td>ER_IPSOCK_ERROR</td>
<td>08S01</td>
<td>08S01</td>
</tr>
<tr>
<td>1082</td>
<td>ER_NO_SUCH_INDEX</td>
<td>S1009</td>
<td>42S12</td>
</tr>
<tr>
<td>1083</td>
<td>ER_WRONG_FIELD_TERMINATORS</td>
<td>S1009</td>
<td>42000</td>
</tr>
<tr>
<td>1084</td>
<td>ER_BLOBS_AND_NO_TERMINATED</td>
<td>S1009</td>
<td>42000</td>
</tr>
<tr>
<td>1090</td>
<td>ER_CANT_REMOVE_ALL_FIELDS</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1091</td>
<td>ER_CANT_DROPFIELD_OR_KEY</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1101</td>
<td>ER_BLOB_CANT_HAVE_DEFAULT</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1102</td>
<td>ER_WRONG_DB_NAME</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1103</td>
<td>ER_WRONG_TABLE_NAME</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1104</td>
<td>ER_TOO_BIG_SELECT</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1106</td>
<td>ER_UNKNOWN_PROCEDURE</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1107</td>
<td>ER_WRONG_PARAMCOUNT_TO_PROCEDURE</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1109</td>
<td>ER_UNKNOWN_TABLE</td>
<td>42S02</td>
<td>42S02</td>
</tr>
<tr>
<td>1110</td>
<td>ER_FIELD_SPECIFIED_TWICE</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1112</td>
<td>ER_UNSUPPORTED_EXTENSION</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1113</td>
<td>ER_TABLE_MUST_HAVE_COLUMNS</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1115</td>
<td>ER_UNKNOWN_CHARACTER_SET</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1118</td>
<td>ER_TOO_BIG_ROWSIZE</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1120</td>
<td>ER_WRONG_OUTER_JOIN</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1121</td>
<td>ER_NULL_COLUMN_IN_INDEX</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1129</td>
<td>ER_HOST_ISBLOCKED</td>
<td>08004</td>
<td>HY000</td>
</tr>
<tr>
<td>1130</td>
<td>ER_HOST_NOT_PRIVILEGED</td>
<td>08004</td>
<td>HY000</td>
</tr>
<tr>
<td>1131</td>
<td>ER_PASSWORD_ANONYMOUS_USER</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1132</td>
<td>ER_PASSWORD_NOT_ALLOWED</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1133</td>
<td>ER_PASSWORD_NO_MATCH</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1136</td>
<td>ER_WRONG_VALUE_COUNT_ON_ROW</td>
<td>21S01</td>
<td>21S01</td>
</tr>
<tr>
<td>1138</td>
<td>ER_INVALID_USE_OF_NULL</td>
<td>S1000</td>
<td>42000</td>
</tr>
<tr>
<td>1139</td>
<td>ER_REGEXP_ERROR</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1140</td>
<td>ER_MIX_OF_GROUP_FUNC_AND_FIELDS</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1141</td>
<td>ER_NONEXISTING_GRANT</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1142</td>
<td>ER_TABLEACCESS_DENIED_ERROR</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1143</td>
<td>ER_COLUMNACCESS_DENIED_ERROR</td>
<td>42000</td>
<td>42000</td>
</tr>
</tbody>
</table>
## Mapping MySQL Error Numbers to JDBC SQLState Codes

<table>
<thead>
<tr>
<th>MySQL Error Number</th>
<th>MySQL Error Name</th>
<th>Legacy (X/Open) SQLState</th>
<th>SQL Standard SQLState</th>
</tr>
</thead>
<tbody>
<tr>
<td>1144</td>
<td>ER_ILLEGAL_GRANT_FOR_TABLE</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1145</td>
<td>ER_GRANT_WRONG_HOST_OR_USER</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1146</td>
<td>ER_NO_SUCH_TABLE</td>
<td>42S02</td>
<td>42S02</td>
</tr>
<tr>
<td>1147</td>
<td>ER_NONEXISTING_TABLE_GRANT</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1148</td>
<td>ER_NOT_ALLOWED_COMMAND</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1149</td>
<td>ER_SYNTAX_ERROR</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1152</td>
<td>ER_ABORTING_CONNECTION</td>
<td>08S01</td>
<td>08S01</td>
</tr>
<tr>
<td>1153</td>
<td>ER_NET_PACKET_TOO_LARGE</td>
<td>08S01</td>
<td>08S01</td>
</tr>
<tr>
<td>1154</td>
<td>ER_NET_READ_ERROR_FROM_PIPE</td>
<td>08S01</td>
<td>08S01</td>
</tr>
<tr>
<td>1155</td>
<td>ER_NET_FCNTL_ERROR</td>
<td>08S01</td>
<td>08S01</td>
</tr>
<tr>
<td>1156</td>
<td>ER_NET_PACKETS_OUT_OF_ORDER</td>
<td>08S01</td>
<td>08S01</td>
</tr>
<tr>
<td>1157</td>
<td>ER_NET_UNCOMPRESS_ERROR</td>
<td>08S01</td>
<td>08S01</td>
</tr>
<tr>
<td>1158</td>
<td>ER_NET_READ_ERROR</td>
<td>08S01</td>
<td>08S01</td>
</tr>
<tr>
<td>1159</td>
<td>ER_NET_READ_INTERRUPTED</td>
<td>08S01</td>
<td>08S01</td>
</tr>
<tr>
<td>1160</td>
<td>ER_NET_ERROR_ON_WRITE</td>
<td>08S01</td>
<td>08S01</td>
</tr>
<tr>
<td>1161</td>
<td>ER_NET_WRITE_INTERRUPTED</td>
<td>08S01</td>
<td>08S01</td>
</tr>
<tr>
<td>1162</td>
<td>ER_TOO_LONG_STRING</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1163</td>
<td>ER_TABLE_CANT_HANDLE_BLOB</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1164</td>
<td>ER_TABLE_CANT_HANDLE_AUTO_INCREMENT</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1166</td>
<td>ER_WRONG_COLUMN_NAME</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1167</td>
<td>ER_WRONG_KEY_COLUMN</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1169</td>
<td>ER_DUP_UNIQUE</td>
<td>23000</td>
<td>23000</td>
</tr>
<tr>
<td>1170</td>
<td>ER_BLOB_KEY_WITHOUT_LENGTH</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1171</td>
<td>ER_PRIMARY_CANT_HAVE_NULL</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1172</td>
<td>ER_TOO_MANY_ROWS</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1173</td>
<td>ERQUIRES_PRIMARY_KEY</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1176</td>
<td>ER_KEY_DOES_NOT_EXITS</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1177</td>
<td>ER_CHECK_NO_SUCH_TABLE</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1178</td>
<td>ER_CHECK_NOT_IMPLEMENTED</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1179</td>
<td>ER_CANT_DO_THIS_DURING_AN_TRANSACTION</td>
<td>25000</td>
<td>25000</td>
</tr>
<tr>
<td>1184</td>
<td>ER_NEW_ABORTING_CONNECTION</td>
<td>08S01</td>
<td>08S01</td>
</tr>
<tr>
<td>1189</td>
<td>ER_MASTER_NET_READ</td>
<td>08S01</td>
<td>08S01</td>
</tr>
<tr>
<td>1190</td>
<td>ER_MASTER_NET_WRITE</td>
<td>08S01</td>
<td>08S01</td>
</tr>
<tr>
<td>1203</td>
<td>ER_TOO_MANY_USER_CONNECTIONS</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1205</td>
<td>ER_LOCK_WAIT_TIMEOUT</td>
<td>40001</td>
<td>40001</td>
</tr>
<tr>
<td>1207</td>
<td>ER_READ_ONLY_TRANSACTION</td>
<td>25000</td>
<td>25000</td>
</tr>
<tr>
<td>MySQL Error Number</td>
<td>MySQL Error Name</td>
<td>Legacy (X/Open) SQLState</td>
<td>SQL Standard SQLState</td>
</tr>
<tr>
<td>-------------------</td>
<td>------------------</td>
<td>--------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>1211</td>
<td>ER_NO_PERMISSION_TO_CREATE_USER</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1213</td>
<td>ER_LOCK_DEADLOCK</td>
<td>40001</td>
<td>40001</td>
</tr>
<tr>
<td>1216</td>
<td>ER_NO_REFERENCED_ROW</td>
<td>23000</td>
<td>23000</td>
</tr>
<tr>
<td>1217</td>
<td>ER_ROW_IS_REFERENCED</td>
<td>23000</td>
<td>23000</td>
</tr>
<tr>
<td>1218</td>
<td>ER_CONNECT_TO_MASTER</td>
<td>08S01</td>
<td>08S01</td>
</tr>
<tr>
<td>1222</td>
<td>ER_WRONG_NUMBER_OF_COLUMNS_IN_SELECT</td>
<td>21000</td>
<td>21000</td>
</tr>
<tr>
<td>1226</td>
<td>ER_USER_LIMIT_REACHED</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1227</td>
<td>ER_SPECIFIC_ACCESS_DENIED_ERROR</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1230</td>
<td>ER_NO_DEFAULT</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1231</td>
<td>ER_WRONG_VALUE_FOR_VAR</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1232</td>
<td>ER_WRONG_TYPE_FOR_VAR</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1234</td>
<td>ER_CANT_USE_OPTION_HERE</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1235</td>
<td>ER_NOT_SUPPORTED_YET</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1239</td>
<td>ER_WRONG_FK_DEF</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1241</td>
<td>ER_OPERAND_COLUMNS</td>
<td>21000</td>
<td>21000</td>
</tr>
<tr>
<td>1242</td>
<td>ER_SUBQUERY_NO_1_ROW</td>
<td>21000</td>
<td>21000</td>
</tr>
<tr>
<td>1247</td>
<td>ER_ILLEGAL_REFERENCE</td>
<td>42S22</td>
<td>42S22</td>
</tr>
<tr>
<td>1248</td>
<td>ER_DERIVED_MUST_HAVE_ALIAS</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1249</td>
<td>ER_SELECT_REduced</td>
<td>01000</td>
<td>01000</td>
</tr>
<tr>
<td>1250</td>
<td>ER_TABLENAME_NOT_ALLOWED_HERE</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1251</td>
<td>ER_NOT_SUPPORTED_AUTH_MODE</td>
<td>08004</td>
<td>08004</td>
</tr>
<tr>
<td>1252</td>
<td>ER_SPATIAL_CANT_HAVE_NULL</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1253</td>
<td>ER_COLLATION_CHARSET_MISMATCH</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1261</td>
<td>ER_WARN_TOO_FEW_RECORDS</td>
<td>01000</td>
<td>01000</td>
</tr>
<tr>
<td>1262</td>
<td>ER_WARN_TOO_MANY_RECORDS</td>
<td>01000</td>
<td>01000</td>
</tr>
<tr>
<td>1263</td>
<td>ER_WARN_NULL_TO_NOTNULL</td>
<td>S1000</td>
<td>01000</td>
</tr>
<tr>
<td>1264</td>
<td>ER_WARN_DATA_OUT_OF_RANGE</td>
<td>01000</td>
<td>01000</td>
</tr>
<tr>
<td>1265</td>
<td>ER_WARN_DATA_TRUNCATED</td>
<td>01000</td>
<td>01000</td>
</tr>
<tr>
<td>1280</td>
<td>ER_WRONG_NAME_FOR_INDEX</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1281</td>
<td>ER_WRONG_NAME_FOR_CATALOG</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1286</td>
<td>ER_UNKNOWN_STORAGEENGINE</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1292</td>
<td>ER_TRUNCATED_WRONG_VALUE</td>
<td>22007</td>
<td>22007</td>
</tr>
<tr>
<td>1303</td>
<td>ER_SP_NO_RECURSIVE_CREATE</td>
<td>S1000</td>
<td>2F003</td>
</tr>
<tr>
<td>1304</td>
<td>ER_SP_ALREADY_EXISTS</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1305</td>
<td>ER_SP_DOES_NOT_EXIST</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1308</td>
<td>ER_SP_LILABEL_MISMATCH</td>
<td>42000</td>
<td>42000</td>
</tr>
</tbody>
</table>
### Mapping MySQL Error Numbers to JDBC SQLState Codes

<table>
<thead>
<tr>
<th>MySQL Error Number</th>
<th>MySQL Error Name</th>
<th>Legacy (X/Open) SQLState</th>
<th>SQL Standard SQLState</th>
</tr>
</thead>
<tbody>
<tr>
<td>1309</td>
<td>ER_SP_LABEL_REDEFINE</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1310</td>
<td>ER_SP_LABEL_MISMATCH</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1311</td>
<td>ER_SP_UNINIT_VAR</td>
<td>01000</td>
<td>01000</td>
</tr>
<tr>
<td>1312</td>
<td>ER_SP_BADSELECT</td>
<td>0A000</td>
<td>0A000</td>
</tr>
<tr>
<td>1313</td>
<td>ER_SP_BADRETURN</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1314</td>
<td>ER_SP_BADSTATEMENT</td>
<td>0A000</td>
<td>0A000</td>
</tr>
<tr>
<td>1315</td>
<td>ER_UPDATE_LOG_DEPRECATED_IGNORED</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1316</td>
<td>ER_UPDATE_LOG_DEPRECATED_TRANSLATED</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1317</td>
<td>ER_QUERY_INTERRUPTED</td>
<td>S1000</td>
<td>70100</td>
</tr>
<tr>
<td>1318</td>
<td>ER_SP_WRONG_NO_OF_ARGS</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1319</td>
<td>ER_SP_COND_MISMATCH</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1320</td>
<td>ER_SP_NORETURN</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1321</td>
<td>ER_SP_NORETURNEND</td>
<td>S1000</td>
<td>2F005</td>
</tr>
<tr>
<td>1322</td>
<td>ER_SP_BAD_CURSOR_QUERY</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1323</td>
<td>ER_SP_BAD_CURSOR_SELECT</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1324</td>
<td>ER_SP_CURSOR_MISMATCH</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1325</td>
<td>ER_SP_CURSOR_ALREADY_OPEN</td>
<td>24000</td>
<td>24000</td>
</tr>
<tr>
<td>1326</td>
<td>ER_SP_CURSOR_NOT_OPEN</td>
<td>24000</td>
<td>24000</td>
</tr>
<tr>
<td>1327</td>
<td>ER_SP_UNDECLARED_VAR</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1329</td>
<td>ER_SP_FETCH_NO_DATA</td>
<td>S1000</td>
<td>02000</td>
</tr>
<tr>
<td>1330</td>
<td>ER_SP_DUP_PARAM</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1331</td>
<td>ER_SP_DUP_VAR</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1332</td>
<td>ER_SP_DUP_COND</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1333</td>
<td>ER_SP_DUP_CURS</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1335</td>
<td>ER_SP_SUBSELECT_NYI</td>
<td>0A000</td>
<td>0A000</td>
</tr>
<tr>
<td>1336</td>
<td>ER_STMT_NOT_ALLOWED_IN_SF_OR_TRG</td>
<td>0A000</td>
<td>0A000</td>
</tr>
<tr>
<td>1337</td>
<td>ER_SP_VARCOND_AFTER_CURSHNDLR</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1338</td>
<td>ER_SP_CURSOR_AFTER_HANDLER</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1339</td>
<td>ER_SP_CASE_NOT_FOUND</td>
<td>S1000</td>
<td>20000</td>
</tr>
<tr>
<td>1365</td>
<td>ER_DIVISION_BY_ZERO</td>
<td>22012</td>
<td>22012</td>
</tr>
<tr>
<td>1367</td>
<td>ER_ILLEGAL_VALUE_FOR_TYPE</td>
<td>22007</td>
<td>22007</td>
</tr>
<tr>
<td>1370</td>
<td>ER_PROCACCESS_DENIED_ERROR</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1397</td>
<td>ER_XAER_NOTA</td>
<td>S1000</td>
<td>XAE04</td>
</tr>
<tr>
<td>1398</td>
<td>ER_XAER_INVAL</td>
<td>S1000</td>
<td>XAE05</td>
</tr>
<tr>
<td>1399</td>
<td>ER_XAER_RMFAIL</td>
<td>S1000</td>
<td>XAE07</td>
</tr>
<tr>
<td>1400</td>
<td>ER_XAER_OUTSIDE</td>
<td>S1000</td>
<td>XAE09</td>
</tr>
</tbody>
</table>
### Mapping MySQL Error Numbers to JDBC SQLState Codes

<table>
<thead>
<tr>
<th>MySQL Error Number</th>
<th>MySQL Error Name</th>
<th>Legacy (X/Open) SQLState</th>
<th>SQL Standard SQLState</th>
</tr>
</thead>
<tbody>
<tr>
<td>1401</td>
<td>ER_XA_RMERR</td>
<td>S1000</td>
<td>XAE03</td>
</tr>
<tr>
<td>1402</td>
<td>ER_XA_RBROLLBACK</td>
<td>S1000</td>
<td>XA103</td>
</tr>
<tr>
<td>1403</td>
<td>ER_NONEXISTING_PROC_GRANT</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1406</td>
<td>ER_DATA_TOO_LONG</td>
<td>22001</td>
<td>22001</td>
</tr>
<tr>
<td>1407</td>
<td>ER_SP_BAD_SQLSTATE</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1410</td>
<td>ER_CANT_CREATE_USER_WITH_GRANT</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1413</td>
<td>ER_SP_DUP_HANDLER</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1414</td>
<td>ER_SP_NOT_VAR_ARG</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1415</td>
<td>ER_SP_NO_RETSET</td>
<td>0A000</td>
<td>0A000</td>
</tr>
<tr>
<td>1416</td>
<td>ER_CANT_CREATE_GEOMETRY_OBJECT</td>
<td>22003</td>
<td>22003</td>
</tr>
<tr>
<td>1425</td>
<td>ER_TOO_BIG_SCALE</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1426</td>
<td>ER_TOO_BIG_PRECISION</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1427</td>
<td>ER_M_BIGGER_THAN_D</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1437</td>
<td>ER_TOO_LONG_BODY</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1439</td>
<td>ER_TOO_BIG_DISPLAYWIDTH</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1440</td>
<td>ER_XAER_DUPID</td>
<td>S1000</td>
<td>XAE08</td>
</tr>
<tr>
<td>1441</td>
<td>ER_DATETIME_FUNCTION_OVERFLOW</td>
<td>22008</td>
<td>22008</td>
</tr>
<tr>
<td>1451</td>
<td>ER_ROW_IS_REFERENCED_2</td>
<td>23000</td>
<td>23000</td>
</tr>
<tr>
<td>1452</td>
<td>ER_NO_REFERENCED_ROW_2</td>
<td>23000</td>
<td>23000</td>
</tr>
<tr>
<td>1453</td>
<td>ER_SP_BAD_VAR_SHADOW</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1458</td>
<td>ER_SP_WRONG_NAME</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1460</td>
<td>ER_SP_NO_AGGREGATE</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1461</td>
<td>ER_MAX_PREPARED_STMT_COUNT_REACHED</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1463</td>
<td>ER_NON_GROUPING_FIELD_USED</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1557</td>
<td>ER_FOREIGN_DUPLICATE_KEY</td>
<td>23000</td>
<td>23000</td>
</tr>
<tr>
<td>1582</td>
<td>ER_WRONG_PARAMCOUNT_TO_NATIVE_FCT</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1583</td>
<td>ER_WRONG_PARAMETERS_TO_NATIVE_FCT</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1584</td>
<td>ER_WRONG_PARAMETERS_TO_STORED_FCT</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1586</td>
<td>ER_DUP_ENTRY_WITH_KEY_NAME</td>
<td>23000</td>
<td>23000</td>
</tr>
<tr>
<td>1613</td>
<td>ER_XA_RBTIMEOUT</td>
<td>S1000</td>
<td>XA106</td>
</tr>
<tr>
<td>1614</td>
<td>ER_XA_RBDEADLOCK</td>
<td>S1000</td>
<td>XA102</td>
</tr>
<tr>
<td>1630</td>
<td>ER_FUNC_INEXISTENT_NAME_COLLISION</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1641</td>
<td>ER_DUP_SIGNAL_SET</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1642</td>
<td>ER_SIGNAL_WARN</td>
<td>01000</td>
<td>01000</td>
</tr>
<tr>
<td>1643</td>
<td>ER_SIGNAL_NOT_FOUND</td>
<td>S1000</td>
<td>02000</td>
</tr>
<tr>
<td>MySQL Error Number</td>
<td>MySQL Error Name</td>
<td>Legacy (X/Open) SQLState</td>
<td>SQL Standard SQLState</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------------------------------------------------------</td>
<td>--------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>1645</td>
<td>ER_RESIGNAL_WITHOUT_ACTIVE_HANDLER</td>
<td>S1000</td>
<td>0K000</td>
</tr>
<tr>
<td>1687</td>
<td>ER_SPATIAL_MUST_HAVE_GEOM_COL</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1690</td>
<td>ER_DATA_OUT_OF_RANGE</td>
<td>22003</td>
<td>22003</td>
</tr>
<tr>
<td>1698</td>
<td>ER_ACCESS_DENIED_NO_PASSWORD_ERROR</td>
<td>28000</td>
<td>28000</td>
</tr>
<tr>
<td>1701</td>
<td>ER_TRUNCATE_ILLEGAL_FK</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td>1758</td>
<td>ER_DA_INVALID_CONDITION_NUMBER</td>
<td>35000</td>
<td>35000</td>
</tr>
<tr>
<td>1761</td>
<td>ER_FOREIGN_DUPLICATE_KEY_WITH_CHILD_INFO</td>
<td>23000</td>
<td>23000</td>
</tr>
<tr>
<td>1762</td>
<td>ER_FOREIGN_DUPLICATE_KEY_WITHOUT_CHILD_INFO</td>
<td>23000</td>
<td>23000</td>
</tr>
<tr>
<td>1792</td>
<td>ER_CANT_EXECUTE_IN_READ_ONLY_TRANSACTION</td>
<td>S1000</td>
<td>25006</td>
</tr>
<tr>
<td>1845</td>
<td>ER_ALTER_OPERATION_NOT_SUPPORTED</td>
<td>0A000</td>
<td>0A000</td>
</tr>
<tr>
<td>1846</td>
<td>ER_ALTER_OPERATION_NOT_SUPPORTED_REASON</td>
<td>0A000</td>
<td>0A000</td>
</tr>
<tr>
<td>1859</td>
<td>ER_DUP_UNKNOWN_IN_INDEX</td>
<td>23000</td>
<td>23000</td>
</tr>
<tr>
<td>1873</td>
<td>ER_ACCESS_DENIED_CHANGE_USER_ERROR</td>
<td>28000</td>
<td>28000</td>
</tr>
<tr>
<td>1887</td>
<td>ER_GET_STACKED_DA_WITHOUT_ACTIVE_HANDLER</td>
<td>S1000</td>
<td>0Z002</td>
</tr>
<tr>
<td>1903</td>
<td>ER_INVALID_ARGUMENT_FOR_LOGARITHM</td>
<td>S1000</td>
<td>2201E</td>
</tr>
</tbody>
</table>
Chapter 6 JDBC Concepts

Table of Contents

6.1 Connecting to MySQL Using the JDBC DriverManager Interface ............................................. 71
6.2 Using JDBC Statement Objects to Execute SQL ............................................................................ 72
6.3 Using JDBC CallableStatements to Execute Stored Procedures ................................................. 73
6.4 Retrieving AUTO_INCREMENT Column Values through JDBC .................................................... 76

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.

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

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.

```java
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 SQL) 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

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.PreparedStatement;
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 (Procedures and Functions).

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()`**

   ```java
   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**

   ```java
   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

...
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

```java
... 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()`

```java
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()

```java
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

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

```java
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.
        */
        
        try {
            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
                */
            }
        } finally {
            /*
            * close any jdbc instances here that weren't
            */
        }
    }
}
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.

* 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;
}
}
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:

```java
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
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

8.1 Configuring Server Failover ................................................................. 85
8.2 Configuring Load Balancing with Connector/J ......................................... 88
8.3 Configuring Master/Slave Replication with Connector/J .......................... 90
8.4 Advanced Load-balancing and Failover Configuration .............................. 94

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):
Configuring Connection Access Mode

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 we are 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`

```java
jdbc:mysql://[primary host][:port],[secondary host 1][:port],[secondary host 2][:port]...[/[database]]
{?propertyName1=propertyValue1&propertyName2=propertyValue2}...
```
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 master 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 master-master 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:

```java
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 getConnection() 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();
        }
    }
}
```
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 Master/Slave 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://[master host][:port],[slave host 1][:port],[slave 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 master hosts are reachable. Such `Connection` objects report they are read-only, and `isMasterConnection()` returns false for them. The `Connection` tests for available master hosts when `Connection.setReadOnly(false)` is called, throwing an `SQLException` if it cannot establish a connection to a master, or switching to a master 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 slave hosts are reachable. A `Connection` then, at runtime, tests for available slave hosts when `Connection.setReadOnly(true)` is called (see explanation for the method below), throwing an `SQLException` if it cannot establish a connection to a slave, 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 Slaves

Connector/J 3.1.7 and higher includes a variant of the driver that will automatically send queries to a read/write master, or a failover or round-robin loadbalanced set of slaves 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 slave connections, which are load-balanced per slave host using a round-robin scheme. A given connection is sticky to a slave until a transaction boundary command (a commit or rollback) is issued, or until the slave 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 master when no slaves are available, set the property readFromMasterWhenNoSlaves to “true.” Notice that the master host will be used in read-only state in those cases, as if it is a slave host. Also notice that setting readFromMasterWhenNoSlaves=true might result in an extra load for the master 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 master 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:

```java
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 slaves
        props.put("autoReconnect", "true");
        // We want to load balance between the slaves
        props.put("roundRobinLoadBalance", "true");
        props.put("user", "foo");
        props.put("password", "bar");
        // Looks like a normal MySQL JDBC url, with a
        // comma-separated list of hosts, the first
        // being the 'master', the rest being any number
        // of slaves that the driver will load balance against
    }
}
```
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-Master Replication Topographies

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

The connection URL for replication discussed earlier (i.e., in the format of `jdbc:mysql:replication://master,slave1,slave2,slave3/test`) assumes that the first (and only the first) host is the master. Supporting deployments with an arbitrary number of masters and slaves requires a different URL syntax for specifying the hosts and the properties for specific hosts, which is just an expansion of the URL syntax discussed in IPv6 Connections with the property `type=[master|slave]`; for example:

```
jdbc:mysql://address=(type=master)(host=master1host),address=(type=master)(host=master2host),address=(type=slave)(host=slave1host)/database
```

Connector/J uses a load-balanced connection internally for management of the master connections, which means that `ReplicationConnection`, when configured to use multiple masters, exposes the same options to balance load across master 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-master) topographies is also supported. This enables users to promote slaves 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
ReplicationConnectionGroupManager 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 masters
- getSlaveHosts(): Returns a collection of strings representing the hosts configured as slaves
- addSlaveHost(String host): Adds new host to pool of possible slave hosts for selection at start of new read-only workload
- promoteSlaveToMaster(String host): Removes the host from the pool of potential slaves for future read-only processes (existing read-only process is allowed to continue to completion) and adds the host to the pool of potential master hosts
- removeSlaveHost(String host, boolean closeGently): Removes the host (host name match must be exact) from the list of configured slaves; 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 masters

Some useful management metrics include:

- getConnectionCountWithHostAsSlave(String host): Returns the number of ReplicationConnection objects that have the given host configured as a possible slave
- getConnectionCountWithHostAsMaster(String host): Returns the number of ReplicationConnection objects that have the given host configured as a possible master
- getNumberOfSlavesAdded(): Returns the number of times a slave host has been dynamically added to the group pool
- getNumberOfSlavesRemoved(): Returns the number of times a slave host has been dynamically removed from the group pool
- getNumberOfSlavePromotions(): Returns the number of times a slave host has been promoted to a master
- 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
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:

```java
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-master deployments, as explained in Section 8.2, "Configuring Load Balancing with Connector/J" and Support for Multiple-Master Replication Topographies. As of Connector/J 5.1.12, this same implementation is used for balancing load between read-only slaves 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` 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:

```java
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:

```java
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":

```java
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:

```java
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 slaves. 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”:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadBalanceAutoCommitStatementThreshold</td>
<td>3</td>
</tr>
<tr>
<td>loadBalanceAutoCommitStatementRegex</td>
<td>.<em>test.</em></td>
</tr>
</tbody>
</table>

**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 5.1, “Driver/Datasource Class Names, URL Syntax and 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 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:

```xml
<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.
These instructions cover JBoss-4.x. To make the JDBC driver classes available to the application server, copy the .jar file that comes with Connector/J to 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:

```xml
<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>
```
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:
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.

```java
// 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.

### 12.1 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.

```java
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
generateRandomCityByCountryCode().

// 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.
12.2 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:

```java
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.

```java
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.

```xml
<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.

```xml
<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.

```xml
<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.

```java
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.

**12.3 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.

```xml
<bean id="dataSource" destroy-method="close"/>
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.
Chapter 13 Using Connector/J with GlassFish

Table of Contents

13.1 A Simple JSP Application with GlassFish, Connector/J and MySQL ............................................. 110
13.2 A Simple Servlet with GlassFish, Connector/J and MySQL ............................................................. 112

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.

### 13.1 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:

```jsp
<%@ 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");
    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
<?xml version="1.0" encoding="UTF-8"?>
    <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
<?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>
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:

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

You will still be able to access the data source directly using:

```java
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.

### 13.2 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
```
The code for the servlet, located in `HelloWebServlet.java`, is as follows:

```java
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>");
        }
```
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 13.1, “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
<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 13.1, “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 13.1, “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 14 Using Connector/J with MySQL Fabric

MySQL Fabric is a system for managing a farm of MySQL servers (and other components). Fabric provides an extensible and easy to use system for managing a MySQL deployment for sharding and high-availability.

For more information on MySQL Fabric, see MySQL Fabric. For instructions on how to use Connector/J with MySQL Fabric, see Using Connector/J with MySQL Fabric.
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**: I am trying to connect to my MySQL server within my application, but I get the following error and stack trace:

```
java.net.SocketException
MESSAGE: Software caused connection abort: recv failed
STACKTRACE:
java.net.SocketException: Software caused connection abort: recv failed
at java.net.SocketInputStream.socketRead0(Native Method)
at java.net.SocketInputStream.read(Unknown Source)
at com.mysql.jdbc.MysqlIO.readFully(MysqlIO.java:1392)
at com.mysql.jdbc.MysqlIO.readPacket(MysqlIO.java:1414)
at com.mysql.jdbc.MysqlIO.doHandshake(MysqlIO.java:625)
at com.mysql.jdbc.Connection.createNewIO(Connection.java:1926)
at com.mysql.jdbc.Connection.<init>(Connection.java:452)
at com.mysql.jdbc.NonRegisteringDriver.connect(NonRegisteringDriver.java:411)
```

• **15.8**: 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.
• **15.9:** [124] When using gcj, a `java.io.CharConversionException` exception is raised when working with certain character sequences.

• **15.10:** [124] 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.

• **15.11:** [125] 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.

• **15.12:** [125] 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”?

• **15.13:** [126] 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?

• **15.14:** [127] How can I use 3-byte UTF8 with Connector/J?

• **15.15:** [127] How can I use 4-byte UTF8, `utf8mb4` with Connector/J?

• **15.16:** [127] 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
SQLException: 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 Syntax](#), 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 \texttt{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:

\begin{verbatim}
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
\end{verbatim}

Either you're running an Applet, your MySQL server has been installed with the "skip-networking" option set, 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" flag, or if it is firewalled.

If MySQL has been started with the "skip-networking" option set (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 5.1, "Driver/Datasource Class Names, URL Syntax and 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**

\begin{verbatim}
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)?
    //
    
\end{verbatim}
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;
        }
    }
} 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;
    }
}
```java
} 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 {
            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** option 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: I am trying to connect to my MySQL server within my application, but I get the following error and stack trace:**

```
java.net.SocketException
MESSAGE: Software caused connection abort: recv failed

STACKTRACE:
java.net.SocketException: Software caused connection abort: recv failed
  at java.net.SocketInputStream.socketRead0(Native Method)
  at java.net.SocketInputStream.read(Unknown Source)
  at com.mysql.jdbc.MysqlIO.readFully(MysqlIO.java:1392)
  at com.mysql.jdbc.MysqlIO.readPacket(MysqlIO.java:1414)
  at com.mysql.jdbc.MysqlIO.doHandshake(MysqlIO.java:625)
  at com.mysql.jdbc.Connection.createNewIO(Connection.java:1926)
  at com.mysql.jdbc.Connection.<init>(Connection.java:452)
  at com.mysql.jdbc.NonRegisteringDriver.connect(NonRegisteringDriver.java:411)
```

The error probably indicates that you are using a older version of the Connector/J JDBC driver (2.0.14 or 3.0.x) and you are trying to connect to a MySQL server with version 4.1x or newer. The older drivers are not compatible with 4.1 or newer of MySQL as they do not support the newer authentication mechanisms.

It is likely that the older version of the Connector/J driver exists within your application directory or your **CLASSPATH** includes the older Connector/J package.

**15.8: 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.

**15.9: 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.

**15.10: 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`.

15.11: 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.

15.12: 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.
15.13: 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:

```java
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.

**15.14: How can I use 3-byte UTF8 with Connector/J?**

To use 3-byte UTF8 with Connector/J set `characterEncoding=utf8` and set `useUnicode=true` in the connection string.

**15.15: 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 as long as `characterEncoding` has not been set in the connection string. This is equivalent to autodetection of the character set.

**15.16: 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:

- 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:

```plaintext
useTimezone=true
useLegacyDatetimeCode=false
serverTimezone=UTC
```
Chapter 17 Connector/J Support

Table of Contents

17.1 Connector/J Community Support ................................................................. 131
17.2 How to Report Connector/J Bugs or Problems ........................................... 131

17.1 Connector/J Community Support

Oracle provides assistance to the user community by means of its mailing lists. For Connector/J related issues, you can get help from experienced users by using the MySQL and Java mailing list. Archives and subscription information is available online at http://lists.mysql.com/java.

For information about subscribing to MySQL mailing lists or to browse list archives, visit http://lists.mysql.com/. See MySQL Mailing Lists.

Community support from experienced users is also available through the JDBC Forum. You may also find help from other users in the other MySQL Forums, located at http://forums.mysql.com. See MySQL Community Support at the MySQL Forums.

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).
How to Report Connector/J Bugs or Problems

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:

```java
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/.
Appendix A Licenses for Third-Party Components

Table of Contents

A.1 Ant-Contrib License ................................................................. 133
A.2 c3p0 JDBC Library License ...................................................... 134
A.3 GNU Lesser General Public License Version 2.1, February 1999 .......... 134
A.4 jboss-common-jdbc-wrapper.jar License ..................................... 142
A.5 NanoXML License ................................................................. 142
A.6 rox.jar License ................................................................. 143
A.7 Simple Logging Facade for Java (SLF4J) License .......................... 143
A.8 Unicode Data Files .............................................................. 144

MySQL Connector/J 5.1

- Section A.1, “Ant-Contrib License”
- Section A.2, “c3p0 JDBC Library License”
- Section A.3, “GNU Lesser General Public License Version 2.1, February 1999”
- Section A.4, “jboss-common-jdbc-wrapper.jar License”
- Section A.5, “NanoXML License”
- Section A.6, “rox.jar License”
- Section A.7, “Simple Logging Facade for Java (SLF4J) License”
- Section A.8, “Unicode Data Files”

A.1 Ant-Contrib License

The following software may be included in this product up to version 5.1.26: Ant-Contrib

Ant-Contrib
Copyright (c) 2001-2003 Ant-Contrib project. All rights reserved.
Licensed under the Apache 1.1 License Agreement, a copy of which is reproduced below.

The Apache Software License, Version 1.1

Copyright (c) 2001-2003 Ant-Contrib project. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. The end-user documentation included with the redistribution, if any, must include the following acknowledgement:
   "This product includes software developed by the Ant-Contrib project (http://sourceforge.net/projects/ant-contrib)."
4. The name Ant-Contrib must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact ant-contrib-developers@lists.sourceforge.net.

5. Products derived from this software may not be called "Ant-Contrib" nor may "Ant-Contrib" appear in their names without prior written permission of the Ant-Contrib project.

THIS SOFTWARE IS PROVIDED "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE ANT-CONTRIB PROJECT OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

A.2 c3p0 JDBC Library License

You are receiving a copy of c3p0-0.9.1-pre6.jar in both source and object code in the following /src/lib/c3p0-0.9.1-pre6.jar. The terms of the Oracle license do NOT apply to c3p0-0.9.1-pre6.jar; it is licensed under the following license, separately from the Oracle programs you receive. If you do not wish to install this library, you may remove the file /src/lib/c3p0-0.9.1-pre6.jar, but the Oracle program might not operate properly or at all without the library.

This component is licensed under Section A.3, "GNU Lesser General Public License Version 2.1, February 1999".

A.3 GNU Lesser General Public License Version 2.1, February 1999

The following applies to all products licensed under the GNU Lesser General Public License, Version 2.1: You may not use the identified files except in compliance with the GNU Lesser General Public License, Version 2.1 (the "License"). You may obtain a copy of the License at http://www.gnu.org/licenses/lgpl-2.1.html. A copy of the license is also reproduced below. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

GNU LESSER GENERAL PUBLIC LICENSE
Version 2.1, February 1999

Copyright (C) 1991, 1999 Free Software Foundation, Inc.
51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

[This is the first released version of the Lesser GPL. It also counts as the successor of the GNU Library Public License, version 2, hence the version number 2.1.]
The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users.

This license, the Lesser General Public License, applies to some specially designated software packages--typically libraries--of the Free Software Foundation and other authors who decide to use it. You can use it too, but we suggest you first think carefully about whether this license or the ordinary General Public License is the better strategy to use in any particular case, based on the explanations below.

When we speak of free software, we are referring to freedom of use, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish); that you receive source code or can get it if you want it; that you can change the software and use pieces of it in new free programs; and that you are informed that you can do these things.

To protect your rights, we need to make restrictions that forbid distributors to deny you these rights or to ask you to surrender these rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link other code with the library, you must provide complete object files to the recipients, so that they can relink them with the library after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method: (1) we copyright the library, and (2) we offer you this license, which gives you legal permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that there is no warranty for the free library. Also, if the library is modified by someone else and passed on, the recipients should know that what they have is not the original version, so that the original author's reputation will not be affected by problems that might be introduced by others.

Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a restrictive license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and is quite different from the ordinary General Public License. We use this license for certain libraries in order to permit linking those libraries into non-free programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.
We call this license the "Lesser" General Public License because it does Less to protect the user's freedom than the ordinary General Public License. It also provides other free software developers Less of an advantage over competing non-free programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.

For example, on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes a de-facto standard. To achieve this, non-free programs must be allowed to use the library. A more frequent case is that a free library does the same job as widely used non-free libraries. In this case, there is little to gain by limiting the free library to free software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in non-free programs enables a greater number of people to use a large body of free software. For example, permission to use the GNU C Library in non-free programs enables many more people to use the whole GNU operating system, as well as its variant, the GNU/Linux operating system.

Although the Lesser General Public License is Less protective of the users' freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, whereas the latter must be combined with the library in order to run.

GNU LESSER GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called "this License"). Each licensee is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification").

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for

136
writing it). Whether that is true depends on what the Library does
and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's
complete source code as you receive it, in any medium, provided that
you conspicuously and appropriately publish on each copy an
appropriate copyright notice and disclaimer of warranty; keep intact
all the notices that refer to this License and to the absence of any
warranty; and distribute a copy of this License along with the
Library.

You may charge a fee for the physical act of transferring a copy,
and you may at your option offer warranty protection in exchange for a
fee.

2. You may modify your copy or copies of the Library or any portion
of it, thus forming a work based on the Library, and copy and
distribute such modifications or work under the terms of Section 1
above, provided that you also meet all of these conditions:

a) The modified work must itself be a software library.

b) You must cause the files modified to carry prominent notices
stating that you changed the files and the date of any change.

c) You must cause the whole of the work to be licensed at no
charge to all third parties under the terms of this License.

d) If a facility in the modified Library refers to a function or a
table of data to be supplied by an application program that uses
the facility, other than as an argument passed when the facility
is invoked, then you must make a good faith effort to ensure that,
in the event an application does not supply such function or
table, the facility still operates, and performs whatever part of
its purpose remains meaningful.

(For example, a function in a library to compute square roots has
a purpose that is entirely well-defined independent of the
application. Therefore, Subsection 2d requires that any
application-supplied function or table used by this function must
be optional: if the application does not supply it, the square
root function must still compute square roots.)

These requirements apply to the modified work as a whole. If
identifiable sections of that work are not derived from the Library,
and can be reasonably considered independent and separate works in
themselves, then this License, and its terms, do not apply to those
sections when you distribute them as separate works. But when you
distribute the same sections as part of a whole which is a work based
on the Library, the distribution of the whole must be on the terms of
this License, whose permissions for other licensees extend to the
entire whole, and thus to each and every part regardless of who wrote
it.

Thus, it is not the intent of this section to claim rights or contest
your rights to work written entirely by you; rather, the intent is to
exercise the right to control the distribution of derivative or
collective works based on the Library.

In addition, mere aggregation of another work not based on the Library
with the Library (or with a work based on the Library) on a volume of
a storage or distribution medium does not bring the other work under
the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public
License instead of this License to a given copy of the Library. To do
this, you must alter all the notices that refer to this License, so
that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work
during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", an object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

b) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user's computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.

c) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

d) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

e) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.

b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.
8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties with this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new
versions of the Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Libraries

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change. You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the ordinary General Public License).

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the library's name and a brief idea of what it does.>  Copyright (C) <year>  <name of author>

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.
This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA

Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the library, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the library 'Frob' (a library for tweaking knobs) written by James Random Hacker.

<signature of Ty Coon>, 1 April 1990
Ty Coon, President of Vice

That's all there is to it!

A.4 jboss-common-jdbc-wrapper.jar License

You are receiving a copy of jboss-common-jdbc-wrapper.jar in both source and object code in the following /src/lib/jboss-common-jdbc-wrapper.jar. The terms of the Oracle license do NOT apply to jboss-common-jdbc-wrapper.jar; it is licensed under the following license, separately from the Oracle programs you receive. If you do not wish to install this library, you may remove the file /src/lib/jboss-common-jdbc-wrapper.jar, but the Oracle program might not operate properly or at all without the library.

This component is licensed under Section A.3, “GNU Lesser General Public License Version 2.1, February 1999”.

A.5 NanoXML License

The following software may be included in this product:

NanoXML

* Copyright (C) 2000-2002 Marc De Scheemaeker, All Rights Reserved.
  *
  * This software is provided 'as-is', without any express or implied warranty.
  * In no event will the authors be held liable for any damages arising from the
  * use of this software.
  *
  * Permission is granted to anyone to use this software for any purpose,
  * including commercial applications, and to alter it and redistribute it
  * freely, subject to the following restrictions:
  *
  * 1. The origin of this software must not be misrepresented; you must not
     * claim that you wrote the original software. If you use this software in
     * a product, an acknowledgment in the product documentation would be
     * appreciated but is not required.
     *
  * 2. Altered source versions must be plainly marked as such, and must not be
     * misrepresented as being the original software.
A.6 rox.jar License

The following software may be included in this product:

rox.jar

Copyright (c) 2006, James Greenfield
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
* Neither the name of the <ORGANIZATION> nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

A.7 Simple Logging Facade for Java (SLF4J) License

The following software may be included in this product:

Simple Logging Facade for Java (SLF4J)

Copyright (c) 2004-2008 QOS.ch
All rights reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
A.8 Unicode Data Files

The following software may be included in this product:

Unicode Data Files

COPYRIGHT AND PERMISSION NOTICE

Copyright © 1991-2014 Unicode, Inc. All rights reserved. Distributed under the Terms of Use in http://www.unicode.org/copyright.html.

Permission is hereby granted, free of charge, to any person obtaining a copy of the Unicode data files and any associated documentation (the "Data Files") or Unicode software and any associated documentation (the "Software") to deal in the Data Files or Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Data Files or Software, and to permit persons to whom the Data Files or Software are furnished to do so, provided that (a) the above copyright notice(s) and this permission notice appear with all copies of the Data Files or Software, (b) both the above copyright notice(s) and this permission notice appear in associated documentation, and (c) there is clear notice in each modified Data File or in the Software as well as in the documentation associated with the Data File(s) or Software that the data or software has been modified.

THE DATA FILES AND SOFTWARE ARE PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THE DATA FILES OR SOFTWARE.

Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in these Data Files or Software without prior written authorization of the copyright holder.