

MySQL Connector/J Developer's Guide

MySQL Connector/J Developer's Guide

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 release notes detailing the changes in each release of Connector/J, see [MySQL Connector/J Release Notes](#).

Document generated on: 2013-10-23 (revision: 36500)

Table of Contents

Preface and Legal Notices	v
1. MySQL Connector/J	1
2. Connector/J Versions	3
2.1. Connector/J Release Notes and Change History	3
2.2. Java Versions Supported	3
3. Connector/J Installation	5
3.1. Installing Connector/J from a Binary Distribution	5
3.2. Installing the Driver and Configuring the <code>CLASSPATH</code>	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 the Development Source Tree	9
4. Connector/J Examples	11
5. Connector/J (JDBC) Reference	13
5.1. Driver/Datasource Class Names, URL Syntax and Configuration Properties for Connector/J	13
5.1.1. Properties Files for the <code>useConfigs</code> Option	32
5.2. JDBC API Implementation Notes	33
5.3. Java, JDBC and MySQL Types	37
5.4. Using Character Sets and Unicode	39
5.5. Connecting Securely Using SSL	40
5.6. Connecting Using PAM Authentication	43
5.7. Using Master/Slave Replication with <code>ReplicationConnection</code>	43
5.8. Mapping MySQL Error Numbers to JDBC <code>SQLState</code> Codes	44
6. JDBC Concepts	49
6.1. Connecting to MySQL Using the JDBC <code>DriverManager</code> Interface	49
6.2. Using JDBC <code>Statement</code> Objects to Execute SQL	50
6.3. Using JDBC <code>CallableStatements</code> to Execute Stored Procedures	51
6.4. Retrieving <code>AUTO_INCREMENT</code> Column Values through JDBC	54
7. Connection Pooling with Connector/J	59
8. Load Balancing with Connector/J	63
9. Failover with Connector/J	67
10. Using the Connector/J Interceptor Classes	69
11. Using Connector/J with Tomcat	71
12. Using Connector/J with JBoss	73
13. Using Connector/J with Spring	75
13.1. Using <code>JdbcTemplate</code>	76
13.2. Transactional JDBC Access	77
13.3. Connection Pooling with Spring	79
14. Using Connector/J with GlassFish	81
14.1. A Simple JSP Application with Glassfish, Connector/J and MySQL	82
14.2. A Simple Servlet with Glassfish, Connector/J and MySQL	84
15. Troubleshooting Connector/J Applications	89
16. Known Issues and Limitations	99
17. Connector/J Support	101
17.1. Connector/J Community Support	101
17.2. How to Report Connector/J Bugs or Problems	101
A. Licenses for Third-Party Components	103
A.1. Ant-Contrib License	103
A.2. c3p0 JDBC Library License	104

A.3. GNU Lesser General Public License Version 2.1, February 1999	104
A.4. jboss-common-jdbc-wrapper.jar License	112
A.5. Simple Logging Facade for Java (SLF4J) License	112

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 © 1997, 2013, 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 software or related documentation is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle USA, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This software 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 which may create a risk of personal injury. If you use this software in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure the safe use of this software. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software in dangerous applications.

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. MySQL is a trademark of Oracle Corporation and/or its affiliates, and shall not be used without Oracle's express written authorization. Other names may be trademarks of their respective owners.

This software and documentation may provide access to or information on 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. 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.

This document in any form, software or printed matter, contains proprietary information that is the exclusive property of Oracle. Your access to and use of this material is subject to the terms and conditions of your Oracle Software License and Service Agreement, which has been executed and with which you agree to comply. This document and information contained herein may not be disclosed, copied, reproduced, or distributed to anyone outside Oracle without prior written consent of Oracle or as specifically provided below. This document is not part of your license agreement nor can it be incorporated into any contractual agreement with Oracle or its subsidiaries or affiliates.

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.

For more information on the terms of this license, or for details on how the MySQL documentation is built and produced, please visit [MySQL Contact & Questions](#).

For additional licensing information, including licenses for third-party libraries used by MySQL products, see [Preface and 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](#).

Chapter 1. MySQL Connector/J

This section explains how to configure and develop Java applications with MySQL Connector/J, the JDBC driver that is integrated with MySQL.

For release notes detailing the changes in each release of Connector/J, see [MySQL Connector/J Release Notes](#).

Chapter 2. Connector/J Versions

Table of Contents

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

There are currently four versions of MySQL Connector/J available:

- Connector/J 5.1 is the Type 4 pure Java JDBC driver, which conforms to the JDBC 3.0 and JDBC 4.0 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 the JDBC-4.0 XML processing, per connection client information, [NCHAR](#), [NVARCHAR](#) and [NCLOB](#) 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

Connector/J version	Driver Type	JDBC version	MySQL Server version	Status
5.1	4	3.0, 4.0	4.1, 5.0, 5.1, 5.5, 5.6, 5.7	Recommended version
5.0	4	3.0	4.1, 5.0	Released version
3.1	4	3.0	4.1, 5.0	Obsolete
3.0	4	3.0	3.x, 4.1	Obsolete

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 2.2. Summary of Java Versions Required by Connector/J

Connector/J version	Java RTE required	JDK required (to build source code)
5.1	1.5.x, 1.6.x, 1.7.x	1.6.x and 1.5.x
5.0	1.3.x, 1.4.x, 1.5.x, 1.6.x	1.4.2, 1.5.x, 1.6.x
3.1	1.2.x, 1.3.x, 1.4.x, 1.5.x, 1.6.x	1.4.2, 1.5.x, 1.6.x
3.0	1.2.x, 1.3.x, 1.4.x, 1.5.x, 1.6.x	1.4.2, 1.5.x, 1.6.x

If you are building Connector/J from source code using the source distribution (see [Section 3.4, “Installing from the Development Source Tree”](#)), you must use JDK 1.4.2 or newer to compile the Connector package. 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.

Chapter 3. Connector/J Installation

Table of Contents

3.1. Installing Connector/J from a Binary Distribution	5
3.2. Installing the Driver and Configuring the <code>CLASSPATH</code>	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 the Development Source Tree	9

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 Mac 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
```

Within Windows 2000, Windows XP, Windows Server 2003 and Windows Vista, 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](#) [Chapter 8, Load Balancing with Connector/J](#), and [Chapter 9, Failover with Connector/J](#). 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.

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

- `user`
- `password`
- `serverName` (see the previous section about fail-over 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.getColumnNames()`. This behavior however is not JDBC compliant, and in Connector/J 5.1 this behavior was 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.getColumnNames()`, use the `useOldAliasMetadataBehavior` option and set the value to `true`.

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

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

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

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

```
useOldUTF8Behavior=true
```

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

```
useServerPrepStmts=false
```

3.3.3. Upgrading from MySQL Connector/J 3.0 to 3.1

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

- **Unicode Character Sets:** See the next section, as well as [Character 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

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.Strings` 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 the Development Source Tree

Caution

Read this section only if you are interested in helping us test our new code. To just get MySQL Connector/J up and running on your system, use a standard binary release distribution.

To install MySQL Connector/J from the development source tree, make sure that you have the following prerequisites:

- A Bazaar client, to check out the sources from our Launchpad repository (available from <http://bazaar-vcs.org/>).
- Apache Ant version 1.7 or newer (available from <http://ant.apache.org/>).
- JDK 1.4.2 or later. Although MySQL Connector/J can be used with older JDKs, compiling it from source requires at least JDK 1.4.2. To build Connector/J 5.1 requires JDK 1.6.x *and* an older JDK such as JDK 1.5.x; point your `JAVA_HOME` environment variable at the older installation.

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

1. Check out the latest code from the branch that you want with one of the following commands.

The source code repository for MySQL Connector/J is located on Launchpad at <https://code.launchpad.net/connectorj>. To check out the latest development branch, use:

```
shell> bzip branch lp:connectorji
```

This creates a `connectorj` subdirectory in the current directory that contains the latest sources for the requested branch.

To check out the latest 5.1 code, use:

```
shell> bzip branch lp:connectorj/5.1
```

This creates a `5.1` subdirectory in the current directory containing the latest 5.1 code.

2. To build Connector/J 5.1, make sure that you have both JDK 1.6.x installed and an older JDK such as JDK 1.5.x. This is because Connector/J supports both JDBC 3.0 (which was prior to JDK 1.6.x) and JDBC 4.0. Set your `JAVA_HOME` environment variable to the path of the older JDK installation.
3. Change your current working directory to either the `connectorj` or `5.1` directory, depending on which branch you intend to build.
4. To build Connector/J 5.1, edit the `build.xml` to reflect the location of your JDK 1.6.x installation. The lines to change are:

```
<property name="com.mysql.jdbc.java6.javac" value="C:\jvms\jdk1.6.0\bin\javac.exe" />
<property name="com.mysql.jdbc.java6.rtjar" value="C:\jvms\jdk1.6.0\jre\lib\rt.jar" />
```

Alternatively, you can set the value of these property names through the Ant `-D` option.

5. Issue the following command to compile the driver and create a `.jar` file suitable for installation:

```
shell> ant dist
```

This creates a `build` directory in the current directory, where all build output will go. A directory is created in the `build` directory that includes the version number of the sources you are building from. This directory contains the sources, compiled `.class` files, and a `.jar` file suitable for deployment. For other possible targets, including ones that will create a fully packaged distribution, issue the following command:

```
shell> ant -projecthelp
```

6. A newly created `.jar` file containing the JDBC driver will be placed in the directory `build/mysql-connector-java-version`.

Install the newly created JDBC driver as you would a binary `.jar` file that you download from MySQL, by following the instructions in [Section 3.2, "Installing the Driver and Configuring the CLASSPATH"](#).

A package containing both the binary and source code for Connector/J 5.1 can also be found at the following location: [Connector/J 5.1 Download](#)

Chapter 4. Connector/J Examples

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

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

Chapter 5. Connector/J (JDBC) Reference

Table of Contents

5.1. Driver/Datasource Class Names, URL Syntax and Configuration Properties for Connector/J	13
5.1.1. Properties Files for the <code>useConfigs</code> Option	32
5.2. JDBC API Implementation Notes	33
5.3. Java, JDBC and MySQL Types	37
5.4. Using Character Sets and Unicode	39
5.5. Connecting Securely Using SSL	40
5.6. Connecting Using PAM Authentication	43
5.7. Using Master/Slave Replication with <code>ReplicationConnection</code>	43
5.8. Mapping MySQL Error Numbers to JDBC <code>SQLState</code> Codes	44

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 otherwise configuring software to use MySQL Connector/J.

JDBC URL Format

The JDBC URL format for MySQL Connector/J is as follows, with items in square brackets ([,]) being optional:

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

If the host name is not specified, it defaults to `127.0.0.1`. If the port is not specified, it defaults to `3306`, the default port number for MySQL servers.

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

Here is a sample connection URL:

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

IPv6 Connections

For IPv6 connections, use this alternative syntax to specify hosts in the URL, `address=(key=value)`. Supported keys are:

- `(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)(user=test)/db
```

Any other parameters are treated as host-specific properties that follow the conventions of the JDBC URL properties. This now allows per-host overrides of any configuration property for multi-host connections (that is, when using failover, load balancing, or replication). Limit the overrides to user, password, network timeouts and statement and metadata cache sizes; the results of other per-host overrides are not defined.

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.

Failover Support

MySQL Connector/J has failover support. This enables the driver to fail over to any number of slave hosts and still perform read-only queries. Failover only happens when the connection is in an `autoCommit(true)` state, because failover cannot happen reliably when a `transaction` is in progress. Most application servers and connection pools set `autoCommit` to `true` at the end of every transaction/connection use.

The failover functionality has the following behavior:

- If the URL property `autoReconnect` is `false`: Failover only happens at connection initialization, and fallback occurs when the driver determines that the first host has become available again.
- If the URL property `autoReconnect` is `true`: Failover happens when the driver determines that the connection has failed (checked before every query), and falls back to the first host when it determines that the host has become available again (after `queriesBeforeRetryMaster` queries have been issued).

In either case, whenever you are connected to a “failed-over” server, the connection is set to read-only state, so queries that attempt to modify data will throw exceptions (the query will **never** be processed by the MySQL server).

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.

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.

Property Name	Definition	Default Value	Since Version
user	The user to connect as		all versions
password	The password to use when connecting		all versions
socketFactory	The name of the class that the driver should use for creating socket connections to the server. This class must implement the interface 'com.mysql.jdbc.SocketFactory' and have public no-args constructor.	com.mysql.jdbc.StandardSocketFactory	3.0.1
connectTimeout	Timeout for socket connect (in milliseconds), with 0 being no timeout. Only works on JDK-1.4 or newer. Defaults to '0'.	0	3.0.1
socketTimeout	Timeout on network socket operations (0, the default means no timeout).	0	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.		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.		3.1.5
authenticationPlugins	Comma-delimited list of classes that implement com.mysql.jdbc.AuthenticationPlugin and which		5.1.19

	will be used for authentication unless disabled by "disabledAuthenticationPlugins" property.			
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).	com.mysql.jdbc.authentication.M...	5.1.19	
disabledAuthenticationPlugins	Comma-delimited list of classes implementing com.mysql.jdbc.AuthenticationPlugin or mechanisms, i.e. "mysql_native_password". The authentication plugins or mechanisms listed will not be used for authentication which will fail if it requires one of them. It is an error to disable the default authentication plugin (either the one named by "defaultAuthenticationPlugin" property or the hard-coded one if "defaultAuthenticationPlugin" property is not set).		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.	true	5.1.23	
interactiveClient	Set the CLIENT_INTERACTIVE flag, which tells MySQL to timeout connections based on INTERACTIVE_TIMEOUT instead of WAIT_TIMEOUT	false	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.		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		3.1.4	
useCompression	Use zlib compression when communicating with the server (true/false)? Defaults to 'false'.	false	3.0.17	

Networking.

Property Name	Definition	Default Value	Since Version
maxAllowedPacket	Maximum allowed packet size to send to server. If not set, the value of system variable	-1	5.1.8

	'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'.		
tcpKeepAlive	If connecting using TCP/IP, should the driver set SO_KEEPALIVE?	true	5.0.7
tcpNoDelay	If connecting using TCP/IP, should the driver set SO_TCP_NODELAY (disabling the Nagle Algorithm)?	true	5.0.7
tcpRcvBuf	If connecting using TCP/IP, should the driver set SO_RCV_BUF to the given value? The default value of '0', means use the platform default value for this property)	0	5.0.7
tcpSndBuf	If connecting using TCP/IP, should the driver set SO_SND_BUF to the given value? The default value of '0', means use the platform default value for this property)	0	5.0.7
tcpTrafficClass	If connecting using TCP/IP, should the driver set traffic class or type-of-service fields ?See the documentation for java.net.Socket.setTrafficClass() for more information.	0	5.0.7

High Availability and Clustering.

Property Name	Definition	Default Value	Since Version
autoReconnect	Should the driver try to re-establish stale and/or dead connections? If enabled the driver will throw an exception for a queries issued on a stale or dead connection, which belong to the current transaction, but will attempt reconnect before the next query issued on the connection in a new transaction. The use of this feature is not recommended, because it has side effects related to session state and data consistency when applications don't handle SQLExceptions properly, and is only designed to be used when you are unable to configure your application to handle SQLExceptions resulting from dead and stale connections properly. Alternatively, as a last option, investigate setting the MySQL server variable "wait_timeout" to a high value, rather than the default of 8 hours.	false	1.1
autoReconnectForPools	Use a reconnection strategy appropriate for connection pools (defaults to 'false')	false	3.1.3
failOverReadOnly	When failing over in autoReconnect mode, should the connection be set to 'read-only'?	true	3.0.12
maxReconnects	Maximum number of reconnects to attempt if autoReconnect is true, default is '3'.	3	1.1
reconnectAtTxEnd	If autoReconnect is set to true, should the driver attempt reconnections at the end of every transaction?	false	3.0.10

retriesAllDown	When using loadbalancing, 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.	120	5.1.6
initialTimeout	If autoReconnect is enabled, the initial time to wait between re-connect attempts (in seconds, defaults to '2').	2	1.1
roundRobinLoadBalance	When autoReconnect is enabled, and failoverReadonly is false, should we pick hosts to connect to on a round-robin basis?	false	3.1.2
queriesBeforeRetryMaster	Number of queries to issue before falling back to master 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 master. Defaults to 50.	50	3.0.2
secondsBeforeRetryMaster	How long should the driver wait, when failed over, before attempting	30	3.0.2
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.	0	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.	0	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		5.0.1

Security.

Property Name	Definition	Default Value	Since Version
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.	false	3.1.1
useSSL	Use SSL when communicating with the server (true/false), defaults to 'false'	false	3.0.2
requireSSL	Require SSL connection if useSSL=true? (defaults to 'false').	false	3.1.0
verifyServerCertificate	If "useSSL" is set to "true", should the driver verify the server's certificate? When using this feature, the keystore parameters should be specified by the	true	5.1.6

	"clientCertificateKeyStore*" properties, rather than system properties.		
clientCertificateKeyStoreUrl	URL to the client certificate KeyStore (if not specified, use defaults)		5.1.0
clientCertificateKeyStoreType	KeyStore type for client certificates (NULL or empty means use the default, which is "JKS". Standard keystore types supported by the JVM are "JKS" and "PKCS12", your environment may have more available depending on what security products are installed and available to the JVM.	JKS	5.1.0
clientCertificateKeyStorePassword	Password for the client certificates KeyStore		5.1.0
trustCertificateKeyStoreUrl	URL to the trusted root certificate KeyStore (if not specified, use defaults)		5.1.0
trustCertificateKeyStoreType	KeyStore type for trusted root certificates (NULL or empty means use the default, which is "JKS". Standard keystore types supported by the JVM are "JKS" and "PKCS12", your environment may have more available depending on what security products are installed and available to the JVM.	JKS	5.1.0
trustCertificateKeyStorePassword	Password for the trusted root certificates KeyStore		5.1.0
allowLoadLocalInfile	Should the driver allow use of 'LOAD DATA LOCAL INFILE...' (defaults to 'true').	true	3.0.3
allowUrlInLocalInfile	Should the driver allow URLs in 'LOAD DATA LOCAL INFILE' statements?	false	3.1.4
paranoid	Take measures to prevent exposure sensitive information in error messages and clear data structures holding sensitive data when possible? (defaults to 'false')	false	3.0.1
passwordCharacterEncoding	What character encoding is used for passwords? Leaving this set to the default value (null), uses the platform character set, which works for ISO8859_1 (i.e. "latin1") passwords. For passwords in other character encodings, the encoding will have to be specified with this property, as it's not possible for the driver to auto-detect this.		5.1.7

Performance Extensions.

Property Name	Definition	Default Value	Since Version
callableStmtCacheSize	If 'cacheCallableStmts' is enabled, how many callable statements should be cached?	100	3.1.2
metadataCacheSize	The number of queries to cache ResultSetMetadata for if cacheResultSetMetadata is set to 'true' (default 50)	50	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	false	3.1.7

	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?			
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?	false	5.1.7	
prepStmtCacheSize	If prepared statement caching is enabled, how many prepared statements should be cached?	25	3.0.10	
prepStmtCacheSqlLimit	If prepared statement caching is enabled, what's the largest SQL the driver will cache the parsing for?	256	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.	com.mysql.jdbc.PerConnectionLF	5.0.4	
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	com.mysql.jdbc.PerVmServerCon	5.0.4	
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.	true	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.	true	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?	false	5.0.0	
blobSendChunkSize	Chunk to use when sending BLOB/CLOBs via ServerPreparedStatement	1048576	3.1.9	
cacheCallableStmts	Should the driver cache the parsing stage of CallableStatements	false	3.1.2	
cachePrepStmts	Should the driver cache the parsing stage of PreparedStatement of client-side prepared statements, the "check" for suitability of server-	false	3.0.10	

	side prepared and server-side prepared statements themselves?		
cacheResultSetMetadata	Should the driver cache ResultSetMetaData for Statements and PreparedStatements? (Req. JDK-1.4+, true/false, default 'false')	false	3.1.1
cacheServerConfiguration	Should the driver cache the results of 'SHOW VARIABLES' and 'SHOW COLLATION' on a per-URL basis?	false	3.1.5
defaultFetchSize	The driver will call setFetchSize(n) with this value on all newly-created Statements	0	3.1.9
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.	false	3.1.7
dynamicCalendars	Should the driver retrieve the default calendar when required, or cache it per connection/session?	false	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)?	false	3.1.3
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.	true	5.0.6
holdResultsOpenOverStatementClose	Should the driver close result sets on Statement.close() as required by the JDBC specification?	false	3.1.7
largeRowSizeThreshold	What size result set row should the JDBC driver consider "large", and thus use a more memory-efficient way of representing the row internally?	2048	5.1.1
loadBalanceStrategy	If using a load-balanced connection to connect to SQL nodes in a MySQL Cluster/ NDB configuration (by using the URL prefix "jdbc:mysql:loadbalance://"), which load balancing algorithm should the driver use: (1) "random" - the driver will pick a random host for each request. This tends to work better than round-robin, as the randomness will somewhat account for spreading loads where requests vary in response time, while round-robin can sometimes lead to overloaded nodes if there are variations in response times	random	5.0.6

	across the workload. (2) "bestResponseTime" - the driver will route the request to the host that had the best response time for the previous transaction.		
locatorFetchBufferSize	If 'emulateLocators' is configured to 'true', what size buffer should be used when fetching BLOB data for getBinaryInputStream?	1048576	3.2.1
rewriteBatchedStatements	Should the driver use multiqueries (irregardless of the setting of "allowMultiQueries") as well as rewriting of prepared statements for INSERT into multi-value inserts when executeBatch() is called? Notice that this has the potential for SQL injection if using plain java.sql.Statements and your code doesn't sanitize input correctly. Notice that for prepared statements, 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.	false	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.	true	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?	true	5.0.6
useFastDateParsing	Use internal String->Date/Time/Timestamp conversion routines to avoid excessive object creation?	true	5.0.5
useFastIntParsing	Use internal String->Integer conversion routines to avoid excessive object creation?	true	3.1.4
useJvmCharsetConverters	Always use the character encoding routines built into the JVM, rather than using lookup tables for single-byte character sets?	false	5.0.1
useReadAheadInput	Use newer, optimized non-blocking, buffered input stream when reading from the server?	true	3.1.5

Debugging/Profiling.

Property Name	Definition	Default Value	Since Version
logger	The name of a class that implements "com.mysql.jdbc.log.Log" that will be	com.mysql.jdbc.log.StandardLog	3.1.1

	used to log messages to. (default is "com.mysql.jdbc.log.StandardLogger", which logs to STDERR)		
gatherPerfMetrics	Should the driver gather performance metrics, and report them via the configured logger every 'reportMetricsIntervalMillis' milliseconds?	false	3.1.2
profileSQL	Trace queries and their execution/fetch times to the configured logger (true/false) defaults to 'false'	false	3.1.0
profileSql	Deprecated, use 'profileSQL' instead. Trace queries and their execution/fetch times on STDERR (true/false) defaults to 'false'		2.0.14
reportMetricsIntervalMillis	If 'gatherPerfMetrics' is enabled, how often should they be logged (in ms)?	30000	3.1.2
maxQuerySizeToLog	Controls the maximum length/size of a query that will get logged when profiling or tracing	2048	3.1.3
packetDebugBufferSize	The maximum number of packets to retain when 'enablePacketDebug' is true	20	3.1.3
slowQueryThresholdMillis	If 'logSlowQueries' is enabled, how long should a query (in ms) before it is logged as 'slow'?	2000	3.1.2
slowQueryThresholdNanos	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.	0	5.0.7
useUsageAdvisor	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')?	false	3.1.1
autoGenerateTestcaseScript	Should the driver dump the SQL it is executing, including server-side prepared statements to STDERR?	false	3.1.9
autoSlowLog	Instead of using slowQueryThreshold* to determine if a query is slow enough to be logged, maintain statistics that allow the driver to determine queries that are outside the 99th percentile?	true	5.1.4
clientInfoProvider	The name of a class that implements the com.mysql.jdbc.JDBC4ClientInfoProvider interface in order to support JDBC-4.0's Connection.get/setClientInfo() methods	com.mysql.jdbc.JDBC4ClientInfoProvider	5.1.6
dumpMetadataOnColumnNotFoundException	Should the driver dump the field-level metadata of a result set into the exception message when ResultSet.findColumn() fails?	false	3.1.13
dumpQueriesOnException	Should the driver dump the contents of the query sent to the server in the message for SQLExceptions?	false	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	false	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 WARN level?	false	3.1.2
includeInnodbStatusInDeadlockExceptions	Include the output of "SHOW ENGINE INNODB STATUS" in exception messages when deadlock exceptions are detected?	false	5.0.7
includeThreadDumpInDeadlockExceptions	Include a current Java thread dump in exception messages when deadlock exceptions are detected?	false	5.1.15
includeThreadNamesAsStatementComments	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".	false	5.1.15
logSlowQueries	Should queries that take longer than 'slowQueryThresholdMillis' be logged?	false	3.1.2
logXaCommands	Should the driver log XA commands sent by MysqlXaConnection to the server, at the DEBUG level of logging?	false	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.	com.mysql.jdbc.profiler.LoggingP	5.0.6
resultSetSizeThreshold	If the usage advisor is enabled, how many rows should a result set contain before the driver warns that it is suspiciously large?	100	5.0.5
traceProtocol	Should trace-level network protocol be logged?	false	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)?	false	5.0.7

Miscellaneous.

Property Name	Definition	Default Value	Since Version
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'	true	1.1g
characterEncoding	If 'useUnicode' is set to true, what character encoding should the driver use when dealing with strings? (defaults is to 'autodetect')		1.1g
characterSetResults	Character set to tell the server to return results as.		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		5.1.25

	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 will be ignored. Setting connectionAttributes=none will cause connection attribute processing to be bypassed, for situations where Connection creation/initialization speed is critical.		
connectionCollation	If set, tells the server to use this collation via 'set collation_connection'		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.	false	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.		5.1.3
utf8OutsideBmpIncludedColumnNamePattern	Used to specify exclusion rules to "utf8OutsideBmpExcludedColumnNamePattern". The regex must follow the patterns used for the java.util.regex package.		5.1.3
loadBalanceEnableJMX	Enables JMX-based management of load-balanced connection groups, including live addition/removal of hosts from load-balancing pool.	false	5.1.13
sessionVariables	A comma-separated list of name/value pairs to be sent as SET SESSION ... to the server when the driver connects.		3.1.8
useColumnNamesInFindColumn	Prior to JDBC-4.0, the JDBC specification had a bug related to what could be given as a "column name" to ResultSet methods like findColumn(), or getters that took a String property. JDBC-4.0 clarified "column name" to mean the label, as given in an "AS" clause and returned by ResultSetMetaData.getColumnLabel(), and if no AS clause, the column name. Setting this property to "true" will give behavior that is congruent to JDBC-3.0 and earlier versions of the JDBC specification, but which because of the specification bug could give unexpected results. This property is preferred over "useOldAliasMetadataBehavior" unless you need the specific behavior that it provides with respect to ResultSetMetadata.	false	5.1.7

allowNaNAndInf	Should the driver allow NaN or +/- INF values in PreparedStatement.setDouble()?	false	3.1.5
autoClosePstmtStreams	Should the driver automatically call .close() on streams/readers passed as arguments via set*() methods?	false	3.1.12
autoDeserialize	Should the driver automatically detect and de-serialize objects stored in BLOB fields?	false	3.1.5
blobsAreStrings	Should the driver always treat BLOBs as Strings - specifically to work around dubious metadata returned by the server for GROUP BY clauses?	false	5.0.8
capitalizeTypeNames	Capitalize type names in DatabaseMetaData? (usually only useful when using WebObjects, true/false, defaults to 'false')	true	2.0.7
clobCharacterEncoding	The character encoding to use for sending and retrieving TEXT, MEDIUMTEXT and LONGTEXT values instead of the configured connection characterEncoding		5.0.0
clobberStreamingResults	This will cause a 'streaming' ResultSet to be automatically closed, and any outstanding data still streaming from the server to be discarded if another query is executed before all the data has been read from the server.	false	3.0.9
compensateOnDuplicateKeyUpdate	Should the driver compensate for the update counts of "ON DUPLICATE KEY" INSERT statements (2 = 1, 0 = 1) when using prepared statements?	false	5.1.7
continueBatchOnError	Should the driver continue processing batch commands if one statement fails. The JDBC spec allows either way (defaults to 'true').	true	3.0.3
createDatabaseIfNotExist	Creates the database given in the URL if it doesn't yet exist. Assumes the configured user has permissions to create databases.	false	3.1.9
emptyStringsConvertToZero	Should the driver allow conversions from empty string fields to numeric values of '0'?	true	3.1.8
emulateLocators	Should the driver emulate java.sql.Blobs with locators? With this feature enabled, the driver will delay loading the actual Blob data until the one of the retrieval methods (getInputStream(), getBytes(), and so forth) on the blob data stream has been accessed. For this to work, you must use a column alias with the value of the column to the actual name of the Blob. The feature also has the following restrictions: The SELECT that created the result set must reference only one table, the table must have a primary key; the SELECT must alias the original blob column name, specified as a string, to an alternate name; the SELECT must cover all columns that make up the primary key.	false	3.1.0

emulateUnsupportedPstmts	Should the driver detect prepared statements that are not supported by the server, and replace them with client-side emulated versions?	true	3.1.7
exceptionInterceptors	Comma-delimited list of classes that implement com.mysql.jdbc.ExceptionInterceptor. These classes will be instantiated one per Connection instance, and all SQLExceptions thrown by the driver will be allowed to be intercepted by these interceptors, in a chained fashion, with the first class listed as the head of the chain.		5.1.8
functionsNeverReturnBlobs	Should the driver always treat data from functions returning BLOBs as Strings - specifically to work around dubious metadata returned by the server for GROUP BY clauses?	false	5.0.8
generateSimpleParameterMetadata	Should the driver generate simplified parameter metadata for PreparedStatements when no metadata is available either because the server couldn't support preparing the statement, or server-side prepared statements are disabled?	false	5.0.5
getProceduresReturnsFunctions	Pre-JDBC4 DatabaseMetaData API has only getProcedures() and getProcedureColumns() methods, so they return metadata info for both stored procedures and functions. JDBC4 was extended with getFunctions() and getFunctionColumns() methods and the expected behaviour of previous methods is not well defined. For JDBC4 and higher, default 'true' value means that calls of DatabaseMetaData.getProcedures() and DatabaseMetaData.getProcedureColumns() return metadata for both procedures and function, as before, keeping back compatibility. Setting this property to false allows to decouple from pre-JDBC4 behaviour of DatabaseMetaData.getProcedures() and DatabaseMetaData.getProcedureColumns(), forcing these to return metadata for procedures only.	true	5.1.26
ignoreNonTxTables	Ignore non-transactional table warning for rollback? (defaults to 'false').	false	3.0.9
jdbcCompliantTruncation	Should the driver throw java.sql.DataTruncation exceptions when data is truncated as is required by the JDBC specification when connected to a server that supports warnings (MySQL 4.1.0 and newer)? This property has no effect if the server sql-mode includes STRICT_TRANS_TABLES.	true	3.1.2
loadBalanceAutoCommitStatements	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.		5.1.15

Setting Configuration Properties

loadBalanceAutoCommitStatementThreshold	When auto-commit is enabled, the number of statements which should be executed before triggering load-balancing to rebalance. Default value of 0 causes load-balanced connections to only rebalance when exceptions are encountered, or auto-commit is disabled and transactions are explicitly committed or rolled back.	0	5.1.15	
loadBalanceBlacklistTimeout	Time in milliseconds between checks of servers which are unavailable, by controlling how long a server lives in the global blacklist.	0	5.1.0	
loadBalanceConnectionGroup	Logical group of load-balanced connections within a classloader, used to manage different groups independently. If not specified, live management of load-balanced connections is disabled.		5.1.13	
loadBalanceExceptionChecker	Fully-qualified class name of custom exception checker. The class must implement com.mysql.jdbc.LoadBalanceExceptionChecker interface, and is used to inspect SQLExceptions and determine whether they should trigger fail-over to another host in a load-balanced deployment.	com.mysql.jdbc.StandardLoadBalancer	5.0.13	
loadBalancePingTimeout	Time in milliseconds to wait for ping response from each of load-balanced physical connections when using load-balanced Connection.	0	5.1.13	
loadBalanceSQLExceptionSubclassFailover	Comma-delimited list of classes/interfaces used by default load-balanced exception checker to determine whether a given SQLException should trigger failover. The comparison is done using Class.getInstance(SQLException) using the thrown SQLException.		5.1.13	
loadBalanceSQLStateFailover	Comma-delimited list of SQLState codes used by default load-balanced exception checker to determine whether a given SQLException should trigger failover. The SQLState of a given SQLException is evaluated to determine whether it begins with any value in the comma-delimited list.		5.1.13	
loadBalanceValidateConnectionOnSwap	Should a load-balanced Connection explicitly check whether the connection is live when swapping to a new physical connection at commit/rollback?	false	5.1.13	
maxRows	The maximum number of rows to return (0, the default means return all rows).	-1		all versions
netTimeoutForStreamingResults	What value should the driver automatically set the server setting 'net_write_timeout' to when the streaming result sets feature is in use? (value has unit of seconds, the value '0' means the driver will not try and adjust this value)	600	5.1.0	
noAccessToProcedureBodies	When determining procedure parameter types for CallableStatements, and the connected user can't access procedure bodies through "SHOW CREATE PROCEDURE" or select on mysql.proc should the driver instead create basic metadata	false	5.0.3	

	(all parameters reported as IN VARCHARs, but allowing registerOutParameter() to be called on them anyway) instead of throwing an exception?		
noDatetimeStringSync	Don't ensure that <code>ResultSet.getDatetimeType().toString().equals(ResultSet.getString())</code>	false	3.1.7
noTimezoneConversionForTimeType	Do not convert TIME values using the server timezone if 'useTimezone'='true'	false	5.0.0
nullCatalogMeansCurrent	When DatabaseMetadataMethods ask for a 'catalog' parameter, does the value null mean use the current catalog? (this is not JDBC-compliant, but follows legacy behavior from earlier versions of the driver)	true	3.1.8
nullNamePatternMatchesAll	Should DatabaseMetaData methods that accept *pattern parameters treat null the same as '%' (this is not JDBC-compliant, however older versions of the driver accepted this departure from the specification)	true	3.1.8
overrideSupportsIntegrityEnhancementFacility	Should the driver return "true" for <code>DatabaseMetaData.supportsIntegrityEnhancementFacility()</code> even if the database doesn't support it to workaround applications that require this method to return "true" to signal support of foreign keys, even though the SQL specification states that this facility contains much more than just foreign key support (one such application being OpenOffice)?	false	3.1.12
padCharsWithSpace	If a result set column has the CHAR type and the value does not fill the amount of characters specified in the DDL for the column, should the driver pad the remaining characters with space (for ANSI compliance)?	false	5.0.6
pedantic	Follow the JDBC spec to the letter.	false	3.0.0
pinGlobalTxToPhysicalConnection	When using XAConnections, should the driver ensure that operations on a given XID are always routed to the same physical connection? This allows the XAConnection to support "XA START ... JOIN" after "XA END" has been called	false	5.0.1
populateInsertRowWithDefaultValues	When using ResultSets that are CONCUR_UPDATABLE, should the driver pre-populate the "insert" row with default values from the DDL for the table used in the query so those values are immediately available for ResultSet accessors? This functionality requires a call to the database for metadata each time a result set of this type is created. If disabled (the default), the default values will be populated by the an internal call to <code>refreshRow()</code> which pulls back default values and/or values changed by triggers.	false	5.0.5
processEscapeCodesForPrepStatements	Should the driver process escape codes in queries that are prepared?	true	3.1.12

queryTimeoutKillsConnection	If the timeout given in Statement.setQueryTimeout() expires, should the driver forcibly abort the Connection instead of attempting to abort the query?	false	5.1.9
relaxAutoCommit	If the version of MySQL the driver connects to does not support transactions, still allow calls to commit(), rollback() and setAutoCommit() (true/false, defaults to 'false')?	false	2.0.13
retainStatementAfterResultSetClose	Should the driver retain the Statement reference in a ResultSet after ResultSet.close() has been called. This is not JDBC-compliant after JDBC-4.0.	false	3.1.11
rollbackOnPooledClose	Should the driver issue a rollback() when the logical connection in a pool is closed?	true	3.0.15
runningCTS13	Enables workarounds for bugs in Sun's JDBC compliance testsuite version 1.3	false	3.1.7
serverTimezone	Override detection/mapping of timezone. Used when timezone from server doesn't map to Java timezone		3.0.2
statementInterceptors	A comma-delimited list of classes that implement "com.mysql.jdbc.StatementInterceptor" that should be placed "in between" query execution to influence the results. StatementInterceptors are "chainable", the results returned by the "current" interceptor will be passed on to the next in the chain, from left-to-right order, as specified in this property.		5.1.1
strictFloatingPoint	Used only in older versions of compliance test	false	3.0.0
strictUpdates	Should the driver do strict checking (all primary keys selected) of updatable result sets (true, false, defaults to 'true')?	true	3.0.4
tinyInt1isBit	Should the driver treat the datatype TINYINT(1) as the BIT type (because the server silently converts BIT -> TINYINT(1) when creating tables)?	true	3.0.16
transformedBitsBoolean	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?	false	3.1.9
treatUtilDateAsTimestamp	Should the driver treat java.util.Date as a TIMESTAMP for the purposes of PreparedStatement.setObject()?	true	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')	false	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 ...	false	5.1.7

	ON DUPLICATE KEY UPDATE" statements to be returned by the server.		
useGmtMillisForDatetimes	Convert between session timezone and GMT before creating Date and Timestamp instances (value of "false" is legacy behavior, "true" leads to more JDBC-compliant behavior.	false	3.1.12
useHostsInPrivileges	Add '@hostname' to users in DatabaseMetaData.getColumn/TablePrivileges() (true/false), defaults to 'true'.	true	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?	false	5.0.0
useJDBCCompliantTimezoneShift	Should the driver use JDBC-compliant rules when converting TIME/TIMESTAMP/DATETIME values' timezone information for those JDBC arguments which take a java.util.Calendar argument? (Notice that this option is exclusive of the "useTimezone=true" configuration option.)	false	5.0.0
useLegacyDatetimeCode	Use code for DATE/TIME/DATETIME/TIMESTAMP handling in result sets and statements that consistently handles timezone 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?	true	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.getColumnNames() or ResultSetMetaData.getTableNames() rather than the original column/table name? In 5.0.x, the default value was true.	false	5.0.4
useOldUTF8Behavior	Use the UTF-8 behavior the driver did when communicating with 4.0 and older servers	false	3.1.6
useOnlyServerErrorMessages	Don't prepend 'standard' SQLState error messages to error messages returned by the server.	true	3.0.15
useSSPSCompatibleTimezonesShift	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.	false	5.0.5
useServerPrepStmts	Use server-side prepared statements if the server supports them?	false	3.1.0
useSqlStateCodes	Use SQL Standard state codes instead of 'legacy' X/Open/SQL state codes (true/false), default is 'true'	true	3.1.3
useStreamLengthsInPrepStmts	Honor stream length parameter in PreparedStatement/ResultSet.setXXXStream() method calls (true/false, defaults to 'true')?	true	3.0.2

<code>useTimezone</code>	Convert time/date types between client and server timezones (true/false, defaults to 'false')?	false	3.0.2
<code>useUnbufferedInput</code>	Don't use <code>BufferedInputStream</code> for reading data from the server	true	3.0.11
<code>yearIsDateType</code>	Should the JDBC driver treat the MySQL type "YEAR" as a <code>java.sql.Date</code> , or as a <code>SHORT</code> ?	true	3.1.9
<code>zeroDateTimeBehavior</code>	What should happen when the driver encounters <code>DATETIME</code> values that are composed entirely of zeros (used by MySQL to represent invalid dates)? Valid values are "exception", "round" and "convertToNull".	exception	3.1.4

Connector/J also supports access to MySQL using named pipes on Windows NT, Windows 2000, or Windows XP using the `NamedPipeSocketFactory` as a plugin-socket factory using the `socketFactory` property. 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, use:

```
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:

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

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

3-0-Compat

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

5-0-Compat

```
useDirectRowUnpack=false
```

clusterBase

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

coldFusion

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

fullDebug

```
profileSQL=true
gatherPerMetrics=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 passes all of the tests in the publicly available version of Sun's JDBC compliance test suite. This section gives details on a interface-by-interface level about implementation decisions that might affect how you code applications with MySQL Connector/J. The JDBC specification is vague about how certain functionality should be implemented, or the specification enables leeway in implementation.

- **BLOB**

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

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

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

You must also follow these rules:

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

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

- **CallableStatement**

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

- **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**

PreparedStatement 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:

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

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

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

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

If the statement is within scope of a transaction, then locks are released when the transaction completes (which implies that the statement needs to complete first). As with most other databases, statements

are not complete until all the results pending on the statement are read or the active result set for the statement is closed.

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

- **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

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

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 `java.sql.Types` classes see [Java 2 Platform Types](#).

Table 5.2. MySQL Types to Java Types for ResultSet.getObject()

MySQL Type Name	Return value of <code>GetColumnName</code>	Returned as Java Class
<code>BIT(1)</code> (new in MySQL-5.0)	BIT	<code>java.lang.Boolean</code>
<code>BIT(> 1)</code> (new in MySQL-5.0)	BIT	<code>byte[]</code>
TINYINT	TINYINT	<code>java.lang.Boolean</code> if the configuration property <code>tinyIntIsBit</code> is set to <code>true</code> (the default) and the storage size is 1, or <code>java.lang.Integer</code> if not.
BOOL, BOOLEAN	TINYINT	See TINYINT, above as these are aliases for TINYINT(1), currently.
SMALLINT[(M)] [UNSIGNED]	SMALLINT [UNSIGNED]	<code>java.lang.Integer</code> (regardless if UNSIGNED or not)
MEDIUMINT[(M)] [UNSIGNED]	MEDIUMINT [UNSIGNED]	<code>java.lang.Integer</code> , if UNSIGNED <code>java.lang.Long</code> (C/J 3.1 and earlier), or <code>java.lang.Integer</code> for C/J 5.0 and later
INT, INTEGER[(M)] [UNSIGNED]	INTEGER [UNSIGNED]	<code>java.lang.Integer</code> , if UNSIGNED <code>java.lang.Long</code>
BIGINT[(M)] [UNSIGNED]	BIGINT [UNSIGNED]	<code>java.lang.Long</code> , if UNSIGNED <code>java.math.BigInteger</code>
FLOAT[(M,D)]	FLOAT	<code>java.lang.Float</code>
DOUBLE[(M,B)]	DOUBLE	<code>java.lang.Double</code>
DECIMAL[(M[,D])]	DECIMAL	<code>java.math.BigDecimal</code>
DATE	DATE	<code>java.sql.Date</code>
DATETIME	DATETIME	<code>java.sql.Timestamp</code>
TIMESTAMP[(M)]	TIMESTAMP	<code>java.sql.Timestamp</code>
TIME	TIME	<code>java.sql.Time</code>
YEAR[(2 4)]	YEAR	If <code>yearIsDateType</code> configuration property is set to <code>false</code> , then the returned object type is <code>java.sql.Short</code> . If set to <code>true</code> (the default), then the returned object is of type <code>java.sql.Date</code> with the date set to January 1st, at midnight.
CHAR(M)	CHAR	<code>java.lang.String</code> (unless the character set for the column is BINARY, then <code>byte[]</code> is returned.
VARCHAR(M) [BINARY]	VARCHAR	<code>java.lang.String</code> (unless the character set for the column is BINARY, then <code>byte[]</code> is returned.
BINARY(M)	BINARY	<code>byte[]</code>
VARBINARY(M)	VARBINARY	<code>byte[]</code>
TINYBLOB	TINYBLOB	<code>byte[]</code>
TINYTEXT	VARCHAR	<code>java.lang.String</code>
BLOB	BLOB	<code>byte[]</code>
TEXT	VARCHAR	<code>java.lang.String</code>

MySQL Type Name	Return value of <code>getColumnClassName</code>	Returned as Java Class
MEDIUMBLOB	MEDIUMBLOB	<code>byte[]</code>
MEDIUMTEXT	VARCHAR	<code>java.lang.String</code>
LOB	LOB	<code>byte[]</code>
LONGTEXT	VARCHAR	<code>java.lang.String</code>
ENUM('value1','value2')	CHAR..)	<code>java.lang.String</code>
SET('value1','value2')	CHAR..)	<code>java.lang.String</code>

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

Number of Encodings Per Connection

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 `ResultSet`.

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

MySQL Character Set Name	Java-Style Character Encoding Name
<code>ascii</code>	<code>US-ASCII</code>
<code>big5</code>	<code>Big5</code>

MySQL Character Set Name	Java-Style Character Encoding Name
gbk	GBK
sjis	SJIS (or Cp932 or MS932 for MySQL Server < 4.1.11)
cp932	Cp932 or MS932 (MySQL Server > 4.1.11)
gb2312	EUC_CN
ujis	EUC_JP
euackr	EUC_KR
latin1	Cp1252
latin2	ISO8859_2
greek	ISO8859_7
hebrew	ISO8859_8
cp866	Cp866
tis620	TIS620
cp1250	Cp1250
cp1251	Cp1251
cp1257	Cp1257
macroman	MacRoman
macce	MacCentralEurope
utf8	UTF-8
ucs2	UnicodeBig

Warning

Do not issue the query `set names` with Connector/J, as the driver will not detect that the character set has changed, and will continue to use the character set detected during the initial connection setup.

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://developer.java.sun.com/developer/bugParade/bugs/4273544.html>
- 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 [Configuring MySQL for SSL](#).
- 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=MySQL AB, L=Orenburg, ST=Some-State, C=RU  
Issuer: EMAILADDRESS=walrus@example.com, CN=Walrus,  
O=MySQL AB, L=Orenburg, ST=Some-State, C=RU  
Serial number: 0  
Valid from:  
Fri Aug 02 16:55:53 CDT 2002 until: Sat Aug 02 16:55:53 CDT 2003  
Certificate fingerprints:  
MD5: 61:91:A0:F2:03:07:61:7A:81:38:66:DA:19:C4:8D:AB  
SHA1: 25:77:41:05:D5:AD:99:8C:14:8C:CA:68:9C:2F:B8:89:C3:34:4D:6C  
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:

```
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]: MySQL AB
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=MySQL AB,
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   ...7..=.....J.
0010: 36 F4 00 A8 37 67 D7 40 10 8A E1 BE 84 99 02 D9   6...7g.@.....
0020: DB EF CA 13 79 4E 00 00 10 00 05 00 04 00 09 00   ....yN.....
0030: 0A 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
*** ServerHello, v3.1
RandomCookie:  GMT: 1018577560 bytes = { 116, 50, 4, 103, 25, 100, 58, »
    202, 79, 185, 178, 100, 215, 66, 254, 21, 83, 187, 190, 42, 170, 3, »

```



```

    132, 110, 82, 148, 160, 92 }
Session ID:  {163, 227, 84, 53, 81, 127, 252, 254, 178, 179, 68, 63, »
    182, 158, 30, 11, 150, 79, 170, 76, 255, 92, 15, 226, 24, 17, 177, »
    219, 158, 177, 187, 143}
Cipher Suite:  { 0, 5 }
Compression Method: 0
***
%% Created:  [Session-1, SSL_RSA_WITH_RC4_128_SHA]
** SSL_RSA_WITH_RC4_128_SHA
[read] MD5 and SHA1 hashes:  len = 74
0000: 02 00 00 46 03 01 3D B6 43 98 74 32 04 67 19 64  ...F..=.C.t2.g.d
0010: 3A CA 4F B9 B2 64 D7 42 FE 15 53 BB BE 2A AA 03  :.O..d.B..S..*..
0020: 84 6E 52 94 A0 5C 20 A3 E3 54 35 51 7F FC FE B2  .nR..\ ..T5Q....
0030: B3 44 3F B6 9E 1E 0B 96 4F AA 4C FF 5C 0F E2 18  .D?.....O.L.\...
0040: 11 B1 DB 9E B1 BB 8F 00 05 00  .....
main, READ:  SSL v3.1 Handshake, length = 1712
...

```

JSSE provides debugging (to `stdout`) when you set the following system property: -
[Djavax.net.debug=all](#) This will tell you what keystores and truststores are being used, as well as what is going on during the SSL handshake and certificate exchange. It will be helpful when trying to determine what is not working when trying to get an SSL connection to happen.

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

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)`, this replication-aware connection will use one of the slave connections, which are load-balanced per-vm using a round-robin scheme (a given connection is sticky to a slave unless that slave is removed from service). 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:

```
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
        //
        Connection conn =
            driver.connect("jdbc:mysql:replication://master,slave1,slave2,slave3/test",
                props);
        //
        // Perform read/write work on the master
        // by setting the read-only flag to "false"
        //
        conn.setReadOnly(false);
        conn.setAutoCommit(false);
        conn.createStatement().executeUpdate("UPDATE some_table ....");
        conn.commit();
        //
        // Now, do a query from a slave, the driver automatically picks one
        // from the list
        //
        conn.setReadOnly(true);
        ResultSet rs =
            conn.createStatement().executeQuery("SELECT a,b FROM alt_table");
        .....
    }
}
```

Consider investigating 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 Pool \(lbpool\)](#).

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

MySQL Error Number	MySQL Error Name	Legal (X/) Open SQL State	SQL Standard SQL State
1022	ER_DUPESKEY	23000	KEY
1037	ER_OUTOFMEMORY		OUTOFMEMORY
1038	ER_OUTOFMEMORY		OUTOFMEMORY
1040	ER_CONNOT4ERROR		CONNOT4ERROR
1042	ER_BADINDEX		BADINDEX
1043	ER_HANDSHAKE		HANDSHAKE
1044	ER_DBACCESS_DENIED_ERROR		DBACCESS_DENIED_ERROR
1045	ER_ACCESS_DENIED_ERROR		ACCESS_DENIED_ERROR
1047	ER_UNKNOWN_ERROR		UNKNOWN_ERROR
1050	ER_TABLE_EXISTS_ERROR		TABLE_EXISTS_ERROR
1051	ER_BADTABLE		BADTABLE
1052	ER_NONUNIQUE		NONUNIQUE
1053	ER_SERVERSHUTDOWN		SERVERSHUTDOWN
1054	ER_BADFIELD		BADFIELD
1055	ER_WRONGFIELD_WITH_GROUP		WRONGFIELD_WITH_GROUP
1056	ER_WRONGGROUP_FIELD		WRONGGROUP_FIELD
1057	ER_WRONG_SELECT		WRONG_SELECT
1058	ER_WRONGSUBQUERY_COUNT		WRONGSUBQUERY_COUNT
1059	ER_TOOSLOWIDENT		TOOSLOWIDENT
1060	ER_DUPESFIELD		DUPESFIELD
1061	ER_DUPESKEY		DUPESKEY
1062	ER_DUPESINDEX		DUPESINDEX
1063	ER_WRONGFIELD_SPEC		WRONGFIELD_SPEC
1064	ER_PARSE_ERROR		PARSE_ERROR
1065	ER_EMPTYQUERY		EMPTYQUERY
1066	ER_NONTABLE		NONTABLE
1067	ER_INVALIDDEFAULT		INVALIDDEFAULT
1068	ER_MULTIPLEPRI_KEY		MULTIPLEPRI_KEY
1069	ER_TOOMANYKEYS		TOOMANYKEYS
1070	ER_TOOMANYKEY_PARTS		TOOMANYKEY_PARTS
1071	ER_TOOSLOWKEY		TOOSLOWKEY
1072	ER_KEY_COLUMN_DOES_NOT_EXISTS		KEY_COLUMN_DOES_NOT_EXISTS
1073	ER_BLOCKEDAS_KEY		BLOCKEDAS_KEY
1074	ER_TOOSMALLLENGTH		TOOSMALLLENGTH
1075	ER_WRONG_KEY		WRONG_KEY

MySQL Error Number	MySQL Error Name	Legal (X/ Open) SQL State	SQL Standard SQL State
1080	ER_FORSiNG	08S01	08S01
1081	ER_IPSiNG	08S01	08S01
1082	ER_NO_SiNG	08S01	08S01
1083	ER_WRONG	08S01	08S01
1084	ER_BLOSiNG	08S01	08S01
1090	ER_CANSiNG	08S01	08S01
1091	ER_CANSiNG	08S01	08S01
1101	ER_BLOSiNG	08S01	08S01
1102	ER_WRONG	08S01	08S01
1103	ER_WRONG	08S01	08S01
1104	ER_TOOSiNG	08S01	08S01
1106	ER_UNKSiNG	08S01	08S01
1107	ER_WRONG	08S01	08S01
1109	ER_UNKSiNG	08S01	08S01
1110	ER_FIELSiNG	08S01	08S01
1112	ER_UNSiNG	08S01	08S01
1113	ER_TABSiNG	08S01	08S01
1115	ER_UNKSiNG	08S01	08S01
1118	ER_TOOSiNG	08S01	08S01
1120	ER_WRONG	08S01	08S01
1121	ER_NULSiNG	08S01	08S01
1129	ER_HOSiNG	08S01	08S01
1130	ER_HOSiNG	08S01	08S01
1131	ER_PASSiNG	08S01	08S01
1132	ER_PASSiNG	08S01	08S01
1133	ER_PASSiNG	08S01	08S01
1136	ER_WRONG	08S01	08S01
1138	ER_INVAiNG	08S01	08S01
1139	ER_REGiNG	08S01	08S01
1140	ER_MIXiNG	08S01	08S01
1141	ER_NONiNG	08S01	08S01
1142	ER_TABiNG	08S01	08S01
1143	ER_COLiNG	08S01	08S01
1144	ER_ILLEiNG	08S01	08S01
1145	ER_GRAiNG	08S01	08S01
1146	ER_NO_SiNG	08S01	08S01

MySQL Error Number	MySQL Error Name	Legacy (X/ Open) SQL State	SQL Standard SQLState
1147	ER_NONEXISTING_TABLE_GRANT	HY000	
1148	ER_NOT_ALLOWED_COMMAND	42000	
1149	ER_SYNTAX_ERROR	42000	
1152	ER_ABORTING_CONNECTION	22000	
1153	ER_NET_PACKET_TOO_LARGE	10000	
1154	ER_NET_READ_ERROR_FROM_PIPE	10000	
1155	ER_NET_PACKET_TOO_SMALL	10000	
1156	ER_NET_PACKETS_OUT_OF_ORDER	10000	
1157	ER_NET_UNCOMPRESS_ERROR	10000	
1158	ER_NET_READ_ERROR	10000	
1159	ER_NET_READ_INTERRUPTED	10000	
1160	ER_NET_ERROR_ON_WRITE	10000	
1161	ER_NET_WRITE_INTERRUPTED	10000	
1162	ER_TOO_LONG_STRING	22000	
1163	ER_TABLE_CANT_HANDLE_BLOB	42000	
1164	ER_TABLE_CANT_HANDLE_AUTO_INCREMENT	42000	
1166	ER_WRONG_COLUMN_NAME	42000	
1167	ER_WRONG_COLUMN	42000	
1169	ER_DUPESOME	42000	
1170	ER_BLOB_WITHOUT_LENGTH	22000	
1171	ER_PRIMARY_CANT_HAVE_NULL	42000	
1172	ER_TOO_MANY_ROWS	22000	
1173	ER_REQUEST_PRIMARY_KEY	42000	
1177	ER_CHECK_NO_SUCH_TABLE	42000	
1178	ER_CHECK_NOT_IMPLEMENTED	42000	
1179	ER_CANT_OPEN_FILE_DURING_AN_TRANSACTION	22000	
1184	ER_NEW_ABORTING_CONNECTION	22000	
1189	ER_MASTER_READ	10000	
1190	ER_MASTER_WRITE	10000	
1203	ER_TOO_MANY_USER_CONNECTIONS	22000	
1205	ER_LOCK_TIMEOUT	40000	
1207	ER_READ_ONLY_TRANSACTION	22000	
1211	ER_NO_PERMISSION_TO_CREATE_USER	42000	
1213	ER_LOCK_DEADLOCK	40000	
1216	ER_NO_REFERENCED_ROW	40000	
1217	ER_ROW_IS_REFERENCED	40000	

MySQL Error Number	MySQL Error Name	Legacy (X/) Open SQL State	SQL Standard SQL State
1218	ER_CONSISTENT_MASTER	HY000	08000
1222	ER_WRONG_NUMBER_OF_COLUMNS_IN_SELECT	HY000	21000
1226	ER_USER_LIMIT_REACHED	HY000	08000
1230	ER_NO_DEFAULT	HY000	21000
1231	ER_WRONG_VALUE_FOR_VAR	HY000	22000
1232	ER_WRONG_TYPE_FOR_VAR	HY000	22000
1234	ER_CANT_USE_OPTION_HERE	HY000	08000
1235	ER_NOT_SUPPORTED_YET	HY000	08000
1239	ER_WRONG_KEYDEF	HY000	23000
1241	ER_OPERAND_COLUMNS	HY000	21000
1242	ER_SUBQUERY_NO_1_ROW	HY000	21000
1247	ER_ILLEGAL_REFERENCE	HY000	08000
1248	ER_DERIVED_MUST_HAVE_ALIAS	HY000	08000
1249	ER_SELECT_REDUCED	HY000	08000
1250	ER_TABLE_NOT_ALLOWED_HERE	HY000	08000
1251	ER_NOT_SUPPORTED_AUTH_MODE	HY000	08000
1252	ER_SPATIAL_CANT_HAVE_NULL	HY000	23000
1253	ER_COLLATION_CHARSET_MISMATCH	HY000	08000
1261	ER_WARN_TOO_FEW_RECORDS	HY000	08000
1262	ER_WARN_TOO_MANY_RECORDS	HY000	08000
1263	ER_WARN_NULL_TO_NOTNULL	HY000	08000
1264	ER_WARN_DATA_OUT_OF_RANGE	HY000	08000
1265	ER_WARN_DATA_TRUNCATED	HY000	08000
1280	ER_WRONG_NAME_FOR_INDEX	HY000	23000
1281	ER_WRONG_NAME_FOR_CATALOG	HY000	23000
1286	ER_UNKNOWN_STORAGE_ENGINE	HY000	08000

Chapter 6. JDBC Concepts

Table of Contents

6.1. Connecting to MySQL Using the JDBC <code>DriverManager</code> Interface	49
6.2. Using JDBC <code>Statement</code> Objects to Execute SQL	50
6.3. Using JDBC <code>CallableStatements</code> to Execute Stored Procedures	51
6.4. Retrieving <code>AUTO_INCREMENT</code> Column Values through JDBC	54

This section provides some general JDBC background.

6.1. Connecting to MySQL Using the JDBC `DriverManager` Interface

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

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

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

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
// Notice, do not import com.mysql.jdbc.*
// or you will have problems!
public class LoadDriver {
    public static void main(String[] args) {
        try {
            // The newInstance() call is a work around for some
            // broken Java implementations
            Class.forName("com.mysql.jdbc.Driver").newInstance();
        } catch (Exception ex) {
            // handle the error
        }
    }
}
```

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

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

If you have not already done so, please review the section [Section 6.1, “Connecting to MySQL Using the JDBC `DriverManager` Interface”](#) before working with these examples.

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

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
Connection conn = null;
...
try {
    conn =
        DriverManager.getConnection("jdbc:mysql://localhost/test?" +
                                   "user=monty&password=greatsqlldb");
    // Do something with the Connection
    ...
} catch (SQLException ex) {
    // handle any errors
    System.out.println("SQLException: " + ex.getMessage());
    System.out.println("SQLState: " + ex.getSQLState());
    System.out.println("VendorError: " + ex.getErrorCode());
}
```

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

6.2. Using JDBC `Statement` Objects to Execute SQL

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

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

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

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

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

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

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.ResultSet;
// assume that conn is an already created JDBC connection (see previous examples)
Statement stmt = null;
ResultSet rs = null;
try {
    stmt = conn.createStatement();
```



```

rs = stmt.executeQuery("SELECT foo FROM bar");
// or alternatively, if you don't know ahead of time that
// the query will be a SELECT...
if (stmt.execute("SELECT foo FROM bar")) {
    rs = stmt.getResultSet();
}
// Now do something with the ResultSet ....
}
catch (SQLException ex){
    // handle any errors
    System.out.println("SQLException: " + ex.getMessage());
    System.out.println("SQLState: " + ex.getSQLState());
    System.out.println("VendorError: " + ex.getErrorCode());
}
finally {
    // it is a good idea to release
    // resources in a finally{} block
    // in reverse-order of their creation
    // if they are no-longer needed
    if (rs != null) {
        try {
            rs.close();
        } catch (SQLException sqlEx) { } // ignore
        rs = null;
    }
    if (stmt != null) {
        try {
            stmt.close();
        } catch (SQLException sqlEx) { } // ignore
        stmt = null;
    }
}
}

```

6.3. Using JDBC `CallableStatements` to Execute Stored Procedures

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

For more information on MySQL stored procedures, please refer to <http://dev.mysql.com/doc/mysql/en/stored-routines.html>.

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

Note

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

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

Example 6.3. Connector/J: Calling Stored Procedures

```

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

```

```

SET inOutParam = z;
SELECT inputParam;
SELECT CONCAT('zyxw', inputParam);
END

```

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

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

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

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

```

import java.sql.CallableStatement;
...
//
// Prepare a call to the stored procedure 'demoSp'
// with two parameters
//
// Notice the use of JDBC-escape syntax ({call ...})
//
CallableStatement cStmt = conn.prepareCall("{call demoSp(?, ?)}");
cStmt.setString(1, "abcdefg");

```

Note

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

2. Register the output parameters (if any exist)

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

Example 6.5. Connector/J: Registering output parameters

```

import java.sql.Types;
...
//
// Connector/J supports both named and indexed
// output parameters. You can register output
// parameters using either method, as well
// as retrieve output parameters using either
// method, regardless of what method was
// used to register them.
//
// The following examples show how to use
// the various methods of registering
// output parameters (you should of course
// use only one registration per parameter).
//
//
// Registers the second parameter as output, and
// uses the type 'INTEGER' for values returned from
// getObject()
//
cStmt.registerOutParameter(2, Types.INTEGER);

```

```
//  
// Registers the named parameter 'inOutParam', and  
// uses the type 'INTEGER' for values returned from  
// getObject()  
//  
cStmt.registerOutParameter("inOutParam", Types.INTEGER);  
...
```

3. Set the input parameters (if any exist)

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

Example 6.6. Connector/J: Setting `CallableStatement` input parameters

```
...  
//  
// Set a parameter by index  
//  
cStmt.setString(1, "abcdefg");  
//  
// Alternatively, set a parameter using  
// the parameter name  
//  
cStmt.setString("inputParameter", "abcdefg");  
//  
// Set the 'in/out' parameter using an index  
//  
cStmt.setInt(2, 1);  
//  
// Alternatively, set the 'in/out' parameter  
// by name  
//  
cStmt.setInt("inOutParam", 1);  
...
```

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

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

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

```
...  
boolean hadResults = cStmt.execute();  
//  
// Process all returned result sets  
//  
while (hadResults) {  
    ResultSet rs = cStmt.getResultSet();  
    // process result set  
    ...  
    hadResults = cStmt.getMoreResults();  
}  
//  
// Retrieve output parameters  
//  
// Connector/J supports both index-based and  
// name-based retrieval  
//  
int outputValue = cStmt.getInt(2); // index-based  
outputValue = cStmt.getInt("inOutParam"); // name-based
```

...

6.4. Retrieving `AUTO_INCREMENT` Column Values through JDBC

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

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

```
Statement stmt = null;
ResultSet rs = null;
try {
    //
    // Create a Statement instance that we can use for
    // 'normal' result sets assuming you have a
    // Connection 'conn' to a MySQL database already
    // available
    stmt = conn.createStatement(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))");

    //
    // 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
    }
    rs.close();
    rs = null;
    System.out.println("Key returned from getGeneratedKeys(): "
        + autoIncKeyFromApi);
}
```

```

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

```

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

```

Statement stmt = null;
ResultSet rs = null;
try {
    //
    // Create a Statement instance that we can use for
    // 'normal' result sets.
    stmt = conn.createStatement();
    //
    // Issue the DDL queries for the table for this example
    //
    stmt.executeUpdate("DROP TABLE IF EXISTS autoIncTutorial");
    stmt.executeUpdate(
        "CREATE TABLE autoIncTutorial ( "
        + "priKey INT NOT NULL AUTO_INCREMENT, "
        + "dataField VARCHAR(64), PRIMARY KEY (priKey))");
    //
    // Insert one row that will generate an AUTO INCREMENT
    // key in the 'priKey' field
    //
    stmt.executeUpdate(
        "INSERT INTO autoIncTutorial (dataField) "
        + "values ('Can I Get the Auto Increment Field?')");
    //
    // Use the MySQL LAST_INSERT_ID()
    // function to do the same thing as getGeneratedKeys()
    //
    int autoIncKeyFromFunc = -1;
    rs = stmt.executeQuery("SELECT LAST_INSERT_ID()");
    if (rs.next()) {
        autoIncKeyFromFunc = rs.getInt(1);
    } else {
        // throw an exception from here
    }
    rs.close();
    System.out.println("Key returned from " +
        "'SELECT LAST_INSERT_ID()': " +
        autoIncKeyFromFunc);
} finally {
    if (rs != null) {
        try {
            rs.close();
        } catch (SQLException ex) {
            // ignore
        }
    }
    if (stmt != null) {

```

```

        try {
            stmt.close();
        } catch (SQLException ex) {
            // ignore
        }
    }
}

```

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

```

Statement stmt = null;
ResultSet rs = null;
try {
    //
    // Create a Statement instance that we can use for
    // 'normal' result sets as well as an 'updatable'
    // one, assuming you have a Connection 'conn' to
    // a MySQL database already available
    //
    stmt = conn.createStatement(java.sql.ResultSet.TYPE_FORWARD_ONLY,
                                java.sql.ResultSet.CONCUR_UPDATABLE);

    //
    // Issue the DDL queries for the table for this example
    //
    stmt.executeUpdate("DROP TABLE IF EXISTS autoIncTutorial");
    stmt.executeUpdate(
        "CREATE TABLE autoIncTutorial ( "
        + "priKey INT NOT NULL AUTO_INCREMENT, "
        + "dataField VARCHAR(64), PRIMARY KEY (priKey))");

    //
    // Example of retrieving an AUTO INCREMENT key
    // from an updatable result set
    //
    rs = stmt.executeQuery("SELECT priKey, dataField "
        + "FROM autoIncTutorial");
    rs.moveToInsertRow();
    rs.updateString("dataField", "AUTO INCREMENT here?");
    rs.insertRow();
    //
    // the driver adds rows at the end
    //
    rs.last();
    //
    // We should now be on the row we just inserted
    //
    int autoIncKeyFromRS = rs.getInt("priKey");
    rs.close();
    rs = null;
    System.out.println("Key returned for inserted row: "
        + autoIncKeyFromRS);
} finally {
    if (rs != null) {
        try {
            rs.close();
        } catch (SQLException ex) {
            // ignore
        }
    }
    if (stmt != null) {
        try {
            stmt.close();
        } catch (SQLException ex) {
            // ignore
        }
    }
}

```

```
}
```

Running the preceding example code should produce the following output:

```
Key returned from getGeneratedKeys(): 1  
Key returned from SELECT LAST_INSERT_ID(): 1  
Key returned for inserted row: 2
```

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 lead to unpredictable behavior under load.

Using Connection Pooling with Connector/J

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

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

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

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

```

import javax.naming.InitialContext;
import javax.sql.DataSource;
public class MyServletJspOrEjb {
    public void doSomething() throws Exception {
        /*
         * Create a JNDI Initial context to be able to
         * lookup the DataSource
         *
         * In production-level code, this should be cached as
         * an instance or static variable, as it can
         * be quite expensive to create a JNDI context.
         *
         * Note: This code only works when you are using servlets
         * or EJBs in a J2EE application server. If you are
         * using connection pooling in standalone Java code, you
         * will have to create/configure datasources using whatever
         * mechanisms your particular connection pooling library
         * provides.
         */
        InitialContext ctx = new InitialContext();
        /*
         * Lookup the DataSource, which will be backed by a pool
         * that the application server provides. DataSource instances
         * are also a good candidate for caching as an instance
         * variable, as JNDI lookups can be expensive as well.
         */
        DataSource ds =
            (DataSource)ctx.lookup("java:comp/env/jdbc/MySQLDB");
        /*
         * The following code is what would actually be in your
         * Servlet, JSP or EJB 'service' method...where you need
         * to work with a JDBC connection.
         */
        Connection conn = null;
        Statement stmt = null;
        try {
            conn = ds.getConnection();
            /*
             * Now, use normal JDBC programming to work with
             * MySQL, making sure to close each resource when you're
             * finished with it, which permits the connection pool
             * resources to be recovered as quickly as possible
             */
            stmt = conn.createStatement();
            stmt.execute("SOME SQL QUERY");
            stmt.close();
            stmt = null;
            conn.close();
            conn = null;
        } finally {
            /*
             * close any jdbc instances here that weren't
             * explicitly closed during normal code path, so
             * that we don't 'leak' resources...
             */
            if (stmt != null) {
                try {
                    stmt.close();
                } catch (SQLException sqlex) {
                    // ignore, as we can't do anything about it here
                }
                stmt = null;
            }
            if (conn != null) {
                try {
                    conn.close();
                } catch (SQLException sqlex) {

```

```
        // ignore, as we can't do anything about it here
    }
    conn = null;
}
}
```

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

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

Sizing the Connection Pool

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

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

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

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

Validating Connections

MySQL Connector/J can validate the connection by executing a lightweight ping against a server. In the case of load-balanced connections, this is performed against all active pooled internal connections that are retained. This is beneficial to Java applications using connection pools, as the pool can use this feature to validate connections. Depending on your connection pool and configuration, this validation can be carried out at different times:

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

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

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

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

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

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

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

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

Chapter 8. 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.

There are two connection string options 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 string options, a number of monitoring properties are available:

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

The following management operations can also be performed:

- Add host.
- Remove host.

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

- `int getActiveHostCount(String group);`
- `int getTotalHostCount(String group);`
- `long getTotalLogicalConnectionCount(String group);`
- `long getActiveLogicalConnectionCount(String group);`
- `long getActivePhysicalConnectionCount(String group);`
- `long getTotalPhysicalConnectionCount(String group);`

- `long getTotalTransactionCount(String group);`
- `void removeHost(String group, String host) throws SQLException;`
- `void stopNewConnectionsToHost(String group, String host) throws SQLException;`
- `void addHost(String group, String host, boolean forExisting);`
- `String getActiveHostsList(String group);`
- `String getRegisteredConnectionGroups();`

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

You can test this setup with the following code:

```
public class Test {
    private static String URL = "jdbc:mysql:loadbalance://" +
        "localhost:3306,localhost:3310/test?" +
        "loadBalanceConnectionGroup=first&loadBalanceEnableJMX=true";
    public static void main(String[] args) throws Exception {
        new Thread(new Repeater()).start();
        new Thread(new Repeater()).start();
        new Thread(new Repeater()).start();
    }
    static Connection getNewConnection() throws SQLException, ClassNotFoundException {
        Class.forName("com.mysql.jdbc.Driver");
        return DriverManager.getConnection(URL, "root", "");
    }
    static void executeSimpleTransaction(Connection c, int conn, int trans){
        try {
            c.setAutoCommit(false);
            Statement s = c.createStatement();
            s.executeQuery("SELECT SLEEP(1) /* Connection: " + conn + ", transaction: " + trans + " */");
            c.commit();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
    public static class Repeater implements Runnable {
        public void run() {
            for(int i=0; i < 100; i++){
                try {
                    Connection c = getNewConnection();
                    for(int j=0; j < 10; j++){
                        executeSimpleTransaction(c, i, j);
                        Thread.sleep(Math.round(100 * Math.random()));
                    }
                    c.close();
                    Thread.sleep(100);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

After compiling, the application can be started with the `-Dcom.sun.management.jmxremote` flag, to enable remote management. `jconsole` can then be started. The `Test` main class

will be listed by `jconsole`. Select this and click `Connect`. You can then navigate to the `com.mysql.jdbc.jmx.LoadBalanceConnectionGroupManager` bean. At this point, you can click on various operations and examine the returned result.

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

For further information on the combination of load balancing and failover, see [Chapter 9, Failover with Connector/J](#).

Chapter 9. Failover with Connector/J

Connector/J provides a useful load-balancing implementation for Cluster or multi-master deployments, as explained in [Chapter 8, Load Balancing with Connector/J](#). 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 [SQLExceptions](#) trigger failover.

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

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

A [SQLException](#) is passed in, and a boolean returned. A value of [true](#) triggers a failover, [false](#) does not.

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

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

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

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

```
loadBalanceSQLStateFailover=00,12345
```

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

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

While the three fail-over 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":

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

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

Chapter 10. 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 11. Using Connector/J with Tomcat

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

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

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

```
<Context ....>
...
<Resource name="jdbc/MySQLDB"
          auth="Container"
          type="javax.sql.DataSource"/>
<ResourceParams name="jdbc/MySQLDB">
  <parameter>
    <name>factory</name>
    <value>org.apache.commons.dbcp.BasicDataSourceFactory</value>
  </parameter>
  <parameter>
    <name>maxActive</name>
    <value>10</value>
  </parameter>
  <parameter>
    <name>maxIdle</name>
    <value>5</value>
  </parameter>
  <parameter>
    <name>validationQuery</name>
    <value>SELECT 1</value>
  </parameter>
  <parameter>
    <name>testOnBorrow</name>
    <value>true</value>
  </parameter>
  <parameter>
    <name>testWhileIdle</name>
    <value>true</value>
  </parameter>
  <parameter>
    <name>timeBetweenEvictionRunsMillis</name>
    <value>10000</value>
  </parameter>
  <parameter>
    <name>minEvictableIdleTimeMillis</name>
    <value>60000</value>
  </parameter>
  <parameter>
    <name>username</name>
    <value>someuser</value>
  </parameter>
  <parameter>
    <name>password</name>
    <value>somepass</value>
  </parameter>
  <parameter>
    <name>driverClassName</name>
    <value>com.mysql.jdbc.Driver</value>
  </parameter>
</parameter>
```

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

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

```
<parameter>
  <name>validationQuery</name>
  <value>/* ping */</value>
</parameter>
```

Note that `/* ping */` has to be specified exactly.

In general, follow the installation instructions that come with your version of Tomcat, as the way you configure datasources in Tomcat changes from time to time, and if you use the wrong syntax in your XML file, you will most likely end up with an exception similar to the following:

```
Error: java.sql.SQLException: Cannot load JDBC driver class 'null ' SQL
state: null
```

Note that the auto-loading of drivers having the `META-INF/service/java.sql.Driver` class in JDBC 4.0 causes an improper undeployment of the Connector/J driver in Tomcat on Windows. Namely, the Connector/J jar remains locked. This is an initialization problem that is not related to the driver. The possible workarounds, if viable, are as follows: use `"antiResourceLocking=true"` as a Tomcat Context attribute, or remove the `META-INF/` directory.

Chapter 12. Using Connector/J with JBoss

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:

```
<datasources>
  <local-tx-datasource>
    <jndi-name>MySQLDB</jndi-name>
    <connection-url>jdbc:mysql://localhost:3306/dbname</connection-url>
    <driver-class>com.mysql.jdbc.Driver</driver-class>
    <user-name>user</user-name>
    <password>pass</password>
    <min-pool-size>5</min-pool-size>
    <max-pool-size>20</max-pool-size>
    <idle-timeout-minutes>5</idle-timeout-minutes>
    <exception-sorter-class-name>
com.mysql.jdbc.integration.jboss.ExtendedMysqlExceptionSorter
    </exception-sorter-class-name>
    <valid-connection-checker-class-name>
com.mysql.jdbc.integration.jboss.MysqlValidConnectionChecker
    </valid-connection-checker-class-name>
  </local-tx-datasource>
</datasources>
```

Chapter 13. Using Connector/J with Spring

Table of Contents

13.1. Using <code>JdbcTemplate</code>	76
13.2. Transactional JDBC Access	77
13.3. Connection Pooling with Spring	79

The Spring Framework is a Java-based application framework designed for assisting in application design by providing a way to configure components. The technique used by Spring is a well known design pattern called Dependency Injection (see [Inversion of Control Containers and the Dependency Injection pattern](#)). This article will focus on Java-oriented access to MySQL databases with Spring 2.0. For those wondering, there is a .NET port of Spring appropriately named Spring.NET.

Spring is not only a system for configuring components, but also includes support for aspect oriented programming (AOP). This is one of the main benefits and the foundation for Spring's resource and transaction management. Spring also provides utilities for integrating resource management with JDBC and Hibernate.

For the examples in this section the MySQL world sample database will be used. The first task is to set up a MySQL data source through Spring. Components within Spring use the "bean" terminology. For example, to configure a connection to a MySQL server supporting the world sample database, you might use:

```
<util:map id="dbProps">
  <entry key="db.driver" value="com.mysql.jdbc.Driver"/>
  <entry key="db.jdbcurl" value="jdbc:mysql://localhost/world"/>
  <entry key="db.username" value="myuser"/>
  <entry key="db.password" value="mypass"/>
</util:map>
```

In the above example, we are assigning values to properties that will be used in the configuration. For the datasource configuration:

```
<bean id="dataSource"
  class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="driverClassName" value="${db.driver}"/>
  <property name="url" value="${db.jdbcurl}"/>
  <property name="username" value="${db.username}"/>
  <property name="password" value="${db.password}"/>
</bean>
```

The placeholders are used to provide values for properties of this bean. This means that you can specify all the properties of the configuration in one place instead of entering the values for each property on each bean. We do, however, need one more bean to pull this all together. The last bean is responsible for actually replacing the placeholders with the property values.

```
<bean
  class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <property name="properties" ref="dbProps"/>
</bean>
```

```
</bean>
```

Now that we have our MySQL data source configured and ready to go, we write some Java code to access it. The example below will retrieve three random cities and their corresponding country using the data source we configured with Spring.

```
// Create a new application context. this processes the Spring config
ApplicationContext ctx =
    new ClassPathXmlApplicationContext("exlappContext.xml");
// Retrieve the data source from the application context
DataSource ds = (DataSource) ctx.getBean("dataSource");
// Open a database connection using Spring's DataSourceUtils
Connection c = DataSourceUtils.getConnection(ds);
try {
    // retrieve a list of three random cities
    PreparedStatement ps = c.prepareStatement(
        "select City.Name as 'City', Country.Name as 'Country' " +
        "from City inner join Country on City.CountryCode = Country.Code " +
        "order by rand() limit 3");
    ResultSet rs = ps.executeQuery();
    while(rs.next()) {
        String city = rs.getString("City");
        String country = rs.getString("Country");
        System.out.printf("The city %s is in %s\n", city, country);
    }
} catch (SQLException ex) {
    // something has failed and we print a stack trace to analyse the error
    ex.printStackTrace();
    // ignore failure closing connection
    try { c.close(); } catch (SQLException e) { }
} finally {
    // properly release our connection
    DataSourceUtils.releaseConnection(c, ds);
}
```

This is very similar to normal JDBC access to MySQL with the main difference being that we are using `DataSourceUtils` instead of the `DriverManager` to create the connection.

While it may seem like a small difference, the implications are somewhat far reaching. Spring manages this resource in a way similar to a container managed data source in a J2EE application server. When a connection is opened, it can be subsequently accessed in other parts of the code if it is synchronized with a transaction. This makes it possible to treat different parts of your application as transactional instead of passing around a database connection.

13.1. Using JdbcTemplate

Spring makes extensive use of the Template method design pattern (see [Template Method Pattern](#)). Our immediate focus will be on the `JdbcTemplate` and related classes, specifically `NamedParameterJdbcTemplate`. The template classes handle obtaining and releasing a connection for data access when one is needed.

The next example shows how to use `NamedParameterJdbcTemplate` inside of a DAO (Data Access Object) class to retrieve a random city given a country code.

```
public class Ex2JdbcDao {
    /**
     * Data source reference which will be provided by Spring.
     */
    private DataSource dataSource;
```

```

/**
 * Our query to find a random city given a country code. Notice
 * the ":country" parameter toward the end. This is called a
 * named parameter.
 */
private String queryString = "select Name from City " +
    "where CountryCode = :country order by rand() limit 1";
/**
 * Retrieve a random city using Spring JDBC access classes.
 */
public String getRandomCityByCountryCode(String centryCode) {
    // 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", centryCode);
    // We query for an Object and specify what class we are expecting
    return (String)template.queryForObject(queryString, params, String.class);
}
/**
 * A JavaBean setter-style method to allow Spring to inject the data source.
 * @param dataSource
 */
public void setDataSource(DataSource dataSource) {
    this.dataSource = dataSource;
}
}

```

The focus in the above code is on the `getRandomCityByCountryCode()` method. We pass a country code and use the `NamedParameterJdbcTemplate` to query for a city. The country code is placed in a Map with the key "country", which is the parameter is named in the SQL query.

To access this code, you need to configure it with Spring by providing a reference to the data source.

```

<bean id="dao" class="code.Ex2JdbcDao">
    <property name="dataSource" ref="dataSource"/>
</bean>

```

At this point, we can just grab a reference to the DAO from Spring and call `getRandomCityByCountryCode()`.

```

// Create the application context
ApplicationContext ctx =
    new ClassPathXmlApplicationContext("ex2appContext.xml");
// Obtain a reference to our DAO
Ex2JdbcDao dao = (Ex2JdbcDao) ctx.getBean("dao");
String countryCode = "USA";
// Find a few random cities in the US
for(int i = 0; i < 4; ++i)
    System.out.printf("A random city in %s is %s\n", countryCode,
        dao.getRandomCityByCountryCode(countryCode));

```

This example shows how to use Spring's JDBC classes to completely abstract away the use of traditional JDBC classes including `Connection` and `PreparedStatement`.

13.2. Transactional JDBC Access

You might be wondering how we can add transactions into our code if we do not deal directly with the JDBC classes. Spring provides a transaction management package that not only replaces JDBC

transaction management, but also enables declarative transaction management (configuration instead of code).

To use transactional database access, we will need to change the storage engine of the tables in the world database. The downloaded script explicitly creates MyISAM tables which do not support transactional semantics. The InnoDB storage engine does support transactions and this is what we will be using. We can change the storage engine with the following statements.

```
ALTER TABLE City ENGINE=InnoDB;
ALTER TABLE Country ENGINE=InnoDB;
ALTER TABLE CountryLanguage ENGINE=InnoDB;
```

A good programming practice emphasized by Spring is separating interfaces and implementations. What this means is that we can create a Java interface and only use the operations on this interface without any internal knowledge of what the actual implementation is. We will let Spring manage the implementation and with this it will manage the transactions for our implementation.

First you create a simple interface:

```
public interface Ex3Dao {
    Integer createCity(String name, String countryCode,
        String district, Integer population);
}
```

This interface contains one method that will create a new city record in the database and return the id of the new record. Next you need to create an implementation of this interface.

```
public class Ex3DaoImpl implements Ex3Dao {
    protected DataSource dataSource;
    protected SqlUpdate updateQuery;
    protected SqlFunction idQuery;
    public Integer createCity(String name, String countryCode,
        String district, Integer population) {
        updateQuery.update(new Object[] { name, countryCode,
            district, population });
        return getLastId();
    }
    protected Integer getLastId() {
        return idQuery.run();
    }
}
```

You can see that we only operate on abstract query objects here and do not deal directly with the JDBC API. Also, this is the complete implementation. All of our transaction management will be dealt with in the configuration. To get the configuration started, we need to create the DAO.

```
<bean id="dao" class="code.Ex3DaoImpl">
    <property name="dataSource" ref="dataSource"/>
    <property name="updateQuery">...</property>
    <property name="idQuery">...</property>
</bean>
```

Now you need to set up the transaction configuration. The first thing you must do is create transaction manager to manage the data source and a specification of what transaction properties are required for the [dao](#) methods.

```
<bean id="transactionManager"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource" />
</bean>
<tx:advice id="txAdvice" transaction-manager="transactionManager">
  <tx:attributes>
    <tx:method name="*" />
  </tx:attributes>
</tx:advice>
```

The preceding code creates a transaction manager that handles transactions for the data source provided to it. The `txAdvice` uses this transaction manager and the attributes specify to create a transaction for all methods. Finally you need to apply this advice with an AOP pointcut.

```
<aop:config>
  <aop:pointcut id="daoMethods"
    expression="execution(* code.Ex3Dao.*(..))" />
  <aop:advisor advice-ref="txAdvice" pointcut-ref="daoMethods" />
</aop:config>
```

This basically says that all methods called on the `Ex3Dao` interface will be wrapped in a transaction. To make use of this, you only have to retrieve the `dao` from the application context and call a method on the `dao` instance.

```
Ex3Dao dao = (Ex3Dao) ctx.getBean("dao");
Integer id = dao.createCity(name, countryCode, district, pop);
```

We can verify from this that there is no transaction management happening in our Java code and it is all configured with Spring. This is a very powerful notion and regarded as one of the most beneficial features of Spring.

13.3. Connection Pooling with Spring

In many situations, such as web applications, there will be a large number of small database transactions. When this is the case, it usually makes sense to create a pool of database connections available for web requests as needed. Although MySQL does not spawn an extra process when a connection is made, there is still a small amount of overhead to create and set up the connection. Pooling of connections also alleviates problems such as collecting large amounts of sockets in the `TIME_WAIT` state.

Setting up pooling of MySQL connections with Spring is as simple as changing the data source configuration in the application context. There are a number of configurations that we can use. The first example is based on the [Jakarta Commons DBCP library](#). The example below replaces the source configuration that was based on `DriverManagerDataSource` with DBCP's `BasicDataSource`.

```
<bean id="dataSource" destroy-method="close"
      class="org.apache.commons.dbcp.BasicDataSource">
  <property name="driverClassName" value="${db.driver}" />
  <property name="url" value="${db.jdbcurl}" />
  <property name="username" value="${db.username}" />
  <property name="password" value="${db.password}" />
  <property name="initialSize" value="3" />
</bean>
```

The configuration of the two solutions is very similar. The difference is that DBCP will pool connections to the database instead of creating a new connection every time one is requested. We have also set a parameter here called `initialSize`. This tells DBCP that we want three connections in the pool when it is created.

Another way to configure connection pooling is to configure a data source in our J2EE application server. Using JBoss as an example, you can set up the MySQL connection pool by creating a file called `mysql-local-ds.xml` and placing it in the `server/default/deploy` directory in JBoss. Once we have this setup, we can use JNDI to look it up. With Spring, this lookup is very simple. The data source configuration looks like this.

```
<jee:jndi-lookup id="dataSource" jndi-name="java:MySQL_DS" />
```

Chapter 14. Using Connector/J with GlassFish

Table of Contents

14.1. A Simple JSP Application with Glassfish, Connector/J and MySQL	82
14.2. A Simple Servlet with Glassfish, Connector/J and MySQL	84

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 you will need to make sure it can access MySQL Connector/J. To do this copy the MySQL Connector/J JAR file to the directory `GLASSFISH_INSTALL/glassfish/lib`. For example, copy `mysql-connector-java-5.1.12-bin.jar` to `C:\glassfishv3\glassfish\lib`. Restart the Glassfish Application Server.

You are now ready to create JDBC Connection Pools and JDBC Resources.

Creating a Connection Pool

1. In the Glassfish Administration Console, using the navigation tree navigate to **Resources, JDBC, Connection Pools**.
2. In the **JDBC Connection Pools** frame click **New**. You will enter a two step wizard.
3. In the **Name** field under **General Settings** enter the name for the connection pool, for example enter `MySQLConnPool`.
4. In the **Resource Type** field, select `javax.sql.DataSource` from the drop-down listbox.
5. In the **Database Vendor** field, select `MySQL` from the drop-down listbox. Click **Next** to go to the next page of the wizard.
6. You can accept the default settings for General Settings, Pool Settings and Transactions for this example. Scroll down to Additional Properties.
7. In Additional Properties you will need to ensure the following properties are set:
 - **ServerName** - The server to connect to. For local testing this will be `localhost`.
 - **User** - The user name with which to connect to MySQL.
 - **Password** - The corresponding password for the user.
 - **DatabaseName** - The database to connect to, for example the sample MySQL database `World`.
8. Click **Finish** to exit the wizard. You will be taken to the **JDBC Connection Pools** page where all current connection pools, including the one you just created, will be displayed.
9. In the **JDBC Connection Pools** frame click on the connection pool you just created. Here, you can review and edit information about the connection pool. Because Connector/J does not support optimized validation queries, go to the **Advanced** tab, and under Connection Validation, configure the following settings:
 - **Connection Validation** - select **Required**.

- **Validation Method** - select **table** from the drop-down menu.
- **Table Name** - enter **DUAL**.

10. To test your connection pool click the **Ping** button at the top of the frame. A message will be displayed confirming correct operation or otherwise. If an error message is received recheck the previous steps, and ensure that MySQL Connector/J has been correctly copied into the previously specified location.

Now that you have created a connection pool you will also need to create a JDBC Resource (data source) for use by your application.

Creating a JDBC Resource

Your Java application will usually reference a data source object to establish a connection with the database. This needs to be created first using the following procedure.

- Using the navigation tree in the Glassfish Administration Console, navigate to **Resources**, **JDBC**, **JDBC Resources**. A list of resources will be displayed in the **JDBC Resources** frame.
- Click **New**. The **New JDBC Resource** frame will be displayed.
- In the **JNDI Name** field, enter the JNDI name that will be used to access this resource, for example enter **jdbc/MySQLDataSource**.
- In the **Pool Name** field, select a connection pool you want this resource to use from the drop-down listbox.
- Optionally, you can enter a description into the **Description** field.
- Additional properties can be added if required.
- Click **OK** to create the new JDBC resource. The **JDBC Resources** frame will list all available JDBC Resources.

14.1. A Simple JSP Application with Glassfish, Connector/J and MySQL

This section shows how to deploy a simple JSP application on Glassfish, that connects to a MySQL database.

This example assumes you have already set up a suitable Connection Pool and JDBC Resource, as explained in the preceding sections. It is also assumed you have a sample database installed, such as **world**.

The main application code, **index.jsp** is presented here:

```
<%@ page import="java.sql.*, javax.sql.*, java.io.*, javax.naming.*" %>
<html>
<head><title>Hello world from JSP</title></head>
<body>
<%
    InitialContext ctx;
    DataSource ds;
    Connection conn;
    Statement stmt;
    ResultSet rs;
```



```
try {
    ctx = new InitialContext();
    ds = (DataSource) ctx.lookup("java:comp/env/jdbc/MySQLDataSource");
    //ds = (DataSource) ctx.lookup("jdbc/MySQLDataSource");
    conn = ds.getConnection();
    stmt = conn.createStatement();
    rs = stmt.executeQuery("SELECT * FROM Country");
    while(rs.next()) {
%>
        <h3>Name: <%= rs.getString("Name") %></h3>
        <h3>Population: <%= rs.getString("Population") %></h3>
<%
    }
    }
    catch (SQLException se) {
%>
        <%= se.getMessage() %>
<%
    }
    catch (NamingException ne) {
%>
        <%= ne.getMessage() %>
<%
    }
%>
</body>
</html>
```

In addition two XML files are required: [web.xml](#), and [sun-web.xml](#). There may be other files present, such as classes and images. These files are organized into the directory structure as follows:

```
index.jsp
WEB-INF
|
- web.xml
- sun-web.xml
```

The code for [web.xml](#) is:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee" xmlns:xsi="http://www.w3.org/2001/XMLSchema
    <display-name>HelloWebApp</display-name>
    <istributable/>
    <resource-ref>
        <res-ref-name>jdbc/MySQLDataSource</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
        <res-sharing-scope>Shareable</res-sharing-scope>
    </resource-ref>
</web-app>
```

The code for [sun-web.xml](#) is:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sun-web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Application Server 8.1 Servlet 2.4//EN" "http
<sun-web-app>
    <context-root>HelloWebApp</context-root>
    <resource-ref>
        <res-ref-name>jdbc/MySQLDataSource</res-ref-name>
```

```
<jndi-name>jdbc/MySQLDataSource</jndi-name>
</resource-ref>
</sun-web-app>
```

These XML files illustrate a very important aspect of running JDBC applications on Glassfish. On Glassfish it is important to map the string specified for a JDBC resource to its JNDI name, as set up in the Glassfish administration console. In this example, the JNDI name for the JDBC resource, as specified in the Glassfish Administration console when creating the JDBC Resource, was `jdbc/MySQLDataSource`. This must be mapped to the name given in the application. In this example the name specified in the application, `jdbc/MySQLDataSource`, and the JNDI name, happen to be the same, but this does not necessarily have to be the case. Note that the XML element `<res-ref-name>` is used to specify the name as used in the application source code, and this is mapped to the JNDI name specified using the `<jndi-name>` element, in the file `sun-web.xml`. The resource also has to be created in the `web.xml` file, although the mapping of the resource to a JNDI name takes place in the `sun-web.xml` file.

If you do not have this mapping set up correctly in the XML files you will not be able to lookup the data source using a JNDI lookup string such as:

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

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

```
ds = (DataSource) ctx.lookup("jdbc/MySQLDataSource");
```

With the source files in place, in the correct directory structure, you are ready to deploy the application:

1. In the navigation tree, navigate to **Applications** - the **Applications** frame will be displayed. Click **Deploy**.
2. You can now deploy an application packaged into a single WAR file from a remote client, or you can choose a packaged file or directory that is locally accessible to the server. If you are simply testing an application locally you can simply point Glassfish at the directory that contains your application, without needing to package the application into a WAR file.
3. Now select the application type from the **Type** drop-down listbox, which in this example is `web application`.
4. Click OK.

Now, when you navigate to the **Applications** frame, you will have the option to **Launch**, **Redeploy**, or **Restart** your application. You can test your application by clicking **Launch**. The application will connection to the MySQL database and display the Name and Population of countries in the `Country` table.

14.2. A Simple Servlet with Glassfish, Connector/J and MySQL

This section describes a simple servlet that can be used in the Glassfish environment to access a MySQL database. As with the previous section, this example assumes the sample database `world` is installed.

The project is set up with the following directory structure:

```
index.html
WEB-INF
|
- web.xml
- sun-web.xml
```

```
- classes
|
- HelloWorldServlet.java
- HelloWorldServlet.class
```

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

```
import javax.servlet.http.*;
import javax.servlet.*;
import java.io.*;
import java.sql.*;
import javax.sql.*;
import javax.naming.*;

public class HelloWorldServlet extends HttpServlet {
    InitialContext ctx = null;
    DataSource ds = null;
    Connection conn = null;
    PreparedStatement ps = null;
    ResultSet rs = null;
    String sql = "SELECT Name, Population FROM Country WHERE Name=?";

    public void init () throws ServletException {
        try {
            ctx = new InitialContext();
            ds = (DataSource) ctx.lookup("java:comp/env/jdbc/MySQLDataSource");
            conn = ds.getConnection();
            ps = conn.prepareStatement(sql);
        }
        catch (SQLException se) {
            System.out.println("SQLException: "+se.getMessage());
        }
        catch (NamingException ne) {
            System.out.println("NamingException: "+ne.getMessage());
        }
    }

    public void destroy () {
        try {
            if (rs != null)
                rs.close();
            if (ps != null)
                ps.close();
            if (conn != null)
                conn.close();
            if (ctx != null)
                ctx.close();
        }
        catch (SQLException se) {
            System.out.println("SQLException: "+se.getMessage());
        }
        catch (NamingException ne) {
            System.out.println("NamingException: "+ne.getMessage());
        }
    }

    public void doPost(HttpServletRequest req, HttpServletResponse resp){
        try {
            String country_name = req.getParameter("country_name");
            resp.setContentType("text/html");
            PrintWriter writer = resp.getWriter();
            writer.println("<html><body>");
            writer.println("<p>Country: "+country_name+"</p>");
            ps.setString(1, country_name);
            rs = ps.executeQuery();
            if (!rs.next()){
                writer.println("<p>Country does not exist!</p>");
            }
            else {
```

```

        rs.beforeFirst();
        while(rs.next()) {
            writer.println("<p>Name: "+rs.getString("Name")+"</p>");
            writer.println("<p>Population: "+rs.getString("Population")+"</p>");
        }
        writer.println("</body></html>");
        writer.close();
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}

public void doGet(HttpServletRequest req, HttpServletResponse resp){
    try {
        resp.setContentType("text/html");
        PrintWriter writer = resp.getWriter();
        writer.println("<html><body>");
        writer.println("<p>Hello from servlet doGet()</p>");
        writer.println("</body></html>");
        writer.close();
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

In the preceding code a basic `doGet()` method is implemented, but is not used in the example. The code to establish the connection with the database is as shown in the previous example, [Section 14.1, “A Simple JSP Application with Glassfish, Connector/J and MySQL”](#), and is most conveniently located in the servlet `init()` method. The corresponding freeing of resources is located in the destroy method. The main functionality of the servlet is located in the `doPost()` method. If the user enters into the input form a country name that can be located in the database, the population of the country is returned. The code is invoked using a POST action associated with the input form. The form is defined in the file [index.html](#):

```

<html>
  <head><title>HelloWebServlet</title></head>

  <body>
    <h1>HelloWebServlet</h1>

    <p>Please enter country name:</p>

    <form action="HelloWebServlet" method="POST">
      <input type="text" name="country_name" length="50" />
      <input type="submit" value="Submit" />
    </form>

  </body>
</html>

```

The XML files [web.xml](#) and [sun-web.xml](#) are as for the example in the preceding section, [Section 14.1, “A Simple JSP Application with Glassfish, Connector/J and MySQL”](#), no additional changes are required.

When compiling the Java source code, you will need to specify the path to the file [javaee.jar](#). On Windows, this can be done as follows:

```

shell> javac -classpath c:\glassfishv3\glassfish\lib\javaee.jar HelloWebServlet.java

```

Once the code is correctly located within its directory structure, and compiled, the application can be deployed in Glassfish. This is done in exactly the same way as described in the preceding section, [Section 14.1, “A Simple JSP Application with Glassfish, Connector/J and MySQL”](#).

Once deployed the application can be launched from within the Glassfish Administration Console. Enter a country name such as “England”, and the application will return “Country does not exist!”. Enter “France”, and the application will return a population of 59225700.

Chapter 15. Troubleshooting Connector/J Applications

This section explains the symptoms and resolutions for the most commonly encountered issues with applications using MySQL Connector/J.

Questions

- [15.1: \[90\]](#) 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: \[90\]](#) My application throws an SQLException 'No Suitable Driver'. Why is this happening?
- [15.3: \[91\]](#) 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: \[91\]](#) I have a servlet/application that works fine for a day, and then stops working overnight
- [15.5: \[93\]](#) 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: \[93\]](#) I cannot connect to the MySQL server using Connector/J, and I'm sure the connection parameters are correct.
- [15.7: \[93\]](#) 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: \[94\]](#) 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: \[94\]](#) When using `gcj`, a `java.io.CharConversionException` exception is raised when working with certain character sequences.

- [15.10: \[94\]](#) 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: \[94\]](#) 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: \[94\]](#) 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: \[95\]](#) 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: \[96\]](#) How can I use 3-byte UTF8 with Connector/J?
- [15.15: \[96\]](#) How can I use 4-byte UTF8, [utf8mb4](#) with Connector/J?
- [15.16: \[96\]](#) Using [useServerPrepStmts=false](#) and certain character encodings can lead to corruption when inserting BLOBs. How can this be avoided?

Questions and Answers

15.1: When I try to connect to the database with MySQL Connector/J, I get the following exception:

```
SQLException: Server configuration denies access to data source
SQLState: 08001
VendorError: 0
```

What is going on? I can connect just fine with the MySQL command-line client.

MySQL Connector/J must use TCP/IP sockets to connect to MySQL, as Java does not support Unix Domain Sockets. Therefore, when MySQL Connector/J connects to MySQL, the security manager in MySQL server will use its grant tables to determine whether the connection is permitted.

You must add the necessary security credentials to the MySQL server for this to happen, using the [GRANT](#) statement to your MySQL Server. See [GRANT 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 [jdbc.drivers](#) system property has not been populated with the location of the Connector/J driver.

15.3: I'm trying to use MySQL Connector/J in an applet or application and I get an exception similar to:

```
SQLException: Cannot connect to MySQL server on host:3306.  
Is there a MySQL server running on the machine/port you  
are trying to connect to?  
(java.security.AccessControlException)  
SQLState: 08S01  
VendorError: 0
```

Either you're running an Applet, your MySQL server has been installed with the "skip-networking" 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

```
public void doBusinessOp() throws SQLException {  
    Connection conn = null;  
    Statement stmt = null;  
    ResultSet rs = null;  
    //  
    // How many times do you want to retry the transaction  
    // (or at least _getting_ a connection)?  
    //  
    int retryCount = 5;  
    boolean transactionCompleted = false;  
    do {  
        try {  
            conn = getConnection(); // assume getting this from a  
                                   // javax.sql.DataSource, or the
```

```

// java.sql.DriverManager
conn.setAutoCommit(false);
//
// Okay, at this point, the 'retry-ability' of the
// transaction really depends on your application logic,
// whether or not you're using autocommit (in this case
// not), and whether you're using transactional storage
// engines
//
// For this example, we'll assume that it's _not_ safe
// to retry the entire transaction, so we set retry
// count to 0 at this point
//
// If you were using exclusively transaction-safe tables,
// or your application could recover from a connection going
// bad in the middle of an operation, then you would not
// touch 'retryCount' here, and just let the loop repeat
// until retryCount == 0.
//
retryCount = 0;
stmt = conn.createStatement();
String query = "SELECT foo FROM bar ORDER BY baz";
rs = stmt.executeQuery(query);
while (rs.next()) {
}
rs.close();
rs = null;
stmt.close();
stmt = null;
conn.commit();
conn.close();
conn = null;
transactionCompleted = true;
} catch (SQLException sqlEx) {
//
// The two SQL states that are 'retry-able' are 08S01
// for a communications error, and 40001 for deadlock.
//
// Only retry if the error was due to a stale connection,
// communications problem or deadlock
//
String sqlState = sqlEx.getSQLState();
if ("08S01".equals(sqlState) || "40001".equals(sqlState)) {
    retryCount -= 1;
} else {
    retryCount = 0;
}
} finally {
    if (rs != null) {
        try {
            rs.close();
        } catch (SQLException sqlEx) {
            // You'd probably want to log this...
        }
    }
    if (stmt != null) {
        try {
            stmt.close();
        } catch (SQLException sqlEx) {
            // You'd probably want to log this as well...
        }
    }
    if (conn != null) {
        try {
            //
            // If we got here, and conn is not null, the
            // transaction should be rolled back, as not

```

```

        // all work has been done
        try {
            conn.rollback();
        } finally {
            conn.close();
        }
    } catch (SQLException sqlEx) {
        //
        // If we got an exception here, something
        // pretty serious is going on, so we better
        // pass it up the stack, rather than just
        // logging it...
        throw sqlEx;
    }
}
} while (!transactionCompleted && (retryCount > 0));
}

```

Note

Use of the `autoReconnect` option is not recommended because there is no safe method of reconnecting to the MySQL server without risking some corruption of the connection state or database state information. Instead, use a connection pool, which will enable your application to connect to the MySQL server using an available connection from the pool. The `autoReconnect` facility is deprecated, and may be removed in a future release.

15.5: I'm trying to use JDBC 2.0 updatable result sets, and I get an exception saying my result set is not updatable.

Because MySQL does not have row identifiers, MySQL Connector/J can only update result sets that have come from queries on tables that have at least one [primary key](#), the query must select every primary key column, and the query can only span one table (that is, no joins). This is outlined in the JDBC specification.

Note that this issue only occurs when using updatable result sets, and is caused because Connector/J is unable to guarantee that it can identify the correct rows within the result set to be updated without having a unique reference to each row. There is no requirement to have a unique field on a table if you are using [UPDATE](#) or [DELETE](#) statements on a table where you can individually specify the criteria to be matched using a [WHERE](#) clause.

15.6: I cannot connect to the MySQL server using Connector/J, and I'm sure the connection parameters are correct.

Make sure that the `skip-networking` 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 an 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.

Note

Do not expect to be able to reuse a connection without problems, if it has been lying idle for a period. If a connection is to be reused after being idle for any length of time, ensure that you explicitly test it before reusing it.

15.13: 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:

```
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:

```
useTimezone=true  
useLegacyDatetimeCode=false  
serverTimezone=UTC
```

Chapter 17. Connector/J Support

Table of Contents

17.1. Connector/J Community Support	101
17.2. How to Report Connector/J Bugs or Problems	101

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>](mailto:secalert_us@oracle.com). Exception: Support customers should report all problems, including security bugs, to Oracle Support at <http://support.oracle.com/>.

Writing a good bug report takes patience, but doing it right the first time saves time both for us and for yourself. A good bug report, containing a full test case for the bug, makes it very likely that we will fix the bug in the next release.

This section will help you write your report correctly so that you do not waste your time doing things that may not help us much or at all.

If you have a repeatable bug report, please report it to the bugs database at <http://bugs.mysql.com/>. Any bug that we are able to repeat has a high chance of being fixed in the next MySQL release.

To report other problems, you can use one of the MySQL mailing lists.

Remember that it is possible for us to respond to a message containing too much information, but not to one containing too little. People often omit facts because they think they know the cause of a problem and assume that some details do not matter.

A good principle is this: If you are in doubt about stating something, state it. It is faster and less troublesome to write a couple more lines in your report than to wait longer for the answer if we must ask you to provide information that was missing from the initial report.

The most common errors made in bug reports are (a) not including the version number of Connector/J or MySQL used, and (b) not fully describing the platform on which Connector/J is installed (including the JVM version, and the platform type and version number that MySQL itself is installed on).

This is highly relevant information, and in 99 cases out of 100, the bug report is useless without it. Very often we get questions like, “Why doesn't this work for me?” Then we find that the feature requested wasn't implemented in that MySQL version, or that a bug described in a report has already been fixed in newer MySQL versions.

Sometimes the error is platform-dependent; in such cases, it is next to impossible for us to fix anything without knowing the operating system and the version number of the platform.

If at all possible, create a repeatable, standalone testcase that doesn't involve any third-party classes.

To streamline this process, we ship a base class for testcases with Connector/J, named `'com.mysql.jdbc.util.BaseBugReport'`. To create a testcase for Connector/J using this class, create your own class that inherits from `com.mysql.jdbc.util.BaseBugReport` and override the methods `setUp()`, `tearDown()` and `runTest()`.

In the `setUp()` method, create code that creates your tables, and populates them with any data needed to demonstrate the bug.

In the `runTest()` method, create code that demonstrates the bug using the tables and data you created in the `setUp` method.

In the `tearDown()` method, drop any tables you created in the `setUp()` method.

In any of the above three methods, use one of the variants of the `getConnection()` method to create a JDBC connection to MySQL:

- `getConnection()` - Provides a connection to the JDBC URL specified in `getUrl()`. If a connection already exists, that connection is returned, otherwise a new connection is created.
- `getNewConnection()` - Use this if you need to get a new connection for your bug report (that is, there is more than one connection involved).
- `getConnection(String url)` - Returns a connection using the given URL.
- `getConnection(String url, Properties props)` - Returns a connection using the given URL and properties.

If you need to use a JDBC URL that is different from `'jdbc:mysql:///test'`, override the method `getUrl()` as well.

Use the `assertTrue(boolean expression)` and `assertTrue(String failureMessage, boolean expression)` methods to create conditions that must be met in your testcase demonstrating the behavior you are expecting (vs. the behavior you are observing, which is why you are most likely filing a bug report).

Finally, create a `main()` method that creates a new instance of your testcase, and calls the `run` method:

```
public static void main(String[] args) throws Exception {
    new MyBugReport().run();
}
```

Once you have finished your testcase, and have verified that it demonstrates the bug you are reporting, upload it with your bug report to <http://bugs.mysql.com/>.

Appendix A. Licenses for Third-Party Components

Table of Contents

A.1. Ant-Contrib License	103
A.2. c3p0 JDBC Library License	104
A.3. GNU Lesser General Public License Version 2.1, February 1999	104
A.4. jboss-common-jdbc-wrapper.jar License	112
A.5. Simple Logging Facade for Java (SLF4J) License	112

MySQL Connector/J

- [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, “Simple Logging Facade for Java \(SLF4J\) License”](#)

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)."
   Alternately, this acknowledgement may appear in the software itself,
   if and wherever such third-party acknowledgements normally appear.

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.]

Preamble

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 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", as 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.

NO WARRANTY

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. 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 FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.