JCA/JDBC Driver

Release Notes v 2.2.2

Table of Contents

General Notes

Jaybird is a JCA/JDBC driver suite to connect to Firebird database servers.

This driver is based on both the JCA standard for application server connections to enterprise information systems and the well-known JDBC standard. The JCA standard specifies an architecture in which an application server can cooperate with a driver so that the application server manages transactions, security, and resource pooling, and the driver supplies only the connection functionality. While similar to the JDBC `XADataSource` concept, the JCA specification is considerably clearer on the division of responsibility between the application server and driver.

Supported Firebird versions

Jaybird 2.2.2 was tested against Firebird 2.1.5 and 2.5.2, but should also support other Firebird versions from 1.0 and up. The Type 2 and embedded server JDBC drivers require the appropriate JNI library. Precompiled JNI binaries for Win32 and Linux platforms are shipped in the default installation, other platforms require porting/building the JNI library for that platform.

This driver does not support InterBase servers due to Firebird-specific changes in the protocol and database attachment parameters that are sent to the server.

Supported Java versions

Jaybird 2.2.2 supports Java 5 (JDBC 3.0), Java 6 (JDBC 4.0) and Java 7 (JDBC 4.1). Support for earlier Java versions has been dropped.

Specification support

Driver supports the following specifications:

JDBC 4.1	Driver implements all JDBC 4.1 methods added to existing interfaces. The driver explicitly supports `closeOnCompletion`, most other JDBC 4.1 specific methods throw `SQLFeatureNotSupportedException`.
JDBC 4.0	Driver implements all JDBC 4.0 interfaces and supports exception chaining.
JDBC 3.0	Driver implements all JDBC 3.0 interfaces (but will throw `FBDriverNotCapableException` for some methods)
JCA 1.0	Jaybird provides implementation of `javax.resource.spi.ManagedConnectionFactory` and related interfaces. CCI interfaces are not supported. Although Jaybird depends on the JCA 1.5 classes, JCA 1.5 compatibility is currently not guaranteed.
JTA 1.0.1	Driver provides an implementation of `javax.transaction.xa.XAResource` interface via JCA framework and `XADataSource` implementation.
JMX 1.2	Jaybird provides a MBean to manage Firebird servers and installed databases via JMX agent.

What's new in Jaybird 2.2

Jaybird 2.2 introduces the following new features and fixes:

Changes and fixes in Jaybird 2.2.2

The following has been changed or fixed in Jaybird 2.2.2:

- Fixed: `FBMaintenanceManager.listLimboTransactions()` reports incorrect transaction id when the result contains multi-site transactions in limbo (*JDBC-266*)
- Fixed: Calling `PreparedStatement.setClob(int, Clob)` with a non-Firebird `Clob` (eg like Hibernate does) or calling `PreparedStatement.setClob(int, Reader)` throws `FBSQLException`: "You can't start before the beginning of the blob" (*JDBC-281*)
- Fixed: Connection property types not properly processed from `isc_dpb_types.properties` (*JDBC-284*)
- Fixed: JNI implementation of parameter buffer writes incorrect integers (*JDBC-285*, *JDBC-286*)
- Changed: Throw `SQLException` when calling `execute`, `executeQuery`, `executeUpdate` and `addBatch` methods accepting a query string on a `PreparedStatement` or `CallableStatement` as required by JDBC 4.0 (*JDBC-288*)
  NOTE: Be aware that this change can break existing code if you depended on the old, non-standard behavior! With `addBatch(String)` the old behavior lead to a memory leak and unexpected results.
- Fixed: `LIKE` escape character JDBC escape (`{escape '<char>'}`) doesn't work (*JDBC-290*)
- Added: Support for a connect timeout using connection property `connectTimeout`. This property can be specified in the JDBC URL or `Properties` object or on the `DataSource`. If the `connectTimeout` property is not specified, the general `DriverManager` property `loginTimeout` is used. The value is the timeout in seconds. (*JDBC-295*)
  For the Java wire protocol the connect timeout will detect unreachable hosts. In the JNI implementation (native protocol) the connect timeout works as the DPB item `isc_dpb_connect_timeout` which only works after connecting to the server for the `op_accept` phase of the protocol. This means that – for the native protocol – the connect timeout will not detect unreachable hosts within the timeout. As that might be unexpected, an `SQLWarning` is added to the connection if the property is specified with the native protocol.
- As part of the connect timeout change, hostname handling (if the hostname is an IP-address) in the Java wire protocol was changed. This should not have an impact in recent Java versions, but on older Java versions (Java 5 up to update 5) this might result in a delay in connecting using an IP-address, if that address can't be reverse-resolved to a hostname. Workaround is to add an entry for that IP-address to the `/etc/hosts` or `%WINDIR%\System32\Drivers\etc\hosts` file.

Changes and fixes in Jaybird 2.2.1

The following has been changed or fixed in Jaybird 2.2.1:

- Fixed: `UnsatisfiedLinkError` in `libjaybird22(_x64).so undefined symbol: _ZTVN10__cxxabiv117__class_type_infoE` on Linux *(JDBC-259)*
- Added connection property `columnLabelForName` for backwards compatible behavior of

`ResultSetMetaData#getColumnName(int)` and compatibility with bug in `com.sun.rowset.CachedRowSetImpl` *(JDBC-260)*
Set property to `true` for backwards compatible behavior (`getColumnName()` returns the column label); don't set the property or set it to `false` for JDBC-compliant behavior (recommended).

- Fixed: `setString(column, null)` on "`? IS (NOT) NULL`" condition does not set parameter to `NULL` *(JDBC-264)*
- The `charSet` connection property now accepts all aliases of the supported Java character sets (eg instead of only `Cp1252` now `windows-1252` is also accepted) *(JDBC-267)*
- Fixed: values of `charSet` property are case-sensitive *(JDBC-268)*
- Fixed: setting a parameter as `NULL` with the native protocol does not work when Firebird describes the parameter as not nullable *(JDBC-271)*

Changes and fixes since Jaybird 2.2.0 beta 1

The following was changed or fixed after the release of Jaybird 2.2.0 beta 1:

- `ConcurrentModificationException` when closing connection obtained from `org.firebirdsql.ds.FBConnectionPoolDataSource` with statements open *(JDBC-250)*
- Memory leak when obtaining multiple connections for the same URL *(JDBC-249)*
- CPU spikes to 100% when using events and Firebird Server is stopped or unreachable *(JDBC-232)*
- Events do not work on Embedded *(JDBC-247)*
- Provide workaround for character set transliteration problems in database filenames and other connection properties *(JDBC-253)*; see also Support for Firebird 2.5.
- `FBBackupManager` does not allow 16kb page size for restore (*JDBC-255*)
- Log warning and add warning on `Connection` when no explicit connection character set is specified (*JDBC-257*)

Support for `getGeneratedKeys()`

Support was added for the `getGeneratedKeys()` functionality for `Statement` and `PreparedStatement`.

There are four distinct use-cases:

1. Methods accepting an `int` parameter with values of `Statement.NO_GENERATED_KEYS` and `Statement.RETURN_GENERATED_KEYS`

   When `NO_GENERATED_KEYS` is passed, the query will be executed as a normal query.

   When `RETURN_GENERATED_KEYS` is passed, the driver will add ***all*** columns of the table in ordinal position order (as in the (JDBC) metadata of the table). It is advisable to retrieve the values from the `getGeneratedKeys()` resultset by column name.

   We opted to include all columns as it is next to impossible to decide which columns are filled by a trigger or otherwise and only returning the primary key will be too limiting

2. Methods accepting an `int[]` parameter with column indexes.

   The values in the `int[]` parameter are the ordinal positions of the columns as specified in the (JDBC) metadata of the table. For a null or empty array the statement is processed as is.

Invalid ordinal positions are ignored and silently dropped (be aware: the JDBC specification is not entirely clear if this is valid behavior, so this might change in the future)

3. Methods accepting a `String[]` parameter with column names.

   The values in the `String[]` are the column names to be returned. The column names provided are processed as is and not checked for validity or the need of quoting. Providing non-existent or incorrectly (un)quoted columns will result in an exception when the statement is processed by Firebird.

   This method is the fastest as it does not retrieve metadata from the server.

4. Providing a query already containing a `RETURNING` clause. In this case all of the previous cases are ignored and the query is executed as is. It is possible to retrieve the resultset using `getGeneratedKeys()`.

This functionality will only be available if the ANTLR 3.4 runtime classes are on the classpath. Except for calling methods with `NO_GENERATED_KEYS`, absence of the ANTLR runtime will throw `FBDriverNotCapableException`.

This functionality should work for `INSERT` (from Firebird 2.0), and for `UPDATE`, `UPDATE OR INSERT` and `DELETE` (from Firebird 2.1).

Java 6 and JDBC 4.0 API support

Support was added for the following JDBC 4.0 features:

- Automatic driver loading: on Java 6 and later it is no longer necessary to use `Class.forName("org.firebirdsql.jdbc.FBDriver")` to load the driver
- Implementation of `java.sql.Wrapper` interface on various JDBC classes; in general it only unwraps to the specific implementation class (and superclasses) and implemented interfaces
- Support for chained exceptions (use `getNextException()` and `iterator()` to view other, related exceptions) and `getCause()` to retrieve the cause (deprecating similar `getInternalException()`)
- Support for `getClientInfo()` and `setClientInfo()` on `Connection`

Java 7 and JDBC 4.1 API support

Support was added for the following JDBC 4.1 features:

- try-with-resources[1]
- Statement closeOnCompletion

Other methods added by JDBC 4.1 will throw `FBDriverNotCapableException` (a subclass of `SQLFeatureNotSupportedException`).

Jaybird on Maven

Jaybird 2.2.2 is available on maven, with a separate artifact for each supported Java version.

Groupid: `org.firebirdsql.jdbc`, artifactid: `jaybird-jdkXX` (where XX is 15, 16 or 17).

Version: `2.2.2`

When deploying to a JavaEE environment, exclude the `javax.resource connector-api` dependency as this will be provided by the application server.

1 See http://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html

Native and Embedded (JNI) 64-bit Windows and Linux support

The JNI libraries for native and embedded support now also have a 64 bit version.

Support for Firebird 2.5

Added support for Firebird 2.5 Services API enhancements:

- The security database can be set
- Support for SET/DROP AUTO ADMIN
- Mapping for new role RDB$ADMIN in security database
- Added new Firebird 2.1 shutdown/online modes available in Firebird 2.5 via the Services API
- Support for NBackup via Services API in Firebird 2.5
- Support for Trace/Audit via Services API in Firebird 2.5

Since Firebird 2.5, Firebird supports full UTF-8 database filenames and other connection properties (Database Parameter Buffer values). Jaybird does not yet support these changes, but a workaround is available:

This workaround consists of two steps

1. Ensure your Java application is executed with the system property `file.encoding=UTF-8` (either because that is the default encoding for your OS, or by explicitly specifying this property on the commandline of your application using `-Dfile.encoding=UTF-8`)
2. Include property `utf8_filename=1` in the JDBC URL or (non-standard) properties of the datasource

This will only work if the Firebird server is version 2.5 or higher.

Improved support for OpenOffice / LibreOffice Base

The interpretation of the JDBC standard by Jaybird differs from the interpretation by OpenOffice / LibreOffice. To address some of the problems caused by these differences, Jaybird now provides a separate protocol for OpenOffice / LibreOffice.

When connecting from Base, use the protocol prefix `jdbc:firebirdsql:oo:`. Be aware that this is a variant of the pure Java wire protocol and not the native or embedded protocol.

Issues addressed by this protocol:

- ResultSets are not closed when a statements is finished (eg fully read ResultSet or when creating a new Statement in autoCommit mode)
- `DatabaseMetaData#getTablePrivileges(...)` reports privileges granted to `PUBLIC` and to the current role (as reported by `CURRENT_ROLE`) as being granted to the user *(after Jaybird 2.2.0 beta 1)*.

Other fixes and changes

- Replaced `mini-j2ee.jar` with `connector-api-1.5.jar`: make sure to remove the old `mini-j2ee.jar` from the classpath of your application.
- Dropped `jaybird-pool` jar from the distribution (all classes are include in the `jaybird` jar and the `jaybird-full` jar)

- `FBResultSetMetaData#getcolumnName(int)` will now return the original column name (if available) for compliance with the JDBC specification, `getColumnLabel(int)` will still return the alias (or the column name if no alias is defined). See Compatibility with com.sun.rowset.* for potential problems when using the reference implementation of `CachedRowSet`.
  Jaybird 2.2.1 introduced the connection property `columnLabelForName` which will revert to the old behavior when set to `true`. Be aware that the old behavior is not JDBC-compliant.
- `FBDatabaseMetaData` has been updated to include metadata columns defined by JDBC 3.0, 4.0 and 4.1. This also changes the position of `OWNER_NAME` column in the result set of `getTables(..)` as this column is Jaybird-specific and not defined in JDBC.
- `FBDatabaseMetaData#getIndexInfo(..)` now also returns expression indexes. The `COLUMN_NAME` column will contain the expression (if available).
- `FBDatabaseMetaData#getIndexInfo(..)` now correctly limits the returned indexes to unique indexes when parameter `unique` is set to `true`.
- The connection property `octetsAsBytes` can be used to identify fields with `CHARACTER SET OCTETS` as being `(VAR)BINARY` (in `ResultSetMetaData` only)
- The `getTime()`, `getDate()`, `getTimestamp()` methods which take a `Calendar` object now correctly handle conversions around Daylight Savings Time (DST) changes. Before, the time was first converted to the local JVM timezone, and then to the timezone of the provided `Calendar`, this could lose up to an hour in time. Now the time is converted directly to the timezone of the provided `Calendar`. *(JDBC-154)*

A full list of changes is available at:

Jaybird 2.2.2: http://tracker.firebirdsql.org/secure/ReleaseNote.jspa?projectId=10002&styleName=Text&version=10480

Jaybird 2.2.1: http://tracker.firebirdsql.org/secure/ReleaseNote.jspa?version=10474&styleName=Text&projectId=10002&Create=Create

Jaybird 2.2.0: http://tracker.firebirdsql.org/secure/ReleaseNote.jspa?version=10053&styleName=Text&projectId=10002&Create=Create

Compatibility changes

Jaybird 2.2 introduces some changes in compatibility and announces future breaking changes.

Java support

Java 5 support will be dropped for Jaybird 2.3 as Java 5 has been on End-Of-Life[2] status since October 2009.

Firebird support

Jaybird 2.2 supports Firebird 1.0 and higher, but is only tested with Firebird 2.1 and 2.5. For Jaybird 2.3 formal support for Firebird 1.0 and 1.5 will be dropped. In general this probably will not impact the use of the driver, but might have impact on the availability and use of metadata information. This also means that from Jaybird 2.3 bugs that only occur with Firebird 1.0 and 1.5 will not be fixed.

Important changes to Datasources

The `ConnectionPoolDataSource` and `XADataSource` implementations in `org.firebirdsql.pool` and `org.firebirdsql.pool.sun` contain several bugs with regard to pool and connection management when used by a JavaEE application server. The decision was made to write new implementations in the package `org.firebirdsql.ds`.

The following implementation classes have been deprecated and will be removed in Jaybird 2.3:

- `org.firebirdsql.pool.DriverConnectionPoolDataSource`
- `org.firebirdsql.pool.FBConnectionPoolDataSource`
- `org.firebirdsql.pool.sun.AppServerDataSource`
- `org.firebirdsql.pool.sun.AppServerXADataSource`
- `org.firebirdsql.jca.FBXADataSource`
- `org.firebirdsql.pool.SimpleDataSource`

Their replacement classes are:

- `org.firebirdsql.ds.FBConnectionPoolDataSource`
- `org.firebirdsql.ds.FBXADataSource`
- `org.firebirdsql.pool.FBSimpleDataSource` (a normal `DataSource`)

We strongly urge you to switch to these new implementations if you are using these classes in an application server. The bugs are described in JDBC-86, JDBC-93, JDBC-131 and JDBC-144.

The deprecated classes can still be used with the `DataSource` implementations `WrappingDataSource` as the identified bugs do not occur with this implementation, but we advise you to switch to `FBSimpleDataSource`. If you require a standalone connection pool (outside an application server) or statement pooling, please consider using a third-party connection pool like C3P0, DBCP or BoneCP.

The new `ConnectionPoolDataSource` and `XADataSource` implementations only provide the basic functionality specified in the JDBC specifications and do **not** provide any pooling itself. The `ConnectionPoolDataSource` and `XADataSource` are intended to be **used by** connection pools (as provided by application servers) and should not be connection pools themselves.

2 See http://www.oracle.com/technetwork/java/eol-135779.html

Future changes to Jaybird

The next versions of Jaybird will include some – potentially – breaking changes. We advise to check your code if you will be affected by these changes and prepare for these changes if possible.

Removal of deprecated classes, packages and (interface) methods

As announced above, the `ConnectionPoolDataSource` implementations in `org.firebirdsql.pool` and `org.firebirdsql.jca` will be removed in Jaybird 2.3. This may included removal of additional classes and interfaces from these packages (or the entire package).

The following (deprecated) classes will also be removed:

- `org.firebirdsql.jdbc.FBWrappingDataSource` (old deprecated class subclassing `org.firebirdsql.pool.FBWrappingDataSource`), only included in `jaybird-full` jar

Furthermore the following interfaces will be removed as they are no longer needed:

- `FirebirdSavepoint` (identical to `java.sql.Savepoint`)

The following interfaces will have some of the methods removed:

- `FirebirdConnection`
  - `setFirebirdSavepoint()` replace with `Connection#setSavepoint()`
  - `setFirebirdSavepoint(String name)` replace with `Connection#setSavepoint(String name)`
  - `rollback(FirebirdSavepoint savepoint)` replace with `Connection#rollback(Savepoint savepoint)`
  - `releaseSavepoint(FirebirdSavepoint savepoint)` replace with `Connection#releaseSavepoint(Savepoint savepoint)`

If you are still using these interfaces or methods, please change your code to use the JDBC interface or method instead.

Handling `(VAR)CHAR CHARACTER SET OCTETS` as `(VAR)BINARY` type

From Jaybird 2.3 on `(VAR)CHAR CHARACTER SET OCTETS` will be considered to be of `java.sql.Types` type `(VAR)BINARY`. This should not impact normal use of methods like `get/setString()`, but will impact the metadata and the type of object returned by `getObject()` (a byte array instead of a String).

Handling connections without explicit connection character set

When no connection character set has been specified (properties `lc_ctype` or `encoding` with Firebird character set, or `charSet` or `localEncoding` with Java character set), Jaybird will currently use the `NONE` character set. This means that the Firebird server will return the bytes for `(VAR)CHAR` columns as they are stored, while Jaybird will convert between bytes and Strings using the local platform encoding.

This default has the potential of corrupting data when switching platforms or using the same database with different local encoding, or for transliteration errors when the database character set does not accept some byte combinations. We are currently discussing changing this behavior (see JDBC-257). We haven't decided on the exact changes yet, but most likely the next version of Jaybird will refuse to connect without an explicit connection character set. For the time being, Jaybird will log a warning and add a warning on the `Connection` when no explicit character set was specified.

Review your use of the connection character sets and change it if you are not specifying it explicitly. Be aware that changing this may require you to fix the data as it is currently stored in your database if your database character set does not match the local platform encoding of your Java application. If you are sure that `NONE` is the correct character set for you, specify it explicitly in your connection string or connection properties.

Distribution package

The following file groups can be found in distribution package:

- `jaybird-2.2.2.jar` – archive containing JCA/JDBC driver, implementation of connection pooling and statement pooling interfaces, and JMX management class. It requires JCA 1.5
- `jaybird-full-2.2.2.jar` – merge of `jaybird-2.2.2.jar` and `connector-api-1.5.jar`. This archive can be used for standalone Jaybird deployments
- `jaybird-2.2.2-sources.jar` – archive containing the sources of Jaybird (specific to this JDK version); for including Jaybird sources in your IDE
- `lib/connector-api-1.5.jar` – archive containing JCA 1.5 classes (required dependency)
- `lib/antlr-runtime-3.4.jar` – archive containing ANTLR runtime classes, required for generated keys functionality (optional dependency)
- `lib/log4j-core.jar` – archive containing core Log4J classes that provide logging (optional dependency)

Jaybird has compile-time and run-time dependencies to the JCA 1.5 classes. Additionally, if Log4J classes are found in the class path, it is possible to enable extensive logging inside the driver. If the ANTLR runtime classes are absent, the generated keys functionality will not be available.

Native dependencies (required only for Type 2 and Embedded):

- `jaybird22.dll` – Windows 32-bit
- `jaybird22_x64.dll` – Windows 64-bit
- `libjaybird22.so` – Linux 32-bit (x86)
- `libjaybird22_x64.so` – Linux 64-bit (AMD/Intel 64)

The Windows DLLs have been built with Microsoft Visual Studio 2010 SP1. To use the native or embedded driver, you will need to install the Microsoft Visual C++ 2010 SP 1 redistributable available at:

- x86: http://www.microsoft.com/download/en/details.aspx?id=8328
- x64: http://www.microsoft.com/download/en/details.aspx?id=13523

License

Jaybird JCA/JDBC driver is distributed under the GNU Lesser General Public License (LGPL). Text of the license can be obtained from http://www.gnu.org/copyleft/lesser.html.

Using Jaybird (by importing Jaybird's public interfaces in your Java code), and extending Jaybird by subclassing or implementation of an extension interface (but not abstract or concrete class) is considered by the authors of Jaybird to be dynamic linking. Hence our interpretation of the LGPL is that the use of the unmodified Jaybird source does not affect the license of your application code.

Even more, all extension interfaces to which application might want to link are released under dual LGPL/modified BSD license. Latter is basically “AS IS” license that allows any kind of use of that source code. Jaybird should be viewed as an implementation of that interfaces and LGPL section for dynamic linking is applicable in this case.

Source Code

The distribution package contains the normal sources in `jaybird-2.2.1-sources.jar`; this file does not include the sources of the tests, nor the sourcecode for other JDK-versions.

Full source code, including tests and build files, can be obtained from the Subversion repository at SourceForge.net. The repository URL is
svn://svn.code.sf.net/p/firebird/code/client-java

Alternatively source code can be viewed online at

http://sourceforge.net/p/firebird/code/

Documentation and Support

Where to get more information on Jaybird

The most detailed information can be found in the Jaybird Frequently Asked Questions (FAQ). The FAQ is included in the distribution, and is available on-line in several places.

JaybirdWiki is available at http://Jaybirdwiki.firebirdsql.org *(note: this resource is out of date and will be discontinued in the near future)*

Jaybird 2.1 Programmers Manual:
http://www.firebirdsql.org/file/documentation/drivers_documentation/Jaybird_2_1_JDBC_driver_manual.pdf

Where to get help

The best place to start is the FAQ. Many details for using Jaybird with various programs are located there. Below are some links to useful web sites.

- The http://groups.yahoo.com/group/Firebird-Java and corresponding mailing-list Firebird-Java@yahoogroups.com
- The code for Firebird and this driver are on http://sourceforge.net/projects/firebird
- The Firebird project home page http://www.firebirdsql.org

Contributing

There are several ways you can contribute to Jaybird or Firebird in general:

- Participate on the mailinglists (see http://www.firebirdsql.org/en/mailing-lists/)
- Report bugs or submit patches on the tracker (see below)
- Become a developer (for Jaybird contact us on Firebird-Java, for Firebird in general, use the Firebird-devel mailing-list)
- Become a paying member or sponsor of the Firebird Foundation (see http://www.firebirdsql.org/en/firebird-foundation/)

See also http://www.firebirdsql.org/en/consider-your-contribution/

Reporting Bugs

The developers follow the Firebird-Java@yahoogroups.com list. Join the list and post information about suspected bugs. List members may be able to help out to determine if it is an actual bug, provide a workaround and get you going again, whereas bug fixes might take awhile.

If you are sure that this is a bug you can report it in the Firebird bug tracker, project "Java Client (Jaybird)" at http://tracker.firebirdsql.org/browse/JDBC

When reporting bugs, please provide a minimal, but complete reproduction, including databases and sourcecode to reproduce the problem. Patches to fix bugs are also appreciated. Make sure the patch is against a recent trunk version of the code.

Corrections/Additions To Release Notes

Please send corrections, suggestions, or additions to these Release Notes to to the mailing list at Firebird-Java@yahoogroups.com.

JDBC URL Format

Jaybird provides different JDBC URLs for different usage scenarios:

Pure Java

`jdbc:firebirdsql://host[:port]/<database>`

Default URL, will connect to the database using Type 4 JDBC driver using the Java implementation of the Firebird wire-protocol. Best suited for client-server applications with dedicated database server. Port can be omitted (default value is 3050), host name must be present.

The `<database>` part should be replaced with the database alias or the path to the database. In general it is advisable to uses database aliases instead of the path the file.

On Linux the root `/` should be included in the path. A database located on `/opt/firebird/db.fdb` should use the URL below (note the double slash after port!).
`jdbc:firebirdsql://host:port//opt/firebird/db.fdb`

Deprecated but still available alternative URL:

`jdbc:firebirdsql:host[/port]:<database>`

Using Firebird client library

`jdbc:firebirdsql:native:host[/port]:<database>`

Type 2 driver, will connect to the database using client library (`fbclient.dll` on Windows, and `libfbclient.so` on Linux). Requires correct installation of the client library.

`jdbc:firebirdsql:local:<database>`

Type 2 driver in local mode. Uses client library as in previous case, however will not use socket communication, but rather access database directly. Requires correct installation of the client library.

Embedded Server

`jdbc:firebirdsql:embedded:<database>`

Similar to the Firebird client library, however `fbembed.dll` on Windows and `libfbembed.so` on Linux are used. Requires correctly installed and configured Firebird embedded library.

Using Type 2 and Embedded Server driver

Jaybird 2.2 provides a Type 2 JDBC driver that uses the native client library to connect to the databases. Additionally Jaybird 2.2 can use the embedded version of Firebird so Java applications do not require a separate server setup.

However the Type 2 driver has its limitations:

Due to multi-threading issues in the Firebird client library as well as in the embedded server version, it is not possible to access a single connection from different threads simultaneously. When using the client library only one thread is allowed to access a connection at a time. Access to different connections from different threads is however allowed. Client library in local mode and embedded server library on Linux do not allow multithreaded access to the library. Jaybird provides necessary synchronization in Java code, however the mutex is local to the classloader that loaded the Jaybird driver.

Care should be taken when deploying applications in web or application servers: put jar files in the main library directory of the web and/or application server, not in the library directory of the web or enterprise application (WEB-INF/lib directory or in the .EAR file).

Configuring Type 2 JDBC driver

The Type 2 JDBC driver requires the Jaybird JNI library to be installed and available to the Java Virtual Machine. Precompiled binaries for Windows and Linux platforms are distributed with Jaybird.

Please note that Jaybird 2.2 provides an update to the JNI libraries to support new features. It is not compatible with the JNI library for Jaybird 2.1 or earlier.

- `jaybird22.dll / jaybird22_x64.dll` is a precompiled binary for the Windows platform. It was successfully tested with Windows XP and Windows 7, but there should be no issues in other Windows OS versions (as long as the MS Visual C++ 2010 SP1 distributable is available).
  The library should be copied into a directory in the `PATH` environment variable, or be made available to the JVM using the `java.library.path` system property.
- `libjaybird22.so / libjaybird22_x64.so` is a precompiled binary for the Linux platform (AMD/Intel). It must be available via the `LD_LIBRARY_PATH` environment variable, or be made available to the JVM using the `java.library.path` system property.
  Dependent libraries (`libfbclient.so` or `libfbembed.so`) need to be on the `LD_LIBRARY_PATH`. The `java.library.path` is ignored for these libraries as they are loaded from the JNI library, and not from Java.
  Some Firebird distributions will not create `libfbclient.so` (but only `libfbclient.so.2` and `.so.2.5`), you will need to add a symlink with the name as expected by Jaybird.
- Other platforms can easily compile the JNI library by checking out the Jaybird sources from the CVS and using `./build.sh compile-native` command in the directory with checked out sources.

After making Jaybird JNI library available to the JVM, the application has to tell driver to start using this by either specifying TYPE2 or LOCAL type in the connection pool or data source properties or using appropriate JDBC URL when connecting via `java.sql.DriverManager`.

Configuring Embedded Server JDBC driver

The Embedded Server JDBC driver uses the same JNI library and configuration steps for the Type 2 JDBC driver.

There is however one issue related to the algorithm of Firebird Embedded Server installation directory resolution. Firebird server uses pluggable architecture for internationalization. By default server loads `fbintl.dll` or `libfbintl.so` library that contains various character encodings and collation orders. This library is expected to be installed in the `intl/` subdirectory of the server installation. The algorithm of directory resolution is the following:

1. `FIREBIRD` environment variable.
2. `RootDirectory` parameter in the `firebird.conf` file.
3. The directory where server binary is located.

When Embedded Server is used from Java and no `FIREBIRD` environment variable is specified, it tries to find `firebird.conf` in the directory where application binary is located. In our case application binary is JVM and therefore Embedded Server tries to find its configuration file in the `bin/` directory of the JDK or JRE installation. Same happens to the last item of the list. In most cases this is not desired behavior.

Therefore, if the application uses character encodings, UDFs or wants to fine-tune server behavior through the configuration file, the `FIREBIRD` environment variable must be specified and point to the installation directory of the Embedded Server, e.g. current working directory.

Support for multiple JNI libraries

Upto Jaybird 2.0 only one client library could be loaded in a single JVM. That could be either an embedded Firebird library (`fbembed.dll`/`libfbembed.so`), or Firebird client library (`fbclient.dll`/`libfbclient.so`). This could lead to problems, For example, if embedded Firebird was used first, the JDBC driver would access the database file directly instead of using the local IPC protocol if only the path to the database was specified. It was not possible to change this without restarting the JVM.

Since Jaybird 2.1, Jaybird is able to correctly load arbitrary number of shared libraries that implement the ISC API and forward the requests correctly depending on the type of the driver being used.

Usage and Reference Manual

Events

Events is one of the unique features in the Firebird RDBMS and allows asynchronous notification of the applications about named events that happen in the database. Information on this feature can found in the free IB 6.0 documentation set as well as in The Firebird Book by Helen Borrie.

The interfaces and classes for the event support can be found in `org.firebirdsql.event` package, which includes:

- `EventManager` interface to register for the synchronous and asynchronous notification about the events in the database;
- `EventListener` interface which has to be implemented by the application that wants to participate in the asynchronous notification;
- `DatabaseEvent` interface which represents the object that will be passed to the `EventListener` notification method;
- Implementation of the above interfaces: `FBEventManager` and `FBDatabaseEvent`.

Please note, that each instance of `FBEventManager` will open a new socket connection to the Firebird server on the port specified by Firebird.

Similar to other JDBC extensions in Jaybird, the interfaces are released under the modified BSD license, the implementation of the code is released under LGPL license.

Default holdable result sets (closed ResultSet in auto-commit mode)

This connection property allows to create holdable result sets by default. This is needed as a workaround for the applications that do not follow JDBC specification in regard to the auto-commit mode.

Specifically, such applications open a result set and, while traversing it, execute other statements using the same connection. According to JDBC specification the result set has to be closed if another statement is executed using the same connection in auto-commit mode. Among others the OpenOffice/LibreOffice Base users have problems with the JDBC compatibility in Jaybird.

The property is called:

- `defaultResultSetHoldable` as connection property for JDBC URL or for `java.sql.DriverManager` class and no or empty value should be assigned to it; it has an alias `defaultHoldable` to simplify the typing;
- `isc_dpb_result_set_holdable` as a DPB member;
- `FirebirdConnectionProperties` interface methods `isDefaultResultSetHoldable()` and `setDefaultResultSetHoldable(boolean)`

Note, the price for using this feature is that each holdable result set will be fully cached in memory. The memory occupied by it will be released when the statement that produced the result set is either closed or re-executed.

Updatable result sets

Jaybird provides support for updatable result sets. This feature allows a Java application to update the current record using the `update`*`XXX`* methods of `java.sql.ResultSet` interface. Updates are performed within the current transaction using a best row identifier in `WHERE` clause. This sets the following limitation on the result set “updatability”:

- the `SELECT` references a single table;
- all columns not referenced in `SELECT` permit `NULL`s (otherwise `INSERT`s will fail);
- the `SELECT` statement does not contain `DISTINCT` predicate, aggregate functions, joined tables or stored procedures;
- the `SELECT` statement references all columns from the table primary key definition or an `RDB$DB_KEY` column.

Firebird management interfaces

Jaybird provides full support of the Firebird Services API that allows Java applications to perform various server management tasks:

- database backup/restore on remote server; it is possible to performs metadata-only backups, switch the garbage collection during backup off, restore databases with no validity constraints or active indices, etc.
- database maintenance, e.g. database shutdown, sweep, changing the forced writes settings, changing SQL dialect of the database, shadow management, etc.
- retrieving database statistics including header page statistics, system table statistics, data page statistics and index statistics.
- user management, including adding, modifying, and deleting user accounts.

Jaybird JDBC extensions

Jaybird provides extensions to some JDBC interfaces. JDBC extension interface classes are released under modified BSD license, on “AS IS” and “do what you want” basis, this should make linking to these classes safe from the legal point of view. All classes belong to `org.firebirdsql.jdbc.*` package. The table below shows all JDBC extensions present in Jaybird with a driver version in which the extension was introduced.

JDBC extensions			
Interface	**Since**	**Method name**	**Description**
`FirebirdDriver`	2.0	`newConnectionProperties()`	Create new instance of `FirebirdConnectionProperties` interface that can be used to set connection properties programmatically.
		`connect(FirebirdConnectionProperties)`	Connect to the Firebird database using the specified connection properties.
`FirebirdConnectionProperties`	2.0	see JDBC connection properties section for more details.	
`FirebirdConnection`	1.5	`createBlob()`	Create new BLOB in the database. Later this BLOB can be passed as a parameter into `PreparedStatement` or `CallableStatement`.
	1.5	`getIscEncoding()`	Get connection character

JDBC extensions			
			encoding.

	2.0	`getTransactionParameters(`
		`  int isolationLevel`
		`)`
			Get the TPB parameters for the specified transaction isolation level.

	2.0	`createTransactionParameterBuffer()`	Create an empty transaction parameter buffer.

	2.0	`setTransactionParameters(`
		`  int isolationLevel,`
		
		`TransactionParameterBuffer tpb`
		`)`
			Set TPB parameters for the specified transaction isolation level. The newly specified mapping is valid for the whole connection lifetime.

	2.0	`setTransactionParameters(`
		
		`TransactionParameterBuffer tpb`
		`)`
			meters are effective until the transaction isolation is changed.

`FirebirdDatabaseMetaData`		`getProcedureSourceCode(String)`	Get source code for the specified stored procedure name.
		`getTriggerSourceCode(String)`	Get source code for the specified trigger name.
		`getViewSourceCode(String)`	Get source code for the specified view name.

`FirebirdStatement`	1.5	`getInsertedRowsCount()`
		`getUpdatedRowsCount()`
		`getDeletedRowsCount()`
			Extension that allows to get more precise information about outcome of some statement.

	1.5	`hasOpenResultSet()`	Check if this statement has open result set. Correctly works only when auto-commit is disabled. Check method documentation for details.
	1.5	`getCurrentResultSet()`	Get current result set. Behaviour of this method is similar to the behavior of the `Statement.getResultSet()`, except that this method can be called as much as you like.
	1.5	`isValid()`	Check if this statement is still valid. Statement might be invalidated when connection is automatically recycled between transactions due to some irrecoverable error.
	2.0	`getLastExecutionPlan()`	Get execution plan for the last

JDBC extensions			
			executed statement.
`FirebirdPreparedStatement`	2.0	`getExecutionPlan()`	Get the execution plan of this prepared statement.
	2.0	`getStatementType()`	Get the statement type of this prepared statement.

`FirebirdCallableStatement`	1.5	`setSelectableProcedure(`
		`  boolean selectable`
		`)`
			Mark this callable statement as a call of the selectable procedure. By default callable statement uses `EXECUTE PROCEDURE` SQL statement to invoke stored procedures that return single row of output parameters or a result set. In former case it retrieves only the first row of the result set.

`FirebirdResultSet`	2.0	`getExecutionPlan()`	Get execution plan for this result set.
`FirebirdBlob`	1.5	`detach()`	Method “detaches” a BLOB object from the underlying result set. Lifetime of “detached” BLOB is limited by the lifetime of the connection.
	1.5	`isSegmented()`	Check if this BLOB is segmented. Seek operation is not defined for the segmented BLOBs.

	1.5	`setBinaryStream(`
		`  long position`
		`)`
			Opens an output stream at the specified position, allows modifying BLOB content. Due to server limitations only position 0 is supported.

`FirebirdBlob.BlobInputStream`	1.5	`getBlob()`	Get corresponding BLOB instance.
	1.5	`seek(int position)`	Change the position from which BLOB content will be read, works only for stream BLOBs.
`FirebirdSavepoint`[3]	2.0	interface is equivalent to the java.sql.Savepoint interface introduced in JDBC 3.0 specification, however allows using Firebird savepoints also in JDBC 2.0 (JDK 1.3.x) applications.	

3 To be removed in Jaybird 2.3

JDBC connection properties

The table below lists the properties for the connections that are obtained from this data source. Commonly used parameters have the corresponding getter and setter methods, the rest of the Database Parameters Block parameters can be set using `setNonStandardProperty` setter method.

Property	Getter	Setter	Description
`database`	+	+	**(deprecated)** Path to the database in the format `[host/port:]<database>` This property is not specified in the JDBC standard. Use the the standard defined `serverName`, `portNumber` and `databaseName` instead
`serverName`	+	+	Hostname or IP address of the Firebird server
`portNumber`	+	+	Portnumber of the Firebird server
`databaseName`	+	+	Database alias or full-path
`type`	+	+	Type of the driver to use. Possible values are: • `PURE_JAVA` or `TYPE4` for type 4 JDBC driver • `NATIVE` or `TYPE2` for type 2 JDBC driver • `EMBEDDED` for using embedded version of the Firebird.
`blobBufferSize`	+	+	Size of the buffer used to transfer BLOB content. Maximum value is 64k-1.
`socketBufferSize`	+	+	Size of the socket buffer. Needed on some Linux machines to fix performance degradation.
`buffersNumber`	+	+	Number of cache buffers (in database pages) that will be allocated for the connection. Makes sense for ClassicServer only.
`charSet`	+	+	Character set for the connection. Similar to `encoding` property, but accepts Java names instead of Firebird ones.
`encoding`	+	+	Character encoding for the connection. See Firebird documentation for more information.
`useTranslation`	+	+	Path to the properties file containing character translation map.

Property	Getter	Setter	Description
password	+	+	Corresponding password.
roleName	+	+	SQL role to use.
userName	+	+	Name of the user that will be used by default.
useStreamBlobs	+	+	Boolean flag tells driver whether stream BLOBs should be created by the driver, by default “false”. Stream BLOBs allow “seek” operation to be called, however due to a bug in gbak utility they are disabled by default.
useStandardUdf	+	+	Boolean flag tells driver to assume that standard UDFs are defined in the database. This extends the set of functions available via escaped function calls. This does not affect non-escaped use of functions.
defaultResultSetHoldable	+	+	Boolean flag tells driver to construct the default result set to be holdable. This prevents it from closing in auto-commit mode if another statement is executed over the same connection.
tpbMapping	+	+	TPB mapping for different transaction isolation modes.
defaultIsolation	+	+	Default transaction isolation level. All newly created connections will have this isolation level. One of: • TRANSACTION_READ_COMMITTED • TRANSACTION_REPEATABLE_READ • TRANSACTION_SERIALIZABLE
defaultTransactionIsolation	+	+	Integer value from java.sql.Connection interface corresponding to the transaction isolation level specified in isolation property.
nonStandardProperty	getNonStandardProperty(String)	+ setNonStandardProperty(String) setNonStandardProperty(String, String)	Allows to set any valid connection property that does not have corresponding setter method. Two setters are available: setNonStandardProperty(String) method takes only one parameter in form “propertyName[=propertyValue]”, this allows setting non-standard

Property	Getter	Setter	Description
			parameters using configuration files. `setNonStandardProperty(String, String)` takes property name as first parameter, and its value as the second parameter.
`connectTimeout`	+	+	The connect timeout in seconds. For the Java wire protocol detects unreachable hosts, for JNI (native protocol) only defines a timeout during the `op_accept` phase after connecting to the server.

JDBC Compatibility

The Jaybird driver is not officially JDBC-compliant as the certification procedure is too expensive. The following lists some of the differences between JDBC specification and Jaybird implementation. This list is not exhaustive.

JDBC deviations and unimplemented features

The following optional features and the methods for their support are not implemented:

- `java.sql.Array` data type is not (yet) supported
- `java.sql.Blob` does not implement following methods:
  - `position(Blob, long)` and `position(byte[], long)`; Firebird does not provide any server-side optimization for these calls, client application must fetch complete BLOB content from the server to do pattern search.
  - `truncate(long)`; Firebird does not provide such functionality on the server side, application must fetch old BLOB from the server and pump old content into a newly created BLOB.
- `java.sql.Connection`
  - `getCatalog()` and `setCatalog(String)` are not supported by Firebird server
  - `getTypeMap()` and `setTypeMap(Map)` are not supported
- `java.sql.Ref` data type is not supported by Firebird server
- `java.sql.SQLData` data type is not supported by Firebird server
- `java.sql.SQLInput` is not supported
- `java.sql.SQLOutput` is not supported
- `java.sql.SQLXML` is not supported
- `java.sql.RowId` is not supported
- `java.sql.NClob` is not supported
- `java.sql.Statement`
  - `cancel()` is implemented, but not fully supported by Jaybird
- `java.sql.Struct` data type is not supported by server.

The following methods are implemented, but deviate from the specification:

- `java.sql.Statement`
  - `get/setMaxFieldSize` does nothing, Firebird server does not support this feature.
  - `get/setQueryTimeout` does nothing, Firebird server does not support this feature.
- `java.sql.PreparedStatement`
  - `setObject(int index, Object object, int type)` Target SQL type is determined from the class of the passed object and corresponding parameter is ignored.
  - `setObject(int index, Object object, int type, int scale)` Same as above, type and scale are ignored.
- `java.sql.ResultSetMetaData`

- `isReadOnly()` always returns false
- `isWritable()` always returns true
- `isDefinitivelyWritable()` always returns true

Jaybird Specifics

Jaybird has some implementation-specific issues that should be considered during development.

Result sets

Jaybird behaves differently not only when different result set types are used but also whether the connection is in auto-commit mode or not.

- `ResultSet.TYPE_FORWARD_ONLY` result sets when used in auto-commit mode are completely cached on the client before the execution of the query is finished. This leads to the increased time needed to execute statement, however the result set navigation happens almost instantly. When auto-commit mode is switched off, only part of the result set specified by the fetch size is cached on the client.
- `ResultSet.TYPE_SCROLL_INSENSITIVE` result sets are always cached on the client. The reason is quite simple – the Firebird API does not provide scrollable cursor support, navigation is possible only in one direction.
- `ResultSet.HOLD_CURSORS_OVER_COMMIT` holdability is supported in Jaybird only for result sets of type `ResultSet.TYPE_SCROLL_INSENSITIVE`. For other result set types driver will throw an exception.

Using java.sql.ParameterMetaData with Callable Statements

This interface can be used only to obtain information about the IN parameters. Also it is not allowed to call the `PreparedStatement.getParameterMetaData` method before all of the OUT parameters are registered. Otherwise the corresponding method of `CallableStatement` throws an `SQLException`, because the driver tries to prepare the procedure call with incorrect number of parameters.

Using ResultSet.getCharacterStream with BLOB fields

Jaybird JDBC driver always uses connection encoding when converting array of bytes into character stream. The BLOB SUB_TYPE 1 fields allow setting the character encoding for the field. However when the contents of the field is sent to the client, it is not converted according to the character set translation rules in Firebird, but is sent “as is”. When such fields are accessed from a Java application via Jaybird and character set of the connection does not match the character encoding of the field, conversion errors might happen. Therefore it is recommended to convert such fields in the application using the appropriate encoding.

Heuristic transaction completion support

Current JCA implementation does not support `XAResource.forget(Xid)`. It might be important in cases where a distributed transaction - that was at some time in-limbo - was either committed or rolled back by the database administrator. Such transactions appear to Jaybird as successfully completed, however XA specification requires resource manager to “remember” such transaction until the `XAResource.forget(Xid)` is called.

Compatibility with `com.sun.rowset.*`

The reference implementation of `javax.sql.rowset` included with Java in package `com.sun.rowset` does not correctly look up columns by name as it ignores column aliases and only allows look up by the original column name[4] (this specifically applies to

4 See JDBC-162 and http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=7046875 for details

`com.sun.rowset.CachedRowSetImpl`).

We advise you to either only access columns by their index or use an implementation which correctly uses the column label for column lookup (which is either the alias or the original column name if no alias was defined).

Jaybird 2.2.1 introduced the connection property `columnLabelForName` for backwards compatible behavior of `ResultSetMetaData#getColumnName(int)`. Set property to `true` for backwards compatible behavior (`getColumnName()` returns the column label); don't set the property or set it to `false` for JDBC-compliant behavior (recommended).

Connection pooling with Jaybird

As described in Important changes to Datasources, the `ConnectionPoolDataSource` implementations in `org.firebirdsql.pool` contain some serious issues. The connection pool capability which depends on these classes will be removed in Jaybird 2.3.

This change leaves only the `ConnectionPoolDataSource` implementations in `org.firebirdsql.ds` (for use by application server connection pools). There are no plans to reintroduce a new standalone connection pooling capability. We probably will migrate some of the features like statement pooling to the normal JDBC driver.

If you require standalone connection pooling, or use an application server which has no built-in connectionpool, please consider using a third-party connection pool like C3P0, DBCP or BoneCP.

Description of deprecated `org.firebirdsql.pool` classes

> **WARNING: This section provides information on deprecated classes,**
>
> See Important changes to Datasources

Connection pooling provides effective way to handle physical database connections. It is believed that establishing new connection to the database takes some noticeable amount or time and in order to speed things up one has to reuse connections as much as possible. While this is true for some software and for old versions of Firebird database engine, establishing connection is hardly noticeable with Firebird 1.0.3 and Firebird 1.5. So why is connection pooling needed?

There are few reasons for this. Each good connection pool provides a possibility to limit number of physical connections established with the database server. This is an effective measure to localize connection leaks. Any application cannot open more physical connections to the database than allowed by connection pool. Good pools also provide some hints where connection leak occurred. Another big advantage of connection pool is that it becomes a central place where connections are obtained, thus simplifying system configuration. However, main advantage of good connection pool comes from the fact that in addition to connection pooling, it can pool also prepared statement. Tests executed using AS3AP benchmark suite show that prepared statement pooling might increase speed of the application by 100% keeping source code clean and understandable.

Usage scenario

When some statement is used more than one time, it makes sense to use prepared statement. It will be compiled by the server only once, but reused many times. It provides significant speedup when some statement is executed in a loop. But what if some prepared statement will be used during lifetime of some object? Should we prepare it in object's constructor and link object lifetime to JDBC connection lifetime or should we prepare statement each time it is needed? All such cases make handling of the prepared statements hard, they pollute application's code with irrelevant details.

Connection and statement pooling remove such details from application's code. How would the code in this case look like? Here's the example

Example 1. Typical JDBC code with statement pooling

```
001  ...
002  Connection connection = dataSource.getConnection();
003  try {
004      PreparedStatement ps = connection.prepareStatement(
005          "SELECT * FROM test_table WHERE id = ?");
006      try {
007          ps.setInt(1, id);
008          ResultSet rs = ps.executeQuery();
009          while (rs.next()) {
010              // do something here
011          }
012      } finally {
013          ps.close();
014      }
015  } finally {
016      connection.close();
017  }
018  ...
```

Lines 001-018 show typical code when prepared statement pooling is used. Application obtains JDBC connection from the data source (instance of `javax.sql.DataSource` interface), prepares some SQL statement as if it is used for the first time, sets parameters, and executes the query. Lines 012 and 015 ensure that statement and connection will be released under any circumstances. Where do we benefit from the statement pooling? Call to prepare a statement in lines 004-005 is intercepted by the pool, which checks if there's a free prepared statement for the specified SQL query. If no such statement is found it prepares a new one. In line 013 prepared statement is not closed, but returned to the pool, where it waits for the next call. Same happens to the connection object that is returned to the pool in line 016.

Connection Pool Classes (deprecated)

Jaybird connection pooling classes belong to `org.firebirdsql.pool.*` package.

Description of some connection pool classes.

`AbstractConnectionPool`	Base class for all connection pools. Can be used for implementing custom pools, not necessarily for JDBC connections.
`BasicAbstractConnectionPool`	Subclass of `AbstractConnectionPool`, implements `javax.sql.ConnectionPoolDataSource` interface. Also provides some basic properties (minimum and maximum number of connections, blocking and idle timeout, etc) and code to handle JNDI-related issues.
`DriverConnectionPoolDataSource`	Implementation of `javax.sql.ConnectionPoolDataSource` for arbitrary JDBC drivers, uses `java.sql.DriverManager` to obtain connections, can be used as JNDI object factory.
`FBConnectionPoolDataSource`	Jaybird specific implementation of `javax.sql.ConnectionPoolDataSource` and `javax.sql.XADataSource` interfaces, can be used as JNDI object factory.
`FBSimpleDataSource`	Implementation of `javax.sql.DataSource` interface, no connection and statement pooling is available,

Description of some connection pool classes.	
	connections are physically opened in `getConnection()` method and physically closed in their `close()` method.
`FBWrappingDataSource`	Implementation of `javax.sql.DataSource` interface that uses `FBConnectionPoolDataSource` to allocate connections. This class defines some additional properties that affect allocated connections. Can be used as JNDI object factory.
`SimpleDataSource`	Implementation of `javax.sql.DataSource` interface that uses `javax.sql.ConnectionPoolDataSource` to allocate physical connections.

org.firebirdsql.pool.FBConnectionPoolDataSource (deprecated)

This class is a corner stone of connection and statement pooling in Jaybird. It can be instantiated within the application as well as it can be made accessible to other applications via JNDI. Class implements both `java.io.Serializable` and `javax.naming.Referenceable` interfaces, which allows using it in a wide range of web and application servers.

Class implements both `javax.sql.ConnectionPoolDataSource` and `javax.sql.XADataSource` interfaces. Pooled connections returned by this class implement `javax.sql.PooledConnection` and `javax.sql.XAConnection` interfaces and can participate in distributed JTA transactions.

Class provides following configuration properties:

Standard JDBC Properties

This group contains properties defined in the JDBC specification and should be standard to all connection pools.

Property	Getter	Setter	Description
`maxIdleTime`	+	+	Maximum time in milliseconds after which idle connection in the pool is closed.
`maxPoolSize`	+	+	Maximum number of open physical connections.
`minPoolSize`	+	+	Minimum number of open physical connections. If value is greater than 0, corresponding number of connections will be opened when first connection is obtained.
`maxStatements`	+	+	Maximum size of prepared statement pool. If 0, statement pooling is switched off. When application requests more statements than can be kept in pool, Jaybird will allow creating that statements, however closing them would not return them back to the pool, but rather immediately release the resources.

Pool Properties

This group of properties are specific to the Jaybird implementation of the connection pooling classes.

Property	Getter	Setter	Description
`blockingTimeout`	+	+	Maximum time in milliseconds during which application can be blocked waiting for a connection from the pool. If no free connection can be obtained, exception is thrown.
`retryInterval`	+	+	Period in which pool will try to obtain new connection while blocking the application.
`pooling`	+	+	Allows to switch connection pooling off.
`statementPooling`	+	+	Allows to switch statement pooling off.
`pingStatement`	+	+	Statement that will be used to "ping" JDBC connection, in other words, to check if it is still alive. This statement must always succeed.
`pingInterval`	+	+	Time during which connection is believed to be valid in any case. Pool "pings" connection before giving it to the application only if more than specified amount of time passed since last "ping".

Runtime Pool Properties

This group contains read-only properties that provide information about the state of the pool.

Property	Getter	Setter	Description
`freeSize`	+	-	Tells how many free connections are in the pool. Value is between 0 and `totalSize`.
`workingSize`	+	-	Tells how many connections were taken from the pool and are currently used in the application.
`totalSize`	+	-	Total size of open connection. At the pool creation – 0, after obtaining first connection – between `minPoolSize` and `maxPoolSize`.
`connectionCount`	+	-	*Deprecated*. Same as `freeSize`.

org.firebirdsql.pool.FBWrappingDataSource

This class is a wrapper for `FBConnectionPoolDataSource` converting interface from `javax.sql.ConnectionPoolDataSource` to `javax.sql.DataSource`. It defines same properties as `FBConnectionPoolDataSource` does.

Runtime object allocation and deallocation hints

Pool implementation shipped with Jaybird can provide hints for the application where the connection was obtained from the pool, when it was released back to the pool, when the statement was prepared. Such information is written into the log when appropriate system properties are set to `true`.

List of properties

Property name	Description
`FBLog4j`	Enables logging inside driver. This is the essential property, if it is not present or set to `false`, no debug information is available. When it is set to `true`, pool automatically prints the following information: • When physical connection is added to the pool – `DEBUG` • When a maximum pool capacity is reached – `DEBUG` • When connection is obtained from pool – `DEBUG` • When connection is released back to pool – `DEBUG` • Whether pool supports open statements across transaction boundaries – `INFO`
`FBPoolShowTrace`	Enables logging of the thread stack trace when debugging is enabled and: • Connection is allocated from the pool – `DEBUG` • Thread is blocked while waiting for a free connection – `WARN`
`FBPoolDebugStmtCache`	When statement caching is used and debugging is enabled, following information is logged: • When a statement is prepared – `INFO` • When statement cache is cleaned – `INFO` • When statement is obtained from or returned back to pool – `INFO`