JayBird JCA/JDBC Driver

Release Notes v 1.5.0

Table of Contents

General Notes

JayBird is JCA/JDBC driver suite to connect to Firebird database server. Historically Borland opened sources of type 3 JDBC driver called InterClient. However due to some inherent limitations of Firebird client library it was decided that type 3 driver is a dead end, and Firebird team developed pure Java implementation of wire protocol. This implementation became basis for JayBird, pure Java driver for Firebird relational database.

This driver is based on both the new 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 2 XADataSource idea, the JCA specification is considerably clearer on the division of responsibility between the application server and driver.

Supported Firebird versions

JayBird 1.5 supports Firebird 1.0.x SuperServer and Classic, Firebird 1.5 SuperServer and Classic independently of the platform on which server runs when type 4 JDBC driver is used. Type 2 and embedded server JDBC drivers require JNI library to have precompiled binaries for Win32 and Linux platforms, other platforms require porting/building JNI library for that platform. Firebird 0.9.x and InterBase 6.0.x OpenSource can be accessed with JayBird 1.5, but are not officially supported by the project.

Specification support

Driver supports following specifications:

JDBC 2.0	Driver passed complete JDBC compatibility test suite, though some features are not implemented. It is not officially JDBC compliant, because of high certification costs.
JDBC 2.0 Standard Extensions	JayBird provides implementation of following interfaces from `javax.sql.*` package: • `ConnectionPoolDataSource` implementation provides connection and prepared statement pooling. • `DataSource` implementation provides seamless integration with major web and application servers. • `XADataSource` implementation provides means to use driver in distributed transactions.
JCA 1.0	JayBird provides implementation of `javax.resource.spi.ManagedConnectionFactory` and related interfaces. CCI interfaces are not supported.
JTA 1.0.1	Driver provides implementation of `javax.transaction.xa.XAResource` interface via JCA framework and `XADataSource` implementation.
JMX 1.2	JayBird provides MBean that allows creating and dropping databases via JMX agent.

Using JayBird with InterBase

JayBird 1.5 can be used with InterBase 6.0, 6.5 and 7.1 (not tested with versions with applied service packs). Above mentioned servers implement same wire protocol as Firebird 1.0 and 1.5.

JayBird passes almost all JDBC 2.0 compatibility tests when used with InterBase, failed tests are caused by the unavailability of some Firebird features in InterBase. JayBird 1.5 provides much more better JDBC compatibility compared to InterClient 4 shipped with InterBase 7.1 and provides many features that are not available in InterClient 4.

What's new in JayBird 1.5.0

JayBird 1.5 introduces number of new features and improvements.

Type 2 and embedded server

JayBird 1.5 provides type 2 JDBC driver that uses native client library to connect to the databases. Additionally JayBird 1.5 can use embedded version of Firebird relational database allowing to create Java applications that does not require separate server setup.

Native client library has possibility to use IPC when connecting to the database on the same host. This significantly increases speed of the applications that connect to the server on the same host.

Support of IN and OUT stored procedure parameters

JayBird 1.5 parses procedure call statement and converts it into corresponding SQL call. JayBird 1.0 supported only IN procedure parameters, new version provides full support for OUT parameters too (`CallableStatement.registerOutParameter(...)` methods).

New connection pooling framework

JayBird 1.0 provided built-in connection pooling using `FBWrappingDataSource` class. JayBird 1.5 provides new connection and prepared statement pooling framework. See corresponding chapter for more information.

Firebird specific extensions

JayBird 1.5 provides extensions to some JDBC interfaces. All extension interfaces are released under modified BSD license, on "AS IS" basis, this should make linking to these classes safe from the legal point of view. All classes belong to `org.firebirdsql.jdbc.*` package.

JayBird 1.5 JDBC extensions		
FirebirdConnection	createBlob()	Create new BLOB in the database. Later this BLOB can be passed as a parameter into `PreparedStatement` or `CallableStatement`.
	getIscEncoding()	Get connection character encoding.

	setTransactionParameters(
	   int isolationLevel,
	   int[] tpbParams
	)
		Set TPB parameters for the specified transaction isolation level.

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

	JayBird 1.5 JDBC extensions	
	`hasOpenResultSet()`	Check if this statement has open result set. Correctly works only when auto-commit is disabled. Check method documentation for details.
	`isValid()`	Check if this statement is still valid. Statement might be invalidated when connection is automatically recycled between transactions due to some irrecoverable error.
FirebirdBlob	`detach()`	Method "detaches" a BLOB object from the underlying result set. Lifetime of "detached" BLOB is limited by the lifetime of the connection.
	`isSegmented()`	Check if this BLOB is segmented. Seek operation is not defined for the segmented BLOBs.

	`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	`getBlob()`	Get corresponding BLOB instance.
	`seek(int position)`	Change the position from which BLOB content will be read, works only for stream BLOBs.

Batch executions

JayBird 1.5 supports batch execution of update statements. Batch is executed until first failure. If error happens during batch execution and auto-commit is enabled, whole batch is rolled back.

Multi-threading fixes

JayBird 1.5 connections can safely be used from multiple threads. All necessary synchronization is performed internally.

Scrollable result sets

JayBird 1.5 provides support for `ResultSet.TYPE_SCROLL_INSENSITIVE`. This feature provides scrollable result sets on the client. However use it with care – complete result set is fetched to the client. Big result sets might decrease speed of the application and cause `OutOfMemoryException`.

Also note that result sets in Firebird are not updatable in terms of `ResultSet.updateXXX(...)`

methods. Also scrollable result sets cannot be combined with named cursors.

JDBC 2.0 compatibility

JayBird 1.5 includes number of fixes that allow it pass JDBC 2.0 compatibility suite. It successfully passes 1216 tests, 60 tests were excluded, because they are either not applicable to Firebird or fail due to some server problems (math rounding issues, limitations of NUMERIC data type, etc.).

Distribution package

JayBird driver has compile-time and run-time dependencies to JCA 1.0, JTA 1.0.1, JAAS 1.0, JDBC 2.0 Optional Package and to Doug Lea concurrent package[1]. Additionally, if Log4J classes are found in the class path, it is possible to enable debugging inside the driver.

Following file groups can be found in distribution package:

*   `firebirdsql.jar` – archive containing JCA/JDBC driver and JMX management class. It requires JCA 1.0, JTA 1.0.1, and JAAS 1.0.

*   `firebirdsql-pool.jar` – archive contains implementation of connection pooling and statement pooling interfaces. It requires JDBC 2.0 Optional Package and Doug Lea `concurrent.jar`.

*   `firebirdsql-test.jar` – archive contains JUnit test cases that are used to assure the driver's quality. Requires JUnit 3.8.

*   `firebirdjmx.jar` – archive contains JMX management class that allows creating and dropping databases.

*   `firebirdsql.rar` – resource archive ready for deployment in JCA-enabled application servers.

*   `lib/jaas.jar` – archive containing JAAS 1.0 classes.

*   `lib/log4j-core.jar` – archive containing core Log4J classes that provide a possibility to log into the file.

*   `lib/mini-j2ee.jar` – archive containing JCA 1.0, JTA 1.0.1 and JDBC 2.0 Optional Package classes.

*   `lib/mini-concurrent.jar` – archive containing subset of classes from Doug Lea `concurrent.jar` that are needed by connection pooling implementation.

*   `firebirdsql-full.jar` – merge of `firebirdsql.jar`, `mini-j2ee.jar` and `mini-concurrent.jar`. This archive can be used for standalone[2] JayBird deployments.

*   `jaybird.dll` – Windows version of the JNI library for Type 2 and Embedded Server drivers.

*   `libjaybird.so` – Linux version of the JNI library for Type 2 and Embedded Server drivers.

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.

Source Code

Source code can be obtained from the CVS at SourceForge.net. The CVSROOT is `:pserver:anonymous@cvs.sourceforge.net:/cvsroot/firebird`, the module name is `client-java`. Alternatively source code can be viewed online at http://cvs.sourceforge.net/viewcvs.py/firebird/client-java/

---

1   Latest version of Doug Lea concurrent package can be obtained from the following address: http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html
2   You have to ensure that your class path contains JAAS 1.0 classes when using JDK 1.3.x.

JDBC URL Format

Driver provides different JDBC URLs for different usage scenarios:

Pure Java

```
jdbc:firebirdsql:host[/port]:/path/to/db.fdb
jdbc:firebirdsql://host[:port]/path/to/db.fdb
```

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

First format is considered official, second – compatibility mode for InterClient migration.

Using Firebird client library

```
jdbc:firebirdsql:native:host[/port]:/path/to/db.fdb
jdbc:firebirdsql:native://host[:port]/path/to/db.fdb
```

Type 2 driver, will connect to the database using client library (either `fbclient.dll` or `gds32.dll` on Windows, and `libfbclient.so` or `libgds.so` on Linux). Requires correct installation of the client library.

```
jdbc:firebirdsql:local:/path/to/db.fdb
```

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:/path/to/db.fdb
```

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 server.

Using Type 2 and Embedded Server driver

JayBird 1.0 provided only pure Java wire protocol implementation. While being most effective in client-server setups (even more effective than native client libraries), it performed worser when connected to the server residing on the same host compared to native (C/C++/Delphi/etc) solutions. Reason is that type 4 driver communicates with the server using network sockets, which introduce additional overhead.

JayBird 1.5 provides type 2 JDBC driver that uses native client library to connect to the databases. Additionally JayBird 1.5 can use embedded version of Firebird relational database allowing to create Java applications that does not require separate server setup.

However type 2 driver has also limitations. Due to multi-threading issues in Firebird client library as well as in embedded server version, it is not possible to access them from different threads simultaneously. When using client library only one thread is allowed to access connection at a time, however it is allowed to access different connections from different threads. 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 corresponding mutex is local to the classloader that loaded 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). This issue will be fixed in future releases on JNI level.

Configuring Type 2 JDBC driver

Type 2 JDBC driver requires JNI library to be installed and available for Java Virtual Machine. Library is not distributed together with the JayBird, but is available from the SourceForge.net download area[3]:

- `jaybird.dll` is precompiled binary for Windows platform. Successfully tested with Windows 2000 and Windows XP SP1, but there should be no issues also in other Win32 OS.
  Library should be either copied into the directory specified in %PATH% environment variable, or made available to JVM using the `java.library.path` system property.

- `libjaybird.so` is precompiled binary for Linux platform. It must be available via the LD_LIBRARY_PATH environment variable, e.g. copied into `/usr/lib/` directory. Another possibility is to specify path to the directory with JayBird JNI library in `java.library.path` system property during the JVM startup.

- 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 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

Embedded Server JDBC driver uses 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 application uses character encodings, UDFs or wants to fine-tune server's 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.

JDBC 2.0 Compatibility

As it was mentioned before, JayBird 1.5 JCA/JDBC driver passed Sun JDBC CTS 1.3.1 test suite. All tests except those that do not apply to Firebird RDBMS succeeded. However driver is not officially JDBC-compliant because of high costs involved in the certification process.

In particular driver meets following requirements:

- JDBC specification requires driver to support Entry level SQL 92 plus "DROP TABLE" SQL

3 http://sourceforge.net/project/showfiles.php?group_id=9028&package_id=44761

command. JayBird 1.5 does not implement any SQL parsing code and relies on Firebird database server.

- Driver support escaped syntax except the escaped character in LIKE clause.

- Driver supports transactions, both local and distributed.

- Driver correctly provides information about supported features through the `java.sql.DatabaseMetaData` interface.

Unimplemented JDBC 2.0 Features

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

- `ResultSet.TYPE_SCROLL_SENSITIVE`

  Server does not support scrollable cursors, so we are not able to implement this feature correctly. When you create a statement with result set type `TYPE_SCROLL_SENSITIVE`, it is replaced with `TYPE_SCROLL_INSENSITIVE` and corresponding warning is added to connection warnings.

- `ResultSet.CONCUR_UPDATABLE` and related methods to update result sets:
    - `rowUpdated`
    - `rowInserted`
    - `rowDeleted`
    - `updateXXX` methods
    - `insertRow`
    - `updateRow`
    - `deleteRow`
    - `refreshRow`
    - `cancelRowUpdates`
    - `moveToInsertRow`
    - `moveToCurrentRow`
- `Ref`, `Clob` and `Array` types and corresponding methods from `PreparedStatement` and `ResultSet` interfaces.
    - `java.sql.PreparedStatement`
        - `setRef`
        - `setClob`
        - `setArray`
    - `java.sql.ResultSet`
        - `getArray`
        - `getRef`
        - `getClob`
- User Defined Types/Type Maps and corresponding methods from `ResultSet` and `Connection` interfaces
    - `java.sql.ResultSet`
        - `getObject(int i, java.util.Map map)`
        - `getObject(String columnName, java.util.Map map)`
    - `java.sql.Connection`
        - `getTypeMap()`
        - `setTypeMap(java.util.Map map)`

- Excluding the unsupported features, the following methods are not yet implemented:

    - `java.sql.Statement`
        - `cancel` is currently not supported by Firebird, however it support is planned for the next major Firebird release. At that time also support in JDBC driver will be provided.
    - `java.sql.Blob`
        - `position(byte pattern[], long start)`

- – `position(Blob pattern, long start)`
- ● The following methods are implemented, but do not work as expected:
  - • `java.sql.Statement`
    - • `get/setMaxFieldSize` does nothing
    - • `get/setQueryTimeout` does nothing
  - • `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
  - • `java.sql.DatabaseMetaData`
    - • `getBestRowIdentifier` always returns empty result set.

JayBird Specifics

JayBird driver has also 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 the behavior depends whether 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 – Firebird API does not provide scrollable cursor support, navigation is possible only in one direction.

PreparedStatement.setTimestamp(int, Timestamp, Calendar)

JDBC specification tells that "With a Calendar object, the driver can calculate the timestamp taking into account a custom timezone.". However it does not tell how this phrase should be interpreted and there is no agreement between driver developers about the meaning of this phrase.

All previous versions of JayBird, including all release candidates, interpreted it as "the timestamp is specified in the time zone of the passed calendar, it is written into the database using the local time zone of the JVM". However it was decided that this interpretation is incorrect. New interpretation is "Timestamp is specified in local JVM time zone, it should be written into the database converted to the time zone of the specified calendar".

All applications that rely on the old interpretation of this method should specify `timestamp_uses_local_timezone` connection parameter

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 field is accessed from 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.

Connection pooling with JayBird 1.5

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

JayBird 1.5 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, connections are physically opened in `getConnection()` method and physically closed in their `close()` method.

Description of some connection pool classes.	
`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

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".
isolation	+	+	Default transaction isolation level. All connections returned from the pool will have this isolation level. One of: • TRANSACTION_READ_COMMITTED • TRANSACTION_REPEATABLE_READ • TRANSACTION_SERIALIZABLE
transactionIsolationLevel	+	+	Integer value from `java.sql.Connection` interface corresponding to the transaction isolation level specified in `isolation` property.

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.

Property	Getter	Setter	Description
`totalSize`	+	-	Total size of open connection. At the pool creation – 0, after obtaining first connection – between `minPoolSize` and `maxPoolSize`.
`connectionCount`	+	-	*Deprecated*. Same as `freeSize`.

Firebird-specific Properties

This group contains properties that specify parameters of the connections that are obtained from this data source. Commonly used parameters have the corresponding getter and setter methods, rest of the Database Parameters Block parameters can be set using `setNonStandardProperty` setter method.

Property	Getter	Setter	Description
`database`	+	+	Path to the database in the format `[host/port:]/ path/to/database.fdb`
`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.
`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.
`userName`	+	+	Name of the user that will be used by default.
`password`	+	+	Corresponding password.
`roleName`	+	+	SQL role to use.
`tpbMapping`	+	+	TPB mapping for different transaction isolation modes.

Property	Getter	Setter	Description
nonStandardProperty	-	+ setNonStanda rdProperty (String) setNonStanda rdProperty (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 parameters using configuration files. setNonStandardProperty(String, String) takes property name as first parameter, and its value as the second parameter.

org.firebirdsql.pool.DriverConnectionPoolDataSource

This class provides connection and statement pooling capabilities for arbitrary any JDBC driver. It is very similar to the previous class, however instead of allocating physical connections via JayBird JCA framework, it uses java.sql.DriverManager. It also does not implement javax.sql.XADataSource interface and pooled connections cannot participate in distributed JTA transactions.

This class supports all properties from "Standard JDBC Properties", "Pool Properties" and "Runtime Pool Properties" groups that are described before.

Database-specific Properties

Property	Getter	Setter	Description
driverClassName	+	+	Name of the JDBC driver class.
jdbcUrl	+	+	JDBC URL that will be used to allocate connections.
property	getProperty (String)	setProperty (String, String)	Allows setting additional connection properties.

Note that this class does not provide getters and setters for standard propertied like user name, password, etc. This might be viewed as a limitation/inconvenience, however we believe that if there is situation where DriverConnectionPoolDataSource.setProperty(String, String) cannot be used, one can easily create a subclass providing all necessary setters.

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. Additionally, when connection or prepared statement is closed twice, driver will throw an

SQL exception with an attached stack trace of previous call to `close()` method.

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`

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.

Also a new resource, JayBirdWiki is available at http://jaybirdwiki.firebirdsql.org. You are welcome to provide information and tips in the SandBox area.

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.com.

Reporting Bugs

The developers follow the Firebird-Java@yahoogroups.com list. Join the list and post information

about suspected bugs. This is a good idea because what is often thought to be a bug turns out to be something else. List members may be able o help out and get you going again, whereas bug fixes might take awhile.

If you are sure that this is a bug you may report it in the Firebird bug tracker, "Java Client (JayBird)" at SourceForge.net project area (http://sourceforge.net/projects/firebird).

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.