JayBird JCA/JDBC Driver

Release Notes v 2.0

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 2.0 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, precompiled binaries for Win32 and Linux platforms are shipped in the default installation, 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 2.0, but are not officially supported by the project.

Specification support

Driver supports following specifications:

JDBC 3.0	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 2.0 provides MBean to manage Firebird server and installed databases via JMX agent.

Using JayBird with InterBase

JayBird 2.0 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 3.0 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 2.0

JayBird 2.0 underwent significant internal changes that are not directly visible to the client code, and not necessarily introduced new features. However those changes significantly increase code modularity, stability and maintainability.

Refactorings

- Significant refactoring of the org.firebirdsql.gds.* package. Goal is to create an official Firebird API for Java programming language. Implementations of the interfaces defined in this package are now dynamically loaded from the classpath and provide possibilities to extend driver with custom code without changing the driver itself.
- JCA code was significantly changed, bringing a long awaited stability when critical errors happen into that vital part of code. There is no more so-called “fatal” errors that can cause driver to recycle connections and transactions.
- Refactoring of the transaction and object life-cycle management in the JDBC code. Main reason for this change was the need to remove result set caching in auto-commit mode and make it JDBC specification compliant, but it has also very positive effect on the code stability and maintainability.
- Unified connection property handling in JCA, driver manager and pooling code. This solves a longstanding issue which caused configuration
- Refactoring of the pooling code to support JDK 5.0[1].

Updatable result sets

JayBird 2.0 provides now support for the updatable result sets. Feature allows Java application to update current record using the `updateXXX` 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 single table;
- all columns not referenced in `SELECT` permit `NULLS` (otherwise `INSERTS` 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 2.0 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 maintanence, 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.

1 Sun has changed the API of some JNDI classes, which in JDK 5.0 require Hashtable<String, String> as parameter, which is not compatible with JDK 1.4.x interface declarations.

JayBird JDBC extensions

JayBird 2.0 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. Table below shows all JDBC extensions present in JayBird with a driver version in which the extension was introduced.

JayBird 2.0 JDBC extensions

Interface	Since	Method name	Description
`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 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 )`	Set TPB parameters for the specified transaction isolation level. The newly specified parameters are effective until the transaction isolation is changed.
`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.

JayBird 2.0 JDBC extensions			
	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.
`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.
`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.

JDBC 3.0 compatibility

JayBird 2.0 includes number of fixes that allow it pass JDBC 3.0 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[2]. Additionally, if Log4J classes are found in the class path, it is possible to enable extensive logging 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-full.jar` – merge of `firebirdsql.jar`, `mini-j2ee.jar` and `mini-concurrent.jar`. This archive can be used for standalone[3] JayBird deployments.
- `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.
- `jaybird2.dll` – Windows version of the JNI library for Type 2 and Embedded Server drivers.
- `libjaybird2.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. Using JayBird (by importing JayBird's public interfaces in your Java code), and extending JayBird by subclassing or implemention 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

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/

2 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

3 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.

JDBC connection properties

Table below 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.

Property	Getter	Setter	Description
`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 containg character translation map.
`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.
`tpbMapping`	+	+	TPB mapping for different transaction isolation modes.
`defaultIsolation`	+	+	Default transaction isolation level. All newly created connections will have this isolation level. One of:

Property	Getter	Setter	Description
			• `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 parameters using configuration files.
			`setNonStandardProperty(String, String)` takes property name as first parameter, and its value as the second parameter.

Using Type 2 and Embedded Server driver

JayBird 2.0 provides type 2 JDBC driver that uses native client library to connect to the databases. Additionally JayBird 2.0 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 Firebird client library after merge with Vulcan project.

Configuring Type 2 JDBC driver

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

- `jaybird2.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.

- `libjaybird2.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 3.0 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 certification procedure is too expensive.

JDBC 3.0 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.CallableStatement`
  - `addBatch()`, `clearBatch()` and `executeBatch()` are not yet supported, a simple workaround is to use `EXECUTE PROCEDURE` statement with `PreparedStatement` batches.
- `java.sql.Clob` data type is not yet supported.
- `java.sql.Connection`
  - `getCatalog()` and `setCatalog(String)` are not supported by Firebird server.
  - `getTypeMap()` and `setTypeMap(Map)` are not supported.
  - `prepareStatement(String, int)`, `prepareStatement(String, int[])` and `prepareStatement(String, String[])` that return auto-generated keys are not implemented, because this functionality is not provided by server.
- `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.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.Statement`
  - `cancel()` is not supported by Firebird server.
  - `execute(String, int)`, `execute(String, int[])`, `execute(String, String[])`, `executeUpdate(String, int)`, `executeUpdate(String, int[])`, and `executeUpdate(String, String[])` that return auto-generated keys are not implemented, because of server incapability.
  - `getGeneratedKeys()` is not implemented because of server incapability.
- `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 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.

`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 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 2.0

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

Description of some connection pool classes.	
	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.
`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

Property	Getter	Setter	Description
			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`. 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

Property name	Description
	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.

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.