Frequently Asked Questions

JayBird JCA/JDBC Driver for Firebird

- Last Updated 01/13/04 -

1- How much of JDBC 2.0 is supported by JayBird?

return to top

JayBird complies with the JDBC 2.0 core with some features and methods not implemented. Some of the unimplemented items are required by the specification and some are optional.

Implemented features:

Most useful JDBC functionality ("useful" in the opinion of the developers).

Complete JCA API support: may be used directly in JCA-supporting application servers such as JBoss and WebLogic.

XA transactions with true two phase commit when used as a JCA resource adapter in a managed environment (with a TransactionManager and JCA deployment support).

Includes optional internal connection pooling for standalone use and use in non-JCA environments such as Tomcat 4.

ObjectFactory implementation for use in environments with JNDI but no TransactionManager such as Tomcat 4.

DataSource implementations with or without pooling.

Driver implementation for use in legacy applications.

Complete access to all Firebird database parameter block and transaction parameter block settings.

Optional integrated logging through log4j.

JMX mbean for database management (so far just database create and drop).

2- What parts of JDBC 2.0 are NOT supported by JayBird?

return to top

For more information see the JDBC 2.0 Conformance document in the source code in the src/etc folder.

The following optional features are NOT supported:

The following optional features and the methods that support it are not implemented:

- **Batch Updates.**
  - java.sql.Statement
    - addBatch(String sql)
    - clearBatch()
    - executeBatch()
  - java.sql.Statement
    - addBatch()
- **Scrollable cursors.**
  - java.sql.ResultSet
    - beforeFirst()
    - afterLast()
    - first()
    - last()
    - absolute(int row)
    - relative(int rows)
    - previous()
- **Updatable cursors.**
  - java.sql.ResultSet
    - rowUpdated()
    - rowInserted()
    - rowDeleted()
    - updateXXX(....)
    - insertRow()
    - updateRow()
    - deleteRow()
    - refreshRow()
    - cancelRowUpdates()
    - moveToInsertRow()
    - moveToCurrentRow()
- **Cursors/Positioned update/delete**
  - java.sql.Statement
    - setCursorName()
  - java.sql.ResultSet
    - getCursorName()
- **Ref, Clob and Array types.**
  - java.sql.PreparedStatement
    - setRef(int i, Ref x)
    - setClob (int i, Clob x)
    - setArray(int i, Array x)
  - java.sql.ResultSet

        - getArray(int i)
        - getArray(String columnName)
        - getRef(int i)
        - getRef(String columnName)
        - getClob(int i)
        - getClob(String columnName)
- **User Defined Types/Type Maps.**
    - 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()
- **java.sql.CallableStatement**
    - registerOutParameter(int parameterIndex, int sqlType)
    - registerOutParameter(int parameterIndex, int sqlType, int scale)
    - wasNull()
- **java.sql.Blob**
    - length()
    - getBytes(long pos, int length)
    - 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(index,object,type) This method is implemented but behaves as setObject (index,object)
    - setObject(index,object,type,scale) This method is implemented but behaves as setObject (index,object)
- **java.sql.ResultSetMetaData**
    - isReadOnly(i) always returns false
    - isWritable(i) always returns true
    - isDefinitivelyWritable(i) always returns true
- **java.sql.DatabaseMetaData**
    - getBestRowIdentifier(i) always returns empty resultSet

3- Where do I get JayBird?

return to top

In your browser go to:

http://sourceforge.net/projects/firebird/

Scroll down the page and find the row containing:

firebird-jca-jdbc-driver

Click the link "Download" in the rightmost column of the table in that row. This links you to another page with the zip files of the various versions of JayBird available to be downloaded. JayBird will have the heading "firebird-jca-jdbc-driver". Click on the version you wish to download. It looks something like: FirebirdSQL-1.x.zip.

This will download the driver files to your system. Unzip the file and copy the contents to the appropriate directories on your system as described in #4 "How do I install JayBird?" below.

More recent bugfix versions of JayBird can be obtained by downloading the daily snapshot of the project through CVS and building the project. See #13 below for details.

4- How do I install JayBird?

return to top

The classes from firebirdsql.jar must be in the classpath of the Java application being compiled, or otherwise made available to your application. The classes from the following packages must also be available:

- mini-concurrent.jar
- jaas.jar (included in jdk 1.4)
- mini-j2ee.jar (now including JDBC classes)
- log4j-core.jar (if you want logging available)

The firebirdsql-full.jar file has all of the needed files in one archive for ease of use.

These archives are included in the binary package.

You can use the jmx management mbean either in a jmx agent (MBeanServer) or as a standalone class. So far it has been tested in jbossmx and the jmxri, although since it is extremely simple it should have no problems in any jmx implementation. Use of jbossmx is highly recommended due to its smaller bug count and because it is in active development.

For use in a managed environment as a JCA resource adapter, deploy firebirdsql.rar according to the environment's deployment mechanism.

For installation in Tomcat, JBoss, JBuilder, and for use with stand alone Java programs see the corresponding sections below.

5- How do I use built-in Connection Pooling?

return to top

JayBird features built-in connection pooling. This is very useful because it eliminates the need for external connection pooling routines like Poolman or DBCP.

Use FBWrappingDataSource for pooling. See A simple pooling example below for a code example.

Setting up and shutting down JDBC connections to databases tend to be very time and CPU intensive operations. Connection pooling allows connections to be stored and reused by applications, or even by different applications without going through the time consuming task of setting up the connection again.

Using a connection pool effectively while preventing data corruption and unauthorized access requires a slightly different mindset.

The Firebird database uses the concept of user logins. To access the database a user must log in and provide a password that the database recognizes. This is set up by the database administrator.

This can lead to problems with a connection pool. Suppose "joe" logs in using his password and works with the database. If he logs out and his connection is kept alive and given to the next user, "bob", without re-authorizing, "bob" may be able to see confidential data that he is not intended to see. Worse he can change or delete data he should not have access to.

If we only allow users to use connections previously opened with their username, we drastically reduce the effectiveness of the pool.

To make the pool most effective we have to login with the same username every time. This requires us to manage the user access to the database in our code rather that allowing Firebird to do it for us. This is usually accomplished by creating a table of usernames, passwords, and roles separate from those maintained by the database.

When a user logs in, all database access is done under a single username and password known to Firebird, for example username "calendar", password "calpass". The program then opens a user table created for the application and retrieves the username, password, and role of the user, which might be "Joe", "joespassword", and "manager". The retrieved username and password is compared by your

program to those provided by the user. If they match, your program allows the user to perform actions on the database allowed for users with their role, all under the Firebird username of "calendar".

When the next user, "bob", logs in, transactions are still done with your program using the connection opened with the "calendar" Firebird username with the database, but the program checks the "bob" password and role in the application user table before allowing transactions.

This results in maximum efficiency of connection pools. It is also compatible with the way that web application servers handle container managed security. So even if your program starts out as a standalone Java program, you can "webify" the database access part of it very easily.

It is possible to obtain connections with different usernames/passwords with pooling enabled, and the connections will be kept separate, but this is apt to result in inefficient connection usage.

Be aware that JayBird provides no encryption. If you use JayBird in a standalone Java program, anyone who can listen to your network connection can see your usernames, passwords, and data as it crosses the net. You must take steps to secure your data. You can do this by all the standard methods: Secure networks, VPN, etc.

A popular way to provide wide access to your database while still providing security is to write your application as a web application that is viewed in a browser, rather than as a stand-alone application. Then you can restrict your web app to running under secure HTTP.

If you are using JayBird in stand-alone Java applications there is little need to use anything other than the built-in pooling.

However, in some cases, you may not want to use the built-in connection pooling. If you are using JayBird with a J2EE server that manages connection pooling, like JBoss, you should deploy JayBird as a JCA resource adapter.

In such cases the container (J2EE Server) needs JayBird to be deployed as a JCA resource adapter so that it can manage connection usage and pooling, hook the connections up to the transaction manager, and manage the security and supply connections logged as appropriate for the current application user.

If you are using a limited J2EE server you may need to use the built-in pooling. Versions of Tomcat before 4.1 are an example. From version 4.1, Tomcat has provided connection pooling via DBCP. Tomcat is a Servlet and Java Server Pages (JSP) server, but does not provide full J2EE web application server support.

Since a large number of installations use only servlets and JSP's, application servers like Tomcat, JRun, Cold Fusion, Servlet Exec, and Resin have become very popular. These have varying support for connection pooling. You will have to check the features of the particular version of your app server to see if pooling is offered or if you will need to use the built-in pool from JayBird.

See the sections below on JBoss, Tomcat, and Using JayBird in Java code for specific information on those environments.

6- How do I use JayBird in Java code?

return to top

Two forms of JayBird can be used. FBDriver is used much like the old Interclient driver. FBWrappingDataSource has internal connection pooling capability. Examples of both are included here. Code examples of many of the classes and methods used by JayBird can be found in the src/test subdirectory of the source code available on SourceForge.net, see question 13 below.

JayBird supports two URL syntax formats:

Standard format= jdbc:firebirdsql:[//host[:port]/]<database>

FB old format= jdbc:firebirdsql:[host[/port]:]<database>

For all environments that do not support JCA deployment, make the classes in firebirdsql.jar available to your application. You will probably have to use some of the jars mentioned above as necessary.

For use in a somewhat managed environment with JNDI but no JCA support or transaction manager, use the FBDataSourceObjectFactory to bind a reference to a DataSource into JNDI. Tomcat 4 is an example of this scenario. The JNDI implementation must support use of References/Referenceable. This will not work if the JNDI implementation only supports binding serialized objects.

For use in a standalone application that only needs one connection, use either FBWrappingDataSource or FBDriver.

A typical use of the FBDriver class would use code something like this:

```
Class.forName("org.firebirdsql.jdbc.FBDriver");

Connection conn = DriverManager.getConnection
("jdbc:firebirdsql:localhost/3050:/firebird/test.gdb",
"sysdba", "masterkey");
```

Or in windows:

```
DriverManager.getConnection
("jdbc:firebirdsql:localhost/3050:E:\\database\\carwash.gdb",
"sysdba", "masterkey");
```

For use in a standalone application with multiple connections that would benefit from connection pooling, use an instance of FBWrappingDataSource configured for pooling.

A simple pooling example:

```
Boolean FBDriverLoaded=false;

if (!FBDriverLoaded)

{ // don't load JayBird more than once.

try

{

org.firebirdsql.jdbc.FBWrappingDataSource fbwds = new
org.firebirdsql.jdbc.FBWrappingDataSource();

FBDriverLoaded = true;

}

catch (ResourceException e)

{

System.out.println("Could Not create
org.firebirdsql.jdbc.FBWrappingDataSource, error:"+e+"\n");

}



fbwds.setDatabase("//localhost:3050/dir1/subdir/myDatabase.gdb");

// an old format version of the same url

// fbwds.setDatabase("localhost/3050:/dir1/subdir/myDatabase.gdb");

fbwds.setUserName("sysdba");

fbwds.setPassword("masterkey");

fbwds.setIdleTimeoutMinutes(30);

fbwds.setPooling(true); // this turns on pooling for this data
source. Max and min must be set.

fbwds.setMinSize(5); // this sets the minimum number of connections
to keep in the pool

fbwds.setMaxSize(30); // this sets the maximum number of connections
that can be open at one time.
```

```
Try

{

fbwds.setLoginTimeout(10);

}

catch (SQLException e)

{

System.out.println("Could not set Login Timeout in SQLDriver\n");

}

}

else

{

//System.out.println("Firebird Driver already exists, not
reloaded.\n");

}



Connection c;

try

{

c = fbwds.getConnection();

}

catch (SQLException e)

{

system.out.println("getting new fbwds connection failed! Error:
"+e+"\n");

handleError(e);
```

```
}

// Use the connection "c" like any other connection then close it.

// To release the connection back to the pool you must also close all
result sets

// and close all statements associated with the connection.

c.close();

// closing c returns it to the pool.
```

Be aware that no security or encryption is built into JayBird. If you use stand-alone Java programs you must provide secure access to your database to protect your passwords and data.

More Java code for a driver example and a DataSource example are included in the Code Examples section below.

7- How do I use JayBird with JBoss?

return to top

Deployment in JBoss 3.0.0 and later:

The additional jars/classes mentioned above are already available in JBoss. Put firebirdsql.rar in the deploy directory. Get firebird-service.xml from your binary jboss distribution or jboss CVS at connector [jbosscx]/src/etc/example-config/firebird-service.xml and modify the URL to point to the desired database location. If you get a configuration from CVS, please be very sure and check twice that you have the correct version for your JBoss version. There are hard-to-spot incompatibilities between every minor release.

For simplicity, start by setting the UserName and Password in the firebird-service.xml configuration file. If you need more advanced JAAS based login, set that up based on the instructions in the jboss 3 manual or the quickstart guide after you have a simple configuration working.

8- How do I use JayBird with Tomcat?

return to top

CATALINA_HOME is the installation directory for Tomcat 4.x or greater. An environment variable of that name is set up on the Tomcat server. TOMCAT_HOME was used in versions before Tomcat 4.

To use JayBird with Tomcat you must put the jar files where your web apps can access them. Once they are available to your web apps you use JayBird in your servlets or beans just as you would in standalone programs.

If you have only one webapp using JayBird or you need to keep JayBird separate from other web apps, put the jar files in the WEB-INF/lib/ subdirectory of your web app.

It is more likely that Firebird will be used by all of your web apps and Tomcat itself. To have universal access to JayBird, put the jars in CATALINA_HOME/common/lib/.

A simple example web app that creates a Firebird database and does a few transactions directly is included below. It is called test.

Below that is the same web app set up to use a DataSource and connection pooling via DBCP. It is called dbTest.

To use JayBird's internal connection pooling, you can configure an FBWrapping DataSource for pooling just as you would in a stand-alone program. If you put a class for doing that in a servlet and start it when Tomcat is initialized, you can share a pool among web applications. If you do this, take care to make it thread safe and synchronize access to methods.

Tomcat can also use Firebird for BASIC or FORM based user authentication. See the Tomcat docs for more details. An example realm for this is listed below. This goes in the CATALINA_HOME/conf/server.xml file.

```
<Realm className="org.apache.catalina.realm.JDBCRealm" debug="0"

driverName="org.firebirdsql.jdbc.FBDriver"

userNameCol="USER_NAME"

connectionName="sysdba"

userTable="USERS"

userCredCol="USER_PASS"

validate="true"

connectionURL="jdbc:firebirdsql:localhost/3050:/dir1/subdir/usersdb.gd

userRoleTable="USER_ROLES"

roleNameCol="ROLE_NAME"
```

```
connectionPassword="masterkey"/>
```

If your web app is set up to allow Tomcat to authenticate users this tells Tomcat to use Firebird to look up user names and passwords. It does this by calling the Firebird driver FBDriver to login to a database named usersdb.gdb located on localhost in the directory /dir/subdir/, using the username sysdba and the password masterkey.

Tomcat then takes the username that is typed into the browser by the person logging into the web app and searches the table named USERS to see if it is in the field USER_NAME of a record. If it is, it checks to see if the password typed into the browser is the same as the one in the field USER_PASS for that record. If it is, the user is allowed to login and Tomcat opens the table USER_ROLES and searches for all entries with USER_NAME. For each record it finds, it looks in the ROLE_NAME column and adds that role name to a list of roles for the user. It then allows access to web apps based on the roles listed by the database.

You can configure your web apps in WEB-INF/web.xml to only allow users with certain roles access to the web app. You can even use the role inside your JSP's to only draw certain parts of an HTML page if a user has the appropriate role. This allows you to customize each page based on a user's role.

To use Tomcat's online GUI management web app to control your Tomcat installation, you must have the role "manager" in the USER_ROLES table under your name.

See the Tomcat docs for more information.

Sample Web Apps

These examples assume that the username SYSDBA and password masterkey are valid. These samples create and search for the database in /databases/test.gdb. To use a different directory change those entries below. For windows change /databases/test.gdb to c:\\databases\\test.gdb, for example.

Sample web app test:

To use this sample web app create a folder called "test" in the webapps directory of your Tomcat installation. Put the HTML file below into a file called index.htm inside the folder called test. Put the contents of the XML file following this into test/WEB-INF/web.xml. Finally, put the contents of the jsp file following that into test/search.jsp. Start Tomcat and run test in your browser by calling http://localhost/test/index.htm.

Put this in CATALINA_HOME/webapps/test/index.htm:

```
<html>
< head>
< title>index.htm</title>
< meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
< /head>

<body bgcolor="#CCCCFF" text="#000000">
< /p>
< H1 align="center">Enter Name</H1>
```

< /H1>
< p>This demo of a simple web app will take the name you type above and search
a table called CUSTOMERS in a Firebird database named test.gdb using the JayBird
JDBC
driver.
The table CUSTOMERS has two fields (columns): FIRST_NAME and LAST_NAME.</p>
< p>If the database or tables don't exist they will be created for you. You will
need to edit the Strings at the top of the file search.jsp to appropriate
ones for your system.</p>
< p>If there are no entries with the same last name that you type in, the program
will add 3 so you will get some output: "Aaron, Bob, and Calvin".</p>
< p> </p>
< form name="form1" method="post" action="search.jsp">
< p align="center">
Type a Last Name to search for here:
<input name="name" type="text" id="name">
then click "Submit".</p>
< p align="center"> <input type="submit" name="Submit" value="Submit">
< /p>
< /form>
< p> </p>
< /body>
< /html>

Put this in CATALINA_HOME/webapps/test/WEB-INF/web.xml:

<?xml version="1.0" encoding="UTF-8"?>

< !DOCTYPE web-app PUBLIC '-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN'
'http://java.sun.com/j2ee/dtds/web-app_2.2.dtd'>

<web-app>
<session-config>
<session-timeout>
30
</session-timeout>
</session-config>
<welcome-file-list>
<welcome-file>index.htm</welcome-file>
</welcome-file-list>
< /web-app>

Put this in CATALINA_HOME/webapps/test/search.jsp:

<%@page contentType="text/html; charset=iso-8859-1" language="java"
import="java.sql.*,org.firebirdsql.management.*"%>

```
<html>
< head>
< title>search.jsp</title>
< meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
< /head>

<body bgcolor="#CCCCFF" text="#000000"><p>
< /p>
< H1 align="center">Test Search</H1>

<%
// Edit the strings below as needed for your system.

String DB_SERVER_URL = "localhost";
int DB_SERVER_PORT = 3050;

// Use forward slashes here, even on Windows systems.
String DB_PATH = "c:/databases";

String DB_NAME = "test.gdb";
String DB_USER = "sysdba";
String DB_PASSWORD = "masterkey";
String DB_CONNECTION_STRING =
"jdbc:firebirdsql:"+DB_SERVER_URL+"/"+DB_SERVER_PORT+":"+DB_PATH+"/"+DB_NAME;

String lastName = request.getParameter("name");

// To help you debug, you can set a session attribute to values indicating
// the prograss through the web app, then use an HTTP monitor to see how far
// you have progressed.

// This stuff creates the database with the JMX management tools
// if it doesn't already exist.
FBManager fbManager = new FBManager();

fbManager.setServer(DB_SERVER_URL);
fbManager.setPort(DB_SERVER_PORT);

fbManager.start();

fbManager.createDatabase(DB_PATH + "/" + DB_NAME, DB_USER, DB_PASSWORD);


// Load the JayBird driver. This is OK for a test but your real web apps
// should use a dataSource for efficiency and flexibility.
Class.forName("org.firebirdsql.jdbc.FBDriver");

%>
Opening a connection with connection string: <%=DB_CONNECTION_STRING%><BR><BR>
< %
```

```
// Get a connection to the database.
Connection connRSFind = DriverManager.getConnection(DB_CONNECTION_STRING, DB_USER,
DB_PASSWORD);

// Back to HTML, show the user the LAST_NAME typed in.
%>
< p>
The user entered name is: <%=lastName%>
< p>
< %

String sqlString = "SELECT * FROM CUSTOMERS WHERE LAST_NAME = \'" + lastName +"\'";

// Switch back to html so we can print out the sqlString on the web page.
%>

The SQL String is: <%=sqlString%><p>

<%
PreparedStatement StatementRSFind=null;
ResultSet RSFind=null;
boolean resultException=false;
boolean rsReady = false;

try
{
StatementRSFind = connRSFind.prepareStatement(sqlString);

RSFind = StatementRSFind.executeQuery();
rsReady = RSFind.next();
}
catch(SQLException e1)
{
resultException = true; // Set a flag so that we know there was an error, not just an empty result set.

%>
Could not find table, I'll try to add it: <%=e1.getMessage()%><BR>
< %
}

if (!rsReady || resultException)
{ // If there are no database entries with this last name, enter a few so the user has something to look at.
// Or, if there was an error, the table probably doesn't exist yet.

// try to create table CUSTOMERS, in case it doesn't exist yet.

if (resultException)
{// The table CUSTOMERS probably doesn't exist, so we create it here.
Statement statement2=null;
```

```
try
{
statement2 = connRSFind.createStatement();
}
catch(SQLException e2)
{
%>
Could not create statement2 with this connection! Check your database settings in search.jsp: <%
=e2.getMessage()%><BR>
< %
}

if (statement2 != null)
{
try
{ // try to crate the table CUSTOMERS.
sqlString = "CREATE TABLE CUSTOMERS(LAST_NAME VARCHAR(50), FIRST_NAME
VARCHAR(50))";
statement2.execute(sqlString);
statement2.close();
}
catch(SQLException e2a)
{
%>
Table CUSTOMERS already exists and we tried to create it again, or it could not be created: <%
=e2a.getMessage()%> <BR>
< %
}
}
} // Table CUSTOZMERS should exist now, if it didn't before or things are really messed up.
else
{
%>
First attempt at getting a result set produced an empty result set.<BR>
< %
}

// Insert some names into table CUSTOMERS. Either the table is new and empty, or there were
// no entries with the last name the user gave us, so we'll enter a few, so the user has
// something to look at.
Statement statement3=null;

try
{
statement3 = connRSFind.createStatement();
}
catch(SQLException e3)
{
%>
Could not create statement3 with this connection! Check your database settings in search.jsp: <%
=e3.getMessage()%><BR>
```

```
< %
}

if (statement3 != null)
{
try
{
sqlString = "INSERT INTO CUSTOMERS(LAST_NAME, FIRST_NAME) VALUES
(\'"+lastName+"\', 'Aaron')";
%>
Executing SQL: <%=sqlString%><BR>
< %
statement3.execute(sqlString);
sqlString = "INSERT INTO CUSTOMERS(LAST_NAME, FIRST_NAME) VALUES
(\'"+lastName+"\', 'Bob')";
%>
Executing SQL: <%=sqlString%><BR>
< %
statement3.execute(sqlString);
sqlString = "INSERT INTO CUSTOMERS(LAST_NAME, FIRST_NAME) VALUES
(\'"+lastName+"\', 'Calvin')";
%>
Executing SQL: <%=sqlString%><BR><BR>
< %
statement3.execute(sqlString);
statement3.close();
}
catch(SQLException e3a)
{
%>
We could not enter data in table CUSTOMERS for some reason: <%=e3a.getMessage()%> <BR>
< %
}
}

// try again to get a result set.
sqlString = "SELECT * FROM CUSTOMERS WHERE LAST_NAME = \'" + lastName +"\'";
StatementRSFind = connRSFind.prepareStatement(sqlString);
RSFind = StatementRSFind.executeQuery();
rsReady = RSFind.next();
}

int i = 0;

if (rsReady)
{
boolean done=false;
while (!done)
{
i++;
String RSFind_Last = (String) RSFind.getObject("LAST_NAME");
```

```
String RSFind_First = (String) RSFind.getObject("FIRST_NAME");
// display the names in the browser.
%>
Name <%=i%>: <%=RSFind_First%> <%=RSFind_Last%> <BR>
< %
done = !RSFind.next();
} //End while loop

RSFind.close();
}
else
{
%>
< BR>The result set was empty. Check to be sure database is running and settings in search.jsp are
correct.<BR>
< %

}
if (StatementRSFind != null)
StatementRSFind.close();

if (connRSFind != null)
connRSFind.close();

%>

</body>
< /html>
```

Sample web app dbTest:

This sample web app shows how to use a Fiorebird DataSource with Tomcat. To use it, create a folder called "dbTest" in the webapps directory of your Tomcat installation. Put the HTML file below into a file called index.htm inside the folder called dbTest. Put the contents of the XML file following this into dbTtest/WEB-INF/web.xml. Put the contents of the jsp file following that into dbTest/search.jsp. Put the context XML code after that into the conf/server.xml file of your Tomcat installation just before the entry: </Host>. Start Tomcat and run dbTest in your browser by calling http://localhost/dbTest/index.htm.

Put this in CATALINA_HOME/webapps/dbTest/index.htm:

```
<html>
< head>
< title>index.htm</title>
< meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
< /head>

<body bgcolor="#CCCCFF" text="#000000">
< /p>
< H1 align="center">Enter Name</H1>
```

```
< /H1>
< p>This demo of a simple web app will take the name you type above and search
a table called CUSTOMERS in a Firebird database named test.gdb using the JayBird
JDBC
driver.
The table CUSTOMERS has two fields (columns): FIRST_NAME and LAST_NAME.</p>
< p>If the database or tables don't exist they will be created for you. You will
need to edit the Strings at the top of the file search.jsp to appropriate
ones for your system.</p>
< p>If there are no entries with the same last name that you type in, the program
will add 3 so you will get some output: &amp;quot;Aaron, Bob, and Calvin&amp;quot;.</p>
< p>&amp;nbsp;</p>
< form name="form1" method="post" action="search.jsp">
< p align="center">
Type a Last Name to search for here:
<input name="name" type="text" id="name">
then click &amp;quot;Submit&amp;quot;.</p>
< p align="center"> <input type="submit" name="Submit" value="Submit">
< /p>
< /form>
< p>&amp;nbsp;</p>
< /body>
< /html>
```

Put this in CATALINA_HOME/webapps/dbTest/WEB-INF/web.xml:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE web-app PUBLIC '-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN'
'http://java.sun.com/j2ee/dtds/web-app_2.2.dtd'>

<web-app>
<session-config>
<session-timeout>
30
</session-timeout>
</session-config>
<welcome-file-list>
<welcome-file>index.htm</welcome-file>
</welcome-file-list>
<resource-ref>
<description>Test SQL DB Connection</description>
<res-ref-name>jdbc/dbTest</res-ref-name>
<res-type>javax.sql.DataSource</res-type>
<res-auth>Container</res-auth>
</resource-ref>
< /web-app>
```

Put this in CATALINA_HOME/webapps/dbTest/search.jsp:

```
<%@page contentType="text/html; charset=iso-8859-1" language="java"
import="java.sql.*,javax.sql.*,javax.servlet.*,
javax.servlet.http.*,javax.naming.*,org.firebirdsql.management.*"%>

<html>
< head>
< title>search.jsp</title>
< meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
< /head>

<body bgcolor="#CCCCFF" text="#000000"><p>
< /p>
< H1 align="center">Test Search</H1>

<%
// Edit the strings below as needed for your system.

String DB_SERVER_URL = "localhost";
int DB_SERVER_PORT = 3050;

// Use forward slashes here, even on Windows systems.
String DB_PATH = "c:/databases";

String DB_NAME = "test.gdb";
String DB_USER = "sysdba";
String DB_PASSWORD = "masterkey";
String DB_CONNECTION_STRING =
"jdbc:firebirdsql:"+DB_SERVER_URL+"/"+DB_SERVER_PORT+":"+DB_PATH+"/"+DB_NAME;

String lastName = request.getParameter("name");

// To help you debug, you can set a session attribute to values indicating
// the prograss through the web app, then use an HTTP monitor to see how far
// you have progressed.

// This stuff creates the database with the JMX management tools
// if it doesn't already exist.
FBManager fbManager = new FBManager();

fbManager.setServer(DB_SERVER_URL);
fbManager.setPort(DB_SERVER_PORT);

fbManager.start();

fbManager.createDatabase(DB_PATH + "/" + DB_NAME, DB_USER, DB_PASSWORD);

%>
Opening a connection with connection string: <%=DB_CONNECTION_STRING%><BR><BR>
```

```
< %

// Get a connection to the database from the dataSource.
DataSource ds=null;
Connection connRSFind=null;

try
{
Context ctx = new InitialContext();

if(ctx == null )
throw new Exception("Boom - No Context");

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

try
{
connRSFind = ds.getConnection();
}
catch (SQLException e)
{
System.out.println("getting new data source connection failed! Error: "+e+"\n");
}

}
catch (Exception e)
{
System.out.println("Could Not get data source, error: "+e+"\n");
}

// Back to HTML, show the user the LAST_NAME typed in.
%>
< p>
The user entered name is: <%=lastName%>
< p>
< %

String sqlString = "SELECT * FROM CUSTOMERS WHERE LAST_NAME = \'" + lastName +"\'";

// Switch back to html so we can print out the sqlString on the web page.
%>

The SQL String is: <%=sqlString%><p>

<%
PreparedStatement StatementRSFind=null;
ResultSet RSFind=null;
boolean resultException=false;
boolean rsReady = false;
```

```
try
{
StatementRSFind = connRSFind.prepareStatement(sqlString);

RSFind = StatementRSFind.executeQuery();
rsReady = RSFind.next();
}
catch(SQLException e1)
{
e1.printStackTrace(System.out);
resultException = true; // Set a flag so that we know there was an error, not just an empty result set.

%>
Could not find table, I'll try to add it: <%=e1.getMessage()%><BR>
< %
}

if (!rsReady || resultException)
{ // If there are no database entries with this last name, enter a few so the user has something to look at.
// Or, if there was an error, the table probably doesn't exist yet.

// try to create table CUSTOMERS, in case it doesn't exist yet.

if (resultException)
{// The table CUSTOMERS probably doesn't exist, so we create it here.
Statement statement2=null;

try
{
statement2 = connRSFind.createStatement();
}
catch(SQLException e2)
{
%>
Could not create statement2 with this connection! Check your database settings in search.jsp: <%
=e2.getMessage()%><BR>
< %
}

if (statement2 != null)
{
try
{ // try to crate the table CUSTOMERS.
sqlString = "CREATE TABLE CUSTOMERS(LAST_NAME VARCHAR(50), FIRST_NAME
VARCHAR(50))";
statement2.execute(sqlString);
statement2.close();
}
catch(SQLException e2a)
{
```

```
%>
Table CUSTOMERS already exists and we tried to create it again, or it could not be created: <%
=e2a.getMessage()%> <BR>
< %
}
}
} // Table CUSTOZMERS should exist now, if it didn't before or things are really messed up.
else
{
%>
First attempt at getting a result set produced an empty result set.<BR>
< %
}

// Insert some names into table CUSTOMERS. Either the table is new and empty, or there were
// no entries with the last name the user gave us, so we'll enter a few, so the user has
// something to look at.
Statement statement3=null;

try
{
statement3 = connRSFind.createStatement();
}
catch(SQLException e3)
{
%>
Could not create statement3 with this connection! Check your database settings in search.jsp: <%
=e3.getMessage()%><BR>
< %
}

if (statement3 != null)
{
try
{
sqlString = "INSERT INTO CUSTOMERS(LAST_NAME, FIRST_NAME) VALUES
(\'"+lastName+"\', 'Aaron')";
%>
Executing SQL: <%=sqlString%><BR>
< %
statement3.execute(sqlString);
sqlString = "INSERT INTO CUSTOMERS(LAST_NAME, FIRST_NAME) VALUES
(\'"+lastName+"\', 'Bob')";
%>
Executing SQL: <%=sqlString%><BR>
< %
statement3.execute(sqlString);
sqlString = "INSERT INTO CUSTOMERS(LAST_NAME, FIRST_NAME) VALUES
(\'"+lastName+"\', 'Calvin')";
%>
Executing SQL: <%=sqlString%><BR><BR>
```

```
< %
statement3.execute(sqlString);
statement3.close();
}
catch(SQLException e3a)
{
%>
We could not enter data in table CUSTOMERS for some reason: <%=e3a.getMessage()%> <BR>
< %
}
}

// try again to get a result set.
sqlString = "SELECT * FROM CUSTOMERS WHERE LAST_NAME = \'" + lastName +"\'";
StatementRSFind = connRSFind.prepareStatement(sqlString);
RSFind = StatementRSFind.executeQuery();
rsReady = RSFind.next();
}

int i = 0;

if (rsReady)
{
boolean done=false;
while (!done)
{
i++;
String RSFind_Last = (String) RSFind.getObject("LAST_NAME");
String RSFind_First = (String) RSFind.getObject("FIRST_NAME");
// display the names in the browser.
%>
Name <%=i%>: <%=RSFind_First%> <%=RSFind_Last%> <BR>
< %
done = !RSFind.next();
} //End while loop

RSFind.close();
}
else
{
%>
< BR>The result set was empty. Check to be sure database is running and settings in search.jsp are
correct.<BR>
< %

}
if (StatementRSFind != null)
StatementRSFind.close();

if (connRSFind != null)
connRSFind.close();
```

%>

</body>
< /html>

Put this in CATALINA_HOME/conf/server.xml before </Host>:

<Context path="/dbTest" docBase="dbTest"
debug="0" reloadable="true" crossContext="true">

<Logger className="org.apache.catalina.logger.FileLogger"
prefix="localhost_dbTest_log." suffix=".txt"
timestamp="true"/>

<Resource name="jdbc/dbTest"
auth="Container"
type="javax.sql.DataSource"/>

<ResourceParams name="jdbc/dbTest">

<parameter>
<name>factory</name>
<value>org.apache.commons.dbcp.BasicDataSourceFactory</value>
</parameter>

<parameter>
<name>removeAbandoned</name>
<value>true</value>
</parameter>

<parameter>
<name>removeAbandonedTimeout</name>
<value>300</value>
</parameter>

<parameter>
<name>logAbandoned</name>
<value>true</value>
</parameter>

<!-- Maximum number of dB connections in pool. Make sure you
configure your Firebird max_connections large enough to handle
all of your db connections. Set to 0 for no limit.
-->
<parameter>
<name>maxActive</name>
<value>100</value>
</parameter>

```
<!-- Maximum number of idle dB connections to retain in pool.
Set to 0 for no limit.
-->
<parameter>
<name>maxIdle</name>
<value>30</value>
</parameter>

<!-- Maximum time to wait for a dB connection to become available
in ms, in this example 10 seconds. An Exception is thrown if
this timeout is exceeded. Set to -1 to wait indefinitely.
-->
<parameter>
<name>maxWait</name>
<value>10000</value>
</parameter>

<!-- Firebird dB username and password for dB connections -->
<parameter>
<name>username</name>
<value>SYSDBA</value>
</parameter>
<parameter>
<name>password</name>
<value>masterkey</value>
</parameter>

<!-- Class name for JayBird JDBC driver -->
<parameter>
<name>driverClassName</name>
<value>org.firebirdsql.jdbc.FBDriver</value>
</parameter>

<!-- The JDBC connection url for connecting to your MySQL dB.
The autoReconnect=true argument to the url makes sure that the
mm.mysql JDBC Driver will automatically reconnect if mysqld closed the
connection. mysqld by default closes idle connections after 8 hours.
-->
<parameter>
<name>url</name>
<value>jdbc:firebirdsql:localhost/3050:/databases/test.gdb</value>
</parameter>
</ResourceParams>

</Context>
```

9. How do I use JayBird with JBuilder?

Last updated: 2003.11.04

return to top

Thanks to Marcelo Lopez Ruiz for the Jbuilder 6 Personal section.

Thanks to John B. Moore for the JBuilder 7 Enterprise notes.

Thanks to Sergio Samayoa for the JBuilder 9 notes.

JBuilder 9 Notes:

JBuilder 9 installation is similar to JBuilder 7.

1. Unpack JayBird onto your disk.

2. Create a Library for JayBird:

    Open JBuilder and pull down the "Tools" menu from the main menu bar. Select "Configure Libraries", this opens a dialog box.

    Click the "New" button.This opens the "New Library Wizard" dialog box.

    Type "JayBird" in the "Name" box.

    In "Location" select JBuilder.

    Click the "Add" button near the "Library Paths" list.

    Type in the path to the firebirdsql-full.jar file and click the "OK" button.

    Close the "New Library Wizard" dialog by clicking "OK".

3. Create a file "JayBird.config" in <JBuilder dir>\lib\ext containing the line (for example):

    addpath c:/JayBird/firebirdsql-full.jar

4. Restart JBuilder.

You can now use DataExpress. Add the "JayBird" library to your project. Note that the driver will not be listed in the driver selection combo boxes.

JBuilder 7 Enterprise Notes:

Create a file...

<JBuilderDir>\Lib\Ext\jaybird.config

Containing the following line...

addpath /JavaJars/jars/firebirdsql.jar

Added Libraries
Make two libraries..
FireBird_JCA_JDBC - firebirdsql.jar
FireBird_ext - mini-j2ee.jar

Note: the mini-j2ee is needed IF you get a ClassDefNotFound on javax/resources/ResourceException. This was encountered using Tomcat 3.3..

JBuilder 6 Personal

If you have any comments/suggestions, drop me an mail at marcelo.lopezruiz@xlnet.com.ar. If you have some spare time, check out the rest of the site at http://www.xlprueba.com.ar/marce/index.htm.

1. First, download the .zip file from SourceForge and unzip it to a temporary directory.

2. Read the release_notes.html document.

3. Unless you are doing funny things with your JDK, you use the default JDK installed with JBuilder6. In this case, you will need to download the javax.sql.* package, named the JDBC 2.0 Optional Package API (formerly known as the JDBC 2.0 Standard Extension API) from Sun, at http://java.sun.com/products/jdbc/download.html. Select the last option (option package binary), accept the license agreement, and download the .jar file.

4. Start JBuilder 6 Personal.

5. Create a new project (File | New Project...). Select a directory and a project name, then click Next. In step 2, select the Required Libraries tab - here, the required libraries will be registered with JBuilder and then added to the project.

6. Click the Add button. A list of libraries JBuilder is aware of will be shown. Click the New button to register the required libraries. Enter FireBird JCA-JDBC in the Name field, select JBuilder in the Location combo box, and click the Add button. Select the firebirdsql.jar you unzipped, and click OK. Click OK again to close the library. Verify that the new library is selected, and click OK to close the "Select One or More Libraries" dialog box.

7. Repeat the previous steps with the following libraries, found in the lib subdirectory of the binary distribution (except for the last package, which you downloaded from Sun).

concurrent.jar, named Concurrency Utilities

connector.jar, named Connector

jta-spec1_0_1.jar, named Java Transaction API

jdbc2_0-stdext.jar, named JDBC 2 Optional Package

8. Click Next, enter the desired project information, and click Finish.

9. Create a new application (File | New, then Application in the New tab). Enter the information you want for your application, and complete the wizard.

10. Click on the Design tab, and double-click on the button with the opening folder icon to create an event handler.

11. Type the following code for the event handler (note the small helper method above).

```
private void feedback(String text) {

statusBar.setText(text);

}

void jButton1_actionPerformed(ActionEvent e) {

// Hard-coded parameters

String pathToDatabase = "C:\\Program
Files\\Firebird\\examples\\EMPLOYEE.GDB";

String userName = "sysdba";

String password = "masterkey";

String sql = "SELECT * FROM EMPLOYEE";



// Load the FireBird driver.

Try {

Class.forName("org.firebirdsql.jdbc.FBDriver");

} catch(ClassNotFoundException cnfe) {
```

```
feedback("org.firebirdsql.jdbc.FBDriver not found");

return;

}


// Retrieve a connection.

Try {

Statement stmt = null;

ResultSet rst = null;

Connection conn = DriverManager.getConnection(

"jdbc:firebirdsql:localhost/3050:" + pathToDatabase, userName,
password);

try {

// Create a statement and retrieve its result set.

stmt = conn.createStatement();

rst = stmt.executeQuery(SQL);

// Show the result set through the standard output.

int columnCount = rst.getMetaData().getColumnCount();

int recordIndex = 0;

while(rst.next()) {

recordIndex++;

System.out.println("Record: " + recordIndex);

for (int i=1;i<=columnCount;i++) {

System.out.print(rst.getMetaData().getColumnName(i));

System.out.print(": ");

System.out.println(rst.getString(i));
```

```
}

}

} finally {

// close the database resources immediately, rather than waiting

// for the finalizer to kick in later

if (rst != null) rst.close();

if (stmt != null) stmt.close();

conn.close();

}

} catch(SQLException se) {

feedback(se.toString());

se.printStackTrace();

}

}
```

12. Type the following code at the beginning of the file, after the import statements.

import java.sql.*;

13. Run the application, click the button, and view the output.

14. For the morbidly curious, the following link provides an overview of the classes offered by the concurrent.jar package.

http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html

15. When using this library, note that there are still unimplemented things. To view the source for the Connection class and check which methods will return null values or an unimplemented exception, see the following URL. http://cvs.sourceforge.net/cgi-bin/viewcvs.cgi/firebird/client-java/src/org/firebirdsql/jdbc/FBConnection.java?rev=HEAD&content-type=text/vnd.viewcvs-markup

10- How do I use Blobs with JayBird?

return to top

Blobs are Binary Large OBjects. Blobs are used to store blocks of binary data of varying size in the database. An example would be a database table to store gif or jpeg images for a photo album program. Such a program could be written using the database to store file names of image files to be loaded from disk, but this could easily be corrupted if a file name were changed, a file inadvertently deleted, or the database moved to a system with different file naming conventions.

Blobs allow the binary image data to be stored, retrieved, backed up, and migrated like any other database data.

The Interclient JDBC driver did not implement many of the blob handling methods in the JDBC interfaces. So, the programmer had limited tools for using blobs with Interclient. That is no longer the case with JayBird. If you come across old code or help files dealing with blobs and Interbase/Interclient, be aware that it may be outdated and easier ways of dealing with blobs are available with JayBird.

Firebird BLOB fields are accessed through different JDBC interfaces depending on their subtype. For subtype <0, they are accessed as Blob fields. Subtype 1 is a LongVarChar, and Subtype 2 is LongVarBinary.

Below is some example code written for use with Interclient that may be helpful.

Storing BLOB Data:

The example given below shows a method that inserts an array of bytes into a BLOB column in the database. The PreparedStatement class is used so we can set the parameters independent of the actual SQL command string.

Inserting a BLOB

```
import java.io.*;

Import java.sql.*;

...

public void insertBlob( int rowid, byte[] bindata ) {

// In this example I'm assuming there's an open, active

// Connection instance called 'con'.

// This examples uses an imaginary SQL table of the following
```

```
// form:
//
// CREATE TABLE blobs (
// ROWID INT NOT NULL,
// ROWDATA BLOB,
//
// PRIMARY KEY (rowid)
// );

try {

ByteArrayInputStream bais = new ByteArrayInputStream(bindata);
String SQL = "INSERT INTO blobs ( rowid, rowdata ) VALUES ( ?, ? )";
PreparedStatement ps = con.prepareStatement(SQL);
// Set up the parameter index for convenience (JDBC column
// indices start from 1):
int paramindex = 1;
// Set the first parameter, the Row ID:
ps.setInt(paramindex++, rowid);
// Now set the actual binary column data by passing the
// ByteArrayInputStream instance and its length:
ps.setBinaryStream(paramindex++, bais, bindata.length);
// Finally, execute the command and close the statement:
ps.executeUpdate();
ps.close();
```

```
} catch ( SQLException se ) {
System.err.println("Couldn't insert binary data: "+se);
} catch ( IOException ioe ) {
System.err.println("Couldn't insert binary data: "+ioe);
} finally {
con.close();
}
}
```

Retrieving BLOB Data:

The example given below shows a method that retrieves an array of bytes from the database.

Selecting a BLOB

```
import java.io.*;
Import java.sql.*;
...
public byte[] selectBlob( int rowid ) {
// In this example I'm assuming there's an open, active
// Connection instance called 'con'.
// This examples uses an imaginary SQL table of the following
// form:
//
// CREATE TABLE blobs (
// ROWID INT NOT NULL,
// ROWDATA BLOB,
//
```

```
// PRIMARY KEY (rowid)

// );

try {

Statement sment = con.createStatement();

String SQL = "SELECT rowid, rowdata FROM blobs WHERE rowid = " +
rowid;

ResultSet rs = sment.executeQuery(SQL);

byte[] returndata = null;

if ( rs.next() ) {

try {

// The ByteArrayOutputStream buffers all bytes written to it

// until we call getBytes() which returns to us an array of bytes:

ByteArrayOutputStream baos = new ByteArrayOutputStream(1024);

// Create an input stream from the BLOB column. By default,
rs.getBinaryStream()

// returns a vanilla InputStream instance. We override this for
efficiency

// but you don't have to:

BufferedInputStream bis = new BufferedInputStream( rs.getBinaryStream
("fieldblob") );

// A temporary buffer for the byte data:

byte bindata[1024];

// Used to return how many bytes are read with each read() of the
input stream:

int bytesread = 0;

// Make sure its not a NULL value in the column:

if ( !rs.wasNull() ) {
```

```
if ( (bytesread = bis.read(bindata,0,bindata.length)) != -1 ) {
// Write out 'bytesread' bytes to the writer instance:
baos.write(bindata,0,bytesread);
} else {
// When the read() method returns -1 we've hit the end of the stream,
// so now we can get our bytes out of the writer object:
returndata = baos.getBytes();
}
}
// Close the binary input stream:
bis.close();
} catch ( IOException ioe ) {
System.err.println("Problem retrieving binary data: " + ioe);
} catch ( ClassNotFoundException cnfe ) {
System.err.println("Problem retrieving binary data: " + cnfe);
}
}
rs.close();
sment.close();
} catch ( SQLException se ) {
System.err.println("Couldn't retrieve binary data: " + se);
} finally {
con.close();
}
return returndata;
```

11- How do I use different character sets with JayBird?

return to top

Character Encodings:

Support for character encodings has just been added. This is easily accessible only from the FBDriver class. To use it, request a connection with a properties object containing a name-value pair lc_ctype=WIN1250 or other appropriate encoding name.

URL-encoded params are fully supported only when you get a connection from java.sql.DriverManager (using FBDriver class). For example: jdbc:firebirdsql://localhost//home/databases/sample.gdb?lc_ctype=UNICODE_FSS

It is also possible to set lc_ctype in a deployment descriptor by adding the following to your deployment descriptor:

```
<config-property>

<config-property-name>Encoding</config-property-name>

<config-property-type>java.lang.String</config-property-type>

<config-property-value>UNICODE_FSS</config-property-value>

</config-property>
```

12- How do I report bugs?

return to top

The developers attempt to follow the Firebird-Java@yahoogroups.com list. Join the list (see #16 below) 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.

You may also report bugs in the firebird bugtracker at:

http://sourceforge.net/tracker/?group_id=9028&atid=109028

13- How do I get sources from SourceForge using CVS?

return to top

CVS is a source code control system for managing access to, and synchronizing source code in a project that is being developed by multiple programmers. A full explanation of CVS is beyond the scope of this document, but information is available at sorceforge.net.

To get the project containing the source from SourceForge, install CVS on your system and use it to download the module named "client-Java". The user name is "anonymous". CVSROOT is :pserver:anonymous@sourceforge.net:client-Java The server is sourceforge.net. If you have secure shell capability and a SourceForge account use ":ext:" instead of ":pserver:", then type in your password when prompted.

If you use a GUI based CVS tool like WinCVS or TortoiseCVS on Windows, be sure that you also have your system set up to run CVS from the command line.

This is necessary because the newest version of JayBird uses libraries from the JBoss project to avoid licensing issues related to distributing some libraries from Sun. The Ant build scripts use command line CVS to check out the code needed from the JBoss project.

Tortoise is another good Windows based CVS tool.

Once you install these tools and download the source, you should be able to go into the client-Java folder (wherever you had CVS put it) and type "build" from the command line to build JayBird.

There is a file in the client-Java directory called "build.bat" for Windows systems and one called "build.sh" for Unix systems. Under certain UNIX shells you will have to type "build.sh" from the command line, the csh under Solaris, for example.

Tortoise CVS for Windows Users:

There is a very easy CVS program for Windows users called TortoiseCVS.

If you have access to a Windows box and don't want to learn CVS, this is an easy way to get the daily updates.

NOTE: because the build scripts use the command line version of CVS to download other projects, you need to also download WinCVS from SourceForge and install it before you can do a build.

You can get it at www.tortoisecvs.org. Once you install it on your windows box you can download a

CVS project by simply right clicking in a Windows Explorer subdirectory window and selecting CVS checkout.

A window will appear. Under module type in: client-java
Under server type: cvs.firebird.sourceforge.net
Under repository directory type: /cvsroot/firebird
Under username type: anonymous

Click OK and a window will appear and show the files being downloaded. A folder called client-java is created for the files.

Install WinCVS or another command line version of CVS.

It is a short download and has an installer so it is very easy to install under windows.

After installing you should be able to open a command prompt window and type cvs <return> and see an error from CVS.

If you try to build without command line CVS there will be an error. If you then install command line CVS and try a build, it will still fail unless you go into the client-java folder and delete the folder called thirdparty.

The thirdparty folder is where all the external stuff is. If it exists build will think everything is there, even though it isn't because command line CVS failed.

To build the release, double click the build.bat file in the client-java directory that is created.

Depending on the load on the SourceForge server, it might take several minutes to download the files.

Subsequent updates only download the changed files so things go quicker.

A window will open and show the build but will automatically close when the build is done. If you want to see the progress of the build (recommended) open a command prompt window cd to the client-java directory and type "build" and hit return.

After the build is done you can scroll through the output messages in the window.

To get to the jar files you need go into ...\client-java\output\lib.

With TortoiseCVS installed, the client-java directory icon looks like a fuzzy green file folder. To get a new daily build, right click on the folder and select CVS update.

If a particular build doesn't work properly, you can right click and select CVS... to get a popup menu, select update special to bring up a calendar. Pick the date that the code last worked for you and TortoiseCVS will roll back your code to the daily build for that day.

14- How can I participate in the development project?

return to top

Use the Firebird-Java@yahoogroups.com list to contact the developers about helping out (see #16 below). Perhaps the most crucial need right now is for more and better setup and usage guides, and to produce a reasonable section of the Firebird web site devoted to this driver. Work on this would be greatly appreciated.

15- Where can I get help?

return to top

The Firebird-Java mail list: Firebird-Java@yahoogroups.com (see #16 below).

The code for Firebird and this driver are on www.sourceforge.net under the Firebird project.

The SourceForge Firebird project home page www.firebirdsql.com or firebird.sourceforge.net.

The Firebird web site: www.IBPhoenix.com

The Sun Java web site: java.sun.com

The Jboss web site: www.jboss.org

The Jakarta Project Web Site for Ant, log4j, and Tomcat: jakarta.apache.org

---

16- How do I join the mailing list?

return to top

To join the Firebird-java mailing list, go to www.yahoogroups.com. Follow the instructions there for

joining a group. The name of the group is Firebird-java.

Please set your Yahoo settings to use plain text instead of HTML. All the ads drive some folks nuts and cause an automatic Internet hookup for some folks in Europe who have to pay for outgoing phone calls.

17- Are there any known bugs?

return to top

1) 575397: SERIALIZABLE is not isc_tpb_consistency - there is no final agreement between developers about mapping.

2) 630749: implement isc_info_sql_stmt_savepoint - will be implemented in JayBird 1.5.

3) 631090: Calling stored procedures - cannot be fixed with current Firebird implementation.

4) 638074: JUnit Errors in JayBird Build - known issue, not driver problem, but test case problem.

5) 645725: getBestRowIdentifier() not working - not yet implemented.

Monitor firebird-java@yahoogroups.com for daily updates.

You may report bugs in the firebird bugtracker at http://sourceforge.net/tracker/?group_id=9028&atid=109028

18- What JVM and JARs are needed to use JayBird?

return to top

If you use JVM 1.3.1 or higher the jars included in the distribution should have all the classes you need.

In addition to the jars in the distribution you may need jndi.jar with some applications that use JVMs before 1.3.1. It is available free from Sun at http://java.sun.com/products/jndi. DreamWeaver UltraDev 4.0 is an example of such an application. It uses a pre-1.3 JVM instead of the JVM installed in the system. The jndi.jar file must be added for JayBird to work with UltraDev.

Many development systems also use internal VM's. If they are pre-1.3.1 you may need to add jar files to the program's external library folder or to the system classpath to add classes that the older VM's lack.

If you see an errors about javax.naming.Referenceable, you are probably missing jndi.jar, for example.

Many app servers already include jndi.jar, so you may not need to download it.

All of the jars in the distribution are pretty small so it is not much of a problem to include them all. If it does become a problem and you want to eliminate classes that your app doesn't need, you will need to go through the source files to see what you can live without. Then you can use the jar utility to strip the unneeded files out of the jars.

The list below gives you a rough idea what is in each jar.

firebirdsql.jar	The primary jar file. Required.
mini-j2ee.jar	Contains DataSource and XA routines. Usually required.
mini-concurrent.jar	Concurrency routines. Required.
firebirdsql.rar	This is a J2EE server deployment file. Only needed for J2EE environments.
firebirdjmx.jar	Java Management Extension bean. Only needed if you want to play around with JMX.
jaas.jar	This is the Java Authentication and Authorization Service. It is a standard part of jdk 1.4. If you are using jdk 1.4 or above, you do not need this file.
firebirdsql-test.jar	Test routines, not needed for apps.
log4j-core.jar	Include if you want logging available. Some environments like application servers may already include this.

19- Why isn't JayBird in one jar file?

return to top

JayBird is now available as a single jar file: firebirdsql-full.jar.

Some of the code was originally from Sun and they didn't allow distribution by third parties without a license. To be scrupulously legal you had to download the Sun jars separately. This was a pain so equivalent classes from the JBoss Project were used instead. These could be distributed but needed to be updated from the JBoss project source, so they are in a separate jar (mini-j2ee.jar).

This combined with the fact that people are using the driver for everything from running a backing store for J2EE servers to simple database access in applets makes it hard to please everybody. So, multiple

jars are used and you can pick and choose from them as needed.

The concurrency files are also from another project and are in a separate jar with a subset of only the needed classes in the package (mini-concurrent.jar), see: http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html for details.

The firebirdsql-test.jar has routines that are used only for testing at build time.

The other jars are less frequently used and are packaged separately for easy removal.

If you are trying to build an app that you can distribute as an executable jar, you are free to tear apart the jars and roll your own.

20- What are some common errors?

return to top

1. Wrong URL

The most common error is having the wrong URL format for the database.

The two formats accepted are:

Standard format= jdbc:firebirdsql:[//host[:port]/]<database>

FB old format= jdbc:firebirdsql:[host[/port]:]<database>

2. May need to put ".gdb" in the URL

Most Firebird/Interbase database file names end in ".gdb". People frequently forget to put that in the URL.

3. May need to define "localhost"

If you use "localhost" as the host name in your URL, be sure that your server can resolve "localhost" to itself. A simple test is to execute "ping localhost" from the command line on the system that will be running JayBird. If the system can't find localhost then you or the administrator must configure DNS on it to do so, or you must use the IP number or the full domain name, i.e. host.domain.com.

4. Incorrect spelling in xml files.

When using JBoss or Tomcat or any environment that uses XML files for configuration, don't forget that case is important. Windows users often make this mistake because Windows is insensitive to case.

---

21. Is there any compliance and performance information available?

return to top

Compliance Tests:

The compliance was checked with jDataMaster, which include more than 1000 tests, excluding:

- CallableStatements
- Escape Syntax

FB type 4 driver pass all the tests excluding

- ResultSetMetaData.isReadOnly(i)
- ResultSetMetaData.isWritable(i)
- ResultSetMetaData.isDefinitivelyWritable(i)

The set/get tests don't include Blobs.

Firebird driver is the most JDBC compliant driver of those tested with this tool, the next one fails on 90 tests, including all of the most well known RDBMS.

Performance tests compared with Interclient

The results of the jDataMaster basic performance tests, are the following

Local test

Duron 800MHz with 512MB and FB 1.0 without forced Writes

	Interclient	Firebird Type 4	FB as % of Interclient
Insert 5000 records with autocommit	8510 ms	14307 ms	168 %
Read 5000 records with autocommit (x5)	3052 ms	3056 ms	100 %
Insert 5000 records with one transaction	6650 ms	5552 ms	83 %
Read 5000 records inside one transaction(x5)	3990 ms	3819 ms	95 %

Remote test

Client: Windows 2000 on Duron 800MHz with 512MB.
Server: Suse 7.2 on AMD 500MHz with 256MB and FB 1.0.

	Interclient	Firebird Type 4	FB as % of Interclient
Insert 5000 records with autocommit	16588 ms	17525 ms	106 %
Read 5000x5 records with autocommit	8082 ms	6247 ms	77 %
Insert 5000 records with one transaction	13009 ms	6311 ms	49 %
Read 5000x5 records inside one transaction	11639 ms	8670 ms	75 %

The time in each test is the average of 5 executions.

June-24-2002

22- What is the history of JayBird?

return to top

The idea of writing a Java translation of the C client library and using it in an all-Java driver originated in some discussions between David Jencks and Jim Starkey. Alejandro Alberola provided the initial translation of most of the client library functionality. David still doesn't understand how he did it so quickly. David Jencks wrote the JCA support and initial JDBC interfaces. Roman Rokytskyy wrote the CallableStatement support, many of the trickier JDBC details, the FBField implementation, and the character encoding support (among many other things). Several others have contributed bug fixes.

23- Where can I find an updated FAQ and Release Notes?

return to top

This FAQ is available in the JayBird download on SourceForge and on the project home page at:

http://firebird.sourceforge.net

The release notes are also in the download or can be read on the SourceForge Firebird site at:

http://sourceforge.net/projects/firebird

Find the entry "firebird-jca-jdbc-driver" in the table and click on the small book icon to the right to see the release notes.

The following link takes you to the JayBird FAQ at www.IBPhoenix.com:

http://www.ibphoenix.com/main.nfs?a=ibphoenix&l=;IBPHOENIX.FAQS;NAME='JayBird'

24- What tasks are left to do?

return to top

JayBird is a stable product but there are numerous additions and improvements that can be made. See the firebird-java discussion group to make suggestions or find out what others are working on.

1. Implement the unimplemented methods listed in question 2 above.
2. Implement the unimplemented optional features listed in section 2 above, where possible.
3. Implement JDBC 3.0 functionality.
4. Optimize Performance.

25- Where can I submit corrections/additions to the release notes and FAQ?

return to top

Please send corrections, suggestions, or additions to these Release Notes to Rick Fincher at rnf@tbird.com, or to the developers on the SourceForge site, or post them to the newsgroup at Firebird-Java@yahoogroups.com.

26- How do I turn off logging?

return to top

JayBird uses Log4J for logging. See: http://jakarta.apache.org/log4j/docs/index.html for more details about Log4J.

If you are using JayBird in JBoss, Tomcat, or other server environments, you can usually control the logging from within that application. See the documentation of those programs for more info.

In a stand-alone app you need to put an entry like the following in either a log4j.properties or xml file:

log4j.logger.org.firebirdsql=[DEBUG, INFO, WARN, ERROR, FATAL]

To tell Java where to find the Log4J configuaration file, you can use a flag like the following when you run your app:

java -Dlog4j.configuration=file:/c:/foobar.lcf myApp

27- How do I use JayBird with my development environment?

return to top

You can use JayBird in most Java development environments. Examples are NetBeans, Eclipse, SunOne Studio (formerly Forte for Java) among many others. Look at your IDE's (Integrated Development Environment) documentation under JDBC to find out how to install the drivers.

DreamWeaver Ultradev from MacroMedia can also use the JayBird driver to access Firebird databases. See question 31 below.

Many IDE's will let you develop web apps using servlets and JSP's (Java Server Pages). Where you put the jars when developing web apps may be different from the directories used for developing stand-alone apps. See your IDE's directions for details.

In SunOne Studio (formerly Forte for Java) and NetBeans (an open source version of the same program) a built-in version of Tomcat is used as the default JSP/servlet container for the IDE when developing web apps. The installation directories are different from those described in the Tomcat instructions above. See the help files included with these programs to find out where to install the jar files.

Typically, there is a directory called ext in the directory tree of the IDE where external jars are stored.

If all else fails when developing web apps, you can put the jar files in the WEB-INF/lib folder during development. When you deploy the web app you can remove theJayBird jars from WEB-INF/lib and create your war file for deployment. Just be sure you have the JayBird jars properly installed in the app server before using the web app.

28- Does JayBird provide any kind of security?

return to top

Not directly. Firebird and InterBase do not currently support secure connections internally. Therefore JayBird cannot. You must use SSL with an application server, or an IP tunneling program to secure your data in transit. These are both readily available. Secure connections are possible, you just have to use external software to get them.

One popular tunneling program is ZeBeDee. See http://www.ibphoenix.com/a471.htm for more information and examples of a secure Firebird setup.

There is a widely held misconception that Interclient/Interserver provide a secure connection. All they do is encode the password used to log in to the database. Data transactions after that are insecure unless the above means are used to secure them.

Another method of securing your data is to write your application as a web application that runs on an application server and displays in a browser. Most app servers have secure access capability. In that scenario Firebird and the app server are run on systems behind a firewall. The web browser makes secure calls to the app server. The app server then makes unsecure calls to the Firebird server using Jaybird. Since both the app server and the firebird server are behind the firewall, none of their transactions are visible to the outside world. Port 3050 should be blocked at the firewall in such a scenario to prevent unauthorized access or spoofing of Firebird.

This scenario will not prevent snooping from other systems behind your firewall, so for best security you may want to put your servers behind their own firewall.

If the application server and the Firebird server are on the same machine, packets never reach the network, so snooping is more difficult. Port 3050 will have to remain available, however, so the system will not be completely secure.

29- Does JayBird support dialect 1, 2, and 3 databases?

return to top

JayBird was written to support dialect 3 databases and may have problems with the earlier dialect 1 and 2 databases. Having said that, many people are running legacy dialect 1 and 2 databases without problems.

It is recommended that you migrate to dialect 3 databases as soon as practical and always use dialect 3 for new development.

Firebird 1.5 and later releases will not support dialect 1 and 2 databases.

30- Why don't arguments or ampersands (&) work in my URL?

return to top

If you are using an XML file to pass the parameters (Tomcat, etc.) you must use & or & instead of the ampersand (&) alone.

In XML, the & symbol is used to make references to entities, so it is treated like a special character. The XML parser expects to find something like &entityname; That is why it must end with the ';' delimiter.

If you see an error message like:

org.apache.commons.digester.Digester fatalError
SEVERE: Parse Fatal Error at line 62 column 94: The reference to entity "lc_ctype" must end with the ';' delimiter.
org.xml.sax.SAXParseException: The reference to entity "lc_ctype" must end with the ';' delimiter.

This is what is causing it.

Instead of: ...my.gdb?sql_role_name=guest&lc_ctype=WIN1251

Use: ...my.gdb?sql_role_name=guest&lc_ctype=WIN1251

31- Why do I see "????" instead of the correct characters?

return to top

This problem is seen a lot with Tomcat and alternate character sets. Resin and Jetty seem to work OK. There are two possible sources of this error: a) database contains incorrect data or b) The application handles this incorrecly.

The chain of translations is like this:

database byte[] -> client-side byte[] (using lc_ctype) -> String instance (using corresponding Java encoding) -> HTML page encoding (depends on the implementation).

With lc_ctype=CP1254, for example, some Java code may use String.getBytes() instead of String.getBytes("Cp1254"), the result depends on the default encoding of the JVM. If the default encoding is not Cp1254, it will translate the correct unicode string into "????".

InterClient works because it receives the byte array from InterServer in the encoding that it asked (Cp1254). The String is constructed as "new String(byte[])", and not as "new String(byte[], "Cp1254")". Later "str.getBytes()" will return bytes from that string without doing any unicode/charset translation.

MySQL and JayBird work correctly, InterClient has a bug that compensates for the bug in the code.

To fix it, try to run the JVM with -Dfile.encoding=Cp1254, so the default encoding will be correct.

32- Can I use JayBird with Open Office?

return to top

Yes. Use the standard JayBird settings for the driver class and URL. You must have Java installed on your system as well.

For example:

JDBC Driver class- org.firebirdsql.jdbc.FBDriver
URL- jdbc:firebirdsql:host.domain.com/3050:/(pathtodb)/(dbfile).gdb

Make sure the locations of the driver jars are in the classpath box in the menu: tools->options->openoffice.org->security.

You can then use the data table tool in Open Office to extract rows of the database directly into your spreadsheets and use the data for any of the Open Office apps.

33- How do I create a new database with JayBird?

return to top

You can use the methods in FBManager in the package org.firebirdsql.management to easily create a new database in your programs. After creation you can connect to the database and use the standard SQL calls to create tables and populate them.

This is an example:

```
String DB_SERVER_URL = "localhost";
int DB_SERVER_PORT = 3050;
String DB_PATH = "c:/database";
String DB_NAME = "test.gdb";
String DB_USER = "sysdba";
String DB_PASSWORD = "masterkey";

fbManager.setServer(DB_SERVER_URL);
fbManager.setPort(DB_SERVER_PORT);

fbManager.start();

fbManager.createDatabase(DB_PATH + "/" + DB_NAME, DB_USER,
DB_PASSWORD);
```

After you have created the database you can set its default character set with a command like:

```
UPDATE rdb$database SET rdb$character_set_name='ISO8859_1'
```

34- How do I use JayBird with Windows 95/98?

return to top

You must upgrade to Winsock 2.0 to use JayBird with Windows 95. See the Microsoft web site for details. Windows 98 uses Winsock 2.0 by default, but in a very few cases when Windows is upgraded from 95 to 98, Winsock 2.0 is not installed. See http://www.sockets.com/winsock2.htm for more info.

35- Why does building the CVS code fail?

return to top

One of the primary reasons is that a command line version of CVS is not installed. If you see a message about a cvs failure, that is probably the reason. Type cvs<return> at a command prompt. You should see error messages from CVS. If you get a message from the system that CVS isn't found, it isn't properly installed, or you need to set up a path to it for your shell.

If you see a message about missing files in the thirdparty directory, you probably tried to build without command line CVS. This creates a directory called thirdparty, but CVS can't put the necessary files in it. The build script assumes the files are there if the thirdparty directory exists, so the build fails. Delete the thirdparty directory, make sure command line CVS is working and try build again.

36- Why aren't my connections ever returned to the pool?

return to top

Be sure that you close all statements and result sets associated with a connection before closing a pooled connection. Closing a pooled connection doesn't actually close it but returns it to the pool. If it has active statements or result sets it can't be reused.

Since closing a non-pooled connection automatically closed statements and result sets associated with it, a lot of code has been written that doen not explicitly close statements and result sets. If you simply substitute a pooled connection in old code, it may never properly close statements and result sets.

37- How do I use JayBird with DreamWeaver UltraDev?

return to top

UltraDev uses an internal Java JVM. The JVM in UltraDev 4 is a pre-1.3 version so you will need to obtain the jndi.jar file and install it. The jndi.jar file is available from Sun at: http://java.sun.com/products/jndi.

You must install the jars firebirdsql.jar, mini-concurrent.jar, mini-j2ee.jar, and jndi.jar into the UltraDev JDBCDrivers folder. On Windows machines this is typically at:

C:\Program Files\Macromedia\Dreamweaver UltraDev 4\Configuration\JDBCDrivers

See the UltraDev documentation for instructions on installing on the Macintosh. This has not been tested by me but should work OK.

To set up a connection pull down UltraDev's "Modify" menu and select "Connections...". A dialog box will pop up. Select "New" and you will see a list of JDBC drivers. Select "Custom JDBC Connection".

Another dialog box will pop up. Type in the a name for your new connection, the driver name, database URL string, username, and password. Click the "Test" button to be sure that you can connect to the database.

To add JayBird to the list JDBC drivers in the menu go to the configuration/connections/jsp folder. Then open the Mac or Win folder, depending on your OS. Under Windows this is typically at:

C:\Program Files\Macromedia\Dreamweaver UltraDev 4\Configuration\Connections\JSP\Win

Copy one of the existing html files (.htm under Windows) into a new file with a name like "jaybird_conn.htm". Open your new file and change the following entries to the proper info for Jaybird as in the example below:

<TITLE>JayBird JDBC Driver for Firebird</TITLE>

...

```
//Global vars
var DEFAULT_DRIVER = "org.firebirdsql.jdbc.FBDriver";
var DEFAULT_TEMPLATE = "jdbc:firebirdsql:localhost/3050:[database name]";
var MSG_JayBirdDriverNotFound = "JayBird Driver not found on machine"
var MSG_DriverNotFound = MSG_JayBirdDriverNotFound;
var FILENAME = "jaybird_conn.htm"
```

Save this file and restart UltraDev. When you follow the procedure above for creating a new connection you shoul now see "JayBird JDBC Driver for Firebird" in the list of JDBC drivers.

38- Why do I see a "javax.naming.Referenceable" error?

return to top

If you see this error you probably need to include jndi.jar in your classpath, or your application's external libraries directory. You will need this jar if you are using JayBird with an application that uses an internal JVM with a version prior to 1.3.

39- How do I use stored procedures with JayBird?

return to top

Jaybird does not fully support escaped syntax for procedure calls (output parameters are not supported). However if you use native Firebird syntax, stored procedures are fully supported.

Native Firebird syntax:

EXECUTE PROCEDURE proc_name(?, ?, ?, ...) for executable procedures;
SELECT * FROM proc_name(?, ?, ...) for selectable procedures.

{call proc_name(?, ?, ...)} is translated into EXECUTE PROCEDURE
proc_name(?, ?, ?, ...) without any analysis. This means that:

a) It is not usable for selectable stored procedures;
b) Positions of output params are independent of input params (in JDBC they're not) and start with 1.

When you use native syntax, stored procedures can be called both from PreparedStatement and from CallableStatement, in case of escaped syntax - only from CallableStatement.

Examples

The following example is a stripped-down version of code taken from the project source code tests in the directory:

client-java/src/test/org/firebirdsql/jdbc/TestFBCallableStatement.java

It creates a stored procedure called "factorial" on the Firebird server that recursively calls itself to calculate factorials. The method testRun shows how the stored procedure would be called from Java.

To make this work you would have to add code to open the java.sql.Connection called connection in this code snippet. You would also have to close the connection, statements, etc. when you are done

```
public class TestFBCallableStatement {
  public static final String CREATE_PROCEDURE = ""
    + "CREATE PROCEDURE factorial(number INTEGER, mode INTEGER) RETURNS (result
INTEGER) "
    + "AS "
    + " DECLARE VARIABLE temp INTEGER; "
    + "BEGIN "
    + " temp = number - 1; "
    + " IF (NOT temp IS NULL) THEN BEGIN "
    + " IF (temp > 0) THEN "
    + " EXECUTE PROCEDURE factorial(:temp, 0) RETURNING_VALUES :temp; "
    + " ELSE "
    + " temp = 1; "
    + " result = number * temp; "
    + " END "
    + " IF (mode = 1) THEN "
    + " SUSPEND; "
    + "END";
```

```
   public static final String DROP_PROCEDURE =
      "DROP PROCEDURE factorial;";

  public static final String SELECT_PROCEDURE =
     "SELECT * FROM factorial(?, 1)";

  public static final String EXECUTE_PROCEDURE =
     "{call factorial(?, 0)}";

   private java.sql.Connection connection;

  public TestFBCallableStatement(String testName) {
     //
   }

   public void testRun() throws Exception {

     // set up the connection before you call this
     java.sql.CallableStatement cstmt = connection.prepareCall(EXECUTE_PROCEDURE);
     try {
         cstmt.setInt(1, 5);
         cstmt.execute();
         int ans = cstmt.getInt(1);
         assertTrue("got wrong answer, expected 120: " + ans, ans == 120);
     } finally {
        cstmt.close();
     }

     java.sql.PreparedStatement stmt = connection.prepareStatement(SELECT_PROCEDURE);
     try {
         stmt.setInt(1, 5);
         java.sql.ResultSet rs = stmt.executeQuery();
         assertTrue("Should have at least one row", rs.next());
         int result = rs.getInt(1);
         assertTrue("Wrong result: expecting 120, received " + result, result == 120);

         assertTrue("Should have exactly one row.", !rs.next());
         rs.close();
     } finally {
         stmt.close();
     }

  // Tear down the connection here when you are done.
   }
}
```

40- Why are VARCHAR strings sometimes padded?

return to top

Till v.6.5 InterBase sent VARCHAR columns padded to the full length with spaces (0x20) similar to CHAR. In 6.5 this was changed by Borland so that VARCHARs are no longer padded, as expected. IB 6.0 and FB 1.0 pads VARCHARs too. However, if FB 1.0 is accessed via JayBird, it does NOT pad VARCHARs and sends them correctly.

From one of the posts in the Firebird devel list this issue seems to appear when using the gds32.dll under Windows but not with the INET protocol.

41- Can I use CachedRowSet with JayBird?

return to top

Yes, but third party code must be used. CachedRowSet allows you to store a copy of a rowSet and disconnect from the Firebird server for processing. This is useful because it allows you to release a connection immediately after getting a rowSet instead of holding it open while the rowSet is processed. That allows the server to run more efficiently

Two third-party implementations have been tested and seem to work the same.

One is from Oracle: OracleCachedRowSet -- it doesn't require an Oracle database but does require the Oracle JDBC jar file. You need both ocrs12.zip and ojdbc14.jar.

See: http://otn.oracle.com/software/tech/java/sqlj_jdbc/content.html

Another is an open source project called jxutil: XDisconnectedRowSet -- only requires one jar file, jxRowSet-0.8a.jar.

See: http://sourceforge.net/projects/jxutil/

Depending on the intended use, it would be appropriate to study the licenses both implementations come under.
Jxutil is LGPL (Limited Gnu Public License) software, while OracleCachedRowSet uses the Oracle Technology Network Development and Distribution License. See: http://otn.oracle.com/software/htdocs/distlic.html.

42- How do I pass Database Parameter Buffer (DPB) parameters?

return to top

If you use FBWrappingDataSource, you specify DPB parameters in FBConnectionRequestInfo.

If you are using java.sql.DriverManager you can pass any DPB parameter by stripping the "isc_dpb_" from the DPB parameter name and putting it in the URL. In case of roles and character encoding this could be:

...my.gdb?sql_role_name=guest&lc_ctype=WIN1251

See the InterBase API Guide Chapter 4, "Working with Databases" for more details. The following table is a listing of DPB parameters.

DPB Parameter	Description	Length	Values
isc_dpb_activate_shadow	Directive to activate the database shadow, which is an optional, duplicate, in-sync copy of the database	1	1 (Ignored) 0 (Ignored)
isc_dpb_damaged	Number signifying whether or not the database should be marked as damaged	1	1 = mark as damaged 0 = do not mark as damaged
isc_dpb_dbkey_scope	Scope of dbkey context	1	0 limits scope to the current transaction, 1 extends scope to the database session
isc_dpb_delete_shadow	Directive to delete a database shadow that is no longer needed	1	1(Ignored) 0 (Ignored)
isc_dpb_encrypt_key	String encryption key	up to 255 characters	String containing key
isc_dpb_force_write	Specifies whether database writes are synchronous or asynchronous.	1	0 = asynchronous; 1 = synchronous
isc_dpb_lc_ctype	String specifying the character set to be utilized	number of bytes in string	String containing character set name
isc_dpb_lc_messages	String specifying a language-specific message file	Number of bytes in string	String containing message file name
	Specifies whether or		

isc_dpb_no_reserve	not a small amount of space on each database page is reserved for holding backup versions of records when modifications are made; keep backup versions on the same page as the primary record to optimize update activity.	1	0 (default) = reserve space 1= do not reserve space
isc_dpb_num_buffers	Number of database cache buffers to allocate for use with the database; default=75	1	Number of buffers to allocate
isc_dpb_password	String password	up to 255 characters	String containing password
isc_dpb_password_enc	String encrypted password	up to 255 characters	String containing password
isc_dpb_sql_role_name	String Role Name	up to 255 characters	String containing Role Name
isc_dpb_sys_user_name	String system DBA name	up to 255 characters	String containing SYSDBA name
isc_dpb_user_name	String user name	up to 255 characters	String containing user name

---

43- What is a good validation query to use with JayBird?

return to top

Some programs and web applications need a query that they can send to the database to verify that it is online and connected, or to insure that a connection hasn't timed out. That is called a validation query. A good one for Firebird/JayBird is:

SELECT CAST(1 AS INTEGER) FROM rdb$database

This select will always succeed if the connection is still alive and will have exactly one row in the result set..

44- How can I get the key of the new record I just inserted?

return to top

Sometimes it is necessary to get the Primary Key of a record just inserted so that it can be used as the relation in an insert into another table. If you are using a generator and trigger to auto-generate keys and insert them in new records, you do not know what the key is.

A solution is to get a key from the generator before you do the insert, then use that key to do the insert in both tables.

Getting a new key is simple but you have to modify the trigger to skip getting a new key if a key is supplied in the insert statement. In this way the database will still auto-generate keys if you like or it will use keys you supply.

To get a new primary key to insert with use:

```
SELECT gen_id(my_generator, 1) FROM RDB$DATABASE
```

This fires your generator and returns the new key in the result set. The following sets up the trigger so that it will auto create the primary key if you don't pass it one in the insert, but will also use one that you give it (generated above) without generating another one. So insert will work either way.

Create the trigger:

```
active before insert
as
begin
if (new.YourPk is null) then
new.YourPk = gen_id(my_generator, 1);
end
```

Trigger generation is outside of transaction control (the only thing that is!) so it is guaranteed to be 100% atomic. In other words, this will always safely give you a unique key.

You may get better performance by creating a pool of keys to use for inserts. You could generate a block of keys at startup, then your inserts will go quicker.

To create a block of 100 keys, for example, use:

```
SELECT gen_id(my_generator, 100) FROM RDB$DATABASE
```

This will return the last number in the block. If that number is myKey, you can use numbers from (myKey-99) through myKey.

45-How do I set the default character set of a database?

return to top

After you have created the database (See FAQ item 33) you can set its default character set with a command like:

```
UPDATE rdb$database SET rdb$character_set_name='ISO8859_1'
```

Be sure to do this before creating tables. Tables created before the default charset is changed will have the old default charset and will not be affected by the change, unless explicitly set to something else.

You must also open the connection to the database with lc_ctype set to the same character set as the database for character translations to work properly.

46- Can you explain how character sets work?

return to top

Character handling with JayBird can be confusing because the Java VM, Firebird database, browser, and JayBird Connection all have a charset associated with them.

Also, the Firebird server attempts to transliterate the internal charset of a database to the charset specified in the connection. JayBird also attempts to translate the JVM charset to the Firebird server charsets specfied in the connection.

With Firebird the character encoding of the text data stored in the database is set when the database is created. That applies to the char and varchar columns, and type 1 blobs (text blobs). You can override the default charset for columns with the appropriate SQL commands when the columns are created. Be careful if you do this or you may end up with two columns in the same table that you can't read with the same connection.

The only situation when this problem can happen is when you have a table with columns that have the "NONE" character set and some other character set ("UNICODE_FSS", "WIN1252", etc). The server tries to convert characters from the encoding specified for the column into the encoding specified for connection. The "NONE" character set allows only one-way conversion: from <any> to "NONE". In this case server simply returns you bytes written in the database. So if you have table:

```
CREATE TABLE charset_table(
col1 VARCHAR(10) CHARACTER SET WIN1252,
```

```
col2 VARCHAR(10) CHARACTER SET NONE
)
```

you will not be able to modify both columns in the same SQL statement, and it does not matter whether you use "NONE", "WIN1252" or " UNICODE_FSS" for the connection.

The only possible way to solve this problem is to use character set "OCTETS". This is some kind of artificial character set, similar to "NONE" (data are written and read as byte arrays), however there exist bi-directional translation rules between any character set (incl."NONE") and "OCTETS". You can specify "OCTETS" for connection and then decode byte arrays you receive from the server yourself, the driver will do byte-array-to-string conversion incorrectly, since it does not get a hint about the character set from the server.

Let's say your program or web app prompts the user to type a string. If the user types "abc" into a Java text box in your app, or into a text box in a browser for a web app, Java creates a string for that 3 character string in Unicode. So the string would actually have 9 bytes in it- three Unicode characters of three bytes each.

INSERTing this in a text column in the database would result in nine bytes being inserted without translation.

Note: You have to pass correct unicode strings to the driver. What is "correct unicode" string? It is easier to explain what is not a correct unicode string.

Let's assume you have normal text file in WIN1251 encoding. In this case cyrillic characters from the unicode table (values between 0-65535) are mapped into the characters with values 0-255. However, your regional settings say that you're in Germany. This means that file.encoding will be set to Cp1252 on JVM start. If you now open the file and construct a reader without specifying that character encoding is Cp1251, Java will read the file and construct your strings. However all cyrillic characters will be replaced by characters from the Cp1252 encoding that have the same number representation as the cyrillic ones.

These strings are valid unicode strings, however their content is not the content you read from the file. Interestingly enough, if you write such strings back into the file, and open it in some text editor saying that this is WIN1251 text, you will see correct text.

If you open your connection to the database without specifying a character set (lc_ctype) it defaults to NONE and no translation is done. So when you SELECT the previously inserted data from the database and display it in your program you get the same string you entered, right? Well, not necessarily.

You will get a string with the same nine bytes in it that were stored, but if the user getting that string from the database has a different default charset in his Java VM those bytes will display differently.

The JVM usually picks up its locale dependent character encoding from the underlying operating system, but it can also be set when you invoke the JVM by using -Dfile.encoding=Cp1252, for example. If you attempt to display characters that aren't in your default JVM encoding they apear as '?'.

The only way to insure you always get back what you put in is to create the database with a charset and set lc_ctype to the same charset when you open the connection to that database.

If want to use charsets other than NONE and you have lots of data in databases with a charset of NONE, you may have to set up a new database with a different charset and use a data pump to transfer data over, or write a small program to do the transfer.

Using UNICODE_FSS works well nearly everywhere but may increase the size of your databases if you have lots of text because Unicode uses characters up to 3 bytes long.

There are some limitations regarding UNICODE_FSS character set: there's only one collation, where strings are sorted by the natural order, and not collation rules for different languages; there are some issues when converting them to upper case, etc. More information on these anomalies can be found in the Firebird-Support group.

Again, the default charset for Firebird is NONE. The Firebird server does no translation with this charset.

If your database has a charset of NONE and you set a charset type on the connection (lc_ctype) that is not NONE, you can write to the database but you can't read from it without getting the "Cannot transliterate between character sets" exception.

Let's follow a string as it gets inserted into the database and later selected from the database. For this example we will set the database charset to NONE.

See the freely available Interbase 6 PDF manuals "Data Definition Guide" for a list of charsets available.

The WIN125X or ISO8859_1 charsets may be a good choice for you if you need the non-English characters but want the compactness of the 1 byte characters. With these char sets you can specify many different national language collation orders in the ORDER BY clause of the SELECT statement.

Let's look at the same example above, but this time we will insert into a database that has been created with a charset of WIN1251.

When you open the connection to the database you set the lc_ctype=WIN1251. Then insert the string 'abc' into the appropriate column. JayBird has to take the Unicode encoded Java String "abc"and convert it to WIN1251 format and send it to the database server for insertion. Since the database is already in WIN1251 format, the server does not have to translate. When the string is read back from the database it is converted back to the Java VM format by JayBird.

It is also possible to set an lc_ctype in a connection that is different from the charset of the database. This lets the database server do the translating from one charset to another. This is a feature of the Firebird server that lets programming languages or programs that require specific character formats to connect to the database without requiring the data to be stored in that format.

You can also avoid problems by using java.sql.PreparedStatement instead of java.sql.Statement and not building SQL strings out of concatenated Java strings. For example:

```
String sqlString, firstName="John", lastName="O'Neal";

sqlString = "INSERT INTO nameTable (LNAME, FNAME) VALUES
('"+lastName+"','"+firstName+"')";
```

```
Statement stmt = connection.createStatement();
int insertedRows = stmt.executeUpdate(sqlString);
```

The problem here is that if the user types in data for these strings you might end up with illegal characters, or the translation might not be correct.

In the example above, the following illegal SQL string would be generated and cause an exception to be thrown because of the apostrophe in O'Neil:

```
INSERT INTO nameTable (LNAME, FNAME) VALUES('O'Neal', 'John')
```

To avoid this, use a prepared statement like in the example below.

```
PreparedStatement stmt = connection.prepareStatement("INSERT INTO
nameTable(LNAME,FNAME) VALUES(?, ?)");

stmt.setString(1, lastName);
stmt.setString(2, firstName);

int insertedRows = stmt.executeUpdate();

if (insertedRows != 1)
throw new MyInsertFailedException("Could not insert data");
```

47- Can you give me some code examples?

return to top

Yes we can. There are two examples below, a driver example and a DataSource example that are extensively commented.

Driver Example:

```
// Original version of this file was part of InterClient 2.01
examples

//

// Copyright InterBase Software Corporation, 1998.

// Written by com.inprise.interbase.interclient.r&d.PaulOstler :-)

//

// Code was modified by Roman Rokytskyy to show that Firebird JCA-
JDBC driver
```

```
// does not introduce additional complexity in normal driver usage
scenario.

//

// A small application to demonstrate basic, but not necessarily
simple, JDBC features.

//

// Note: you will need to hardwire the path to your copy of
employee.gdb

// as well as supply a user/password in the code below at the

// beginning of method main().



public class DriverExample

{

// Make a connection to an employee.gdb on your local machine,

// and demonstrate basic JDBC features.

// Notice that main() uses its own local variables rather than

// static class variables, so it need not be synchronized.

public static void main (String args[]) throws Exception

{

// Modify the following hardwired settings for your environment.

// Note: localhost is a TCP/IP keyword which resolves to your local
machine's IP address.

// If localhost is not recognized, try using your local machine's
name or

// the loopback IP address 127.0.0.1 in place of localhost.

String databaseURL =
"jdbc:firebirdsql:localhost/3050:c:/database/employee.gdb";

String user = "sysdba";
```

```
String password = "masterkey";

String driverName = "org.firebirdsql.jdbc.FBDriver";


// As an exercise to the reader, add some code which extracts
databaseURL,

// user, and password from the program args[] to main().

// As a further exercise, allow the driver name to be passed as well,

// and modify the code below to use driverName rather than the
hardwired

// string "org.firebirdsql.jdbc.FBDriver" so that this code becomes

// driver independent. However, the code will still rely on the

// predefined table structure of employee.gdb.


// See comment about closing JDBC objects at the end of this main()
method.

System.runFinalizersOnExit (true);


// Here are the JDBC objects we're going to work with.

// We're defining them outside the scope of the try block because

// they need to be visible in a finally clause which will be used

// to close everything when we are done.

// The finally clause will be executed even if an exception occurs.

java.sql.Driver d = null;

java.sql.Connection c = null;

java.sql.Statement s = null;

java.sql.ResultSet rs = null;
```

```
// Any return from this try block will first execute the finally
clause

// towards the bottom of this file.

try {



// Let's try to register the Firebird JCA-JDBC driver with the driver
manager

// using one of various registration alternatives...

int registrationAlternative = 1;

switch (registrationAlternative) {



case 1:

// This is the standard alternative and simply loads the driver
class.

// Class.forName() instructs the java class loader to load

// and initialize a class. As part of the class initialization

// any static clauses associated with the class are executed.

// Every driver class is required by the JDBC specification to
automatically

// create an instance of itself and register that instance with the
driver

// manager when the driver class is loaded by the java class loader

// (this is done via a static clause associated with the driver
class).

//

// Notice that the driver name could have been supplied dynamically,

// so that an application is not hardwired to any particular driver
```

```
// as would be the case if a driver constructor were used, eg.

// new org.firebirdsql.jdbc.FBDriver().

try {

Class.forName ("org.firebirdsql.jdbc.FBDriver");

}

catch (java.lang.ClassNotFoundException e) {

// A call to Class.forName() forces us to consider this exception :-
)...

System.out.println ("Firebird JCA-JDBC driver not found in class
path");

System.out.println (e.getMessage ());

return;

}

break;



case 2:

// There is a bug in some JDK 1.1 implementations, eg. with Microsoft

// Internet Explorer, such that the implicit driver instance created
during

// class initialization does not get registered when the driver is
loaded

// with Class.forName().

// See the FAQ at http://java.sun.com/jdbc for more info on this
problem.

// Notice that in the following workaround for this bug, that if the
bug

// is not present, then two instances of the driver will be
registered
```

```
// with the driver manager, the implicit instance created by the
driver

// class's static clause and the one created explicitly with
newInstance().

// This alternative should not be used except to workaround a JDK 1.1

// implementation bug.

try {

java.sql.DriverManager.registerDriver (

(java.sql.Driver) Class.forName
("org.firebirdsql.jdbc.FBDriver").newInstance ()

);

}

catch (java.lang.ClassNotFoundException e) {

// A call to Class.forName() forces us to consider this exception :-
)...

System.out.println ("Driver not found in class path");

System.out.println (e.getMessage ());

return;

}

catch (java.lang.IllegalAccessException e) {

// A call to newInstance() forces us to consider this exception :-
)...

System.out.println ("Unable to access driver constructor, this
shouldn't happen!");

System.out.println (e.getMessage ());

return;

}

catch (java.lang.InstantiationException e) {
```

```
// A call to newInstance() forces us to consider this exception :-
)...

// Attempt to instantiate an interface or abstract class.

System.out.println ("Unable to create an instance of driver class,
this shouldn't happen!");

System.out.println (e.getMessage ());

return;

}

catch (java.sql.SQLException e) {

// A call to registerDriver() forces us to consider this exception :-
)...

System.out.println ("Driver manager failed to register driver");

showSQLException (e);

return;

}

break;



case 3:

// Add the Firebird JCA-JDBC driver name to your system's
jdbc.drivers property list.

// The driver manager will load drivers from this system property
list.

// System.getProperties() may not be allowed for applets in some
browsers.

// For applets, use one of the Class.forName() alternatives above.

java.util.Properties sysProps = System.getProperties ();

StringBuffer drivers = new StringBuffer
("org.firebirdsql.jdbc.FBDriver");
```

```
String oldDrivers = sysProps.getProperty ("jdbc.drivers");

if (oldDrivers != null)

drivers.append (":" + oldDrivers);

sysProps.put ("jdbc.drivers", drivers.toString ());

System.setProperties (sysProps);

break;


case 4:

// Advanced: This is a non-standard alternative, and is tied to

// a particular driver implementation, but is very flexible.

//

// It may be possible to configure a driver explicitly, either thru

// the use of non-standard driver constructors, or non-standard

// driver "set" methods which somehow tailor the driver to behave

// differently from the default driver instance.

// Under this alternative, a driver instance is created explicitly

// using a driver specific constructor. The driver may then be

// tailored differently from the default driver instance which is

// created automatically when the driver class is loaded by the java
class loader.

// For example, perhaps a driver instance could be created which

// is to behave like some older version of the driver.

//

// d = new org.firebirdsql.jdbc.FBDriver ();

// DriverManager.registerDriver (d);
```

```
// c = DriverManager.getConnection (...);

//

// Since two drivers, with differing behavior, are now registered
with

// the driver manager, they presumably must recognize different JDBC

// subprotocols. For example, the tailored driver may only recognize

// "jdbc:interbase:old_version://...", whereas the default driver
instance

// would recognize the standard "jdbc:interbase://...".

// There are currently no methods, such as the hypothetical
setVersion(),

// for tailoring an Firebird JCA-JDBC driver so this 4th alternative
is academic

// and not necessary for Firebird JCA-JDBC driver.

//

// It is also possible to create a tailored driver instance which

// is *not* registered with the driver manager as follows

//

// d = new org.firebirdsql.jdbc.FBDriver ();

// c = d.connect (...);

//

// this is the most usual case as this does not require differing

// JDBC subprotocols since the connection is obtained thru the driver

// directly rather than thru the driver manager.

d = new org.firebirdsql.jdbc.FBDriver ();

}
```

```
// At this point the driver should be registered with the driver
manager.

// Try to find the registered driver that recognizes interbase
URLs...

try {

// We pass the entire database URL, but we could just pass
"jdbc:interbase:"

d = java.sql.DriverManager.getDriver (databaseURL);

System.out.println ("Firebird JCA-JDBC driver version " +

d.getMajorVersion () +

"." +

d.getMinorVersion () +

" registered with driver manager.");

}

catch (java.sql.SQLException e) {

System.out.println ("Unable to find Firebird JCA-JDBC driver among
the registered drivers.");

showSQLException (e);

return;

}



// Advanced info: Class.forName() loads the java class for the
driver.

// All JDBC drivers are required to have a static clause that
automatically

// creates an instance of themselves and registers that instance

// with the driver manager. So there is no need to call
```

```
// DriverManager.registerDriver() explicitly unless the driver allows

// for tailored driver instances to be created (each instance
recognizing

// a different JDBC sub-protocol).



// Now that the JayBird driver is registered with the driver manager,

// try to get a connection to an employee.gdb database on this local
machine

// using one of two alternatives for obtaining connections...

int connectionAlternative = 1;

switch (connectionAlternative) {



case 1:

// This alternative is driver independent;

// the driver manager will find the right driver for you based on the
JDBC subprotocol.

// In the past, this alternative did not work with applets in some
browsers because of a

// bug in the driver manager. I believe this has been fixed in the
jdk 1.1 implementations.

try {

c = java.sql.DriverManager.getConnection (databaseURL, user,
password);

System.out.println ("Connection established.");

}

catch (java.sql.SQLException e) {

System.out.println ("Unable to establish a connection through the
driver manager.");
```

```
showSQLException (e);

return;

}

break;

case 2:

// If you're working with a particular driver d, which may or may not
be registered,

// you can get a connection directly from it, bypassing the driver
manager...

try {

java.util.Properties connectionProperties = new java.util.Properties
();

connectionProperties.put ("user", user);

connectionProperties.put ("password", password);

c = d.connect (databaseURL, connectionProperties);

System.out.println ("Connection established.");

}

catch (java.sql.SQLException e) {

System.out.println ("Unable to establish a connection through the
driver.");

showSQLException (e);

return;

}

break;

}



// Let's disable the default autocommit so we can undo our changes
```

```
later...

try {

c.setAutoCommit (false);

System.out.println ("Auto-commit is disabled.");

}

catch (java.sql.SQLException e) {

System.out.println ("Unable to disable autocommit.");

showSQLException (e);

return;

}


// Now that we have a connection, let's try to get some meta data...

try {

java.sql.DatabaseMetaData dbMetaData = c.getMetaData ();

// Ok, let's query a driver/database capability

if (dbMetaData.supportsTransactions ())

System.out.println ("Transactions are supported.");

else

System.out.println ("Transactions are not supported.");


// What are the views defined on this database?

java.sql.ResultSet tables = dbMetaData.getTables (null, null, "%",
new String[] {"VIEW"});

while (tables.next ()) {

System.out.println (tables.getString ("TABLE_NAME") + " is a view.");
```

```
}

tables.close ();

}

catch (java.sql.SQLException e) {

System.out.println ("Unable to extract database meta data.");

showSQLException (e);

// What the heck, who needs meta data anyway ;-(, let's continue
on...

}


// Let's try to submit some static SQL on the connection.

// Note: This SQL should throw an exception on employee.gdb because

// of an integrity constraint violation.

try {

s = c.createStatement ();

s.executeUpdate ("update employee set salary = salary + 10000");

}

catch (java.sql.SQLException e) {

System.out.println ("Unable to increase everyone's salary.");

showSQLException (e);

// We expected this to fail, so don't return, let's keep going...

}


// Let's submit some static SQL which produces a result set.

// Notice that the statement s is reused with a new SQL string.
```

```
try {

rs = s.executeQuery ("select full_name from employee where salary <
50000");

}

catch (java.sql.SQLException e) {

System.out.println ("Unable to submit a static SQL query.");

showSQLException (e);

// We can't go much further without a result set, return...

return;

}


// The query above could just as easily have been dynamic SQL,

// eg. if the SQL had been entered as user input.

// As a dynamic query, we'd need to query the result set meta data

// for information about the result set's columns.

try {

java.sql.ResultSetMetaData rsMetaData = rs.getMetaData ();

System.out.println ("The query executed has " +

rsMetaData.getColumnCount () +

" result columns.");

System.out.println ("Here are the columns: ");

for (int i = 1; i <= rsMetaData.getColumnCount (); i++) {

System.out.println (rsMetaData.getColumnName (i) +

" of type " +

rsMetaData.getColumnTypeName (i));
```

```
}

}

catch (java.sql.SQLException e) {

System.out.println ("Unable to extract result set meta data.");

showSQLException (e);

// What the heck, who needs meta data anyway ;-(, let's continue
on...

}



// Ok, lets step thru the results of the query...

try {

System.out.println ("Here are the employee's whose salary <
$50,000");

while (rs.next ()) {

System.out.println (rs.getString ("full_name"));

}

}

catch (java.sql.SQLException e) {

System.out.println ("Unable to step thru results of query");

showSQLException (e);

return;

}



// As an exercise to the reader, rewrite this code so that required

// table structures are created dynamically using executeUpdate() on
DDL.
```

```
// In this way the code will be able to run against any database file
rather

// than just a previously setup employee.gdb.

// Just to get you started, you'll want to define a method something
like

// the following...

//

// private static void createTableStructures (java.sql.Connection c)
throws java.sql.SQLException

// {

// // Some drivers don't force commit on DDL, Firebird JCA-JDBC
driver does,

// // see DatabaseMetaData.dataDefinitionCausesTransactionCommit().

// // This is not necessary for Firebird JCA-JDBC driver, but may be
for other drivers...

// c.setAutoCommit (true);

//

// java.sql.Statement s = c.createStatement();

//

// // Drop table EMPLOYEE if it already exists, if not that's ok too.

// try { s.executeUpdate ("drop table EMPLOYEE"); } catch
(java.sql.SQLException e) {}

//

// // Ok, now that we're sure the table isn't already there, create
it...

// s.executeUpdate ("create table EMPLOYEE (...)");

//

// // Ok, now populate the EMPLOYEE table...
```

```
// s.executeUpdate ("insert into EMPLOYEE values (...)");

//

// s.close();

// c.setAutoCommit (false);

// }

//

}



// This finally clause will be executed even if "return" was called
in case of any exceptions above.

finally {

System.out.println ("Closing database resources and rolling back any
changes we made to the database.");



// Now that we're all finished, let's release database resources.

try { if (rs!=null) rs.close (); } catch (java.sql.SQLException e)
{ showSQLException (e); }

try { if (s!=null) s.close (); } catch (java.sql.SQLException e)
{ showSQLException (e); }



// Before we close the connection, let's rollback any changes we may
have made.

try { if (c!=null) c.rollback (); } catch (java.sql.SQLException e)
{ showSQLException (e); }

try { if (c!=null) c.close (); } catch (java.sql.SQLException e)
{ showSQLException (e); }



// If you don't close your database objects explicitly as above,
```

```
// they may be closed by the object's finalizer, but there's

// no guarantee if or when the finalizer will be called.

// In general, object finalizers are not called on program exit.

// It's recommended to close your JDBC objects explictly,

// but you can use System.runFinalizersOnExit(true), as at the
beginning

// of this method main(), to force finalizers to be called before

// program exit.

}

}



// Display an SQLException which has occured in this application.

private static void showSQLException (java.sql.SQLException e)

{

// Notice that a SQLException is actually a chain of SQLExceptions,

// let's not forget to print all of them...

java.sql.SQLException next = e;

while (next != null) {

System.out.println (next.getMessage ());

System.out.println ("Error Code: " + next.getErrorCode ());

System.out.println ("SQL State: " + next.getSQLState ());

next = next.getNextException ();

}

}

}
```

Data Source Example:

```
// Original version of this file was part of InterClient 2.01
examples

//

// Copyright InterBase Software Corporation, 1998.

// Written by com.inprise.interbase.interclient.r&d.PaulOstler :-)

//

// Code was modified by Roman Rokytskyy to show that Firebird JCA-
JDBC driver

// does not introduce additional complexity in normal driver usage
scenario.

//

// An example of using a JDBC 2 Standard Extension DataSource.

// The DataSource facility provides an alternative to the JDBC
DriverManager,

// essentially duplicating all of the driver manager's useful
functionality.

// Although, both mechanisms may be used by the same application if
desired,

// JavaSoft encourages developers to regard the DriverManager as a
legacy

// feature of the JDBC API.

// Applications should use the DataSource API whenever possible.

// A JDBC implementation that is accessed via the DataSource API is
not

// automatically registered with the DriverManager.

// The DriverManager, Driver, and DriverPropertyInfo interfaces
```

```
// may be deprecated in the future.


import org.firebirdsql.jdbc.FBWrappingDataSource;

import org.firebirdsql.jca.FBConnectionRequestInfo;

import org.firebirdsql.gds.ISCConstants;


public final class DataSourceExample

{

static public void main (String args[]) throws Exception

{

// Create an Firebird data source manually;

FBWrappingDataSource dataSource = new FBWrappingDataSource();


// Set the standard properties

dataSource.setDatabase ("localhost/3050:c:/database/employee.gdb");

dataSource.setDescription ("An example database of employees");


/*

* Following properties were not deleted in order to show differences

* between InterClient 2.01 data source implementation and Firebird
one.

*/

//dataSource.setDataSourceName ("Employee");

//dataSource.setPortNumber (3060);

//dataSource.setNetworkProtocol ("jdbc:interbase:");
```

```
//dataSource.setRoleName (null);

// Set the non-standard properties

//dataSource.setCharSet
(interbase.interclient.CharacterEncodings.NONE);

//dataSource.setSuggestedCachePages (0);

//dataSource.setSweepOnConnect (false);

/*

* This is an example how to use FBConnectionRequestInfo to specify

* DPB that will be used for this data source.

*/

FBConnectionRequestInfo cri = dataSource.getConnectionRequestInfo();

cri.setProperty(ISCConstants.isc_dpb_lc_ctype, "NONE");

cri.setProperty(ISCConstants.isc_dpb_num_buffers, 1);

cri.setProperty(ISCConstants.isc_dpb_sql_dialect, 3);

dataSource.setConnectionRequestInfo(cri);


// Connect to the Firebird DataSource

try {

dataSource.setLoginTimeout (10);

java.sql.Connection c = dataSource.getConnection ("sysdba",
"masterkey");

// At this point, there is no implicit driver instance

// registered with the driver manager!

System.out.println ("got connection");

c.close ();

}
```

```
catch (java.sql.SQLException e) {
e.printStackTrace();
System.out.println ("sql exception: " + e.getMessage ());
}
}
}
```