# Jaybird JDBC Driver Java Programmer's Manual

Roman Rokytskyy, Mark Rotteveel

# Table of Contents

This is a snapshot version of the Jaybird manual. It may contain obvious (and not so obvious) errors, or it may still miss information on more recent features. If you find any problems, don't hesitate to report them on https://github.com/FirebirdSQL/jaybird-manual/issues or submit a pull request with the fix.

# Chapter 1. Introduction

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

This driver is based on both the JCA standard for application server connections to enterprise information systems and the well-known JDBC standard. The JCA standard specifies an architecture in which an application server can cooperate with a driver so that the application server manages transactions, security, and resource pooling, and the driver supplies only the connection functionality.

Jaybird is a driver that provides both Type 4 (pure Java) and Type 2 (native binding) support. The type 2 driver includes support for Firebird Embedded.

## 1.1. This manual

This manual covers Jaybird 3 and may use Java 8 and Firebird 3 specific features, but most examples and information also apply to previous versions of Jaybird, Firebird and Java.

New or removed features are tagged with the version that introduced a feature (eg *Jaybird 3*) or removed feature (eg *Jaybird 3*). This tagging is only done for features introduced (or removed) in Jaybird 2.2 or later, or in Firebird 3 or later, and for features available in Java/JDBC versions 8/4.2 or later.

This manual may include documentation of features of Jaybird versions later than 3 to simplify manual maintenance and versioning.

## 1.2. History

When Borland released an open-source version of the InterBase RDBMS, it included sources for a type 3 JDBC driver called InterClient.[1] However due to some inherent limitations of the InterBase (and later Firebird) client library, it was decided that the type 3 driver was a dead end. Instead, the Firebird team decided to develop a pure Java implementation of the wire protocol. This implementation became the basis for Jaybird, a pure Java driver for Firebird relational database.

## 1.3. Jaybird Architecture

The Jaybird driver consists of three layers, each of which is responsible for its part of the functionality.

- The GDS layer represents a Java translation of the Firebird API. It is represented by a number of interfaces and classes from the `org.firebirdsql.gds` package (and sub-packages).

  This API is implemented by a number of plugins that provide the pure java, native, local, and embedded implementations of the driver.

- The JCA layer represents the heart of the driver. Here all connection and transaction management happens. Additionally this layer adapts the GDS API and proxies the calls to the GDS implementation.

The JCA layer is an implementation of the Java Connector Architecture specification.

- The JDBC layer is an implementation of the JDBC specification.

In addition, the Services API allows you to manage the database and the server itself. The Manager component represents a JMX compatible implementation that utilizes the Services API. Currently only calls to create and drop database are available in the Manager component, other class provide features for database backup/restore, user management, statistics gathering, etc.

# 1.4. Jaybird Distribution

Jaybird 3 supports Firebird 2.0 and higher. See Jaybird versions for detailed information on supported Java and Firebird versions per Jaybird version.

The latest version of Jaybird can be downloaded from https://firebirdsql.org/en/jdbc-driver/

## 1.4.1. Maven

Alternatively, you can use maven to automatically download Jaybird and its dependencies.

Jaybird 3 is available from Maven central:

Groupid: `org.firebirdsql.jdbc`,
Artifactid: `jaybird-XXX` (where `XXX` is `jdk17` or `jdk18`).
Version: `3.0.6`

For example:

```
<dependency>
  <groupId>org.firebirdsql.jdbc</groupId>
  <artifactId>jaybird-jdk18</artifactId>
  <version>3.0.6</version>
</dependency>
```

The Maven definition of Jaybird depends on antlr-runtime by default.

If your application is deployed to a Java EE application server, you will need to exclude the `javax.resource:connector-api` dependency, and add it as a provided dependency:

```
<dependency>
  <groupId>org.firebirdsql.jdbc</groupId>
  <artifactId>jaybird-jdk18</artifactId>
  <version>3.0.6</version>
  <exclusions>
    <exclusion>
      <groupId>javax.resource</groupId>
      <artifactId>connector-api</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>javax.resource</groupId>
  <artifactId>connector-api</artifactId>
  <version>1.5</version>
  <scope>provided</scope>
</dependency>
```

If you want to use Type 2 support (native, local or embedded), you need to explicitly include JNA 4.4.0 as a dependency:

```
<dependency>
  <groupId>net.java.dev.jna</groupId>
  <artifactId>jna</artifactId>
  <version>4.4.0</version>
</dependency>
```

We plan to make native and embedded support a separate library in future releases, and provide Firebird client libraries as Maven dependencies as well.

## 1.5. Quality Assurance

The Jaybird team uses JUnit test cases to assure the quality of the released driver. During development unit tests are extensively used. Committing a code change to the source control is not allowed until it passes all existing unit tests. Each reproducible bug usually gets its own test case. This guarantees that a clean check out can be compiled and will not contain any previously discovered and fixed bugs. Currently there are more than 3900 test cases covering most of the driver code.

## 1.6. Useful resources

### 1.6.1. JDBC

For JDBC documentation, see http://www.oracle.com/technetwork/java/javase/jdbc/index.html.

### 1.6.2. Firebird

General information about the Firebird database is available from the Firebird web site (https://www.firebirdsql.org/).

For information about using SQL in Firebird, see the Firebird 2.5 Language Reference [https://www.firebirdsql.org/file/documentation/reference_manuals/fblangref25-en/html/fblangref25.html] and other documents available from the Reference Manuals [https://www.firebirdsql.org/en/reference-manuals/] section of the Firebird web site.

### 1.6.3. Jaybird Support

Support for Jaybird is available through the following channels:

- The Firebird-Java group [https://groups.yahoo.com/group/Firebird-Java] and corresponding mailing list firebird-java@yahoogroups.com [mailto:firebird-java@yahoogroups.com]

  You can subscribe to the mailing list by sending an email to firebird-java-subscribe@yahoogroups.com [mailto:firebird-java-subscribe@yahoogroups.com]

- On Jaybird Frequently Asked Questions [https://www.firebirdsql.org/file/documentation/drivers_documentation/java/faq.html].
- On Jaybird wiki [https://github.com/FirebirdSQL/jaybird/wiki/].

  This is a place where the community shares information about different aspects of Jaybird usage, configuration examples for different applications/servers, tips and tricks, FAQ, etc.

- On Stack Overflow [https://stackoverflow.com/], please tag your questions with **jaybird** and **firebird**

  Please make sure to familiarize yourself with the rules and expectations of Stack Overflow before asking, see Stack Overflow Tour [https://stackoverflow.com/tour] and Help Center: Asking [https://stackoverflow.com/help/asking]

## 1.7. Contributing

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

- Participate on the mailing lists (see https://www.firebirdsql.org/en/mailing-lists/)
- Report bugs or submit patches on the tracker (see Reporting Bugs)
- Create pull requests on GitHub (https://github.com/FirebirdSQL/jaybird)
- Become a developer (for Jaybird contact us on firebird-java, for Firebird in general, use the Firebird-devel mailing list)
- Become a paying member or sponsor of the Firebird Foundation (see https://www.firebirdsql.org/en/firebird-foundation/)

### 1.7.1. Reporting Bugs

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

You can report bugs in the Firebird bug tracker, project "Jaybird JCA/JDBC Driver" [http://tracker.firebirdsql.org/browse/JDBC]

When reporting bugs, please provide a minimal, but complete reproduction, including databases and source code to reproduce the problem. Patches to fix bugs are also appreciated. Make sure the patch is against a recent master version of the code. You can also fork the jaybird repository and create pull requests.

[1] For those interested in software archaeology, you can find the open sourced Interclient sources archived on https://github.com/FirebirdSQL/x-cvs-interclient

# User Manual

# Chapter 2. Obtaining a connection

Jaybird is a regular JDBC driver and supports two primary ways to obtain connection: via `java.sql.DriverManager` and via an implementation of the `javax.sql.DataSource` interface.

## 2.1. Obtaining connection java.sql.DriverManager

`java.sql.DriverManager` was the first connection factory in Java. It is based on the concept of the JDBC URL, a string that uniquely identifies the database to connect. The driver manager then checks which driver(s) - if any - can establish a connection.

There is also support to specify additional connection parameters, like user name and password.

A JDBC URL consists of three parts:

```
jdbc:firebirdsql://localhost:3050/c:/database/example.fdb
```

- `jdbc`
  JDBC protocol
- `firebirdsql`
  JDBC subprotocol, identifies driver to use
- `//localhost:3050/c:/database/example.fdb`

  This is a database specific part, and identifies the database for the driver to connect, in the case of Jaybird that is `//<host>:<port>/<path to database>`

The first part, `jdbc:firebirdsql:` is required by JDBC and specifies the so called protocol and subprotocol for the JDBC connection. In other words, the type of connection that the application wants to obtain, in this example it is a connection to a Firebird database.

An example of obtaining a connection is shown below.

*Simple example to obtain a JDBC connection*

```
package hello;

import java.sql.*;

public class HelloServer {

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

    Class.forName("org.firebirdsql.jdbc.FBDriver"); ①

    Connection connection = DriverManager.getConnection(
        "jdbc:firebirdsql://localhost:3050/c:/db/employee.fdb",
        "SYSDBA", "masterkey"); ②

    // do something here
  }
}
```

The first line of this code is important – it tells Java to load the Jaybird JDBC driver. As required by the JDBC specification, at this point driver registers itself with `java.sql.DriverManager`.

Since Java 6 (JDBC 4), explicitly loading the driver using `Class.forName("org.firebirdsql.jdbc.FBDriver")` is no longer necessary, except when the driver is not on the system class path. Examples where it may be necessary to explicitly load the driver are web applications that include the driver in the deployment. There, the driver is not on the system class path, so it will need to be loaded explicitly.

We will leave out usages of `Class.forName` in further examples, they will work because of automatic driver loading.

*Registering the JDBC driver*

There are several ways to register JDBC driver:

1. `DriverManager` loads the drivers from the system class path. This happens automatically.
2. The application explicitly loads the driver's class. This is only necessary if the automatic loading (see previous item) is not available. This can be necessary because the driver `jar` is loaded dynamically, through a different classloader, etc).

   The JDBC specification requires that during class initialization the driver registers itself with `DriverManager`.

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

3. The JDBC driver name is listed in the `jdbc.drivers` system property. Multiple drivers can be separated using a colon (`:`).

   You can specify the value of this property during JVM startup:

```
java\
  -Djdbc.drivers=foo.Driver:org.firebirdsql.jdbc.FBDriver\
  -classpath jaybird-full-{jaybird-full-
version}.jar;C:/myproject/classes\
  my.company.SomeJavaExample
```

The second statement of the example tells the `java.sql.DriverManager` to open a database connection to the Firebird server running on localhost, and the path to the database is `c:/database/employee.fdb`.

The connection specification consists of the host name of the database server, optionally you can specify a port (by default port 3050 is used). The host name can be specified using either its DNS name (for example `fb-server.mycompany.com` or just `fb-server`), or its IP address (for example `192.168.0.5`, or `[1080::8:800:200C:417A]` for an IPv6 address).

After the server name and port, the alias or path to the database is specified. We suggest to specify a database alias instead of the absolute database path. For more information about using aliases, see the documentation of Firebird server.

The format of the path depends on the platform of the Firebird server.

On Windows, the path must include the drive letter and path, for example `c:/database/employee.fdb`, which points to the employee database that can be found in the `database` directory of drive `C:`. Java (and Firebird) supports either `/` or `\` (escaped as `\\`) as path separator on the Windows platform. On Unix and Linux platform, you can use only `/` as the path separator.

On Unix platforms the path must include the root, as the path is otherwise interpreted relative to a

server-dependent folder. Having to include the root has the effect that a database in `/var/firebird/employee.fdb` needs to use a double `//` after the host name (and port) in the connection string: `jdbc:firebirdsql://localhost//var/firebird/employee.fdb`

It is possible to specify a relative path, but as this depends on the server configuration, this may be confusing or easily lead to errors. We suggest not to use relative paths, and instead use an alias.

### 2.1.1. Specifying extended properties

What if we want to specify additional connection parameters, for example a client encoding? The JDBC API provides a method to specify additional connection properties:

*Obtaining JDBC connection with additional connection properties*

```
package hello;

import java.sql.*;
import java.util.*;

public class HelloServerWithEncoding {

  public static void main(String[] args) throws Exception {
    Properties props = new Properties();

    props.setProperty("user", "SYSDBA");
    props.setProperty("password", "masterkey");
    props.setProperty("encoding", "UTF8");

    try (Connection connection = DriverManager.getConnection(
            "jdbc:firebirdsql://localhost:3050/C:/db/employee.fdb",
            props)) {

        // do something here

    }
  }
}
```

The `user` and `password` properties are defined in JDBC. All other property names, like `encoding` here, are driver-specific.

Additional properties, for example the SQL role for the connection can be added to the `props` object. The list of properties available in Jaybird can be found in Extended connection properties.

It is not always possible to use the above described method. Jaybird also provides a possibility to specify extended properties in the JDBC URL.

*Extended JDBC URL format*

```
jdbc:firebirdsql://host[:port]/<path to db>?<properties>
<properties> ::= <property>[{& | ;}<properties>]
<property>   ::= <name>[=<value>]
```

The example below shows the specification for extended JDBC properties in the URL.

In this case extended properties are passed together with the URL using the HTTP-like parameter passing scheme: first comes the main part of the URL, then `"?"`, then name-value pairs separated with `&` or `;`. This example is equivalent to the previous example.

Despite the URL-like syntax, Jaybird does not (yet) support URL encoding.

*Specifying extended properties in the JDBC URL*

```
import java.sql.*;

...

Connection connection = DriverManager.getConnection(
    "jdbc:firebirdsql://localhost:3050/C:/db/employee.fdb?encoding=UTF8",
    "SYSDBA",
    "masterkey");
```

### 2.1.2. Obtaining a connection via javax.sql.DataSource

The JDBC 2.0 specification introduced a mechanism to obtain database connections without requiring the application to know any specifics of the underlying JDBC driver. The application is only required to know the logical name to find an instance of the `javax.sql.DataSource` interface using Java Naming and Directory Interface (JNDI). This is a common way to obtain connections in web and application servers. Alternatively, the `DataSource` may be injected by CDI or Spring.

In order to obtain a connection via a `DataSource` object, you can use the code shown below. This code assumes that you have correctly configured the JNDI properties. For more information about configuring JNDI please refer to the documentation provided with your web or application server.

*Typical way to obtain JDBC connection via JNDI*

```
package hello;

import java.sql.*;
import javax.sql.*;
import javax.naming.*;

public class HelloServerJNDI {

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

    InitialContext ctx = new InitialContext();
    DataSource ds = (DataSource)ctx.lookup("jdbc/SomeDB");

    try (Connection connection = ds.getConnection()) {
      // do something here...
    }
  }
}
```

Usually, the binding between the `DataSource` object and its JNDI name happens in the configuration of your web or application server. However under some circumstances (e.g. you are developing your own JNDI-enabled application server/framework), you may have to do this yourself. You can use this code snippet for this purpose:

*Programmatic way to instantiate javax.sql.DataSource implementation*

```
import javax.naming.*;
import org.firebirdsql.ds.*;
...
FBSimpleDataSource ds = new FBSimpleDataSource();

ds.setDatabase("//localhost:3050/C:/database/employee.fdb");
ds.setUser("SYSDBA");
ds.setPassword("masterkey");

InitialContext ctx = new InitialContext();

ctx.bind("jdbc/SomeDB", ds);
```

The `DataSource` implementation supports all connection properties available to the `DriverManager` interface.

## 2.2. Driver types

As mentioned in the section Jaybird Architecture, Jaybird supports multiple implementations of the GDS API. The default Jaybird distribution contains two main categories of the implementations: the pure Java implementation of the Firebird wire protocol, and a JNA proxy that can use a Firebird

`fbclient` library.

The next sections provide a description of these types and their configuration with the corresponding JDBC URLs that should be used to obtain the connection of desired type. The type of the JDBC driver for the `javax.sql.DataSource` is configured via a corresponding property.

### 2.2.1. PURE_JAVA type

The `PURE_JAVA` type (JDBC Type 4) uses a pure Java implementation of the Firebird wire protocol. This type is recommended for connecting to a remote database server using TCP/IP sockets. No installation is required except adding the JDBC driver to the class path. This type of driver provides the best performance when connecting to a remote server.

In order to obtain a connection using the `PURE_JAVA` driver type you have to use a JDBC URL as shown in Obtaining connection java.sql.DriverManager:

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

When using `javax.sql.DataSource` implementation, you can specify either `"PURE_JAVA"` or `"TYPE4"` driver type, however this type is used by default.

### 2.2.2. NATIVE and LOCAL types

The `NATIVE` and `LOCAL` types (JDBC Type 2) use a JNA proxy to access the Firebird client library and requires installation of the Firebird client. The `NATIVE` driver type is used to access the remote database server, the `LOCAL` type accesses the database server running on the same host by means of IPC (Inter-Process Communication). Performance of `NATIVE` driver is approximately 10% lower compared to the `PURE_JAVA` driver, but `LOCAL` type has up to 30% higher performance compared to the `PURE_JAVA` driver when connecting the server on the same host. This is mostly due to the fact that TCP/IP stack is not involved in this mode.

To create a connection using the `NATIVE` JDBC driver to connect to a remote server you have to use the following JDBC URL with the native subprotocol:

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

When connecting to a local database server using the `LOCAL` driver, you should use following:

```
jdbc:firebirdsql:local:<absolute path to database>
```

In addition to Jaybird, this requires a native Firebird client library, and JNA 4.4.0 needs to be on the classpath.

#### Maven dependency for native client

When using Jaybird 3 and later, you can use a library to provide the Firebird client library for the

`native` and `local` protocol. For Windows and Linux, you can add the `org.firebirdsql.jdbc:fbclient` dependency on your classpath. This dependency does not support the `embedded` protocol.

```
<dependency>
    <groupId>org.firebirdsql.jdbc</groupId>
    <artifactId>fbclient</artifactId>
    <version>3.0.4.0</artifactId>
</dependency>
```

You can also download the library (see mrotteveel/jaybird-fbclient [https://github.com/mrotteveel/jaybird-fbclient#download] for download link) and add it your classpath.

See next sections for other solutions.

#### Windows

For Jaybird 3 and later, we recommend using the solution documented in Maven dependency for native client.

On Windows, you need to make sure that `fbclient.dll` is located on the `PATH` environment variable. Alternatively you can specify the directory containing this DLL in the `jna.library.path` system property.

For example, if you put a copy of `fbclient.dll` in the current directory you have to use the following command to start Java:

```
java -cp <relevant claspath> -Djna.library.path=. com.mycompany.MyClass
```

If your Java install is 32 bit, you need a 32 bit `fbclient.dll`, for 64 bit Java, a 64 bit `fbclient.dll`.

#### Linux

For Jaybird 3 and later, we recommend using the solution documented in Maven dependency for native client.

On Linux, you need to make sure that `libfbclient.so` is available through the `LD_PATH` environment variable.

Usually shared libraries are stored in the `/usr/lib/` directory; however you will need root permissions to install the library there. Some distributions will only have, for example, `libfbclient.so.2.5`. In that case you may need to add a symlink from `libfbclient.so` to the client on your system.

Alternatively you can specify the directory containing the library in the `jna.library.path` Java system property. See the Windows example above for more details.

**Limitations**

**TODO** Section might be outdated

Firebird client library is not thread-safe when connecting to a local database server using IPC. Jaybird provides the necessary synchronization in Java code using a static object instance. However, this static object instance is local to the classloader that has loaded the Jaybird classes.

In order to guarantee correct synchronization , the Jaybird driver must be loaded by the top-most classloader. For example, when using the Type 2 JDBC driver with a web or application server, you have to add the Jaybird classes to the main classpath (for example, to the `lib/` directory of your web or application server), but **not** to the web or Jave EE application, e.g. the `WEB-INF/lib` directory.

### 2.2.3. EMBEDDED type

The Embedded server JDBC driver is a Type 2 JDBC driver that, rather than using the Firebird client library, loads the Firebird embedded server library instead. This is the highest performance type of JDBC driver for accessing local databases, as the Java code accesses the database file directly.

In order to obtain a connection via DriverManager you have to use following URL:

```
jdbc:firebirdsql:embedded:<path to database>

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

When host and, optionally, port is specified, embedded server acts as client library (i.e. you get the same Type 2 behavior as you would get with using "native").

This driver tries to load `fbembed.dll/libfbembed.so` and `fbclient.dll/libfbclient.so`, the last - of course - only works if that fbclient provides Firebird embedded.

See also the NATIVE and LOCAL types section.

When using Firebird 3 embedded, you will need to make sure the necessary plugins like `engine12.dll/libengine12.so` are accessible to the client library, consult the Firebird 3 documentation for more information. For an example, see the article Jaybird with Firebird embedded example [https://www.lawinegevaar.nl/firebird/jaybird_embedded_example.html]

**Limitations**

**TODO** Section might be outdated

The Firebird embedded server for Linux is not thread safe. Jaybird provides the needed synchronization in Java code, similar to the one described for the Type 2 JDBC driver. This implies the same restrictions on the classloader that will load the Jaybird classes.

The Firebird embedded server for Windows opens databases in exclusive mode. This means that this particular database is accessible only to one Java virtual machine. *There is no exclusive mode on the POSIX platform. When the same database file is accessed by multiple JVM instances, database will be corrupted!*

### 2.2.4. OOREMOTE type

The `OOREMOTE` type is a JDBC Type 4 specifically for use with OpenOffice.org and LibreOffice. It addresses some differences in interpretation of the JDBC specification, and provides alternative metadata in certain cases to allow OpenOffice.org and LibreOffice to function correctly.

This only describes connecting to Firebird using Jaybird from OpenOffice.org or LibreOffice, it is not about the Firebird embedded use that has been introduced in recent LibreOffice versions.

In order to obtain a connection you have to use following URL:

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

All other notes for PURE_JAVA type apply.

## 2.3. Connection Pooling

Each time a connection is opened via `DriverManager`, a new physical connection to server is opened. It is closed when the connection is closed. In order to avoid the overhead of creating connections, you can use a connection pool implementation to maintain a cache of open physical connections that can be reused between user sessions.

Since version 3.0, Jaybird no longer provides a connection pool. If you need a `javax.sql.DataSource` implementation that provides a connection pool, either use the connection pool support of your application server, or consider using HikariCP [https://brettwooldridge.github.io/HikariCP/], DBCP [http://commons.apache.org/proper/commons-dbcp/], or c3p0 [http://www.mchange.com/projects/c3p0/].

## 2.4. The javax.sql.DataSource implementation

Connection pool implementations, whether provided by a Java EE application server or a third-party library, are exposed as an implementation of the `javax.sql.DataSource` interface.

The most important method exposed by this interface is the `getConnection()` method, which will return a connection based on the configuration of the data source. For a 'basic' (non-pooling) data source this will create a new, physical, connection. For a connection pool, this will create a logical connection that wraps a physical connection from the pool.

The 'user' of a connection should not care whether the connection is pooled or not, the connection should behave the same from the perspective of the user, and the user should use the connection in the same way. This should allow for swapping between a non-pooling and pooling data source in an application without any changes to the code using the data source.

When the application is done with the connection, it should call `close()` on the connection. A connection from a non-pooling data source will be closed. For a logical connection from a connection pool, `close()` will invalidate the logical connection (which will make it behave like a

closed connection), and return the underlying physical connection to the connection pool, where it will be either kept for re-use, or maybe closed.

Use a connection for the shortest scope (and time) necessary for correct behaviour. Get a connection, and close it as soon as you're done. When using a connection pool, this has the added benefit that just a few connections can serve the needs of the application.

## 2.5. The javax.sql.ConnectionPoolDataSource implementation

The `javax.sql.ConnectionPoolDataSource` interface represents a factory that creates `PooledConnection` objects for use by a connection pool. For example, application servers support the use of a `ConnectionPoolDataSource` to populate their connection pool.

A `PooledConnection` instance represents a physical connection to a database and is a source of logical connections that a connection pool can hand out to the application. Closing this logical connection returns the physical connection back into the pool.

Contrary to its name, a `ConnectionPoolDataSource` is not a connection pool!

Jaybird provides `org.firebirdsql.ds.FBConnectionPoolDataSource` as an implementation of the `javax.sql.ConnectionPoolDataSource` interface.

**TODO** Add more info

## 2.6. The javax.sql.XADataSource implementation

The JDBC 2.0 specification introduced the `javax.sql.XADataSource` interface that should be used to access connections that can participate in distributed transactions with JTA-compatible transaction coordinator. This gives applications possibility to use two-phase commit to synchronize multiple resource managers.

Just like `javax.sql.ConnectionPoolDataSource`, applications normally don't access an `XADataSource` implementation directly, instead it is used as a factory of connections for an XA-enabled data source. To the application this is usually exposed as a `javax.sql.DataSource`.

Jaybird provides `org.firebirdsql.ds.FBXADataSource` as an implementation of the `javax.sql.XADataSource` interface.

# Chapter 3. Handling exceptions

Exception handling is probably the most important aspect that directly affects the stability of the application. Correct handling of the error cases guarantees correct functioning of the client code as well as the database server.

All methods of the interfaces defined in the JDBC specification throw instances of `java.sql.SQLException` to notify about error conditions that happen during request processing. The `SQLException` is a checked exception, which forces Java programmers to either handle it with the try/catch clause or redeclare it in the method signature.

## 3.1. Working with exceptions

Exception handling becomes even more important if we consider that this topic is either ignored or presented in incorrect form in most JDBC tutorials. The official JDBC tutorial from Oracle [https://docs.oracle.com/javase/tutorial/jdbc/index.html] briefly mentions that exceptions should be handled by using try/catch blocks only at the end of the course, but neither reasons for doing this nor the best practices are presented.

There are good reasons to think about exception handling in your applications before you start coding. First of all, it is very hard to change the exception handling pattern in existing code. The changes will affect all layers above the place where the changes in exception handling are made and the new application must be thoroughly tested after the change.

Another reason was already mentioned on the beginning of this chapter: instances of `java.sql.SQLException` are the only way for the RDBMS server to notify about the error condition that happened during request processing. By checking the error code which is sent with the exception application can try to recover from the error.

And last but not least, there is resource management. When an exception happens in the method, the execution flow of Java code differs from the normal flow, and only correctly coded application will ensure that all allocated resources will be released. The resources in our case are JDBC connections, statements, prepared statements and callable statements, result sets, etc. All these objects not only take memory in the Java Virtual Machine of the application, but also consume memory on the server, which, worst case, can lead to an unintended Denial-of-Service attack, as the database server can no longer service requests.

A good exception handling strategy requires you do distinguish three kinds of error conditions:

- errors that the database access layer can detect and correctly handle; for example, the application might decide to re-execute the business transaction if the database server returned a deadlock error;
- errors that database access layer can detect, but is unable to handle; usually those are all database errors that do not have special handling routines;
- errors that database access layer cannot detect without additional code unrelated to the functionality of this layer; basically, all runtime exceptions fall into this category.

The handling strategy then consists of

- processing the selected error codes for cases described above;
- converting the generic `SQLException` into a generic business error in the application (this can be throwing some generic exception defined in the application, but can also be an entry in the application event log and short message that asks to retry the operation later);
- some emergency tactics, since the error that happened (e.g. `NullPointerException` or `OutOfMemoryError`) was not considered while the application was created, thus possibly leaving it in an unknown state; further operation should be considered dangerous and the corresponding execution branch has to be halted.

The problem of resource management can be solved if resource allocation and deallocation happens in the same code block and is protected with a try-with-resources block. The code to recover from error conditions should use try/catch blocks. Example of such error and resource handling code is presented below.

*Typical resource allocation and error handling patterns*

```
String updateString = "update COFFEES " +
    "set SALES = ? where COF_NAME like ?";

try (PreparedStatement updateSales = con.prepareStatement(updateString)) {
  int [] salesForWeek = {175, 150, 60, 155, 90};
  String [] coffees = {"Colombian", "French_Roast",
      "Espresso", "Colombian_Decaf",
      "French_Roast_Decaf"};

  int len = coffees.length;

  for(int i = 0; i < len; i++) {
    updateSales.setInt(1, salesForWeek[i]);
    updateSales.setString(2, coffees[i]);

    try {
      updateSales.executeUpdate();
    } catch(SQLException ex) {
      if (ex.getErrorCode() == ...) {
        // do something
      } else {
        throw new BusinessDBException(ex);
      }
    }
  }
}
```

The nested try/catch block shows you an example of handling a deadlock error if it happens (first scenario according to our classification), otherwise the exception is converted and passed to the upper layers (second scenario). As you see, there is no special treatment to the third scenario.

A possible bug in the JDBC driver could have generated runtime exception in the `PreparedStatement.executeUpdate()` method, which would lead to the statement handle leakage if

the try-with-resource block had not been used to do the resource cleanup. As a rule of thumb, always declare and allocate resources in a try-with-resources block: the resource will be automatically closed/freed at the end of the block, even if exceptions occur.

Such coding practice might look weird, because on first sight the whole purpose of using the `PreparedStatement` is neglected: the statement is prepared, used only once and then deallocated. However, when this practice is combined with connection and statement pooling, it brings enormous advantage to the application code. The code becomes much more manageable – resource allocations and deallocations happen in the same method and the software developer does not need to remember the places where the same prepared statement might be used – a statement pool will either reuse the statement or it will prepare a new one, if it detects that all pooled prepared statements are currently in use. As a side effect, the application will always use the minimum number of statements handles, which in turn reduces the used resources on the database side.

Jaybird currently provides no statement pooling itself, availability will depend on the connection pool library used. Consult the documentation of your connection pool to see if - and how - it provides statement pooling.

## 3.2. Warnings

Some errors returned by Firebird are treated as warnings. They are converted into instances of `java.sql.SQLWarning` class in the JDBC layer. These exceptions are not thrown from the driver methods, but added to a connection instance.

Currently no warning is added to `Statement` or `ResultSet` objects.

Each next warning is appended to the tail of the warning chain. In order to read the warning chain, use the code presented below.

*Example how to work with warnings*

```
import java.sql.*;
....
SQLWarning warning = connection.getWarnings();
while (warning != null) {
  .... // do something with the warning
  warning = warning.getNextWarning();
}
```

or

*Alternative example how to work with warnings*

```
import java.sql.*;
....
for (Throwable throwable : connection.getWarnings()) {
  if (throwable instanceof SQLWarning) {
    SQLWarning warning = (SQLWarning) throwable;
    .... // do something with the warning
  }
}
```

This second example will iterate over the first warning, all its causes (if any), and then on to other warnings (if any), and so on.

In order to clear existing warning, call `Connection.clearWarnings()` method.

## 3.3. java.sql.SQLException in Jaybird

**TODO** Information in section is outdated

An `SQLException` is a special exception that is thrown by the JDBC connectivity component in case of an error. Each instance of this exception is required to carry the vendor error code (if applicable) and a SQL state according to the X/Open SQLstate or SQL:2003 specifications. Firebird and Jaybird use SQL:2003 SQL state codes.[2]

When multiple SQL errors happened, they are joined into a chain. Usually the most recent exception is thrown to the application, the exceptions that happened before can be obtained via `SQLException.getNextException()` method. Alternatively, `SQLException.iterator()` can be used to walk over all exceptions in the chain and their causes.

The JDBC specification provides an exception hierarchy that allows an application to react on the error situations using regular exception handling rather than checking the error code. Error codes may still be necessary for handling specific error cases.

The JDBC 4.3 specification defines the following exception hierarchy:[3]

- `java.sql.SQLException` - root of all JDBC exceptions
  - `java.sql.BatchUpdateException` - thrown when batch of the statements did not execute successfully; contains the result of batch execution.
  - `java.sql.SQLClientInfoException` - thrown when client info properties could not be set.
  - `java.sql.SQLNonTransientException` - thrown when retrying the same action without fixing the underlying cause would fail.
    - `java.sql.SQLDataException` - thrown for data-related errors, for example conversion errors, too long values. (SQLstate class `22`)
    - `java.sql.SQLFeatureNotSupportedException` - thrown to indicate that an optional JDBC feature is not supported by the driver or the data source (Firebird). (SQLstate class `0A`)
    - `java.sql.SQLIntegrityConstraintViolationException` - thrown for constraint violations.

(SQLstate class 23)

- `java.sql.SQLInvalidAuthorizationSpecException` - thrown for authorization failures. (SQLstate class 28)
- `java.sql.NonTransientConnectionException` - thrown for connection operations that will not succeed on retry without fixing the underlying cause. (SQLstate class 08)
- `java.sql.SQLSyntaxErrorException` - thrown for syntax errors. (SQLstate class 42)

◦ `java.sql.SQLRecoverableException` - thrown when an action might be retried by taking recovery actions and restarting the transaction.

◦ `java.sql.SQLTransientException` - thrown when the action might succeed if it is retried without further recovery steps.

- `java.sql.SQLTimeoutException` - thrown when the `queryTimeout` or `loginTimeout` has expired.
- `java.sql.SQLTransactionRollbackException` - thrown when the statement was automatically rolled back because of deadlock or other transaction serialization failures. (SQLstate class 40)
- `java.sql.SQLTransientConnectionException` - thrown for connection operations that might succeed on retry without any changes. (SQLstate class 08)

◦ `java.sql.SQLWarning` should only be used to signal warnings, it should never be thrown by a JDBC driver.

- `java.sql.DataTruncation` - thrown when a data truncation error happens, can also be used as a warning.

Unfortunately Jaybird 3.0 does not yet fully use this exception hierarchy, we are working to address this with the next versions of Jaybird.

Each of three layers in Jaybird use exceptions most appropriate to the specific layer. **TODO** List needs revision

- `org.firebirdsql.gds.GDSException` is an exception that directly corresponding to the error returned by the database engine. Instances of this class are thrown by the GDS implementations. Upper layers either convert these exceptions into the ones appropriate to that layer or catch them if driver can handle the error condition.
- Subclasses of `javax.resource.ResourceException` are thrown by the JCA layer when an error happens in the JCA-related code. Upper layer converts this exception into a subclass of `java.sql.SQLException`. If the `ResourceException` was caused by the `GDSException`, latter is extracted during conversion preserving the error code. If `ResourceException` was caused by an error condition not related to an error returned by the database engine, error code of the `SQLException` remains 0.
- Subclasses of `javax.transaction.XAException` are thrown when an XA protocol error happens in JCA layer. Similar to the previous case, `XAException` can wrap the `GDSException`, which are extracted during exception conversion to preserve the error code.
- Subclasses of `java.sql.SQLException` are thrown by the JDBC layer. Jaybird has a few subclasses that might be interesting to the application:

- `org.firebirdsql.jdbc.FBDriverConsistencyCheckException` – this exception is thrown when driver detects an internal inconsistent state. SQL state is `HY000`.
- `org.firebirdsql.jdbc.FBDriverNotCapableException` – this exception is thrown when an unsupported method is called. SQL state is `0A000`.
- `org.firebirdsql.jdbc.FBSQLParseException` – this exception is thrown when incorrect escaped syntax is detected. SQL state is `42000`.
- `org.firebirdsql.jdbc.field.TypeConversionException` – this exception is thrown when the driver is asked to perform a type conversion that is not defined in the JDBC specification. For a table of allowed type conversions see Data Type Conversion Table.

## 3.4. SQL states

Jaybird supports the SQLstate values from the SQL:2003 standard,[2] however only few states nicely map into the Firebird error codes.

Applications can use the SQLstate codes in the error handling routines which should handle errors that are returned from different databases. But since there is little agreement between RDBMS vendors, this method can be used only for very coarse error distinction.

## 3.5. Useful Firebird error codes

Contrary to the SQLstates, the Firebird native error codes are extremely useful to determine the type of an error that happened.

Here you can find a short list of error codes, symbolic names of a corresponding constant in a `org.firebirdsql.gds.ISCConstants` class, the error message and short explanation of an error.

**TODO** Needs revising now Jaybird tries to pull the most important error code to the top

### 3.5.1. DDL Errors

DDL errors happen during execution of DDL requests, and two primary error codes are used in Firebird while executing the DDL operations. There are few other rare cases not mentioned here, but the corresponding error messages contain enough information to understand the reason of an error.

| | | |
|---|---|---|
| 335544351L | `isc_no_meta_update` | `"unsuccessful metadata update"`<br><br>This error is returned when the requested DDL operation cannot be completed, for example the application tries to define a primary key that will exceed the maximum allowed key size. |

| | | |
|---|---|---|
| 335544510L | `isc_lock_timeout` | In combination with `isc_obj_in_use` (335544453L), this means that the DDL command tries to modify an object that is used in some other place, usually in another transaction. The complete error message will contain the name of the locked object. |
| 335544569L | `isc_dsql_error` | If the third error code is either `isc_dsql_datatype_err` or `isc_dsql_command_err`, then additional error codes and arguments specify the reason why the operation has failed. |

### 3.5.2. Lock Errors

Lock errors are reported by Firebird primarily when the application tries to modify a record which is already modified by a concurrent transaction. Depending on the transaction parameters such error can be reported either right after detection or after waiting some defined timeout hoping that concurrent transaction will either commit or rollback and eventually release the resource. More information on transaction locking modes can be found in section Using transactions.

| | | |
|---|---|---|
| 335544345L | `isc_lock_conflict` | `"lock conflict on no wait transaction"`<br><br>This error is returned when a "no wait" transaction needs to acquire a lock but finds another concurrent transaction holding that lock.<br><br>Instead of waiting the predefined timeout hoping that concurrent transaction will either commit or rollback, an error is returned to notify an application about the situation. |
| 335544510L | `isc_lock_timeout` | `"lock time-out on wait transaction"`<br><br>Similar to the `isc_lock_conflict`, but this error is returned when the lock timeout that was specified for the current transaction expired while waiting for a lock.<br><br>Another source of this error are DDL operations that try to obtain a lock on a database object that is currently used in some other place. |
| 335544336L | `isc_deadlock` | `"deadlock"`<br><br>Two transactions experience a deadlock when each of them has a lock on a resource on which the other is trying to obtain a lock. |

### 3.5.3. Referential Integrity Errors

Referential integrity constraints ensure that the database remains in a consistent state after the DML operation and/or whole transaction is completed. Three primary error codes are returned when the defined constraints are violated. The error messages are self-explanatory.

| | | |
|---|---|---|
| 335544665L | `isc_unique_key_violation` | `violation of PRIMARY or UNIQUE KEY constraint "{0}" on table "{1}"` |
| 335544558L | `isc_check_constraint` | `Operation violates CHECK constraint {0} on view or table {1}` |
| 335544466L | `isc_foreign_key` | `violation of FOREIGN KEY constraint "{0}" on table "{1}"` |

### 3.5.4. DSQL Errors

This group contains secondary codes for the primary error code `isc_dsql_error` (`335544569L`), that has a message `"Dynamic SQL Error"`.

In most situations, Jaybird 3 and higher will put this secondary error code in the `SQLException` instead of `isc_dsql_error`.

| | | |
|---|---|---|
| 335544573L | `isc_dsql_datatype_err` | `"Data type unknown"`<br><br>Usually this error is reported during DDL operation when the specified data type is either unknown or cannot be used in the specified statement. However it can also happen in DML operation, e.g. when an `ORDER BY` clause contains unknown collation, or if a parameter is used in a `SELECT` clause without explicit cast. |
| 335544570L | `isc_dsql_command_err` | `"Invalid command"`<br><br>Error happens either during parsing the specified SQL request or by handling the DDL command. |

### 3.5.5. Other Errors

This table contains other errors that might be interesting to the application developer, however they do not fall into any of the previous categories.

| | | |
|---|---|---|
| 335544321L | isc_arith_except | "arithmetic exception, numeric overflow, or string truncation"<br><br>Happens at runtime when an arithmetic exception happens, like division by zero or the numeric overflow (e.g. number does not fit the 64 bits limit).<br><br>Another source of this error are string operations, like string concatenation producing a too long string, impossibility to transliterate characters between character sets, etc.<br><br>Future versions of Firebird will provide a secondary code to distinguish the exact reason of an error. |
| 335544348L | isc_no_cur_rec | "no current record for fetch operation"<br><br>Happens when the application asks Firebird to fetch a record, but no record is available for fetching.<br><br>Java applications should never get this error, since checks in the JDBC driver should prevent the application from executing a fetch operation on the server side. |
| 335544374L | isc_stream_eof | "attempt to fetch past the last record in a record stream"<br><br>Application tries to execute fetch operation after all records have already been fetched.<br><br>Similar to the previous error, Java applications should not get this error due to the checks that happen before issuing the fetch request to the server. |
| 335544517L | isc_except | "exception {0}"<br><br>An custom exception has been raised on the server. Java application can examine the underlying GDSException to extract the exception message. |

| | | |
|---|---|---|
| 335544721L | `isc_network_error` | `Unable to complete network request to host "{0}"`<br><br>This error is thrown when Jaybird cannot establish a connection to the database server due to a network issues, e.g. host name is specified incorrectly, Firebird has not been started on the remote host, firewall configuration prevents client from establishing the connection, etc. |

[2] it is possible sometimes X/Open SQLstates are used

[3] excluding those defined for `javax.sql.rowset`

# Chapter 4. Executing statements

After obtaining a connection, the next thing to do is to execute an SQL statement. The JDBC specification distinguishes three kinds of statements:

1. Regular statements to execute fixed SQL requests,
2. prepared statements to execute SQL code with parameters,
3. and callable statements to execute stored procedures.

## 4.1. The java.sql.Statement interface

The `java.sql.Statement` interface is the simplest interface to execute SQL statements. It distinguishes three types:

- statements that return results, or, in other words, queries;
- statements that change the state of the database but return no results;
- `INSERT` statements (or other statements with similar behaviour) that return the values of the columns which were generated by the database engine while inserting the record.

Let's check one of the typical usages shown below. In general the usage pattern of the statement consists of three steps

*Typical way to execute query to get information about the user*

```
try (Statement stmt = connection.createStatement(); ①
     ResultSet rs = stmt.executeQuery(
         "SELECT firstName, lastName FROM users" +
         " WHERE userId = 5") ②
){
  if (rs.next()) {
    String firstName = rs.getString(1);
    String lastName = rs.getString(2);
  }
} ③
```

① Create a `Statement` object by calling the `createStatement()` method of the `Connection` object.

② Use the Statement object by calling its methods, in our case we execute a simple query `SELECT firstName, lastName FROM users WHERE userId = 5`. Processing of the query result will be discussed in details in Working with result sets.

③ Close the result set and statement to release all allocated resources. In our example this is done using the try-with-resources block. With try-with-resources. Java takes care of closing resources in the right order, even if exceptions occur, or if a resource was not allocated (say, if `executeQuery` throws an exception).

As the connection object is the factory for the statement objects, this puts a constraint on the object lifetime: statements are bound to the connection; when the connection is closed, all statements that

were created by that connection become invalid and the resources allocated by them are released. However, despite that these resources are released when the connection closes, it is strongly recommended to use the try-with-resources block, to guarantee that resources are released as soon as possible because of reasons that will be discussed later.

A statement can be executed using the following methods:

- `Statement.executeQuery(String)` – executes a SELECT statement and returns a result set. If the specified statement does not produce a result set, an `SQLException` is thrown after statement execution.
- `Statement.executeUpdate(String)` – executes INSERT, UPDATE, DELETE or DDL[4] statements and returns the number of updated rows. If the specified statement is a query, an `SQLException` is thrown.
- `Statement.execute(String)` – executes a statement and returns `true` when the statement returned a result set, otherwise an update was executed and `false` is returned. You can use `Statement.getResultSet()` method to get the result of the executed query or you can use `Statement.getUpdateCount()` when you have executed update statement.

These `execute` methods have several variants for additional features covered later (**TODO** check if already covered).

A statement is closed by calling the `Statement.close()` method, or by using a try-with-resources which calls `close()` behind the scenes. After a close, the statement object is invalid and cannot be used any more.

It is allowed to use the same statement object to execute different types of queries one after other. The code below contains a short example which first performs a select to find the ID of the user 'Joe Doe', and if the record is found, it enables his account.

The concatenation of values into a query string as done in this example is not a good practice as it can leave your code vulnerable to SQL injection.

In this specific case it is safe to do as the values are integers. In general: don't do this, use a prepared statement instead.

*Using the same statement object multiple times to enable user account*

```
try (Statement stmt = connection.createStatement();
     ResultSet rs = stmt.executeQuery(
         "SELECT userId FROM users " +
         "WHERE lastName = 'Doe' AND firstName = 'Joe'")) {
  if (rs.next()) {
    int userId = rs.getInt(1);

    int rowsUpdated = stmt.executeUpdate(
        "UPDATE accounts SET accountEnabled = 1 " +
        "WHERE userId = " + userId);

    if (rowsUpdated == 0)
      rowsUpdated = stmt.executeUpdate(
          "INSERT INTO accounts (userId, enabled) " +
          "VALUES (" + userId + ", 1)");

    if (rowsUpdated != 1)
      throw new SomeException(
          "User was not updated correctly.");
  }
}
```

The way the code is constructed is quite tricky because of the result set lifetime constraints that are defined by the JDBC specification, please read the chapter Working with result sets for more details.

However, here it is done intentionally to emphasize that a single object is used to execute `SELECT` and `UPDATE`/`INSERT` statements. It also shows how to check whether the executed statement modified expected number of rows – the application first tries to update the account and only if no rows were updated, it inserts new record into the `accounts` table.

This example of 'try update, then insert' approach can be better handled using `MERGE` [https://www.firebirdsql.org/file/documentation/reference_manuals/fblangref25-en/html/fblangref25-dml-merge.html] or `UPDATE OR INSERT` [https://www.firebirdsql.org/file/documentation/reference_manuals/fblangref25-en/html/fblangref25-dml-update-or-insert.html].

When an application needs to execute DDL statements, it is recommended to use the `Statement.execute(String)` method, as in this case the amount of modified records makes little sense. The next example shows the creation of database tables using this method.

*Example of creating database tables*

```
try (Statement stmt = connection.createStatement()) {
  stmt.execute("CREATE TABLE customer(" +
      "customerId INTEGER NOT NULL PRIMARY KEY, " +
      "firstName VARCHAR(20) NOT NULL, " +
      "lastName VARCHAR(40) NOT NULL)");
}
```

As mentioned earlier, the `Statement.execute(String)` method can also be used to execute statements of an unknown type.

```
try (Statement stmt = connection.createStatement()) {
  boolean hasResultSet = stmt.execute(sql);
  if (hasResultSet) {
    ResultSet rs = stmt.getResultSet();
    ...
  } else {
    int updateCount = stmt.getUpdateCount();
    ...
  }
}
```

It is worth mentioning, that according to the JDBC specification `getResultSet()` and `getUpdateCount()` methods can be only called once per result, and in case of using Firebird, that means once per executed statement, since Firebird does not support multiple results from a single statement. Calling the methods the second time will cause an exception.

## 4.2. Statement behind the scenes

The previous examples requires us to discuss the statement object dynamics, its life cycle and how it affects other subsystems in details.

### 4.2.1. Statement dynamics

When a Java application executes a statement, a lot more operations happen behind the scenes:

1. A new statement object is allocated on the server. Firebird returns a 32-bit identifier of the allocated object, a statement handle, that must be used in next operations.
2. An SQL statement is compiled into an executable form and is associated with the specified statement handle.
3. Jaybird asks the server to describe the statement and Firebird returns information about the statement type and possible statement input parameters (we will discuss this with prepared statements) and output parameters, namely the result set columns.
4. If no parameters are required for the statement, Jaybird tells Firebird to execute statement passing the statement handle into corresponding method.

After this Jaybird has to make a decision depending on the operation that was called.

- If `Statement.execute()` was used, Jaybird only checks the statement type to decide whether it should return true, telling the application that there is a result set for this operation, or false, if statement did not return any result set.
- If `Statement.executeUpdate()` was called, Jaybird asks Firebird to give the information about the number of affected rows. This method can be called only if the statement type tells that no result set can be returned by the statement.

When called for queries, an exception is thrown despite the fact that the statement was successfully executed on the server.

- If `Statement.executeQuery()` was called and the statement type indicates that a result set can be returned, Jaybird constructs a `ResultSet` object and returns it to the application. No additional checks, like whether the result set contains rows, are performed, as it is the responsibility of the `ResultSet` object.

  If this method is used for statements that do not return result set, an exception is thrown despite the fact that the statement was successfully executed on the server.

The described behaviour may change in the future by throwing the exception **before** executing the statement.

When an application does not need to know how many rows were modified, it should use the `execute()` method instead of `executeUpdate()`. This saves an additional call to the server to get the number of modified rows which can increase the performance in the situations where network latency is comparable with the statement execution times.

The `execute()` method is also the only method that can be used when the application does not know what kind of statement is being executed (for example, an application that allows the user to enter SQL statements to execute).

After using the statement object, an application should close it. Two different possibilities exist: to close the result set object associated with the statement handle and to close the statement completely.

If, for example, we want to execute another query, it is not necessary to completely release the allocated statement. Jaybird is required only to compile a new statement before using it, in other words we can skip step 1 (allocating a new statement handle). This saves us one round-trip to the server over the network, which might improve the application performance.

If we close the statement completely, the allocated statement handle is no longer usable. Jaybird could allocate a new statement handle, however the JDBC specification does not allow use of a `Statement` object after `close()` method has been called.

### 4.2.2. Statement lifetime and DDL

Step 2 (compiling the SQL statement) in the previous section is probably the most important, and usually, most expensive part of the statement execution life cycle.

When Firebird server receives the "prepare statement" call, it parses the SQL statement and converts it into the executable form: BLR. BLR, or Binary Language Representation, contains low-level commands to traverse the database tables, conditions that are used to filter records, defines the order in which records are accessed, indices that are used to improve the performance, etc.

When a statement is prepared, it holds the references to all database object definitions that are used during that statement execution. This mechanism preserves the database schema consistency, it saves the statement objects from "surprises" like accessing a database table that has been

removed by another application.

However, holding a reference on the database objects has one very unpleasant effect: it is not possible to upgrade the database schema, if there are active connections to the database with open statements referencing the objects being upgraded. In other words, if two application are running and one is trying to modify the table, view, procedure or trigger definition while another one is accessing those objects, the first application will receive an error 335544453 "object is in use".

To avoid this problem, it is strongly recommended to close the statement as soon as it is no longer needed. This invalidates the BLR and release all references to the database objects, making them available for the modification.

Special care should be taken when the statement pooling is used. In that case statements are not released even if the `close()` method is called. The only possibility to close the pooled statements is to close the pooled connections. Please check the documentation of your connection pool for more information.

## 4.3. The java.sql.PreparedStatement interface

As we have seen, Jaybird already performs internal optimization when it comes to multiple statement execution – it can reuse the allocated statement handle in subsequent calls. However this improvement is very small and sometimes can even be neglible compared to the time needed to compile the SQL statement into the BLR form.

The `PreparedStatement` interface addresses such inefficiencies. An object that implements this interface represents a precompiled statement that can be executed multiple times. If we use the execution flow described in the "Statement dynamics" section, it allows us to go directly to the step 4 for repeated executions.

However, executing the same statement with the same values makes little sense, unless we want to fill the table with the same data, which usually is not the case. Therefore, JDBC provides support for parametrized statements – SQL statements where literals are replaced with question marks (`?`), so called positional parameters. The application then assigns values to the parameters before executing the statement.

Our first example in this chapter can be rewritten as shown below. At first glance the code becomes more complicated without any visible advantage.

*Example for user account update rewritten using prepared statements*

```
try (PreparedStatement stmt1 = connection.prepareStatement(
         "SELECT userId FROM users WHERE " +
         "lastName = ? AND firstName = ?")) {
  stmt1.setString(1, "Doe");
  stmt1.setString(2, "Joe");
  try (ResultSet rs = stmt1.executeQuery()) {

    if (rs.next()) {
      int userId = rs.getInt(1);

      try (PreparedStatement stmt2 =
               connection.prepareStatement(
                   "UPDATE accounts SET accountEnabled = 1 " +
                   "WHERE userId = ?" )) {
        stmt2.setInt(1, userId);

        int rowsUpdated = stmt2.executeUpdate();

        if (rowsUpdated == 0) {
          try (PreparedStatement stmt3 =
                   connection.prepareStatement(
                       "INSERT INTO accounts " +
                       "(userId, enabled) VALUES (?, 1)")) {
            stmt3.setInt(1, userId);
            rowsUpdated = stmt3.executeUpdate();
          }
        }
        if (rowsUpdated != 1)
          throw new SomeException(
              "User was not updated correctly.");
      }
    }
  }
}
```

- First, instead of using just one statement object we have to use three – one per statement.
- Second, before executing the statement we have to set parameters first. As is shown in the example, parameters are referenced by their position. The `PreparedStatement` interface provides setter methods for all primitive types in Java as well as for some widely used SQL data types (BLOBs, CLOBs, etc.). The `NULL` value is set by calling the `PreparedStatement.setNull(int)` method.
- Third, we are now forced to use four nested try-with-resources blocks, which makes code less readable.

So, where's the advantage? First of all, prepared statements parameters protect against SQL injection as the values are sent separately from the statement itself. It is not possible to change the meaning of a statement due to incorrect string concatenation, so data leaks or other problems cause by SQL injection can be avoided. Second of all, the driver handles conversion of Java object

types to the correct format for the target datatype in Firebird: you don't need to convert a Java value to the correct string literal format for Firebirds SQL dialect.

To address some of the identified problems, we can redesign our application to prepare those statements before calling that code (for example in a constructor) and close them when the application ends. In that case the code can be more compact (see the next example). Unfortunately, the application is now responsible for prepared statement management. When a connection is closed, the prepared statement object will be invalidated, but the application will not be notified. And when the application uses similar statements in different parts of the application, the refactoring might affect many classes, possibly destabilizing the code. So, the refactoring on this example is not something we want to do.

*Rewritten example to let application manage prepared statements*

```
// prepared statement management
PreparedStatement queryStmt =
    connection.prepareStatement(queryStr);
PreparedStatement updateStmt =
    connection.prepareStatement(updateStr);
PreparedStatement insertStmt =
    connection.prepareStatement(insertStr);

.....................

// query management
queryStmt.clearParameters();
queryStmt.setString(1, "Doe");
queryStmt.setString(2, "Joe");
try (ResultSet rs = queryStmt.executeQuery()) {
  if (rs.next()) {
    int userId = rs.getInt(1);

    updateStmt.clearParameters();
    updateStmt.setInt(1, userId);
    int rowsUpdated = updateStmt.executeUpdate();

    if (rowsUpdated == 0) {
      insertStmt.clearParameters();
      insertStmt.setInt(1, userId);
      rowsUpdated = insertStmt.executeUpdate();
    }

    if (rowsUpdated != 1)
      throw new SomeException(
          "User was not updated correctly.");
  }
}
.....................

// prepared statement cleanup
insertStmt.close();
updateStmt.close();
queryStmt.close();
```

The answer to the advantage question is hidden in the `prepareStatement(String)` call. Since the same statement can be used for different parameter values, the connection object could have a possibility to perform prepared statement caching. A JDBC driver can ignore the request to close the prepared statement, save it internally and reuse it each time application asks to prepare an SQL statement that is known to the connection.

Jaybird currently does not perform statement caching

# 4.4. The java.sql.CallableStatement interface

The `CallableStatement` interface extends `PreparedStatement` with methods for executing and retrieving results from stored procedures. It was introduced in JDBC specification in order to unify access to the stored procedures across the database system. The main difference to `PreparedStatement` is that the procedure call is specified using the portable escaped syntax:[5]

*Unified escaped syntax for stored procedure execution*

```
procedure call ::= {[?=]call <procedure-name>(<params>)}
params ::= <param> [, <param> ...]
```

Each stored procedure is allowed to take zero or more input parameters, similar to the `PreparedStatement` interface. After being executed, a procedure can either return data in the output parameters or it can return a result set that can be traversed. Though the interface is generic enough to support database engines that can return both and have multiple result sets. These features are of no interest to Jaybird users, since Firebird does not support them.

The IN and OUT parameters are specified in one statement. The syntax above does not allow to specify the type of the parameter, therefore additional facilities are needed to tell the driver which parameter is will contain output values, the rest are considered to be IN parameters.

## 4.4.1. Firebird stored procedures

Firebird stored procedures represent a piece of code written in the PSQL language that allows SQL statement execution at the native speed of the engine and provides capabilities for a limited execution flow control. The PSQL language is not general purpose language therefore its capabilities are limited when it comes to interaction with other systems.

Firebird stored procedures can be classified as follow:

- Procedures that do not return any results. These are stored procedures that do not contain the `RETURNS` keyword in their header.
- Procedures that return only a single row of results. These are stored procedures that contain the `RETURNS` keyword in their header, but do not contain the `SUSPEND` keyword in their procedure body. These procedures can be viewed as functions that return multiple values. These procedures are executed by using the `EXECUTE PROCEDURE` statement.
- Procedures that return a result set, also called "selectable stored procedures". These are stored procedures that contain the `RETURNS` keyword in their header and the `SUSPEND` keyword in their procedure body, usually within a loop. Selectable procedures are executed using the `"SELECT * FROM myProcedure()"` SQL statement. It is also allowed to use the `EXECUTE PROCEDURE` statement, however that might produce strange results, since for selectable procedures is is equivalent to executing a `SELECT` statement, but doing only one fetch after the select. If the procedure implementation relies on the fact that all rows that it returns must be fetched, the logic will be broken.

Consider the following stored procedure that returns factorial of the specified number.

*Source code for the procedure that multiplies two integers*

```
CREATE PROCEDURE factorial(
  max_value INTEGER
) RETURNS (
  factorial INTEGER
) AS
  DECLARE VARIABLE temp INTEGER;
  DECLARE VARIABLE counter INTEGER;
BEGIN
  counter = 0;
  temp = 1;
  WHILE (counter <= max_value) DO BEGIN
    IF (counter = 0) THEN
      temp = 1;
    ELSE
      temp = temp * counter;
    counter = counter + 1;
  END
  factorial = temp;
END
```

This procedure can be executed using the EXECUTE PROCEDURE call. When it is done in isql, the output looks as follow

*Output of the EXECUTE PROCEDURE call in isql*

```
SQL> EXECUTE PROCEDURE factorial(5);

   FACTORIAL
============
         120
```

Now let's modify this procedure to return each intermediate result to the client.

*Modified procedure that returns each intermediate result*

```
CREATE PROCEDURE factorial_selectable(
  max_value INTEGER
) RETURNS (
  row_num INTEGER,
  factorial INTEGER
) AS
  DECLARE VARIABLE temp INTEGER;
  DECLARE VARIABLE counter INTEGER;
BEGIN
  counter = 0;
  temp = 1;
  WHILE (counter <= max_value) DO BEGIN
    IF (row_num = 0) THEN
      temp = 1;
    ELSE
      temp = temp * counter;
    factorial = temp;
    row_num = counter;
    counter = counter + 1;
    SUSPEND;
  END
END
```

If you create this procedure using the isql command line tool and then issue the `"SELECT * FROM factorial_selectable(5)"` statement, the output will be like this:

*Output of the modified procedure*

```
SQL> SELECT * FROM factorial_selectable(5);

     ROW_NUM    FACTORIAL
============ ============
           0            1
           1            1
           2            2
           3            6
           4           24
           5          120
```

### 4.4.2. Using the CallableStatement

Let's see how the procedures defined above can be accessed from Java.

First, we can execute this procedure from the first example in the previous section using the EXECUTE PROCEDURE statement and `PreparedStatement`, however this approach requires some more code for result set handling.

*Example of using the PreparedStatement to call executable procedure*

```
try (PreparedStatement stmt = connection.prepareStatement(
        "EXECUTE PROCEDURE factorial(?)")) {

  stmt.setInt(1, 2);

  try (ResultSet rs = stmt.executeQuery()) {
    rs.next(); // move cursor to the first row

    int result = rs.getInt(1);
  }
}
```

However, the standard for calling stored procedures in JDBC is to use the `CallableStatement`. The call can be specified using the escaped syntax, but native Firebird `EXECUTE PROCEDURE` syntax is also supported.

*Accessing the executable procedure via CallableStatement*

```
try (CallableStatement stmt = connection.prepareCall(
        "{call factorial(?,?)}")) {

  stmt.setInt(1, 2);
  stmt.registerOutParameter(2, Types.INTEGER);

  stmt.execute();

  int result = stmt.getInt(2);
}
```

Please note the difference in the number of parameters used in the examples. The first example contained only IN parameter on position 1 and the OUT parameter was returned in the `ResultSet` on the first position, so it was accessed via index 1.

The latter example additionally contains the OUT parameter in the call. We have used the `CallableStatement.registerOutParameter` method to tell the driver that the second parameter in our call is an OUT parameter of type INTEGER. Parameters that were not marked as OUT are considered by Jaybird as IN parameters. Finally the `"EXECUTE PROCEDURE factorial(?)"` SQL statement is prepared and executed. After executing the procedure call we get the result from the appropriate getter method.

It is worth mentioning that the stored procedure call preparation happens in the `CallableStatement.execute` method, and not in the `prepareCall` method of the `Connection` object. Reason for this deviation from the specification is that Firebird does not allow to prepare a procedure without specifying parameters and set them only after the statement is prepared. It seems that this part of the JDBC specification is modelled after the Oracle RDBMS and a workaround for this issue had to be delivered. Another side effect of this issue is, that it is allowed to intermix input and output parameters, for example in the "IN, OUT, IN, OUT, OUT, IN" order. Not

that it makes much sense to do this, but it might help in some cases when porting applications from another database server.

It is also allowed to use a procedure call parameter both as an input and output parameter. It is recommended to use this only when porting applications from the database servers that allow INOUT parameter types, such as Oracle.

The actual stored procedure call using the `CallableStatement` is equivalent to the call using the prepared statement as shown in the first example. There is no measurable performance differences when using the callable statement interface.

The JDBC specification allows another syntax for the stored procedure calls:

*Calling stored procedure using different syntax*

```
try (CallableStatement stmt = connection.prepareCall(
         "{?= call factorial(?}")) {

  stmt.registerOutParameter(1, Types.INTEGER);
  stmt.setInt(2, 2);

  stmt.execute();

  int result = stmt.getInt(1);
}
```

Note, that input parameter now has index 2, and not 1 as in the previous example. This syntax seems to be more intuitive, as it looks like a function call. It is possible to use this syntax for stored procedures that return more than one parameter by combining code from the second and the last examples.

Firebird stored procedures can also return result sets. This is achieved by using the SUSPEND keyword inside the procedure body. This keyword returns the current values of the output parameters as a single row to the client.

The following example is more complex and shows a stored procedure that computes a set of factorial of the numbers up to the specified number of rows.

The SELECT SQL statement is the natural way of accessing the selectable procedures in Firebird. You "select" from such procedures using the `Statement` or `PreparedStatement` objects.

With minor issues it is also possible to access selectable stored procedures through the `CallableStatement` interface. The escaped call must include all IN and OUT parameters. After the call is prepared, parameters are set the same way. However, the application must explicitly tell the driver that selectable procedure is used and access to the result set is desired. This is done by calling a Jaybird-specific method as shown in the example below. When this is not done, the application has access only to the first row of the result set. **TODO** Outdated?

The getter methods from the `CallableStatement` interface will provide you access only to the first row of the result set. In order to get access to the complete result set you have to either call the

`executeQuery` method or the `execute` method followed by `getResultSet` method.

*Example of using selectable stored procedure via escaped syntax*

```
import java.sql.*;
import org.firebirdsql.jdbc.*;
...
try (CallableStatement stmt = connection.prepareCall(
         "{call factorial(?, ?, ?)}")) {

  FirebirdCallableStatement fbStmt =
      (FirebirdCallableStatement)stmt;
  fbStmt.setSelectableProcedure(true);

  stmt.setInt(1, 5);
  stmt.registerOutParameter(2, Types.INTEGER); // first OUT
  stmt.registerOutParameter(3, Types.INTEGER); // second OUT

  try (ResultSet rs = stmt.executeQuery()) {

    while(rs.next()) {
      int firstCol = rs.getInt(1);             // first OUT
      int secondCol = rs.getInt(2);            // second OUT
      int anotherSecondCol = stmt.getInt(3);   // second OUT
    }
  }
}
```

Note that OUT parameter positions differ when they are accessed through the `ResultSet` interface (the `firstCol` and `secondCol` variables in our example). They are numbered in the order of their appearance in the procedure call starting with 1.

When OUT parameter is accessed through the `CallableStatement` interface (the `anotherSecondCol` parameter in our example), the registered position should be used. In this case the result set can be used for navigation only.

### 4.4.3. Describing Output and Input Parameters

The `PreparedStatement.getMetaData` method is used to obtain description of the columns that will be returned by the prepared SELECT statement. The method returns an instance of `java.sql.ResultSetMetaData` interface that among other descriptions provides the following:

- column type, name of the type, its scale and precision if relevant;
- column name, its label and the display size;
- name of the table, to which this column belongs;
- information whether the column is read-only or writable, whether it contains signed numbers, whether it can contains NULL values, etc.

Additionally, the JDBC 3.0 specification introduced the interface `java.sql.ParameterMetaData` that

provides similar information for the input parameters of both `PreparedStatement` and `CallableStatement` objects.

Due to the implementation specifics of the escaped syntax support for callable statements, it is not allowed to call `getParameterMetaData` before all OUT parameters are registered. Otherwise the driver will try to prepare a procedure with an incorrect number of parameters and the database server will generate an error.

# 4.5. Batch Updates

Batch updates are intended to group multiple update operations to be submitted to a database server to be processed at once. Firebird does not provide support for such functionality[6], but Jaybird emulates it by issuing separate update commands.

## 4.5.1. Batch Updates with java.sql.Statement interface

The `Statement` interface defines three methods for batch updates: `addBatch`, `executeBatch` and `clearBatch`. It is allowed to add arbitrary INSERT/UPDATE/DELETE or DDL statement to the batch group. Adding a statement that returns result set is an error.

*Example of batch updates using Statement object*

```
try (Statement stmt = connection.createStatement()) {

  stmt.addBatch("UPDATE products " +
      "SET amount = amount  1 WHERE id = 1");
  stmt.addBatch("INSERT INTO orders(id, amount) VALUES(1, 1)");

  int[] updateCounts = stmt.executeBatch();
}
```

The JDBC specification recommends to turn the auto-commit mode off to guarantee standard behavior for all databases. The specification explicitly states that behavior in auto-commit case is implementation defined.

In auto-commit mode, Jaybird executes a batch in a single transaction, i.e. the "all-or-nothing" principle. A new transaction is started before the batch execution and is committed if there were no exceptions during batch execution, or is rolled back if at least one batch command generated an error.

The `Statement.executeBatch` method submits the job to the database server. In case of successful execution of the complete batch, it returns an array of integers containing update counts for each of the commands. Possible values are:

- 0 or positive value – an update count for the corresponding update/DDL statement.
- `Statement.SUCCESS_NO_INFO` – driver does not have any information about the update count, but it knows that statement was executed successfully.

The `Statement.executeBatch` method closes the current result set if one is open. After successful execution the batch is cleared. Calling `execute`, `executeUpdate` and `executeQuery` before the batch is executed does not have any effect on the currently added batch statements.

If at least one statement from the batch fails, a `java.sql.BatchUpdateException` is thrown. Jaybird will stop executing statements from batch after the first error. In auto-commit mode it will also rollback the transaction. An application can obtain update counts for the already executed statements using `getUpdateCounts` method of the `BatchUpdateException` class. The returned array will always contain fewer entries than there were statements in the batch, as it will only report the update counts of succesfully executed statements.

### 4.5.2. Batch Updates with java.sql.PreparedStatement and java.sql.CallableStatement

Using batch updates with a prepared statement is conceptually similar to the `java.sql.Statement` approach. The main difference is that only one statement can be used, but with different sets of parameter values.

*Example of batch updates with PreparedStatement*

```
try (PreparedStatement stmt = connection.prepareStatement(
         "INSERT INTO products(id, name) VALUES(?, ?)")) {

  stmt.setInt(1, 1);
  stmt.setString(2, "apple");
  stmt.addBatch();

  stmt.setInt(1, 2);
  stmt.setString(2, "orange");
  stmt.addBatch();

  int[] updateCounts = stmt.executeBatch();
}
```

*Example of batch updates with CallableStatement*

```
try (CallableStatement stmt = connection.prepareCall(
         "{call add_product(?, ?)")) {

  stmt.setInt(1, 1);
  stmt.setString(2, "apple");
  stmt.addBatch();

  stmt.setInt(1, 2);
  stmt.setString(2, "orange");
  stmt.addBatch();

  int[] updateCounts = stmt.executeBatch();
}
```

# 4.6. Escape Syntax

The escape syntax was introduced as a portable JDBC-specific syntax to represent parts of the SQL language that are (or were) usually implemented differently by database vendors. The escaped syntax is also used to define features that might not be implemented by the database server, but can have an appropriate implementation in the driver.

The JDBC specification defines escaped syntax for the following

- scalar functions
- date and time literals
- outer joins
- calling stored procedures
- escape characters for LIKE clauses

## 4.6.1. Scalar Functions

Escaped syntax for the scalar function call is defined as

```
{fn <function-name> (argument list)}
```

For example `{fn concat('Firebird', 'Java')}` concatenates these two words into `'FirebirdJava'` literal. "Supported JDBC Scalar Functions" provides a list of supported scalar functions.

## 4.6.2. Date and Time Literals

It is possible to include date and time literals in SQL statements. In order to guarantee that each database will interpret the literal identically, the JDBC specification provides following syntax to specify them:

Date literal escaped syntax:

```
{d 'yyyy-mm-dd'}
```

Time literal escaped syntax:

```
{t 'hh:mm:ss'}
```

Timestamp literal syntax (fractional seconds part `'.f'` can be omitted):

```
{ts 'yyyy-mm-dd hh:mm:ss.f...'}
```

### 4.6.3. Outer Joins

Due to the various approaches to specify outer joins (for instance, the Oracle "(+)" syntax), the JDBC specification provides the following syntax:

```
{oj <outer join>}
```

where the outer join is specified as

```
<outer join> ::=
    <table name> {LEFT|RIGHT|FULL} OUTER JOIN
    {<table name> | <outer join>} ON >search condition>
```

An example SQL statement would look like this:

```
SELECT * FROM {oj tableA a
    LEFT OUTER JOIN tableB b ON a.id = b.id}
```

### 4.6.4. Stored Procedures

The escaped syntax for stored procedures is described in details in the the section The java.sql.CallableStatement interface.

### 4.6.5. LIKE Escaped Characters

The percent sign (%) and underscore (_) characters are wild cards in LIKE clause of the SQL statement. In order to interpret them literally they must be preceded by the backslash character (\) that is called the escape character. The escaped syntax for this case identifies which character is used as an escape character:

```
{escape '<escape character>'}
```

[4] DDL – Data Definition Language. This term is used to group all statements that are used to manipulate database schema, i.e. creation of tables, indices, views, etc.

[5] escape syntax in limited form also works for `Statement` and `PreparedStatement`

[6] Support is expected in Firebird 4 for prepared statements, Jaybird will add support in a future version

# Chapter 5. Working with result sets

When a SELECT statement is executed, the results of the query are returned through the implementation of the `java.sql.ResultSet` interface.

## 5.1. ResultSet properties

### 5.1.1. ResultSet Types

The JDBC specification defines three types of result sets

- `TYPE_FORWARD_ONLY` – the result set is not scrollable, cursor can only move forward. When the `TRANSACTION_READ_COMMITTED` isolation level is used, the result set will return all rows that are satisfying the search condition at the moment of fetch (which will be every *fetch size* calls to `ResultSet.next()`). In other cases result set will return only rows that were visible at the moment of the transaction start.
- `TYPE_SCROLL_INSENSITIVE` – the result set is scrollable, the cursor can move back and forth, can be positioned on the specified row. Only rows satisfying the condition at the time of query execution are visible.
- `TYPE_SCROLL_SENSITIVE`, is not supported by Firebird and Jaybird. Jaybird allows an application to ask for this type of result set, however in compliance with the JDBC specification, the type is "downgraded" to the `TYPE_SCROLL_INSENSITIVE` and a corresponding warning is added to the connection object.

Due to a missing support of scrollable cursors in Firebird, this support (`TYPE_SCROLL_INSENSITIVE` result set type) is implemented by fetching the complete result set to the client. Scrolling happens in the memory on the client. This can have adverse effect on the system memory usage and performance when the result set is large.

### 5.1.2. ResultSet Concurrency

Result set concurrency specifies whether the result set object can be updated directly or a separate SQL request should be used to update the row. Result sets that allow direct modification using the `ResultSet.updateXXX` methods are usually used in GUI applications which allow in-place editing of the underlying result set.

The result set concurrency is specified during statement creation and cannot be changed later. Jaybird supports two types of result set concurrency:

- `CONCUR_READ_ONLY` is available for all types of result sets. It tells the driver that direct update of the result set is not possible and all `ResultSet.updateXXX` methods should throw an exception.
- `CONCUR_UPDATABLE` is supported only under certain conditions that are needed for the driver to correctly construct a DML request that will modify exactly one row. These conditions are:
  - the SELECT statement that generated the result set references only one table;

- all columns that are not referenced by the SELECT statement allow `NULL` values, otherwise it won't be possible to insert new rows;
- the SELECT statement does not contain the `DISTINCT` predicate, aggregate functions, joined tables, or stored procedures;
- the SELECT statement references all columns of the tables primary key definition or the `RDB$DB_KEY` column.

### 5.1.3. ResultSet Holdability

Result set holdability informs the driver whether result sets should be kept open across commits. `ResultSet.HOLD_CURSORS_OVER_COMMIT` tells the driver to keep the result set object open, while `ResultSet.CLOSE_CURSORS_AT_COMMIT` tells driver to close them on commit.

When an application calls `Connection.commit()`, the Firebird server closes all open result sets. It is not possible to tell the server to keep a result set open over commit unless "commit retaining" mode is used. This mode is global for the complete connection and is not suitable for holdability control on a statement level. Use of "commit retaining" mode is believed to have an undesired side-effect for read-write transactions as it inhibits garbage collection. Because of these reasons "commit retaining" is not used in Jaybird during normal execution. Applications are able to commit the transaction keeping the result sets open by executing a `"COMMIT RETAIN"` SQL statement.

To support holdable result sets, Jaybird will upgrade the result set to `TYPE_SCROLL_INSENSITIVE` to cache all rows locally, even if you asked for a `TYPE_FORWARD_ONLY` result set. See also ResultSet Types.

## 5.2. ResultSet manipulation

`ResultSet` objects are created when either `Statement.executeQuery(String)` or `Statement.getResultSet()` methods are called, or when obtaining the generated keys from `Statement.getGeneratedKeys()`. `Statement.getResultSet()` is used in combination with `Statement.execute(String)` and can be called only once per result set (see the following two examples).

The current implementation does not allow to call `getResultSet()` method after using the `executeQuery(String)` method of the `Statement` class. The JDBC specification is unclear on this topic and JDBC drivers of different vendors treat it differently.

*Using Statement.executeQuery(String) method*

```
try (Statement stmt = connection.createStatement();
     ResultSet rs = stmt.executeQuery("SELECT * FROM myTable")) {
  // process result set
}
```

*Using Statement.getResultSet() method*

```
try (Statement stmt = connection.createStatement()) {
  boolean hasResultSet = stmt.execute("SELECT * FROM myTable");
  if (hasResultSet) {
    try (ResultSet rs = stmt.getResultSet()) {
      // process result set
    }
  }
}
```

### 5.2.1. Accessing the values in the result set

Depending on the type of the result set, it is possible to move the cursor either forward only (next example [#using-forward-only]) or using absolute and relative positioning (second example below [#using-scrollable-updatable]).

Values of the result set are obtained by calling the corresponding getter method depending on the type of column. For example the `ResultSet.getInt(1)` method returns the value of the first column as an `int` value. If value of the column is not integer, Jaybird tries to convert it according to the "Data Type Conversion Table" specified in Data Type Conversion Table. If conversion is not possible, an exception is thrown.

There are two possibilities to obtain data from the result set columns: by column label or by column position. Position of the first column is 1. Names supplied to getter methods are case-insensitive. The search first happens in the column aliases, and if no match found, driver checks the original column names. If there is more then one column matching the specified name (even if the original names were quoted), the first match is taken. **TODO: Behavior changed with Jaybird 2.2, verify and update**

When getters for primitive types are used and original value in the result set is `NULL`, driver returns a default value for that type. For example `getInt()` method will return 0. In order to know whether the value is really 0 or NULL, you have to call `ResultSet.wasNull()` method **after** calling the *get* method.

Getters that return object values (`getString`, `getDate`, `getObject`, etc.) will return a `null` value for columns containing `NULL`. Calling `wasNull` after object *get* methods is possible but unnecessary.

*Example of using forward-only result sets*

```
try (Statement forwardStatement = connection.createStatement();
     ResultSet rs = forwardStatement.executeQuery(
         "SELECT id, name, price FROM myTable")) {

  while(rs.next()) {
    int id = rs.getInt(1);
    String name = rs.getString("name");
    double price = rs.getDouble(3);
  }
}
```

### 5.2.2. Updating records in the result set

Scrollable cursors are especially useful when result of some query is displayed by the application which also allows the user to directly edit the data and post the changes to the database.

*Example of using scrollable and updatable result sets*

```
Statement scrollStatement = connection.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_UPDATABLE);

ResultSet rs = scrollStatement.executeQuery(
    "SELECT id, name, price FROM myTable");

rs.absolute(1);                   // move to the first row
rs.updateString(2, anotherName); // update the name
rs.updateRow();                   // post changes to the db

rs.moveToInsertRow();
rs.updateInt(1, newId);
rs.updateString(2, newName);
rs.updateDouble(3, newPrice);
rs.insertRow();
rs.moveToCurrentRow();

rs.relative(-2);
```

The code example above shows how to update first row, insert new one and after that move two records backwards.

An application can also update the current row using so called "positioned updates" on named cursors. This technique can be used only with forward-only cursors, since application can update only the row to which the server-side cursor points to. In case of scrollable cursors the complete result set is fetched to the client and then the server-side cursor is closed. The example below [#using-positioned-updates] shows how to use positioned updates.

First, the application has to specify the name of the cursor and the list of the columns that will be updated before the query is executed. This name is later used in the `UPDATE` statement as shown in the example.

*Example of using the positioned updates*

```
connections.setAutoCommit(false);
try (Statement selectStmt = connection.createStatement();
     Statement updateStmt = connection.createStatement()) {
  selectStmt.setCursorName("someCursor");

  try (ResultSet rs = selectStmt.executeQuery(
           "SELECT id, name, price FROM myTable " +
           "FOR UPDATE OF myColumn")) {

    while(rs.next()) {
      ...
      if (someCondition) {
        updateStmt.executeUpdate("UPDATE myTable " +
            "SET myColumn = myColumn + 1 " +
            "WHERE CURRENT OF " + rs.getCursorName());
      }
    }
  }
}
```

### 5.2.3. Closing the result set

A result set is closed by calling the `ResultSet.close()` method. This releases the associated server resources and makes the `ResultSet` object available for garbage collection. It is strongly recommended to explicitly close result sets in auto-commit mode or `ResultSet.TYPE_SCROLL_INSENSITIVE` result sets, because this releases memory used for the cached data. Whenever possible, use try-with-resources.

The result set object is also closed automatically, when the statement that created it is closed or re-executed. In auto-commit mode, the result set is closed automatically if any statement is executed on the same connection.

# Chapter 6. Using transactions

Transactions are used to group SQL statements into a single block that satisfies so called ACID properties: atomicity, consistency, isolation and durability. In other words, all statements executed within transaction will either succeed and their results will be permanently stored in the database or the effect of the statement execution will be undone.

## 6.1. JDBC transactions

Firebird supports multiple concurrent transactions over the same database connection. This allows applications that work via the native Firebird API to save the number of network connections, which in turn saves the resources on the server[7].

This model however cannot be applied to each database engine in the world and the designers of the JDBC API have chosen a model where each database connection has one and only one active transaction associated with it. Also, unlike the Firebird model, where transactions require explicit start, JDBC specification requires the driver to start transaction automatically as soon as a transactional context is needed.

The following code shows a very simple example of using transactions in JDBC where a hypothetical intruder that increases salary of each employee twice and uses explicit transaction control in JDBC. It also tries to hide his own identity and if the operations succeed, he commits the transaction, otherwise he rolls the changes back.

*Example of explicit transaction control*

```
Connection connection = ...

connection.setAutoCommit(false); ①

Statement stmt = connection.createStament();
try {
  stmt.executeUpdate("UPDATE employee " +
      " SET salary = salary * 2"); ②

  // ... do some more changes to database
  // to hide the identity of the person
  // that messed up the salary information
  // by deleting the audit trails logs, etc.

  stmt.executeUpdate("DELETE FROM audit_trails");

  connection.commit(); ③
} catch(SQLException ex) {
  connection.rollback(); ④
}
```

In order to use transactions, the application first switches the auto-commit mode off (see below for

more information), then creates a `java.sql.Statement` object, executes an UPDATE statement. Please note, that there is no explicit transaction start, a new transaction will be started right before executing the statement (step 2).

If we work with a database where not only referential integrity is preserved, but also reasonable security rules are encoded in the triggers, it will raise an error preventing cleaning the audit trails information. In this case the intruder chooses to undo all the changes he made, so that nobody notices anything. But if no security rules are implemented, he commits the transaction.

Firebird PSQL has an `IN AUTONOMOUS TRANSACTION` block that can be used to prevent such abuse and prevent audit-records from being wiped out by a transaction rollback.

When a connection uses explicit transaction control, each transaction must be finished by calling the `commit()` or `rollback()` methods of the `Connection` object before the connection is closed. If a transaction was not finished, but the `close` method is called, the active transaction is rolled back automatically. This also happens when the transaction was not finished, the connection was not closed explicitly and that `Connection` object became eligible for garbage collection. In this case, the `close()` method is implicitly invoked by the class finalizer, which in turn rolls the transaction back.

Given the non-deterministic delay of garbage collection, make sure to explicitly end the transaction and close the connection. Do not rely on the garbage collector for this.

## 6.2. Auto-commit mode

Each newly created connection by default has the auto-commit property enabled. In other words, the duration of the transaction is limited by the duration of statement execution, or more formally – the transaction is ended when the statement is completed. The point when statement execution is considered complete, is defined in the specification as:

*Rules when the statement is completed in auto-commit mode*

- For insert, update, delete and DDL statements, the statement is complete as soon as it has finished executing.
- For select statements, statement is complete when the associated result set is closed. The result set is closed as soon as one of the following occurs:
  - all of the rows have been retrieved
  - the associated Statement object is re-executed
  - another Statement object is executed on the same connection
- For CallableStatement objects, the statement is complete, when all of the associated result sets have been closed.

If there is an ongoing transaction and the value of the auto-commit property is changed, the current transaction is committed.

Note, when a connection is obtained via `javax.sql.DataSource` object and container managed

transactions are used (for example, the application is executing inside an EJB container), it is an error to call `setAutoCommit` method.

Special care should be taken when using multiple statements in auto-commit mode. The JDBC 2.0 specification did not fully define the rules for the statement completion as it did not define the behavior of multiple `ResultSet` objects created using the same connection object in auto-commit mode.

Since Firebird does not allow the result set to remain open after the transaction ends, Jaybird 1.5.x and below cached the complete result set in memory when the SELECT statements were executed and corresponding transaction was committed. This had an adverse effect on allocated memory when the result set is big, especially when it contains BLOB fields. The JDBC 3.0 specification addressed this unclear situation (see above) and Jaybird 2.1 was improved to correctly handle them. It also allowed to improve the memory footprint – the driver no longer caches non-scrollable and non-holdable result sets in memory.

However, some Java applications that do not conform the current JDBC specification might no longer work with Jaybird 2.1 and above unless additional steps are taken.

The piece of code below works perfectly with explicit transaction control. However, it won't work correctly with auto-commit with a driver - like Jaybird - that complies with the JDBC 3.0 specification, when the `selectStmt` and `updateStmt` object are created by the same connection object (step 1). When the UPDATE is executed in step 3, the result set produced by the SELECT statement must be closed before the execution. When the Java application tries to fetch the next record by calling the `rs.next()` method, it will receive an `SQLException` with a message *"The result set is closed"*.

The only correct solution to this situation is to fix the application by either using explicit transaction control, or by using two connection objects, one for SELECT statement and one for UPDATE statement.

*Non-compliant usage of nested statements in auto-commit mode*

```
Statement selectStmt = connection.createStatement(); ①
Statement updateStmt = connection.createStatement();

ResultSet rs = selectStmt.executeQuery(
    "SELECT * FROM myTable");

while(rs.next()) { ②
  int id = rs.getInt(1);
  String name = rs.getString(2);

  updateStmt.executeUpdate("UPDATE anotherTable SET " +
      " name = '" + name + "' WHERE id = " + id); ③
}
```

Unfortunately, not all applications can be changed either because there is no source code available or, simply, because any change in the code requires complete release testing of the software. To address this, Jaybird 2.1 introduced the connection parameter `defaultHoldable` which makes result

sets holdable by default. The holdable result sets will be fully cached in memory, but won't be closed automatically when transaction ends.[8] This property also affects the default holdability of result sets when auto-commit is disabled.

See Default holdable result sets for more information.

## 6.3. Read-only Transactions

A transaction can be declared read-only to reduce the possibility of lock conflicts. In general, this makes little sense for Firebird, because of its multi-generational architecture, where readers do not block writers and vice versa. However, in some cases it can be useful.

It is not allowed to connect with a read-write transaction to a database located on a read-only media, for example, a CD-ROM. The reason is that, in order to guarantee consistency of the read-write transactions, Firebird has to increase the transaction identifier when transaction ends, and to store the new value on the so-called Transaction Inventory Page even if no changes were made in that transaction. This requirement can be relaxed if transaction is declared read-only and the engine ensures that no data can be modified.

Another reason is that long running read-write transactions inhibit the process of collecting garbage, i.e. a process of identifying previous versions of the database records that are no longer needed and releasing the occupied space for the new versions. Without garbage collection the database size will grow very fast and the speed of the database operations will decrease, because the database engine will have to check all available record versions to determine the appropriate one.

Therefore, if you are sure that application won't modify the database in the transaction, use the `setReadOnly` method of the `java.sql.Connection` object to tell the server that the transaction is read-only.

## 6.4. Transaction Isolation Levels

The ANSI/ISO SQL standard defines four such levels, each next one weaker than the previous. These isolation levels are also used in the JDBC specification:

*Table 1. JDBC transaction isolation levels and their characteristics*

| JDBC isolation level | Description |
|---|---|
| TRANSACTION_SERIALIZABLE | Transactions with this isolation level prohibit phantom reads, the situation when one transaction reads all rows satisfying the WHERE condition, another transaction inserts a row satisfying that condition, and first transaction re-executes the statement. |
| TRANSACTION_REPEATABLE_READ | This isolation level prevents non-repeatable reads, a situation when a row is read in one transaction, then modified in another transaction, and later re-read in the first transaction. In this case different values had been read within the same transaction. |

| JDBC isolation level | Description |
|---|---|
| TRANSACTION_READ_COMMITTED | Transactions with this isolation level can see only committed records. However, it does not prevent so-called non-repeatable reads and phantom reads. |
| TRANSACTION_READ_UNCOMMITTED | The weakest isolation level, or better to say level with no isolation. Such transactions can see the not yet committed changes to the data in the database from the concurrently running transactions. |

Firebird, however, defines other isolation levels: `read_committed`, `concurrency` and `consistency`. Only the `read_committed` isolation level can be mapped to the same level defined by the ANSI/ISO SQL standard. Dirty reads are prevented, non-repeatable reads as well as phantom reads can occur.

The `concurrency` isolation level is stronger than repeatable read isolation defined in ANSI/SQL standard and satisfies the requirements of a serializable isolation level, however, unlike RDBMSes with locking concurrency control, it guarantees better performance.

And finally Firebird provides a `consistency` isolation level which in combination with table reservation feature guarantees the deadlock-free execution of transactions. A transaction will be prevented from starting if there is already another one with the overlapping sets of the reserved tables. This isolation level guarantees truly serial history of transaction execution.

In order to satisfy the JDBC specification Jaybird provides a following default mapping of the JDBC transaction isolation levels into Firebird isolation levels:

- TRANSACTION_READ_COMMITTED is mapped to `read_committed` isolation level in Firebird – any changes made inside a transaction are not visible outside a transaction until the transaction is committed. A transaction in read-committed mode sees all committed changes made by other transactions even if that happened after start of the current transaction.
- TRANSACTION_REPEATABLE_READ is mapped to `concurrency` isolation level in Firebird – any changes made inside this transaction are not visible outside a transaction until the transaction is committed. A transaction in repeatable-read sees only those changes that were committed before the transaction started. Any committed change in another transaction that happened after the start of this transaction is not visible in this transaction.
- TRANSACTION_SERIALIZABLE is mapped into `consistency` isolation level in Firebird – any modification to a table happens in serial way: all transactions wait until the current modification is done. This mode can be considered as a traditional pessimistic locking scheme, but the lock is placed on the whole table. See section "Table Reservation" for more information.

The default mapping is specified in the `isc_tpb_mapping.properties` file that can be found in the Jaybird archive and can be overridden via the connection properties

- via the `tpbMapping` property that specifies the path to the `PropertiesResourceBundle` with the new mapping of the isolation level;
- via the direct specification of the JDBC transaction isolation level. The following code contains an example of such operation, the values in the mapping are described in section "Transaction Parameter Buffer".

- via the data source configuration.

*Overriding the default isolation level mapping*

```
Properties props = new Properties();
props.setProperty("user", "SYSDBA");
props.setProperty("password", "masterkey");
props.setProperty("TRANSACTION_READ_COMMITTED",
    "isc_tpb_read_committed,isc_no_rec_version," +
    "isc_tpb_write,isc_tpb_nowait");

Connection connection = DriverManager.getConnection(
    "jdbc:firebirdsql://localhost:3050/c:/example.fdb",
    props);
```

The overridden mapping is used for all transactions started within the database connection. If the default mapping is overridden via the data source configuration, it will be used for all connections created by the data source.

## 6.5. Savepoints

Savepoints provide finer-grained control over transactions by providing intermediate steps within a larger transaction. Once a savepoint has been set, transaction can be rollback to that point without affecting preceding work.

In order to set a savepoint, use the following code:

*Example of using savepoints*

```
Connection connection = ...;
connection.setAutoCommit(false);

Statement stmt = connection.createStatement();

stmt.executeUpdate(
    "INSERT INTO myTable(id, name) VALUES (1, 'John')");

Savepoint savePoint1 =
    connection.setSavepoint("savepoint_1");

stmt.executeUpdate(
    "UPDATE myTable SET name = 'Ann' WHERE id = 1");
...

connection.rollback(savePoint1);

// at this point changes done by second update are undone
```

Note, rolling back to the savepoint automatically releases and invalidates any savepoints that were

created after the released savepoint.

If the savepoint is no longer needed, you can use the `Connection.releaseSavepoint` method to release system resources. After releasing a savepoint it is no longer possible to rollback the current transaction to that savepoint. Attempts to call the `rollback(Savepoint)` method will result in an `SQLException`. Savepoints that have been created within a transaction are automatically released when that transaction is committed or rolled back.

## 6.6. Transaction Parameter Buffer

The behavior of Firebird transactions is internally controlled by the Transaction Parameter Buffer (TPB), which specifies different transaction properties:

- the transaction isolation level;
- the transaction's read-only or read-write mode;
- the lock conflict resolution mode – wait or no wait;
- the lock wait timeout;
- and, finally, the table reservations – their names and reservation modes.

The TPB is automatically generated depending on the transaction isolation level specified for the `java.sql.Connection` object. Usually there is no need to manipulate the TPB directly. Additionally, if the connection is set to read-only mode, this is reflected in the TPB by appropriate constant. However, the lock resolution mode as well as table reservations cannot be specified by using the standard JDBC interfaces. For the cases where this is needed, Jaybird provides an extension of the JDBC standard.

*Example of specifying custom TPB*

```
FirebirdConnection fbConnection =
    (FirebirdConnection)connection;

TransactionParameterBuffer tpb =
    fbConnection.createTransactionParameterBuffer();

tpb.addArgument(TransactionParameterBuffer.READ_COMMITTED);
tpb.addArgument(TransactionParameterBuffer.REC_VERSION);
tpb.addArgument(TransactionParameterBuffer.WRITE);
tpb.addArgument(TransactionParameterBuffer.WAIT);
tpb.addArgument(TransactionParameterBuffer.LOCK_TIMEOUT, 15);

fbConnection.setTransactionParameters(tpb);
```

The above presents an example of populating the TPB with custom parameters.

### 6.6.1. Isolation level

Firebird supports three isolation levels: `read_committed`, `concurrency` and `consistency` which are

represented by appropriate constants in the `TransactionParameterBuffer` class. The isolation level specifies the way the database engine processes the record versions on read operations. The `concurrency` isolation level is also often called SNAPSHOT and the `consistency` - SNAPSHOT TABLE STABILITY isolation levels.

In `consistency` and `concurrency` modes Firebird database engine loads the different versions of the same record from disk and checks the "timestamps" of each version and compares it with the "timestamp" of the current transaction. The record version with the highest timestamp that is however lower or equal to the timestamp of the current transaction is returned to the application. This effectively returns the version of the record that was when the current transaction started and guarantees that neither non-repeatable reads nor phantom reads can ever occur.

In `read_committed` mode, the Firebird database engine accesses the record version with the highest timestamp for which the corresponding transaction is marked as committed. This prevents engine from reading the record versions which were modified in concurrent transactions that are not yet committed or were rolled back for whatever reasons. However, such mode allows non-repeatable reads as well as phantom reads if the concurrent transaction that modified records or inserted new ones had been committed.

The `read_committed` isolation mode requires another constant that specifies the behavior of the transaction when it sees a record version with a timestamp that belongs to a currently running transaction which is not yet committed.

Most applications require the `TransactionParameterBuffer.REC_VERSION` mode, which is shown in the code above. In this mode database engine fetches the latest committed version as described before.

The `TransactionParameterBuffer.NO_REC_VERSION` constant tells database engine to report a lock conflict when an uncommitted record version is seen while fetching data from the database. The outcome of the operation is then controlled by the lock resolution mode (see section Lock resolution mode).

### 6.6.2. Read-only transactions

The read-only or read-write transaction mode is controlled by two constants:

- `TranscationParameterBuffer.READ` and
- `TransactionParameterBuffer.WRITE`

When the read-write mode (constant `WRITE`) is specified, the database engine stores the "timestamp" of the new transaction in the database even when no modification will be made in the transaction. The "timestamp" affects the garbage collection process, since the database engine cannot release records that were modified in transactions with higher "timestamps" even when these record versions are no longer needed (in other words, when there are already newer versions of the records). Thus, long-running read-write transaction inhibits the garbage collection even when no modifications are done in it.

Therefore, it is recommended to set the read-only mode for the transaction when it is used for read operations.

### 6.6.3. Lock resolution mode

Relational database systems that use pessimistic locking for concurrency control lock the records regardless of the operation type, read or write. When an application tries to read a record from the database, the database engine tries to obtain a "read lock" to that record. If the operation succeeds and the application later tries to update the record, the lock is upgraded to a "write lock". And finally, if the resource is already locked for write, concurrent transactions cannot lock it for reading, since the system cannot allow the transaction to make a decision based on data that might be rolled back later. This approach significantly decreases concurrency. However, databases systems that employ a record versioning mechanism do not have such restrictions because each transaction "sees" its own version of the record – the only possible conflict happens when two concurrent transactions try to obtain "write lock" for the same database record.

Firebird belongs to the latter, and on `read_committed` and `concurrency` isolation levels it behaves appropriately – there are no lock conflicts between readers and writers, and only writers competing for the same resource raise a lock conflict. However, on the `consistency` isolation level Firebird emulates the behavior of systems with pessimistic locking – read operations will conflict with write operations. Even more, the locks are obtained for whole tables (see "Table Reservation" for details).

The following table summarizes the above for Firebird 2.0. It shows that read-committed or repeatable read transactions conflict only when they simultaneously update the same rows. In contrast, a `consistency` transaction conflicts with any transaction running in read-write mode, e.g. as soon as a `consistency` transaction gets write access to a table, other read-write transactions are not allowed to make changes in that tables.

*Table 2. Lock conflicts within one table depending on the isolation level*

| | Read-commited, Concurrency read-write | Read-commited, Concurrency read-only | Consistency, read-write | Consistency, read-only |
|---|---|---|---|---|
| Read-committed, Concurrency read-write | some updates may conflict | | conflict | conflict |
| Read-committed, Concurrency read-only | | | | |
| Consistency read-write | conflict | | conflict | conflict |
| Consistency read-only | conflict | | conflict | |

## 6.7. Table Reservation

Table reservation allows you to specify the database tables and the corresponding access modes at the beginning of the transaction. When the transaction is started, the engine tries to obtain the requested locks for the specified tables and proceeds only when all of them were successfully obtained. Such behavior allows to create a deadlock-free execution history[9].

The table reservation is specified via a TPB and includes the table to lock, the lock mode (read or write) and lock type (shared, protected and exclusive).

*Example of using table reservation facility in Firebird*

```
Connection connection = ...
TransactionParameterBuffer tpb =
    connection.createTransactionParameterBuffer(); ①

tpb.addArgument(TransactionParameterBuffer.CONSISTENCY); ②
tpb.addArgument(TransactionParameterBuffer.WRITE);
tpb.addArgument(TransactionParameterBuffer.NOWAIT);

tpb.addArgument(TransactionParameterBuffer.LOCK_WRITE,
    "TEST_LOCK");
tpb.addArgument(TransactionParameterBuffer.PROTECTED);

connection.setTransactionParameters(tpb); ③

// next transaction will lock TEST_LOCK table for writing
// in protected mode
```

This shows an example of reserving the TEST_LOCK table for writing in a protected mode. The code does the following:

① Create a new instance of `TransactionParameterBuffer` class.

② Populate the TPB. The first three statements were described in "Transaction Parameter Buffer". The fourth call specifies that the application wants to obtain a lock on the table `TEST_LOCK` for writing. The fifth call specifies the type of the lock to obtain, in our case the protected lock.

③ Set the new TPB to be used for the next transaction.

The lock mode to the table specified in the TPB can be either

- `TransactionParameterBuffer.LOCK_READ` for read-only access to the table;
- or `TransactionParameterBuffer.LOCK_WRITE` for read-write access to the table.

The lock type can be either

- `TransactionParameterBuffer.SHARED` for shared access to the table;
- or, `TransactionParameterBuffer.PROTECTED` for protected access to the table;

The `TransactionParameterBuffer.EXCLUSIVE` mode was introduced in later versions of Firebird, however it behaves like `PROTECTED` mode for all read-write transactions.

The lock conflict table depends on the isolation level of the transactions and has the following properties:

- `LOCK_WRITE` mode always conflicts with another `LOCK_WRITE` mode regardless of the lock type and transaction isolation mode;
- `LOCK_WRITE` always conflicts with another `LOCK_READ` mode if both transactions have `consistency` isolation, but has no conflict with shared-read locks if the other transaction has either

`concurrency` or `read_committed` isolation level;

- `LOCK_READ` mode never conflicts with `LOCK_READ` mode.

[7] Additionally, before the InterBase was open-sourced, this allowed application developers to create multi-threaded application without need to purchase additional user licenses.

[8] Other cases, e.g. closing the statement object or the connection object will still ensure that the result set object is closed. If you need result sets that can be "detached" from the statement object that created them, please check the `javax.sql.RowSet` implementations.

[9] This approach follows the two-phase locking protocol, where all locks are acquired on the beginning of the transaction and are released only when transaction is finished.

# Chapter 7. Working with Services

In addition to normal database connections, Firebird features server-wide connections. These are used to perform various administrative tasks in Firebird, e.g. database backup, maintenance, statistics. The set of API calls to perform such tasks are known under the name "Services API". Additionally, client applications can use the Services API to get some limited information about the server environment and configuration.

The actual execution of the Services API calls can be viewed as a tasks triggered from the client application to be executed on server. The parameters passed in the calls are internally used to construct the string similar to the one that is passed to command-line tools. Later this string is passed into entry routine of the gbak, gfix, gsec or gstat utility. The output of the utility, which in normal case is printed to standard out, is in this case transmitted over the network to the client application.

Jaybird attempts to hide the complexity of the original API by providing a set of interfaces and their implementations to perform the administrative tasks regardless of the usage mode (i.e. remote server and embedded engine, wire protocol and access via native client library).

This chapter describes the available Java API for the administrative tasks. All classes and interfaces described below are defined in the `org.firebirdsql.management` package. Each management class works as a standalone object and does not require an open connection to the server.

## 7.1. ServiceManager

The `ServiceManager` interface and the `FBServiceManager` class are defined as the common superclasses providing setters and getters for common properties as well as some common routines. The following properties can be specified:

| Name | Type | Description |
|---|---|---|
| `host` | `java.lang.String` | Name or the IP address of the host to which we make the Service API request. *Required.* |
| `port` | `int` | Port to which we make the request, 3050 by default. |
| `database` | `java.lang.String` | Path to the database. The meaning of the property depends on the service being invoked and will be described in each of chapters below. |
| `user` | `java.lang.String` | Name of the user on behalf of which which all Service API calls will be executed. *Required.* |
| `password` | `java.lang.String` | Password corresponding to the specified user. *Required.* |
| `logger` | `java.io.OutputStream` | Stream into which the output of the remote service will be written to. *Optional.* |

The last parameter requires some explanation. The calls to all Services API routines are asynchronous. The client application can start the call, but there are no other means to find out whether execution of the service call is finished or not except reading the output of the service call

– EOF in this case means that execution is finished.

The `FBServiceManager` converts the asynchronous calls into synchronous by constantly polling the service output stream. If the `logger` property is specified the received data is copied into the specified `OutputStream`, otherwise it is simply ignored and the EOF-marker is being watched.

This behavior can be changed by overriding the appropriate method in the `FBServiceManager` class and/or subclasses. The only requirement is to detach from the service manager when it is no longer needed.

## 7.2. Backup and restore

The backup and restore routines are defined in the `BackupManager` interface and are implemented in the `FBBackupManager` class. Additionally to the setters and getters described in the previous section the following methods are used to specify the backup and restore paths and properties:

| Name | Type | Description |
|---|---|---|
| `database` | `String` | For backup operation it specifies the path to the database to backup.<br><br>For restore operation it specifies the path to the database into which the backup file will be restored. In case when multi-file database should be created, use `addRestorePath(String, int)` method instead. |
| `backupPath` | `String` | Path to the backup file. For backup operation specifies the path and the file name of the newly created backup file. If multi-file backup files are to be created, use `addBackupPath(String, int)` method instead. For restore operations specifies path to the single backup file. If database should be restored from multi-file backup, please use the `addBackupPath(String)` method instead. |
| `restorePageBufferCount` | `int` | Number of pages that will be cached of this particular database. Should be used only for restore operation. |
| `restorePageSize` | `int` | Size of the database page. Should be used only for restore operation. Valid values depend on the Firebird version, but should be one of the 1024, 2048, 4096 or 8192. |
| `restoreReadOnly` | `boolean` | Set to `true` if the database should be restored in read-only mode. |

| Name | Type | Description |
|---|---|---|
| `restoreReplace` | `boolean` | Set to `true` if restore should replace the existing database with the one from backup.<br>  <br>It is easy to drop an existing database if the backup can't be restored, as the existing database is first deleted and only after that the restore process starts. To avoid such situation it is recommended to restore a database into some dummy file first and then use file system commands to replace the existing database with the newly created one. |
| `verbose` | `boolean` | Be verbose when writing to the log.<br><br>The service called on the server will produce lots of output that will be written to the output stream specified in logger property. |

In addition to the properties, the following methods are used to configure the paths to backup and database files when multi-file backup or restore operations are used.

| Method | Description |
|---|---|
| `addBackupPath(String)` | Add a path to a backup file from a multi-file backup. Should be used for restore operation only. |
| `addBackupPath(String, int)` | Add a path to the multi-file backup. The second parameter specifies the maximum size of the particular file in bytes. Should be used for backup operation only. |
| `addRestorePath(String, int)` | Add a path for the multi-file database. The second parameter specifies the maximum size of the database file in pages (in other words, the maximum size in bytes can be obtained by multiplying this value by `restorePageSize` parameter) |
| `clearBackupPaths()` | Clear all the specified backup paths. This method also clears the path specified in `backupPath` property. |
| `clearRestorePaths()` | Clear all the specified restore paths. This method also clears the path specified in the database property. |

All paths specified are paths specifications on the remote server. This has the following implications:

a. it is not possible to backup to the local or network drive unless it is mounted on the remote server;

b. it is not possible to restore from the local or network drive unless it is mounted on the remote server.

After specifying all the needed properties, the application developer can use `backupDatabase()`, `backupMetadata()` and `restoreDatabase()` methods to perform the backup and restore tasks. These methods will block until the operation is finished. If the `logger` property was set, the output of the service will be written into the specified output stream, otherwise it will be ignored.[10]

*Example of backup and restore process*

```
// backup the database
BackupManager backupManager = new FBBackupManager();

backupManager.setHost("localhost");
backupManager.setPort(3050);
backupManager.setUser("SYSDBA");
backupManager.setPassword("masterkey");
backupManager.setLogger(System.out);
backupManager.setVerbose(true);

backupManager.setDatabase("C:/database/employee.fdb");
backupManager.setBackupPath("C:/database/employee.fbk");

backupManager.backupDatabase();
...
// and restore it back
BackupManager restoreManager = new FBBackupManager();

restoreManager.setHost("localhost");
restoreManager.setPort(3050);
restoreManager.setUser("SYSDBA");
restoreManager.setPassword("masterkey");
restoreManager.setLogger(System.out);
restoreManager.setVerbose(true);

restoreManager.setRestoreReplace(true); // attention!!!

restoreManager.setDatabase("C:/database/employee.fdb");
restoreManager.setBackupPath("C:/database/employee.fbk");

backupManager.restoreDatabase();
```

The methods `backupDatabase(int)` and `restoreDatabase(int)` provide a possibility to specify additional backup and restore options that cannot be specified via the properties of this class. The

parameter value is bitwise combination of the following constants:

| Constant | Description |
|---|---|
| BACKUP_CONVERT | Backup external files as tables.<br><br>By default external tables are not backed up, only references to the external files with data are stored in the backup file. When this option is used, the backup will store the external table as if they were regular tables. After restore the tables will remain regular tables. |
| BACKUP_EXPAND | No data compression.<br><br>The gbak utility uses RLE compression for the strings in backup file. Using this option tells it to write strings in their full length, possibly fully consisting of empty characters, etc. |
| BACKUP_IGNORE_CHECKSUMS | Ignore checksums.<br><br>The backup utility can’t backup a database with page checksum errors. Such database is considered corrupted and the completeness and correctness of the backup cannot be guaranteed. However in some cases such errors can be ignored, e.g. when the index page is corrupted. In such case the data in the database are ok and the error disappears when the database is restored and index is recreated.<br><br>Use this option only when checksum errors are detected and can’t be corrected without full backup/restore cycle. Ensure that the restored database contains correct data afterwards. |
| BACKUP_IGNORE_LIMBO | Ignore in-limbo transactions.<br><br>The backup utility can’t backup database with in-limbo transactions. When such situation appears, backup has to wait until the decision about the outcome of the in-limbo transaction. After a wait timeout, an exception is thrown and backup is aborted. This option allows to workaround this situation – the gbak looks for the most recent committed version of the record and writes it into the backup. |

| Constant | Description |
| --- | --- |
| BACKUP_METADATA_ONLY | Backup metadata only.<br><br>When this option is specified, the backup utility creates a backup of only the metadata information (e.g. table an/or view structure, stored procedures, etc.), but no data are backed up. This allows restoring a clean database from the backup. |
| BACKUP_NO_GARBAGE_COLLECT | Do not collect garbage during backup.<br><br>The backup process reads all records in the tables one by one. When cooperative garbage collection is enabled [11] the transaction that accesses the latest version of the record is also responsible for marking the previous versions as garbage. This process is time consuming and might be switched off when creating backup, where the most recent version will be read.<br><br>Later, operator can restore the database from the backup. In databases with many backversions of the records, the backup-restore cycle can be faster than traditional garbage collection. |
| BACKUP_NON_TRANSPORTABLE | Use non-transportable backup format.<br><br>By default gbak creates so-called transportable backup where it does not make difference whether it is later restored on the platform with big or little endianess. By using this option a non-transportable format will be used which allows restoring the database only on the same architecture. |
| BACKUP_OLD_DESCRIPTIONS | Save old style metadata descriptions.<br><br>Actually no real information exist for this option, by default it is switched off. |
| RESTORE_DEACTIVATE_INDEX | Deactivate indexes during restore.<br><br>By default indexes are created at the beginning of the restore process and they are updated with each record being restored from the backup file. On a big tables it is more efficient first to store data in the database and to update the index afterwards. When this option is specified, the indexes will be restored in the inactive state. The downside of this option is that the database administrator is required to activate indexes afterwards, it won’t happen automatically. |

| Constant | Description |
|---|---|
| RESTORE_NO_SHADOW | Do not restore shadow database.<br><br>If the shadow database is configured, an absolute path to the shadow is stored in the backup file. If such backup file is restored on a different system where the path does not exist (e.g. moving a database from Windows to Linux or otherwise), the restore will fail. Using this option allows to overcome such situations. |
| RESTORE_NO_VALIDITY | Do not restore validity constraints.<br><br>This option is usually needed when the validity constraints (e.g. NOT NULL constraints) were added after the data were already in the database but the database contains records that do not satisfy such constraints[12].<br><br>When this option is specified, the validity constraints won't be restored. This allows to recover the data and perform cleanup tasks. The application and/or database administrators are responsible for restoring the validity constrains afterwards. |
| RESTORE_ONE_AT_A_TIME | Commit after completing restore of each table.<br><br>By default all data is restored in one transaction. If for some reason a complete restore is not possible, using this option will allow to restore at least some of the data. |
| RESTORE_USE_ALL_SPACE | Do not reserve 20% on each page for the future versions, useful for read-only databases. |

Example of using these options:

*Example of using extended options for restore*

```
BackupManager restoreManager = new FBBackupManager();

restoreManager.setHost("localhost");
restoreManager.setPort(3050);
restoreManager.setUser("SYSDBA");
restoreManager.setPassword("masterkey");
restoreManager.setLogger(System.out);
restoreManager.setVerbose(true);

restoreManager.setRestoreReplace(true); // attention!!!

restoreManager.setDatabase("C:/database/employee.fdb");
restoreManager.setBackupPath("C:/database/employee.fbk");

// restore database with no indexes,
// validity constraints and shadow database
backupManager.restoreDatabase(
    BackupManager.RESTORE_DEACTIVATE_INDEX |
    BackupManager.RESTORE_NO_VALIDITY |
    BackupManager.RESTORE_NO_SHADOW |
    BackupManager.RESTORE_ONE_AT_A_TIME);
```

## 7.3. User management

Starting with Firebird 3, user management through the Services API has been deprecated. You should use the SQL DDL statements for user management instead.

The next service available is the user management. The routines are defined in the `UserManager` interface and are implemented in the `FBUserManager` class. Additionally, there is an `User` interface providing getters and setters for properties of a user account on the server and corresponding implementation in the `FBUser` class.[13] The available properties of the `FBUser` class are:

| Name | Type | Description |
|---|---|---|
| `userName` | `String` | Unique name of the user on the Firebird server. Required. Maximum length is 31 byte. |
| `password` | `String` | Corresponding password. Getter return value only if the password had been set |
| `firstName` | `String` | First name of the user. Optional. |
| `middleName` | `String` | Middle name of the user. Optional. |
| `lastName` | `String` | Last name of the user. Optional. |
| `userId` | `int` | ID of the user on Unix. Optional. |
| `groupId` | `int` | ID of the group on Unix. Optional. |

The management class, `FBUserManager` has following methods to manipulate the user accounts on the server:

| Method | Description |
|---|---|
| `getUsers():Map` | Method delivers a map containing user names as keys and instances of `FBUser` class as values containing all users that are registered on the server. The instances of `FBUser` class do not contain passwords, the corresponding property is `null`. |
| `addUser(User)` | Register the user account on the server. |
| `updateUser(User)` | Update the user account on the server. |
| `deleteUser(User)` | Delete the user account on the server. |

An example of using the `FBUserManager` class:

*Example of FBUserManager class usage*

```
UserManager userManager = new FBUserManager();

userManager.setHost("localhost");
userManager.setPort(3050);
userManager.setUser("SYSDBA");
userManager.setPassword("masterkey");

User user = new FBUser();
user.setUserName("TESTUSER123");
user.setPassword("test123");
user.setFirstName("John");
user.setMiddleName("W.");
user.setLastName("Doe");

userManager.add(user);
```

# 7.4. Database maintenance

Database maintenance is something that everybody would prefer to avoid, and, contrary to the backup/restore and user management procedures, there is little automation that can be done here. Usually the maintenance tasks are performed on the server by the database administrator, but some routines are needed to perform the automated database upgrade or perform periodic checks of the database validity.

This chapter describes the methods declared in the `MaintenanceManager` interface and its implementation, the `FBMaintenanceManager` class.

## 7.4.1. Database shutdown and restart

One of the most often used maintenance operations is database shutdown and/or bringing it back

online. When the database was shutdown only the user that initiated the shutdown, either SYSDBA or the database owner, can connect to the database and perform other tasks, e.g. metadata modification or database validation and repair.

The database shutdown is performed by `shutdownDatabase(int, int)` method. The first parameter is the shutdown mode, the second – maximum allowed time for operation.

There are three shutdown modes:

| Shutdown mode | Description |
|---|---|
| `SHUTDOWN_ATTACH` | The shutdown process is initiated and it is not possible to obtain a new connection to the database, but the currently open connections are fully functional.<br><br>When after the maximum allowed time for operation there are still open connections to the database, the shutdown process is aborted. |
| `SHUTDOWN_TRANSACTIONAL` | The shutdown process is started and it is not possible to start new transactions or open new connections to the database. The transactions that were running at the time of shutdown initiation are fully functional.<br><br>When after the maximum allowed time for operation there are still running transactions, the shutdown process is aborted.<br><br>If no running transactions are found, the currently open connections are allowed to disconnect. |
| `SHUTDOWN_FORCE` | The shutdown process is started and will be completed before or when the maximum allowed time for operation is reached. New connections and transactions are not prohibited during the wait.<br><br>After that any running transaction won't be able to commit. |

After database shutdown, the owner of the database or SYSDBA can connect to it and perform maintenance tasks, e.g. migration to the new data model[14], validation of the database, changing the database file configuration.

To bring the database back online use the `bringDatabaseOnline()` method.

### 7.4.2. Shadow configuration

A database shadow is an in-sync copy of the database that is usually stored on a different hard disk, possibly on a remote computer[15], which can be used as a primary database if the main database server crashes. Shadows can be defined using `CREATE SHADOW` SQL command and are characterized by a *mode* parameter:

- in the AUTO mode database continues operating even if shadow becomes unavailable (disk or file system failure, remote node is not accessible, etc.)

- in the MANUAL mode all database operations are halted until the problem is fixed. Usually it means that DBA has to kill the unavailable shadow and define a new one.

The `MaintenanceManager` provides a `killUnavailableShadows()` method to kill the unavailable shadows. This is equivalent to the `gfix -kill` command.

Additionally, if the main database becomes unavailable, the DBA can decide to switch to the shadow database. In this case the shadow must be activated before use. To activate the shadow use the `activateShadowFile()` method. Please note, that in this case the `database` property of the `MaintenanceManager` must point to the shadow file which must be located on the local file system of the server to which the management class is connected.

### 7.4.3. Database validation and repair

The Firebird server does its best to keep the database file in a consistent form. In particular this is achieved by a special algorithm called *careful writes* which guarantees that the server writes data on disk in such a manner that despite events like a server crash the database file always remains in a consistent state. Unfortunately, it is still possible that under certain conditions, e.g. crash of the file system or hardware failure, the database file might become corrupted. Firebird server can detect such cases including

- Orphan pages. These are the database pages that were allocated for subsequent write, but due to a crash were not used. Such pages have to be marked as unused to return storage space back to the application;
- Corrupted pages. These are the database pages that were caused by the operating system or hardware failures.

The `MaintenanceManager` class provides a `validateDatabase()` method to perform simple health check of the database, and releasing the orphan pages if needed. It also reports presence of the checksum errors. The output of the routine is written to the output stream configured in the `logger` property.

The `validateDatabase(int)` method can be used to customize the validation process:

| Validation mode | Description |
|---|---|
| `VALIDATE_READ_ONLY` | Perform read-only validation. In this case the database file won't be repaired, only the presence of the database file errors will be reported.<br><br>Can be used for periodical health-check of the database. |
| `VALIDATE_FULL` | Do a full check on record and pages structures, releasing unassigned record fragments. |
| `VALIDATE_IGNORE_CHECKSUM` | Ignore checksums during repair operations.<br><br>The checksum error means that the database page was overwritten in a random order and the data stored on it are corrupted. When this option is specified, the validation process will succeed even if checksum errors are present. |

In order to repair the corrupted database use the `markCorruptRecords()` method which marks the corrupted records as unavailable. This method is equivalent to `gfix -mend` command. After this operation database can be backed up and restored to a different place.

The presence of the checksum errors and subsequent use of `markCorruptedRecords()` method will mark all corrupted data as unused space. You have to perform a careful check after backup/restore cycle to assess the damage.

### 7.4.4. Limbo transactions

Limbo transactions are transactions that were prepared for commit but were never committed. This can happen when, for example, the database was accessed by JTA-enabled applications from Java[16]. The in-limbo transactions affect the normal database operation, since the records that were modified in that transactions are not available – Firebird does not know whether the new version will be committed or rolled back and blocks access to them. Also in-limbo transactions prevents garbage collection, since the garbage collector does not know whether it can discard the record versions of the in-limbo transaction.

Jaybird contains functionality to allow the JTA-enabled transaction coordinator to recover the in-limbo transactions and either commit them or perform a rollback. For the cases when this is not possible `MaintenanceManager` provides the following methods to perform this in interactive mode:

| Method | Description |
|---|---|
| `listLimboTransactions()` | Method lists IDs of all in-limbo transactions to the output stream specified in logger property.<br><br>The application has to either parse the output to commit or rollback the transactions in some automated fashion or it should present the output to the user and let him/her make a decision. Alternatively, use one of the following two methods |
| `limboTransactionsAsList()` | Returns a `List<Long>` of the IDs of all in-limbo transactions |
| `getLimboTransactions()` | Returns an array of `long` with the IDs of all in-limbo transactions |
| `commitTransaction(long)` | Commit the transaction with the specified ID. |
| `rollbackTransaction(long)` | Rollback the transaction with the specified ID. |

### 7.4.5. Sweeping the database

The in-limbo transactions are not the only kind of transactions that prevent garbage collection. Another type are transactions are those that were finished by "rollback" and the changes made in such transactions were not automatically undone by the internal savepoint mechanism, e.g. when there were a lot of changes made in the transaction (e.g. 10,000 records and more). Such transactions are marked as "rollback" transactions on Transaction Inventory Page and this prevents advancing the so-called Oldest Interesting Transaction (OIT) – ID of the oldest transaction which created record versions that are relevant to any of the currently running transactions. On each access to the records, Firebird has to check all the record versions between the current transaction and the OIT, which leads to performance degradation on large databases. In order to solve the issue

Firebird periodically starts a database sweeping process, that traverses all database records, removes the changes made by the rolled back transactions and moves forward the OIT.[17]

The sweep process is controlled by a threshold parameter – a difference between the Next Transaction and OIT, by default it equal to 20,000. While this value is ok for the average database, a DBA can decide to increase or decrease the number to fit the database usage scenario. Alternatively, a DBA can trigger the sweep process manually regardless of the current difference between Next Transaction and OIT.

The `MaintenanceManager` provides following methods to help with database sweeping:

| Method | Description |
|---|---|
| `setSweepThreshold(int)` | Set the threshold between Next Transaction and OIT that will trigger the automatic sweep process. Default value is 20,000. |
| `sweepDatabase()` | Perform the sweep regardless of the current difference between Next Transaction and OIT. |

### 7.4.6. Other database properties

There are a few other properties of the database that can be set via `MaintenanceManager`:

| Method | Description |
|---|---|
| `setDatabaseAccessMode(int)` | Change the access mode of the database. Possible values are:<br>• `ACCESS_MODE_READ_ONLY` to make database read-only;<br>• `ACCESS_MODE_READ_WRITE` to allow writes into the database.<br>Please note, only read-only databases can be placed on read-only media, the read-write databases will need to be able to write even if only accessed with read-only transactions. |
| `setDatabaseDialect(int)` | Change the database SQL dialect. The allowed values can be either 1 or 3. |
| `setDefaultCacheBuffer(int)` | Change the number of database pages to cache.<br>This setting applies to this specific database, overriding the system-wide configuration. |
| `setForcedWrites(boolean)` | Change the forced writes setting for the database.<br>When forced writes are switched off, the database engine does not enforce flushing pending changes to disk and they are kept in OS cache. Tthe same page is changed again later, the write happens in memory, which in many cases increases the performance. However, in case of OS or hardware crashes the database will be corrupted. |

| Method | Description |
| --- | --- |
| `setPageFill(int)` | Set the page fill factor.<br><br>Firebird leaves 20% of free space on each database page for future record versions. It is possible to tell Firebird not to reserve the space, this makes sense for read-only databases, since more data fit the page, which increases performance.<br><br>Possible values are:<br><br>• `PAGE_FILL_FULL` – do not reserve additional space for future versions;<br>• `PAGE_FILL_RESERVE` – reserve the free space for future record versions. |

## 7.5. Database statistics

And last but not least is the `StatisticsManager` interface and corresponding implementation in the `FBStatisticsManager` class, which allow to obtain statistical information for the database, like page size, values of OIT and Next transactions, database dialect, database page allocation and its distribution.

The following methods provide the functionality equivalent to the `gstat` command line tool, the output of the commands is written to the output stream specified in the `logger` property. It is the responsibility of the application to correctly parse the text output if needed.

| Method | Description |
| --- | --- |
| `getDatabaseStatistics()` | Get complete statistics about the database. |
| `getDatabaseStatistics(int)` | Get the statistical information for the specified options.<br><br>Possible values are (bit mask, can be combined):<br><br>• `DATA_TABLE_STATISTICS`<br>• `SYSTEM_TABLE_STATISTICS`<br>• `INDEX_STATISTICS`<br>• `RECORD_VERSION_STATISTICS` |
| `getHeaderPage()` | Get information from the header page (e.g. page size, OIT, OAT and Next transaction values, etc.) |
| `getTableStatistics(String[])` | Get statistic information for the specified tables.<br><br>This method allows to limit the reported statistical information to a single or couple of the tables, not for the whole database. |

[10] The output of the service is always transferred over the network regardless whether the logger property is set or not. Additionally to providing a possibility to the user to track the service progress it acts also as a signal of operation completion – in

this case the Java code will receive an EOF marker.

[11] Cooperative garbage collection can be switched off in Firebird 2.0 SuperServer architecture by corresponding configuration option. It can't be switched off in ClassicServer architecture and in previous Firebird versions.

[12] All versions of Firebird upto 2.5 allow to define validity constraints despite the table(s) contain data that do not satisfy them. Only the new records will be validated, and it is responsibility of the database administrator to ensure the validity of existing ones.

[13] The class implementation is a simple bean publishing the properties via getters and setters. You can replace it with any other implementation of the `User` interface.

[14] Until Firebird 2.0 adding a foreign key constraint required exclusive access to the database.

[15] Currently possible only on Unix platforms by using NFS shares.

[16] Another reason for limbo transactions are multidatabase transactions which can be initiated via native Firebird API. However, since Jaybird does not provide methods to initiate them, we do not consider them in this manual.

[17] For more information please read article by Ann Harrison "Firebird for the Database Expert: Episode 4 - OAT, OIT, & Sweep", available, for example, at http://www.ibphoenix.com/resources/documents/design/doc_21

# Chapter 8. Working with Events

Firebird supports events. Events are a feature that provides asynchronous notification to the connected applications about events triggered by the database or other applications. Instead of requiring applications to reread the database tables to check for the changes, events make it possible to avoid that: triggers in the database can post an event in case of a change. And even more, the event can be so specific that an application would need to reread only a limited set of records, possibly only one.

This chapter describes the event mechanism in Firebird and the common usage scenarios.

## 8.1. Database events

An *event* is a message generated in a trigger, stored procedure or execute block that is delivered to subscribed applications. The event is characterized only by a name which is used when the event is posted, therefore two different events must have two different names. The applications that subscribe for events are required to specify the event names of interest, no wildcards are allowed; and applications either provide a callback function that will be invoked in case of event or are required to poll for the posted events periodically.

Events are delivered to the application only on (after) commit of the transaction that generated the event. Firebird does not provide any guarantees about the time of event delivery – it depends on the load of the Firebird engine, application load, network delays between application and the database system. The database engine will continue operating even if no application subscribes to events or when the subscribed application crashed in the meantime.

It can also happen that multiple transactions will be committed before the events are delivered to the client system. But even in such case the callback function will be invoked only once, and only the event name and the count of the events will be passed as parameters. The same applies to periodical polling – the application will receive event names and counts of the events since last polling.

Internally, Firebird can be thought to store the subscription information in a table where columns contain event names, rows correspond to the subscribed applications and the cells contain the count of the particular event for a particular application. When an event is posted in trigger or stored procedure, Firebird checks the subscription information and increases the event count for the subscribed applications. Another thread checks the table periodically and notifies the application about all new events relevant to the particular application. Such mechanism allows Firebird to keep the event notification table very small[18] and to reduce the number of messages sent to the application.

It is not possible to pass parameters with the event, e.g. an ID of the modified records. It is also not possible to encode such information in the event names – wildcards are not supported. For such cases, applications should maintain a change tracking table where the IDs of the modified records are stored and the event mechanism is used to tell the application that new records were added to the table.

## 8.2. Posting events

Events are posted from PSQL procedural code (trigger, stored procedure, execute block, function) using the `POST_EVENT` command. It is possible to create a stored procedure with the sole purpose of posting events:

*Example of posting events from PSQL code*

```
CREATE PROCEDURE sp_post_event(event_name VARCHAR(72))
AS BEGIN
  POST_EVENT :event_name;
END
```

Firebird 2.0 introduced a new command `EXECUTE BLOCK` which allows to execute PSQL statements within DSQL code:

*Using EXECUTE BLOCK to post events*

```
Statement stmt = connection.createStatement();
try {
  stmt.execute(
      "EXECUTE BLOCK AS BEGIN POST_EVENT 'some_evt'; END");
} finally {
  stmt.close();
}
```

## 8.3. Subscribing to events

The design of the classes and interfaces in the `org.firebirdsql.event` package is similar to the Services API support – there is a central manager-class that establishes a database connection and provides service methods to work with the events, a callback interface that applications must implement to use the asynchronous event notification and an interface representing a database event with two properties – event name and occurrence count.

Applications have to configure the following properties before starting use of the implementation `EventManager` interface:

| Name | Type | Description |
|---|---|---|
| `host` | `java.lang.String` | Name or the IP address of the host to which we subscribe for events. *Required.* |
| `port` | `int` | Port to which we connect to, 3050 by default. |
| `database` | `java.lang.String` | Path to the database. The path is specified for the remote host but must be absolute. *Required.* |
| `user` | `java.lang.String` | Name of the user on behalf of which we connect to the database. *Required.* |
| `password` | `java.lang.String` | Password corresponding to the specified user. *Required.* |

After configuring these properties, the application has to invoke the `connect()` method to establish a physical connection to the database. At this point the `EventManager` is ready to receive the event notifications.

Now the application developer has two choices: use asynchronous event notification or use methods that will block until an event is delivered or timeout occurs.

### 8.3.1. Asynchronous event notification

The asynchronous event notification uses a separate daemon thread to wait for the event notifications and to deliver the events to the registered listeners. The listeners are added using the `addEventListener(String, EventListener)` method, where the first parameter contains the name of the event to register on and the second parameter – an instance of `EventListener` interface that will be notified about occurrences of this event. It is allowed to use the same instance of `EventListener` interface to listen on different events. The code below shows an example of using asynchronous event notification.

*Example of registering an event listener for asynchronous event notification*

```
EventManager eventManager = new FBEventManager();

eventManager.setHost("localhost");
eventManager.setUser("SYSDBA");
eventManager.setPassword("masterkey");
eventManager.setDatabase("c:/database/employee.fdb");

eventManager.connect();

eventManager.addEventListener("test_event",
    new EventListener() {
      public void eventOccurred(DatabaseEvent event){
        System.out.println("Event [" +
            event.getEventName() + "] occured " +
            event.getEventCount() + " time(s)");
      }
    }
);
```

### 8.3.2. Using blocking methods

Alternatively, an application can use the synchronous methods, one that blocks until the named event is received – the `waitForEvent(String)` method, or one that will block until the named event is received or timeout specified in the second parameter occurs – the `waitForEvent(String, int)` method. The following shows an example of using the blocking methods.

*Example of blocking waiting for event with a specified timeout*

```
EventManager eventManager = new FBEventManager();

eventManager.setHost("localhost");
eventManager.setUser("SYSDBA");
eventManager.setPassword("masterkey");
eventManager.setDatabase("c:/database/employee.fdb");

eventManager.connect();

int eventCount =
    eventManager.waitForEvent("test_event", 10 * 1000);

System.out.println(
    "Received " + eventCount + " event(s) during 10 sec.");
```

[18] For example, the effective size for 100 applications subscribed for 100 different events is about 40k in memory.

# Reference Manual

# Chapter 9. Connection reference

## 9.1. Authentication plugins

*Jaybird 3*
*Firebird 3*

Firebird 3 introduced authentication plugins together with a new authentication model. By default, Firebird 3 uses the authentication plugin `Srp` (*Secure remote password*). It also includes plugins `Legacy_Auth` that supports the pre-Firebird-3 authentication mechanism, and - *Firebird 3.0.4* - `Srp256`. Firebird 4 introduced the plugins `Srp224`, `Srp384` and `Srp512`.

The original `Srp` plugin uses SHA-1, the new Srp-variants use SHA-224, SHA-256, SHA-384 and SHA-512 respectively.[19]

Support for these plugins depends on support of these hash algorithms in the JVM. For example, SHA-224 is not supported in Oracle Java 7 by default and may require additional JCE libraries.

### 9.1.1. Default authentication plugins

#### Jaybird 3

Jaybird 3 will try - in order - `Srp` and `Legacy_Auth`, or - *Jaybird 3.0.5* - `Srp256`, `Srp` and `Legacy_Auth`. It is not possible to specify a different configuration in Jaybird 3.

Firebird 2.5 and earlier will always use legacy authentication.

#### Jaybird 4

The default plugins applied by Jaybird 4 are - in order - `Srp256` and `Srp`. This applies only for the pure Java protocol and only when connecting to Firebird 3 or higher. The native implementation will use its own default or the value configured through its `firebird.conf`.

When connecting to Firebird 3 or higher, the pure Java protocol in Jaybird 4 will no longer try the `Legacy_Auth` plugin by default as it is an unsafe authentication mechanism. We strongly suggest to use SRP users only, but if you really need to use legacy authentication, you can specify connection property `authPlugins=Legacy_Auth`, see Configure authentication plugins for details.

When connecting to Firebird 3 versions earlier than 3.0.4, or if `Srp256` has been removed from the `AuthServer` setting in Firebird, this might result in slightly slower authentication because more roundtrips to the server are needed. After an attempt to use `Srp256` fails, authentication continues with `Srp`.

To avoid this, consider explicitly configuring the authentication plugins to use, see Configure authentication plugins for details.

Firebird 2.5 and earlier will always use legacy authentication.

### 9.1.2. Configure authentication plugins

*Jaybird 4*

Jaybird 4 introduces the connection property `authPlugins` (alias `auth_plugin_list`) to specify the authentication plugins to try when connecting. The value of this property is a comma-separated list with the plugin names.

The `authPlugins` values can be separated by comma, space, tab, or semi-colon. We recommend using comma as the separator. The semi-colon should not be used in a JDBC URL as there the semi-colon is a separator between connection properties.

Unknown or unsupported plugins will be logged and skipped. When no known plugins are specified, Jaybird will throw an exception with:

- For pure Java

  *Cannot authenticate. No known authentication plugins, requested plugins: [<plugin-names>] [SQLState:28000, ISC error code:337248287]*

- For native

  *Error occurred during login, please check server firebird.log for details [SQLState:08006, ISC error code:335545106]*

The `authPlugins` property only affects connecting to Firebird 3 or later. It will be ignored when connecting to Firebird 2.5 or earlier. The setting will also be ignored for native connections when using a fbclient library of version 2.5 or earlier.

Examples:

- JDBC URL to connect using `Srp256` only:

  ```
  jdbc:firebirdsql://localhost/employee?authPlugins=Srp256
  ```

- JDBC URL to connect using `Legacy_Auth` only (this is unsafe!)

  ```
  jdbc:firebirdsql://localhost/employee?authPlugins=Legacy_Auth
  ```

- JDBC URL to try `Legacy_Auth` before `Srp512` (this order is unsafe!)

  ```
  jdbc:firebirdsql://localhost/employee?authPlugins=Legacy_Auth,Srp512
  ```

The property is also supported by the data sources, service managers and event manager.

### 9.1.3. External authentication plugin support (experimental)

*Jaybird 4*

If you develop your own Firebird authentication plugin (or use a third-party authentication plugin), it is possible - for pure Java only - to add your own authentication plugin by implementing the interfaces

- `org.firebirdsql.gds.ng.wire.auth.AuthenticationPluginSpi`
- `org.firebirdsql.gds.ng.wire.auth.AuthenticationPlugin`

The SPI implementation needs to be listed in `META-INF/services/org.firebirdsql.gds.ng.wire.auth.AuthenticationPluginSpi` in your jar.

This support is experimental and comes with a number of caveats:

- We haven't tested this extensively (except for loading Jaybird's own plugins internally)
- The authentication plugin (and provider) interfaces should be considered unstable; they may change with point-releases (although we will try to avoid that)
- For now it will be necessary for the jar containing the authentication plugin to be loaded by the same class loader as Jaybird itself

If you implement a custom authentication plugin and run into problems, contact us on the Firebird-Java mailing list.

If you use a native connection, check the Firebird documentation how to add third-party authentication plugins to fbclient.

## 9.2. Wire encryption support

*Jaybird 3.0.4*
*Firebird 3*

Firebird 3 and higher have support for encrypting the data sent over the network. This *wire encryption* is configured using the connection property `wireCrypt`, with the following (case-insensitive) values:

`DEFAULT`
: default (value used when `wireCrypt` is not specified; you'd normally not specify `DEFAULT` explicitly)

`ENABLED`
: enable, but not require, wire encryption

`REQUIRED`
: require wire encryption (only if Firebird version is 3.0 or higher)

`DISABLED`
: disable wire encryption

The default value acts as `ENABLED` for pure Java connections, for JNA (native) connections this wil use the fbclient default (either `Enabled` or the configured value of `WireCrypt` from a `firebird.conf` read by the native library).

Connection property `wireCrypt=REQUIRED` will **not** reject unencrypted connections when connecting to Firebird 2.5 or lower. This behavior matches the Firebird 3 client library behavior. The value will also be ignored when using native connections with a Firebird 2.5 client library.

Using `wireCrypt=DISABLED` when Firebird 3 or higher uses setting `WireCrypt = Required` (or vice versa) will yield error *"Incompatible wire encryption levels requested on client and server"* (error: *isc_wirecrypt_incompatible / 335545064*).

The same error is raised when connecting to Firebird 3 and higher with a legacy authentication user with connection property `wireCrypt=REQUIRED`.

Alternative wire encryption plugins are currently not supported, although we made some preparations to support this. If you want to develop such a plugin, contact us on the Firebird-Java mailing list so we can work out the details of adding plugin support.

The implementation comes with a number of caveats:

- we cannot guarantee that the session key cannot be obtained by someone with access to your application or the machine hosting your application (although that in itself would already imply a severe security breach)
- the ARC4 encryption - the default provided by Firebird - is considered to be a weak (maybe even broken) cipher these days
- the encryption cipher uses ARCFOUR with a 160 bits key, this means that the unlimited Cryptographic Jurisdiction Policy needs to be used (or at minimum a custom policy that allows ARCFOUR with 160 bits keys). See also FAQ entry Encryption key did not meet algorithm requirements of Symmetric/Arc4 (337248282) [https://www.firebirdsql.org/file/documentation/drivers_documentation/java/faq.html#encryption-key-did-not-meet-algorithm-requirements-of-symmetricarc4-337248282]

# 9.3. Database encryption support

*Jaybird 3.0.4*
*Firebird 3*

Jaybird 3.0.4 added support for Firebird 3 database encryption callbacks in the pure Java implementation of the version 13 protocol.

The current implementation is simple and only supports replying with a static value from a connection property. Be aware that a static value response for database encryption is not very secure as it can easily lead to replay attacks or unintended key exposure.

Future versions of Jaybird (likely 4, maybe 5) will introduce plugin support for database encryption plugins that require a more complex callback.

The static response value of the encryption callback can be set through the `dbCryptConfig` connection property. Data sources and `ServiceManager` implementations have an equivalent property with the same name. This property can be set as follows:

- Absent or empty value: empty response to callback (depending on the database encryption plugin this may just work or yield an error later)
- Strings prefixed with `base64:`: rest of the string is decoded as base64 to bytes. The `=` padding characters are optional, but when present they must be valid (that is: if you use padding, you must use the right number of padding characters for the length)
- Plain string value: string is encoded to bytes using UTF-8, and these bytes are used as the response

Because of the limitation of connection URL parsing, we strongly suggest to avoid plain string values with `&` or `;`. Likewise, avoid `:` so that we can support other prefixes similar to `base64:` in the future. If you need these characters, consider using a base64 encoded value instead.

For service operations, as implemented in the `org.firebirdsql.management` package, Firebird requires the `KeyHolderPlugin` configuration to be globally defined in `firebird.conf`. Database-specific configuration in `databases.conf` will be ignored for service operations. Be aware that some service operations on encrypted databases are not supported by Firebird 3 (eg `gstat` equivalents other than `gstat -h` or `gstat -e`).

Other warnings and limitations

- Database encryption callback support is only available in the pure Java implementation. Support for native and embedded connections will be added in a future version.
- The database encryption callback does not require an encrypted connection, so the key can be exchanged unencrypted if wire protocol encryption has been disabled client-side or server-side, or if legacy authentication is used. Consider setting connection property `wireCrypt=REQUIRED` to force encryption (caveat: see the next point).
- Firebird may ask for the database encryption key before the connection has been encrypted (for example if the encrypted database itself is used as the security database). *This applies to v15 protocol support, which is not yet available.*
- The improvements of the versions 14 and 15 wire protocol are not implemented, and as a result encrypted security databases (external or security database hosted in the database itself) will not work unless the encryption plugin does not require a callback. Support for the version 15 wire protocol will be added in a future version.
- We cannot guarantee that the `dbCryptConfig` value cannot be obtained by someone with access to your application or the machine hosting your application (although that in itself would already imply a severe security breach).

## 9.4. Default holdable result sets

This connection property enables a connection to create holdable result sets by default. This propery can be used as a workaround for applications that expect a result to remain open after commit, or have expectations regarding result sets in auto-commit mode that do not conform to the JDBC specification.

Specifically, such applications open a result set and, while traversing it, execute other statements using the same connection. According to the JDBC specification the result set has to be closed if another statement is executed using the same connection in auto-commit mode. With the default result set holdability, close on commit, doing this yields a `SQLException` with message *"The result set is closed"*.

The property is called:

- `defaultResultSetHoldable` as connection property with no value, empty value or `true` (aliases: `defaultHoldable` and `result_set_holdable`);
- `isc_dpb_result_set_holdable` as a DPB member;
- `FirebirdConnectionProperties` interface methods `isDefaultResultSetHoldable()` and `setDefaultResultSetHoldable(boolean)`

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

## 9.5. Firebird auto commit mode (experimental)

*Jaybird 2.2.9*

This option is enabled by specifying the connection property `useFirebirdAutocommit=true`.

With this option, Jaybird will configure the transaction to use `isc_tpb_autocommit` with `autoCommit=true`. This means that Firebird server will internally commit the transaction after each statement completion. Jaybird itself will not commit until connection close (or switching to `autoCommit=false`). The exception is if the statement was of type `isc_info_sql_stmt_ddl`, in that case Jaybird will commit on statement success and rollback on statement failure (just like it does for all statements in normal auto commit mode). The reason is that Firebird for some DDL commands only executes at a real commit boundary and relying on the Firebird auto-commit is insufficient.

On statement completion (as specified in JDBC), result sets will still close unless they are holdable over commit. The result set is only closed client side, which means that the cursor remains open server side to prevent roundtrips. This may lead to additional resource usage server side unless explicitly closed in the code. Note that any open blobs will be closed client- and server-side (until this is improved with JDBC-401 [http://tracker.firebirdsql.org/browse/JDBC-401]).

A connection can be interrogated using `FirebirdConnection.isUseFirebirdAutocommit()` if it uses `isc_tpb_autocommit`.

If you manually add `isc_tpb_autocommit` to the transaction parameter buffer and you enable this option, the `isc_tpb_autocommit` will be removed from the TPB if `autoCommit=false`.

Artificial testing with repeated inserts (using a prepared statement) against a Firebird server on localhost shows that this leads to a reduction of execution time of +/- 7%.

Support for this option is experimental, and should only be enabled if you 1) know what you're doing, and 2) really need this feature. Internally `isc_tpb_autocommit` uses `commit_retaining`, which means that using this feature may increase the transaction gap with associated sweep and garbage collection impact.

# 9.6. Process information

*Jaybird 2.1.6*
*Firebird 2.1*

Firebird 2.1 introduced the `MON$ATTACHMENTS` table. This table includes the columns `MON$REMOTE_PID` and `MON$REMOTE_PROCESS` which report the process id and process name of the connected process.

By default, Jaybird does not provide this information. This has two main reasons: until recently Java did not have a portable way of retrieving the process id, and in most cases the process name is just 'java' (or similar), which is not very useful.

Since Firebird 3, the `MON$ATTACHMENTS` table also includes the column `MON$CLIENT_VERSION`. Jaybird (*Jaybird 3.0*) will report its full version (eg `Jaybird 3.0.5-JDK_1.8`).

Do not use this information for security decisions. Treat it as informational only, as clients can report fake information.

It is possible to specify the process name and process id in two ways:

## 9.6.1. System properties for process information

It is possible to specify the process information through Java system properties:

`org.firebirdsql.jdbc.pid`
: Process id

`org.firebirdsql.jdbc.processName`
: Process name

This is the preferred method because you only need to specify it once.

## 9.6.2. Connection properties for process information

It is also possible to specify the process information through connection properties:

`process_id`
: Process id (alias: `processId` (*Jaybird 3*))

`process_name`

Process name (alias: `processName` (*Jaybird 3*))

These properties are not exposed on the data sources. To set on data sources, use `setNonStandardProperty`.

[19] Internally `SrpNNN` continues to uses SHA-1, only the client-proof applies the SHA-NNN hash. See also CORE-5788 [http://tracker.firebirdsql.org/browse/CORE-5788]).

# Chapter 10. Statement reference

## 10.1. Generated keys retrieval

*Jaybird 2.2*
*Firebird 2.0*

Jaybird 2.2 added support for the `getGeneratedKeys()` JDBC feature for `Statement` and `PreparedStatement`. This feature can be used to retrieve the generated ids (and other columns) from DML statements.

This feature is available for `Connection.prepareStatement`, and `Statement.execute`, `Statement.executeUpdate` and `Statement.executeLargeUpdate` (*Java 8/JDBC 4.2*)

There are four distinct use-cases:

1. Methods accepting an `int` parameter with values of `Statement.NO_GENERATED_KEYS` and `Statement.RETURN_GENERATED_KEYS`, see Basic generated keys retrieval,
2. Methods accepting an `int[]` parameter with column indexes, see Generated keys by column index.
3. Methods accepting a `String[]` parameter with column names, see Generated keys by column name.
4. Providing a query already containing a `RETURNING` clause to any of these methods.

   In this case all of the previous cases are ignored and the query is executed as is. It is possible to retrieve the result set using `getGeneratedKeys()`.

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

The required ANTLR runtime version depends on the Jaybird version, check the release notes of your version for details.

This functionality is available for `INSERT` (*Firebird 2.0*), for `UPDATE`, `UPDATE OR INSERT` and `DELETE` (*Firebird 2.1*), and for `MERGE` (*Firebird 3.0*).

Generated keys retrieval modifies the statement to add a `RETURNING`-clause. Currently Firebird only supports `RETURNING` for DML operations that affect a single row. Attempting to use generated keys retrieval with a statement that affects multiple rows will yield the error *"multiple rows in singleton select"*.

The examples in this section use the following (Firebird 3) table:

*Example person table*

```
create table PERSON (
  ID integer generated by default as identity constraint pk_employee primary key,
  FIRSTNAME varchar(20),
  LASTNAME varchar(20),
  BIRTHDATE date,
  "age" integer generated always as (datediff(year, birthdate, current_date))
)
```

### 10.1.1. Basic generated keys retrieval

This form of generated keys execution involves the following methods:

- `Connection.prepareStatement(String sql, int autoGeneratedKeys)`
- `Statement.execute(String sql, int autoGeneratedKeys)`
- `Statement.executeUpdate(String sql, int autoGeneratedKeys)`
- `Statement.executeLargeUpdate(String sql, int autoGeneratedKeys)` (*JDBC 4.2*)

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

When `RETURN_GENERATED_KEYS` is passed, the driver will return *all* columns of the table as generated keys. The columns are ordered by ordinal position (as reported in the JDBC metadata of the table). It is advisable to retrieve the values from the `getGeneratedKeys()` result set by column name.

We opted to include all columns as it is next to impossible to decide which columns are populated by a trigger or otherwise. Only returning the primary key will be too restrictive (consider computed columns, default values, etc).

Passing `NO_GENERATED_KEYS` hardcoded should normally not be done. It would be better to use the equivalent `prepareStatement` or `executeXXX` method that only accepts a `String`. Use of the value `NO_GENERATED_KEYS` only makes sense in code that dynamically decides between `NO_GENERATED_KEYS` and `RETURN_GENERATED_KEYS`.

#### Example using RETURN_GENERATED_KEYS

The following will insert a person using a `Statement` and retrieve the generated id using `Statement.RETURN_GENERATED_KEYS`:

*Statement generated keys retrieval*

```
Connection connection = ...;
try (Statement statement = connection.createStatement()) {
  statement.executeUpdate(
      "insert into person(firstname, lastname, birthdate) "
          + "values ('Mark', 'Rotteveel', date'1979-01-12')",
      Statement.RETURN_GENERATED_KEYS);                  ①

  try (ResultSet keys = statement.getGeneratedKeys()) { ②
    if (keys.next()) {                                  ③
      int generatedId = keys.getInt("id");              ④
      int age = keys.getInt("age");                     ⑤
      String firstName = keys.getString("firstname");   ⑥

      System.out.printf("Inserted: %s, Id: %d, Age: %d%n",
          firstName, generatedId, age);
    }
  }
}
```

① Use of `Statement.RETURN_GENERATED_KEYS` instructs Jaybird to parse the statement and add a `RETURNING` clause with all columns of the `PERSON` table

② Get the generated keys result set from the statement

③ Just like a normal result set, it is positioned before the first row, so you need to call `next()`

④ The generated value of the `ID` column

⑤ The calculated value of the `AGE` column

⑥ The generated keys result set also contains the normal columns like `FIRSTNAME`

The equivalent using `PreparedStatement` is:

*Prepared statement generated keys retrieval*

```
try (PreparedStatement statement = connection.prepareStatement(
        "insert into person(firstname, lastname, birthdate) values (?, ?, ?)",
        Statement.RETURN_GENERATED_KEYS)) {            ①
  statement.setString(1, "Mark");
  statement.setString(2, "Rotteveel");
  statement.setObject(3, LocalDate.of(1979, 1, 12));

  statement.executeUpdate();
  try (ResultSet keys = statement.getGeneratedKeys()) { ②
    if (keys.next()) {                                  ③
      int generatedId = keys.getInt("id");              ④
      int age = keys.getInt("age");                     ⑤
      String firstName = keys.getString("firstname");

      System.out.printf("Inserted: %s, Id: %d, Age: %d%n",
          firstName, generatedId, age);
    }
  }
}
```

① Besides use of parameters, the only real difference is that use of `Statement.RETURN_GENERATED_KEYS` moved from execution to prepare. This makes sense if you consider that once prepared, the statement can be reused.

### 10.1.2. Generated keys by column index

This form of generated keys execution involves the following methods:

- `Connection.prepareStatement(String sql, int[] columnIndexes)`
- `Statement.execute(String sql, int[] columnIndexes)`
- `Statement.executeUpdate(String sql, int[] columnIndexes)`
- `Statement.executeLargeUpdate(String sql, int[] columnIndexes)` (*JDBC 4.2*)

The values in the `int[]` parameter are the ordinal positions of the columns as specified in the (JDBC) metadata of the table.

In Jaybird 3 and earlier, a null or empty array was silently ignored and the statement was executed normally (not producing generated keys). In Jaybird 4, this behaviour has changed and instead will throw an exception with message *"Generated keys array columnIndexes was empty or null. A non-empty array is required."*

In Jaybird 3 and earlier, invalid ordinal positions are ignored and silently dropped: passing `new int[] { 1, 5, 6 }` will work, even though we don't have sixth column. In Jaybird 4, this behavior has changed and instead will throw an exception with message *"Generated keys column position <position> does not exist for table <tablename>. Check DatabaseMetaData.getColumns (column ORDINAL_POSITION) for valid values."*

### Example using column indexes

Retrieval by column index uses the ordinal position as reported in `DatabaseMetaData.getColumns`, column `ORDINAL_POSITION`. In practice this is the value of `RDB$RELATION_FIELDS.RDB$FIELD_POSITION + 1` of that column.

In our example, the columns are

1. `ID`
2. `FIRSTNAME`
3. `LASTNAME`
4. `BIRTHDATE`
5. `age`

*Prepared statement generated keys retrieval by index*

```
try (PreparedStatement statement = connection.prepareStatement(
         "insert into person(firstname, lastname, birthdate) values (?, ?, ?)",
         new int[] { 1, 5 })) {                          ①
  statement.setString(1, "Mark");
  statement.setString(2, "Rotteveel");
  statement.setObject(3, LocalDate.of(1979, 1, 12));

  statement.executeUpdate();
  try (ResultSet keys = statement.getGeneratedKeys()) {
    if (keys.next()) {
      int generatedId = keys.getInt("id");               ②
      int age = keys.getInt(2);                          ③

      System.out.printf("Id: %d, Age: %d%n",
          generatedId, age);
    }
  }
}
```

① Instead of `Statement.RETURN_GENERATED_KEYS`, the column indices are passed in an array, in this

case `1` for `ID` and `5` for `age`.

② Retrieval of the first column, `ID`, by name

③ Retrieval of the second column, `age`, by id. Notice that the index used for retrieval does not match the position (5) passed in the prepare. As this is the second column, it is retrieved from the result set by `2`.

In Jaybird 3 and earlier, the array of indices is sorted in ascending order before use: passing `new int[] { 4, 1, 3 }` will yield columns in order `ID`, `LASTNAME`, `BIRTHDATE`. In Jaybird 4, this sort is no longer applied, so columns will be in the order specified by the array: `BIRTHDATE`, `ID`, `LASTNAME`. To avoid issues, we recommend specifying the columns in ascending order, or always retrieve them by name.

### 10.1.3. Generated keys by column name

This form of generated keys execution involves the following methods:

- `Connection.prepareStatement(String sql, String[] columnNames)`
- `Statement.execute(String sql, String[] columnNames)`
- `Statement.executeUpdate(String sql, String[] columnNames)`
- `Statement.executeLargeUpdate(String sql, String[] columnNames)` (*JDBC 4.2*)

The values in the `String[]` are the column names to be returned. The column names provided are processed as is and are not checked for validity or the need of quoting. Providing non-existent or incorrectly (un)quoted columns will result in an exception when the statement is processed by Firebird (be aware: the JDBC specification is not entirely clear if this is valid behavior, so this might change in the future). This method is the fastest as it does not retrieve metadata from the server.

In Jaybird 3 and earlier, a null or empty array was silently ignored and the statement was executed normally (not producing generated keys). In Jaybird 4, this behaviour has changed and instead will throw an exception with message *"Generated keys array columnNames was empty or null. A non-empty array is required."*

**Example using column names**

*Prepared statement generated keys retrieval by name*

```
try (PreparedStatement statement = connection.prepareStatement(
        "insert into person(firstname, lastname, birthdate) values (?, ?, ?)",
        new String[] { "id", "\"age\"" })) {          ①
  statement.setString(1, "Mark");
  statement.setString(2, "Rotteveel");
  statement.setObject(3, LocalDate.of(1979, 1, 12));

  statement.executeUpdate();
  try (ResultSet keys = statement.getGeneratedKeys()) {
    if (keys.next()) {
      int generatedId = keys.getInt("id");
      int age = keys.getInt("age");

      System.out.printf("Id: %d, Age: %d%n",
          generatedId, age);
    }
  }
}
```

① The column names are passed as is, this means that correct quoting is required for case sensitive columns (and other names that require quoting).

The requirement to pass column names correctly quoted is not specified in the JDBC standard. It may change in future Jaybird versions to conform with column names as returned from `DatabaseMetaData.getColumn`. That is, unquoted exactly as stored in `RDB$RELATION_FIELDS.RDB$FIELD_NAME`. Quoting the column names would then be done by Jaybird.

When this changes, a connection property for backwards compatibility will be provided.

### 10.1.4. Configuring generated keys support

*Jaybird 4*

The connection property `generatedKeysEnabled` (alias `generated_keys_enabled`) allows the behaviour of generated keys support to be configured. This property is also available on data sources.

This property supports the following values (case insensitive):

- `default`: default behaviour to enable generated keys for statement types with `RETURNING` clause in the connected Firebird version. Absence of this property, `null` or empty string implies `default`.
- `disabled`: disable support. Attempts to use generated keys methods other than using `Statement.NO_GENERATED_KEYS` will throw a `SQLFeatureNotSupportedException`.
- `ignored`: ignore generated keys support. Attempts to use generated keys methods will not attempt to detect generated keys support and execute as if the statement generates no keys. The `Statement.getGeneratedKeys()` method will always return an empty result set. This behaviour is

equivalent to using the non-generated keys methods.

- A comma-separated list of statement types to enable.

For `disabled` and `ignored`, `DatabaseMetaData.supportsGetGeneratedKeys` will report `false`.

Because of the behaviour specified in the next section, typos in property values will behave as `ignored` (eg using `generatedKeysEnabled=disable` instead of `disabled` will behave as `ignored`).

**Selectively enable statement types**

This last option allows you to selectively enable support for generated keys. For example, `generatedKeysEnabled=insert` will only enable it for `insert` while ignoring it for all other statement types. Statement types that are not enabled will behave as if they generate no keys and will execute normally. For these statement types, `Statement.getGeneratedKeys()` will return an empty result set.

Possible statement type values (case insensitive) are:

- `insert`
- `update`
- `delete`
- `update_or_insert`
- `merge`

Invalid values will be ignored. If none of he specified statement types are supported by Firebird, it will behave as `ignored`.[20]

Some examples:

- `jdbc:firebird://localhost/testdb?generatedKeysEnabled=insert` will only enable insert support
- `jdbc:firebird://localhost/testdb?generatedKeysEnabled=merge` will only enable merge support. But only on Firebird 3 and higher, for Firebird 2.5 this will behave as `ignored` given the lack of `RETURNING` support for merge.
- `jdbc:firebird://localhost/testdb?generatedKeysEnabled=insert,update` will only enable insert and update support

This feature can be used to circumvent issues with frameworks or tools that always use generated keys methods for prepare or execution. For example with `UPDATE` statements that touch multiple records and - given the Firebird limitations for `RETURNING` - produce the error *"multiple rows in singleton select"*.

### 10.1.5. Limitations

Jaybird 2.2 does not support generated keys retrieval for batch execution of prepared statements. Support for generated key retrieval with batch execution was introduced in Jaybird 3.0.

## 10.2. Connection property ignoreProcedureType

*Jaybird 3.0.6*

On Firebird 2.1 and higher, Jaybird will use the procedure type information from the database metadata to decide how to execute `CallableStatement`. When a procedure is selectable, Jaybird will automatically transform a call-escape or `EXECUTE PROCEDURE` statement to a `SELECT`.

In some cases this automatic transformation to use a `SELECT` leads to problems. You can explicitly set `FirebirdCallableStatement.setSelectableProcedure(false)` to fix most of these issues, but this is not always an option. For example spring-data-jpa's `@Procedure` will not work correctly with selectable procedures, but you can't call `setSelectableProcedure`.

To disable this automatic usage of procedure type information, set connection property `ignoreProcedureType=true`. When necessary you can use `FirebirdCallableStatement.setSelectableProcedure(true)` to execute a procedure using `SELECT`.

Be aware though, when `EXECUTE PROCEDURE` is used with a selectable procedure, it is executed only up to the first `SUSPEND`, and the rest of the stored procedure is not executed.

For Firebird 2.0 and lower this property has no effect, as there the procedure type information is not available.

[20] This is not the case for the unsupported Firebird 1.0 and 1.5 versions. There this will behave similar to `disabled`, and you will need to explicitly specify `ignored` instead to get this behaviour.

# Chapter 11. General

## 11.1. Logging

Jaybird logs a variety of information during its work.

For logging, jaybird uses the following log levels:

| Jaybird log level | Description |
|---|---|
| *trace* | low-level debug information |
| *debug* | debug information |
| *info* | informational messages |
| *warn* | warnings |
| *error* | errors |
| *fatal* | severe/fatal errors (though in general, level *error* will be used instead of *fatal*) |

### 11.1.1. java.util.logging

*Jaybird 3*

Since Jaybird 3, logging will use `java.util.logging` by default.

Jaybird applies the following mapping for its log levels:

| Jaybird log level | *jul* log level |
|---|---|
| `Logger.trace` | `Level.FINER` |
| `Logger.debug` | `Level.FINE` |
| `Logger.info` | `Level.INFO` |
| `Logger.warn` | `Level.WARNING` |
| `Logger.error` | `Level.SEVERE` |
| `Logger.fatal` | `Level.SEVERE` |

### 11.1.2. Disable logging

*Jaybird 3*

To disable logging, specify system property `org.firebirdsql.jdbc.disableLogging` with value `true`.

### 11.1.3. Console logging

*Jaybird 2.2.8*

Jaybird can write its logging to the `System.out` for *info* and lower and `System.err` for *warn* and

above. Levels *debug* and *trace* are disabled in the implementation.

To enable logging console, you can set system property `org.firebirdsql.jdbc.forceConsoleLogger` (*Jaybird 3*) or `org.firebirdsql.jdbc.fallbackConsoleLogger` (*Jaybird 2.2.8 Jaybird 3*) to `true`.

The Jaybird 2.2.8 `org.firebirdsql.jdbc.fallbackConsoleLogger` only logs to console if Log4j logging is not enabled or Log4j is not on the classpath.

### 11.1.4. Custom logging implementation

*Jaybird 3*

Since Jaybird 3, you can provide your own logging implementation if you don't want to use `java.util.logging` or console logging.

To provide a custom logging implementation, you need to implement interface `org.firebirdsql.logging.Logger`. This implementation must be public and must have a public constructor with a single String argument for the logger name. Set system property `org.firebirdsql.jdbc.loggerImplementation` with the fully-qualified name of your implementation.

The `org.firebirdsql.logging.Logger` interface should be considered volatile and might change in minor releases (but not point/bugfix releases).

For example:

```
package org.example.jaybird.logging;

public class CustomLogger implements org.firebirdsql.logging.Logger {
    public CustomLogger(String name) {
        // create the logger
    }
    // implementation of org.firebirdsql.logging.Logger interface
}
```

On the Java command line, specify:

```
-Dorg.firebirdsql.jdbc.loggerImplementation=org.example.jaybird.logging.CustomLogger
```

### 11.1.5. Log4j 1.x

*Jaybird 3*

Jaybird 2.2 and earlier can log to Log4J 1.x. This requires the log4j 1.x library on the classpath and system property `FBLog4j` (or *Jaybird 2.2.8* `org.firebirdsql.jdbc.useLog4j`) set to `true`.

If you want to use Log4j 1.x with Jaybird 3 or higher, you will need to create your own logger implementation (see Custom logging implementation).

# Chapter 12. Datatype reference

This section documents non-standard datatypes supported by Jaybird, behaviour deviating from standard JDBC expectations, or recently introduced features. It does not provide full documentation for all supported datatypes.

## 12.1. Binary types BINARY/VARBINARY

The JDBC (and SQL standard) types `BINARY` and `VARBINARY` are called `CHAR(n) CHARACTER SET OCTETS` and `VARCHAR(n) CHARACTER SET OCTETS` in Firebird.

*Firebird 4* Firebird 4 introduces the names `BINARY` and `VARBINARY`/`BINARY VARYING` as aliases for `(VAR)CHAR(n) CHARACTER SET OCTETS`.

In Java binary and varbinary are usually handled with byte arrays and `InputStream`/`OutputStream`.

### 12.1.1. Connection property octetsAsBytes

*Jaybird 2.1.1*
*Jaybird 3*

In Jaybird 2.2 and earlier, the default for `(VAR)CHAR(n) CHARACTER SET OCTETS` is to handle columns and parameters of this type as a string. That is, `getObject(int/String)` returns `String`, and metadata reports `(VAR)CHAR` type information. The bytes are converted to string using the default encoding or the connection encoding.

Jaybird 2.1.1 introduced the boolean connection property `octetsAsBytes`. When set, `getObject(int/String)` will return `byte[]`, but otherwise columns or parameters will behave as normal string fields. Metadata information from sources like `DatabaseMetaData`, `ParameterMetaData`, and `ResultSetMetaData`, will report information as if it is a `String` (`(VAR)CHAR`) based field. *Jaybird 2.2.9* `ResultSetMetaData` reports `(VAR)BINARY` type information.

This approach was changed in Jaybird 3, see the next section

### 12.1.2. Always BINARY/VARBINARY

*Jaybird 3*

Jaybird 3 and higher no longer handle `(VAR)CHAR(n) CHARACTER SET OCTETS` as JDBC types `CHAR`/`VARCHAR`, but always as `BINARY`/`VARBINARY`. This closer matches their intended usage. The connection property `octetsAsBytes` is no longer supported.

Jaybird will report the JDBC `BINARY`/`VARBINARY` type information in all metadata (`DatabaseMetaData`, `ResultSetMetaData`, `ParameterMetaData`) for columns and parameters of type `(VAR)CHAR(n) CHARACTER SET OCTETS`, and `getObject(int/String)` will always return `byte[]`.

The getters (on result set/callable statement), setters (prepared/callable statement), and update methods (result set) for columns of this type are restricted to:

- `get/set/updateNull`
- `get/set/updateBytes`
- `get/set/updateBinaryStream`
- `get/set/updateAsciiStream`
- `get/set/updateString` (using the default encoding or connection encoding)
- `get/set/updateCharacterStream` (using the default encoding or connection encoding)
- `get/set/updateObject` (with `String`, `byte[]`, `InputStream`, `Reader`)
- `get/setObject` with a `java.sql.RowId` (*Jaybird 4*)
- `get/setRowId` (*Jaybird 4*)

Other getters/setters/updaters or object types supported for 'normal' `(VAR)CHAR` fields are not available.

# 12.2. Type BOOLEAN

*Firebird 3*

Firebird 3 introduced the SQL standard type `BOOLEAN`.

## 12.2.1. Support for BOOLEAN

*Jaybird 2.2*

Jaybird 2.2 introduced support for `BOOLEAN`.

On parameters of type `BOOLEAN`, Jaybird supports most of the other Java types, using the following mapping:

| Type | `setXXX` | `getXXX` |
|---|---|---|
| `String` | `"true"`/`"Y"`/`"T"`/`"1"` sets true, everything else sets false | true is `"true"`, false is `"false"` |
| integer types | `0` sets false, everything else sets true | true is `1`, false is `0` |
| `float` | Exact `0.0f` sets false, everything else sets true [21] | true is `1.0f`, false is `0.0f` |
| `double` | Exact `0.0` sets false, everything else sets true [21] | true is `1.0`, false is `0.0` |
| `BigDecimal` | `0` (`ZERO`) (using `compareTo`) sets false, everything else sets true | true is `1` (`ONE`), false is `0` (`ZERO`) |

We recommend to avoid the `float`, `double` and `BigDecimal` options.

## 12.2.2. Workarounds for booleans

Firebird 2.5 and earlier do not support `BOOLEAN`, but support for booleans can be simulated

The following options are recommended:

- Use `SMALLINT` with values `0` and `1`
- Use `CHAR(1)` (or `VARCHAR(1)`) with values `'Y'` and `'N'`
- Use `CHAR` or `VARCHAR` with length 5 or longer with values `'true'` and `'false'`

We recommend creating a domain for 'simulated' booleans with a check constraint to restrict the possible values. If you do this, name the domain something like `D_BOOLEAN`, and avoid the name `BOOLEAN` to prevent problems when upgrading to Firebird 3.

For non-boolean types, Jaybird supports the following conversions with `setBoolean` and `getBoolean`:

| Type | getBoolean | setBoolean |
|---|---|---|
| `(VAR)CHAR(<5)` | `'Y'`/`'T'`/`'1'`/`'true'` is true (case insensitive), everything else is false | true sets `'Y'`, false sets `'N'` |
| `(VAR)CHAR(>=5)` | `'Y'`/`'T'`/`'1'`/`'true'` is true (case insensitive), everything else is false | true sets `'true'`, false sets `'false'` |
| `BLOB SUB_TYPE TEXT` | `'Y'`/`'T'`/`'1'`/`'true'` is true (case insensitive), everything else is false | true sets `'true'`, false sets `'false'` |
| `SMALLINT`/`INTEGER`/`BIGINT` | `1` is true, everything else is false | true sets `1`, false sets `0` |
| `DECIMAL`/`NUMERIC` | Truncated integral value `1` is true, everything else is false | true sets `1`, false sets `0` |
| `REAL`/`FLOAT` | Exact `1.0f` is true, everything else is false [22] | true sets `1.0f`, false sets `0.0f` |
| `DOUBLE PRECISION` | Exact `1.0` is true, everything else is false [22] | true sets `1.0`, false sets `0.0` |
| `DECFLOAT` | Exact `1E0` is true, everything else is false (including `1.0E0`!) [23] | true sets `1E0`, false sets `0E0` |

We recommend to avoid the `DECIMAL`, `NUMERIC`, `REAL`/`FLOAT`, `DOUBLE PRECISION` or `DECFLOAT` options.

# 12.3. Date/time types

## 12.3.1. Time zones

*Firebird 4*

Firebird 4 introduced time zone types, with types `TIME WITH TIME ZONE` and `TIMESTAMP WITH TIME ZONE`. See the Firebird 4 release notes and `doc/sql.extensions/README.time_zone.md` in the Firebird installation for details on these types.

*Jaybird 3.0.6*
Since Jaybird 3.0.6, two connection properties, `timeZoneBind` and `sessionTimeZone`, are available to provide limited support for these types. See Time zone bind configuration and Connection property sessionTimeZone for details.

*Jaybird 4*
Since Jaybird 4, the time zone types are supported under Java 8 and higher, using the Java 8 (or higher) version of Jaybird. Time zone types are not supported under Java 7, and you will need to enable legacy time zone bind to use these types. With legacy time zone bind, Firebird will convert to the equivalent `TIME` and `TIMESTAMP` (`WITHOUT TIME ZONE`) types using the session time zone. Time zone bind can be configured with connection property `timeZoneBind`, for more information see Time zone bind configuration.

#### Scope of time zone support

*Jaybird 4*

JDBC 4.2 introduced support for time zones, and maps these types to `java.time.OffsetTime` and `java.time.OffsetDateTime`. JDBC does not define explicit setters for these types. Use `setObject(index, value)`, `updateObject(index, value)`, `getObject(index/name)` or `getObject(index/name, classType)`.

Firebird 4 supports both offset and named time zones. Given the definition in JDBC, Jaybird only supports offset time zones. On retrieval of a value with a named zone, Jaybird will make a best effort to convert to the equivalent offset using Java's time zone information. If no mapping is available the time will be returned at UTC (offset zero).

Jaybird 4 supports the following Java types on fields of time zone types (those marked with * are not defined in JDBC)

`TIME WITH TIME ZONE`:

- `java.time.OffsetTime` (default for `getObject`)
  - On get, if the value is a named zone, it will derive the offset using the current date
- `java.time.OffsetDateTime`
  - On get the current date is added
  - On set the date information is removed
- `java.lang.String`
  - On get applies `OffsetTime.toString()` (eg `13:25:13.1+01:00`)
  - On set tries the default parse format of either `OffsetTime` or `OffsetDateTime` (eg `13:25:13.1+01:00` or `2019-03-10T13:25:13+01:00`) and then sets as that type
- `java.sql.Time` (*)
  - On get obtains `java.time.OffsetDateTime`, converts this to epoch milliseconds and uses `new java.sql.Time(millis)`
  - On set applies `toLocalTime()`, combines this with `LocalDate.now()` and then derives the offset time for the default JVM time zone
- `java.sql.Timestamp` (*)
  - On get obtains `java.time.OffsetDateTime`, converts this to epoch milliseconds and uses `new java.sql.Timestamp(millis)`
  - On set applies `toLocalDateTime()` and derives the offset time for the default JVM time zone

`TIMESTAMP WITH TIME ZONE`:

- `java.time.OffsetDateTime` (default for `getObject`)
- `java.time.OffsetTime` (*)
  - On get, the date information is removed
  - On set, the current date is added
- `java.lang.String`
  - On get applies `OffsetDateTime.toString()` (eg `2019-03-10T13:25:13.1+01:00`)
  - On set tries the default parse format of either `OffsetTime` or `OffsetDateTime` (eg `13:25:13.1+01:00` or `2019-03-10T13:25:13+01:00`) and then sets as that type
- `java.sql.Time` (*)
  - On get obtains `java.time.OffsetDateTime`, converts this to epoch milliseconds and uses `new java.sql.Time(millis)`
  - On set applies `toLocalTime()`, combines this with `LocalDate.now()` and then derives the offset date time for the default JVM time zone
- `java.sql.Timestamp` (*)
  - On get obtains `java.time.OffsetDateTime`, converts this to epoch milliseconds and uses `new java.sql.Timestamp(millis)`
  - On set applies `toLocalDateTime()` and derives the offset date time for the default JVM time zone
- `java.sql.Date` (*)
  - On get obtains `java.time.OffsetDateTime`, converts this to epoch milliseconds and uses `new java.sql.Date(millis)`
  - On set applies `toLocalDate()` at start of day and derives the offset date time for the default JVM time zone

#### Support for legacy JDBC date/time types

For the `WITH TIME ZONE` types, JDBC does not define support for the legacy JDBC types (`java.sql.Time`, `java.sql.Timestamp` and `java.sql.Date`). To ease the transition and potential compatibility with tools and libraries, Jaybird does provide support. However, we strongly recommend to avoid using these types.

Compared to the `WITHOUT TIME ZONE` types, there may be small discrepancies in values as Jaybird uses 1970-01-01 for `WITHOUT TIME ZONE`, while for `WITH TIME ZONE` it uses the current date. If this is problematic, then either apply the necessary conversions yourself, enable legacy time zone bind, or define or cast your columns to `TIME` or `TIMESTAMP`.

#### No support for other java.time types

The types `java.time.LocalTime`, `java.time.LocalDateTime` and `java.time.LocalDate` are not supported for the time zone types. Supporting these types would be ambiguous. If you need to use these, then either apply the necessary conversions yourself, enable legacy time zone bind, or define or cast your columns as `TIME` or `TIMESTAMP`.

Jaybird also does not support non-standard extensions like `java.time.Instant`, or `java.time.ZonedDateTime`. If there is interest, we may add them in the future.

#### Time zone bind configuration

*Jaybird 3.0.6*

The connection property `timeZoneBind` (alias `time_zone_bind`) is a connection property to configure the time zone bind (see also `SET TIME ZONE BIND` in the Firebird 4 release notes).

This property needs to be explicitly set if you are using Jaybird 4 on Java 7 or Jaybird 3 (on any Java version), and need to handle the `WITH TIME ZONE` types. It can also be used for tools or applications that expect `java.sql.Time`/`Timestamp` types and cannot use the `java.time.OffsetTime`/`OffsetDateTime` types returned for the `WITH TIME ZONE` types.

Possible values (case insensitive):

- `legacy`

  Firebird will convert a `WITH TIME ZONE` type to the equivalent `WITHOUT TIME ZONE` type using the session time zone to derive the value.

  Result set columns and parameters on prepared statements will behave as the equivalent `WITHOUT TIME ZONE` types. This conversion is not applied to the database metadata which will always report `WITH TIME ZONE` information.

- `native`

  Behaves as default (`WITH TIME ZONE` types supported), but value will be explicitly set.

Any other value will result in error `isc_time_zone_bind` (code 335545255, message *"Invalid time zone bind mode <value>"*) on connect.

This feature requires Firebird 4 beta 2 or higher (or a snapshot build version 4.0.0.1481 or later). It will be ignored in earlier builds as the necessary database parameter buffer item does not exist in earlier versions.

#### Connection property sessionTimeZone

The connection property `sessionTimeZone` (alias `session_time_zone`) does two things:

1. specifies the Firebird 4 session time zone (*Jaybird 3.0.6*),
2. specifies the time zone to use when converting values of without time zone types to the legacy JDBC datetime types on all Firebird versions (*Jaybird 4*).

See Firebird 4 session time zone for information on the effects of `sessionTimeZone` on the server-side.

Valid values are time zone names known by Firebird, we recommend to use the long names (eg `Europe/Amsterdam`) and not the ambiguous short IDs (eg `CET`). Although not required, we recommend to use time zone names that are known by Firebird and Java (see Session time zone for conversion

for caveats).

In Jaybird 3, `sessionTimeZone` will only configure the server-side session time zone. Client-side, Jaybird will continue to use the JVM default time zone for parsing the without time zone values to the `java.sql.Time/Timestamp/Date` types. Setting `sessionTimeZone` to the JVM default time zone will yield the best (ie correct) values, but not setting it (and thus using the server default) will retain behaviour that is backwards compatible with behaviour of previous versions of Jaybird. In Jaybird 4, this property also configures client-side parsing of values to these legacy types.

On Jaybird 3, we recommend not setting this property, or setting it to the default JVM time zone. If you set it to a different time zone, then we recommend that you do not use the legacy `java.sql.Time/Timestamp/Date` types, but instead use `java.time.LocalTime/LocalDateTime/LocalDate`.

The remainder of this section only applies to Jaybird 4 and higher.

By default, Jaybird 4 and higher will use the JVM default time zone as reported by `java.util.TimeZone.getDefault().getID()` as the session time zone. Using the JVM default time zone as the default is the best option in the light of JDBC requirements with regard to `java.sql.Time` and `java.sql.Timestamp` using the JVM default time zone.

To use the default server time zone and the Jaybird 3 and earlier behaviour to use the JVM default time zone, set the connection property to `server`. This will result in the conversion behaviour of Jaybird 3 and earlier. Be aware that this is inconsistent if Firebird and Java are in different time zones.

#### Firebird 4 session time zone

The session time zone is used for conversion between `WITH TIME ZONE` values and `WITHOUT TIME ZONE` values (ie using cast or with legacy time zone bind), and for the value of `LOCALTIME`, `LOCALTIMESTAMP`, `CURRENT_TIME` and `CURRENT_TIMESTAMP`, and other uses of the session time zone as documented in the Firebird 4 documentation.

The value of `sessionTimeZone` must be supported by Firebird 4. It is possible that time zone identifiers used by Java are not supported by Firebird. If Firebird does not know the session time zone, error (`Invalid time zone region: <zone name>`) is reported on connect.

In Jaybird 4 and higher, Jaybird will apply the JVM default time zone as the default session time zone. The use of the JVM default time zone as the default session time zone will result in subtly different behaviour compared to previous versions of Jaybird and - even with Jaybird 4 - Firebird 3 or earlier, as current time values like `LOCALTIMESTAMP` (etc) will now reflect the time in the JVM time zone, and not the server time zone rebased on the JVM default time zone.

As an example, with a Firebird in Europe/London and a Java application in Europe/Amsterdam with Firebird time 12:00, in Jaybird 3, the Java application will report this time as 12:00, in Jaybird 4 with Firebird 4, this will now report 13:00, as that is the time in Amsterdam if it is 12:00 in London (ignoring potential DST start/end differences).

Other examples include values generated in triggers and default value clauses.

**Session time zone for conversion**

*Jaybird 4*

For `WITHOUT TIME ZONE` types, the session time zone will be used to derive the `java.sql.Time`, `java.sql.Timestamp` and `java.sql.Date` values. This is also done for Firebird 3 and earlier.

If Java does not know the session time zone, no error is reported, but when retrieving `java.sql.Time`, `java.sql.Timestamp` or `java.sql.Date` a warning is logged and conversion will happen in GMT, which might yield unexpected values.

We strongly suggest that you use `java.time.LocalTime`, `java.time.LocalDateTime` and `java.time.LocalDate` types instead of these legacy datetime types.

For `WITH TIME ZONE` types, the session time zone has no effect on the conversion to the legacy JDBC date/time types: the offset date/time is converted to epoch milliseconds and used to construct these legacy types directly.

Executing `SET TIME ZONE <zone name>` statements after connect will change the session time zone on the server, but Jaybird will continue to use the session time zone set in the connection property for these conversions.

### Time zone support for CONVERT

Although not defined in JDBC (or ODBC), Jaybird has added a non-standard extension to the `CONVERT` JDBC escape to allow conversion to the time zone types.

In addition to the standard-defined types, it also supports the type names `TIME_WITH_TIME_ZONE`, `TIME_WITH_TIMEZONE`, `TIMESTAMP_WITH_TIME_ZONE` and `TIMESTAMP_WITH_TIMEZONE` (and the same with the `SQL_` prefix).

### Caveats for time zone types

- Time zone fields do not support `java.time.LocalDate`, `java.time.LocalTime`, `java.time.LocalDateTime`.
- Firebird 4 redefines `CURRENT_TIME` and `CURRENT_TIMESTAMP` to return a `WITH TIME ZONE` type. Use `LOCALTIME` and `LOCALTIMESTAMP` (introduced in Firebird 3.0.4) if you want to ensure a `WITHOUT TIME ZONE` type is used.
- The database metadata will always return JDBC 4.2 compatible information on time zone types, even on Java 7, and even when legacy time zone bind is set. For Java 7 compatibility the JDBC 4.2 `java.sql.Types` constants `TIME_WITH_TIMEZONE` and `TIMESTAMP_WITH_TIMEZONE` are also defined in `org.firebirdsql.jdbc.JaybirdTypeCodes`.
- ([.since]_Jaybird 4) The default `sessionTimeZone` is set to the JVM default time zone, this may result in different application behavior for `DATE`, `TIME` and `TIMESTAMP`, including values generated in triggers and default value clauses. To prevent this, either switch those types to a `WITH TIME ZONE` type, or set the `sessionTimeZone` to `server` or to the actual time zone of the Firebird server.

# 12.4. Decimal floating point type DECFLOAT

*Firebird 4*

Firebird 4 introduces the SQL:2016 `DECFLOAT` datatype, a decimal floating point with a precision of 16 or 34 digits (backed by an IEEE-754 Decimal64 or Decimal128). See the Firebird 4 documentation for details on this datatype.

## 12.4.1. Decfloat support in Jaybird 4 and higher

*Jaybird 4*

Jaybird 4 introduces support for the `DECFLOAT` datatype mapping to `java.math.BigDecimal`. For more information, see DECFLOAT support.

## 12.4.2. Workarounds for Jaybird 3

*Jaybird 3.0.6*

Jaybird 3 does not support `DECFLOAT`, but starting with Jaybird 3.0.6, the connection property `decfloatBind` can be used to convert to a datatype that is supported by Jaybird. For earlier Jaybird 3 versions, see Workarounds for Jaybird 2.2 and earlier.

We recommend either `decfloatBind=char` or `decfloatBind=double precision`. Option `char` has our preference as it is able to support the full range of values of the `DECFLOAT` types.

See Connection property decfloatBind for details.

## 12.4.3. Workarounds for Jaybird 2.2 and earlier

Jaybird 2.2 and earlier do not support `DECFLOAT`. As a workaround, you can use the `SET DECFLOAT BIND <bind-type>` statement to configure your connection to map `DECFLOAT` to a different datatype. When set, Firebird will present columns or parameters of `DECFLOAT` as the specified type, allowing clients without support for `DECFLOAT` to read or set values.

The available options are the same as documented in Connection property decfloatBind

For example

```
try (Connection connection = DriverManager.getConnection(..);
     Statement stmt = connection.createStatement()) {
    stmt.execute("SET DECFLOAT BIND char");
    // DECFLOAT will now be mapped to a CHAR datatype
}
```

The effect of the `SET DECFLOAT BIND` statement will be reset to the default when `ALTER SESSION RESET` is executed.

### 12.4.4. Connection property decfloatBind

*Jaybird 3.0.6*

Jaybird 3.0.6 added the connection property `decfloatBind` to configure conversion of `DECFLOAT` to a different datatype. When set Firebird will present columns or parameters of `DECFLOAT` as the specified type, allowing clients without support for `DECFLOAT` to read or set values.

This property is also available in Jaybird 4, but we recommend to not use this property in Jaybird 4 and instead rely on the default behaviour (`native`) and support for `DECFLOAT`.

This feature requires Firebird 4 beta 2 or higher (or a snapshot build version 4.0.0.1481 or later). It will be ignored in earlier builds as the necessary database parameter buffer item does not exist in earlier versions.

The `decfloatBind` (alias: `decfloat_bind`) connection property has the following valid values:

`native`
: (default) produces the normal `DECFLOAT` values (not supported by Jaybird 3)

`char`/`character`
: converts the `DECFLOAT` values to an equivalent string using scientific notation (eg 6.573820132214283E+283).

`double precision`
: converts the `DECFLOAT` values to an equivalent `double precision`. This may lose precision, and this option cannot support the full range of values of `DECFLOAT(34)` and can result in errors on overflow.

`bigint, n`
: where `n` is the target scale to convert to a `NUMERIC(18,n)`. This option cannot support the full range of values of `DECFLOAT` and can result on errors on overflow.

These properties can be used as connection properties with `DriverManager`. For Jaybird data sources, the properties must be set using `setNonStandardProperty` as corresponding setters have not been defined.

The value set using connection property `decfloatBind` will be the default reverted to by `ALTER SESSION RESET`.

### 12.4.5. DECFLOAT support

*Jaybird 4*

Jaybird 4 introduces support for the `DECFLOAT` datatype. The 'default' object type for `DECFLOAT` is a `java.math.BigDecimal`, but conversion from and to the following datatypes is supported:

- `java.math.BigDecimal` (see note 1)

- `byte` (valid range -128 to 127(!); see notes 2, 3)
- `short` (valid range -32768 to 32767; see note 3)
- `int` (valid range -$2^{31}$ to $2^{31}$-1; see note 3)
- `long` (valid range -$2^{63}$ to $2^{63}$-1; see notes 3, 4)
- `float` (valid range -1 * Float.MAX_VALUE to Float.MAX_VALUE; see notes 5, 6, 7, 8, 9)
- `double` (valid range -1 * Double.MAX_VALUE to Double.MAX_VALUE; see notes 6, 7, 8, 9)
- `boolean` (see notes 10, 11)
- `java.lang.String` (see notes 12, 13, 14)
- `java.math.BigInteger` (see notes 15, 16)
- `org.firebirdsql.extern.decimal.Decimal32/64/128` (see notes 17, 18)

The `DECFLOAT` type is not yet defined in the JDBC specification. For the time being, Jaybird defines a Jaybird specific type code with value `-6001`. This value is available through constant `org.firebirdsql.jdbc.JaybirdTypeCodes.DECFLOAT`, or - for JDBC 4.2 and higher - `org.firebirdsql.jdbc.JaybirdType.DECFLOAT`, which is an enum implementing `java.sql.SQLType`.

If you need to use the type code, we suggest you use these constants. If a `DECFLOAT` type constant gets added to the JDBC standard, we will update the value. The enum value will be deprecated when that version of JDBC has been released.

Jaybird uses a local copy of the FirebirdSQL/decimal-java [https://github.com/FirebirdSQL/decimal-java] library, with a custom package `org.firebirdsql.extern.decimal`. This to avoid additional dependencies.

#### Precision and range

The `DECFLOAT` datatype supports values with a precision of 16 or 34 decimal digits, and an exponent [24] between -398 and 369 (`DECFLOAT(16)`), or between -6176 and 6111 (`DECFLOAT(34)`), so the minimum and maximum values are:

| Type | Min/max value | Smallest (non-zero) value |
|---|---|---|
| `DECFLOAT(16)` | +/-9.9..9E+384 (16 digits) | +/-1E-398 (1 digit) |
| `DECFLOAT(34)` | +/-9.9..9E+6144 (34 digits) | +/-1E-6176 (1 digit) |

When converting values from Java types to `DECFLOAT` and retrieving `DECFLOAT` values as `Decimal32` or `Decimal64`, the following rules are applied:

- Zero values can have a non-zero exponent, and if the exponent is out of range, the exponent value is 'clamped' to the minimum or maximum exponent supported. This behavior is subject to change, and future releases may 'round' to exact `0` (or `0E0`)
- Values with a precision larger than the target precision are rounded to the target precision using `RoundingMode.HALF_EVEN`
- If the magnitude (or exponent) is too low, then the following steps are applied:

1. Precision is reduced applying `RoundingMode.HALF_EVEN`, increasing the exponent by the reduction of precision.

   An example: a `DECFLOAT(16)` stores values as an integral coefficient of 16 digits and an exponent between `-398` and `+369`. The value `1.2345678901234567E-394` or `12345678901234567E-409` is coefficient `1234567890123456` and exponent `-409`. The coefficient is 16 digits, but the exponent is too low by 11.

   If we sacrifice least-significant digits, we can increase the exponent, this is achieved by dividing the coefficient by $10^{11}$ (and rounding) and increasing the exponent by 11. We get exponent = round(1234567890123456 / $10^{11}$) = 12346 and exponent = -409 + 11 = -398.

   The resulting value is now `12346E-398` or `1.2346E-394`, or in other words, we sacrificed precision to make the value fit.

2. If after the previous step, the magnitude is still too low, we have what is called an underflow, and the value is truncated to 0 with the minimum exponent and preserving sign, eg for `DECFLOAT(16)`, the value will become +0E+398 or -0E-398 (see note 19). Technically, this is just a special case of the previous step.

- If the magnitude (or exponent) is too high, then the following steps are applied:

  1. If the precision is less than maximum precision, and the difference between maximum precision and actual precision is larger than or equal to the difference between the actual exponent and the maximum exponent, then the precision is increased by adding zeroes as least-significant digits and decreasing the exponent by the number of zeroes added.

     An example: a `DECFLOAT(16)` stores values as an integral coefficient of 16 digits and an exponent between `-398` and `+369`. The value `1E+384` is coefficient `1` with exponent `384`. This is too large for the maximum exponent, however, we have a value with a single digit, leaving us with 15 'unused' most-significant digits.

     If we multiply the coefficient by $10^{15}$ and subtract 15 from the exponent we get: coefficient = 1 * $10^{15}$ = 1000000000000000 and exponent = 384 - 15 = 369. And these values for coefficient and exponent are in range of the storage requirements.

     The resulting value is now `1000000000000000E+369` or `1.000000000000000E+384`, or in other words, we 'increased' precision by adding zeroes as least-significant digits to make the value fit.

  2. Otherwise, we have what is called an overflow, and an `SQLException` is thrown as the value is out of range.

If you need other rounding and overflow behavior, make sure you round the values appropriately before you set them.

**Notes**

1. `java.math.BigDecimal` is capable of representing numbers with larger precisions than `DECFLOAT`, and numbers that are out of range (too large or too small). When performing calculations in Java, use `MathContext.DECIMAL64` (for `DECFLOAT(16)`) or `MathContext.DECIMAL128` (for `DECFLOAT(34)`)

to achieve similar results in calculations as in Firebird. Be aware there might still be differences in rounding, and the result of calculations may be out of range.

1. Firebird 4 snapshots currently allow storing NaN and Infinity values, retrieval of these values will result in a `SQLException`, with a `DecimalInconvertibleException` cause with details on the special. The support for these special values is currently under discussion and may be removed in future Firebird 4 snapshots.

2. `byte` in Java is signed, and historically Jaybird has preserved sign when storing byte values, and it considers values outside -128 and +127 out of range.
3. All integral values are - if within range - first converted to `long` using `BigDecimal.longValue()`, which discards any fractional parts (rounding by truncation).
4. When storing a `long` in `DECFLOAT(16)`, rounding will be applied using `RoundingMode.HALF_EVEN` for values larger than `9999999999999999L` or smaller than `-9999999999999999L`.
5. `float` values are first converted to (or from) double, this may lead to small rounding differences
6. `float` and `double` can be fully stored in `DECFLOAT(16)` and `DECLOAT(34)`, with minor rounding differences.
7. When reading `DECFLOAT` values as `double` or `float`, rounding will be applied as binary floating point types are inexact, and have a smaller precision.
8. If the magnitude of the `DECFLOAT` value is too great to be represented in `float` or `double`, +Infinity or -Infinity may be returned (see `BigDecimal.doubleValue()`). This behavior is subject to change, future releases may throw a `SQLException` instead, see also related note 9.
9. Storing and retrieving values NaN, +Infinity and -Infinity are currently supported, but this may change as this doesn't seem to be allowed by the SQL:2016 standard.

   It is possible that Jaybird or Firebird will disallow storing and retrieving NaN and Infinity values in future releases, causing Jaybird to throw an `SQLException` instead. We strongly suggest not to rely on this support for special values.

   1. Firebird `DECFLOAT` currently discerns four different NaNs (+/-NaN and +/-signaling-NaN). These are all mapped to `Double.NaN` (or `Float.NaN`), Java NaN values are mapped to +NaN in Firebird.

10. Setting `boolean` values will set `0` (or `0E+0`) for `false` and `1` (or `1E+0`) for `true`.
11. Retrieving as `boolean` will return `true` for `1` (exactly `1E+0`) and `false` for **all other values**. Be aware that this means that `1.0E+0` (or `10E-1`) etc will be **`false`** (*this may change before Jaybird 4 final to `getLong() == 1L` or similar, which truncates the value*).

    This behavior may change in the future and only allow `0` for `false` and exactly `1` for `true` and throw an `SQLException` for all other values, or maybe `true` for everything other than `0`. In general we advise to not use numerical types for boolean values, and especially not to retrieve the result of a calculation as a boolean value. Instead, use a real `BOOLEAN`.

12. Setting values as `String` is supported following the format rules of `new BigDecimal(String)`, with extra support for special values `+NaN`, `-NaN`, `+sNaN`, `-sNaN`, `+Infinity` and `-Infinity` (case insensitive). Other non-numerical strings throw an `SQLException` with a `NumberFormatException` as cause. Out of range values are handled as described in Precision and range.

13. Getting values as `String` will be equivalent to `BigDecimal.toString()`, with extra support for the special values mentioned in the previous note.
14. As mentioned in earlier notes, support for the special values is under discussion, and may be removed in the final Jaybird 4 or Firebird 4 release, or might change in future versions.
15. Getting as `BigInteger` will behave as `BigDecimal.toBigInteger()`, which discards the fractional part (rounding by truncation), and may add `(-1 * scale - precision)` least-significant zeroes if the scale exceeds precision. Be aware that use of `BigInteger` for large values may result in significant memory consumption.
16. Setting as `BigInteger` will lose precision for values with more digits than the target type. It applies the rules described in Precision and range.
17. Values can also be set and retrieved as types `Decimal32`, `Decimal64` and `Decimal128` from the `org.firebirdsql.extern.decimal` package. Where `Decimal64` exactly matches the `DECFLOAT(16)` protocol format, and `Decimal128` the `DECFLOAT(34)` protocol format. Be aware that this is an implementation detail that might change in future Jaybird versions (both in terms of support for these types, and in terms of the interface (API) of these types).
18. Setting a `Decimal128` on a `DECFLOAT(16)`, or a `Decimal32` on a `DECFLOAT(16)` or `DECFLOAT(34)`, or retrieving a `Decimal32` from a `DECFLOAT(16)` or `DECFLOAT(34)`, or a `Decimal64` from a `DECFLOAT(34)` will apply the rules described in Precision and range.
19. Zero values can have a sign (eg `-0` vs `0` (`+0`)), this can only be set or retrieved using `String` or the `DecimalXX` types, or the result of rounding. This behaviour is subject to change, and future releases may 'round' to `0` (aka `+0`).

# 12.5. Exact numeric types DECIMAL/NUMERIC

The JDBC types `DECIMAL` and `NUMERIC` are supported by Firebird and Jaybird and map to `java.math.BigDecimal`.

Behaviour in Firebird of `NUMERIC` is closer to the SQL standard `DECIMAL` behaviour. The precision specified is the minimum precison, not the exact precision.

## 12.5.1. Precision and range

In Firebird 3 and earlier, the maximum precision of `DECIMAL` and `NUMERIC` is 18 with a maximum scale of 18.[25]

*Firebird 4 Jaybird 4* In Firebird 4 the maximum precision and scale of `DECIMAL` and `NUMERIC` have been raised to 34. Any `NUMERIC` or `DECIMAL` with a precision between 19 and 34 will allow storage up to a precision of 34.

In the implementation in Firebird, this extended precision is backed by a IEEE-754 Decimal128 which is also used for `DECFLOAT` support.

Values set on a field or parameter will be rounded to the target scale of the field using `RoundingMode.HALF_EVEN`. Values exceeding a precision of 34 after rounding will be rejected with a `TypeConversionException`.

[21] This behaviour may change in a future version to the equivalent of `setLong((long) value)`

[22] This behaviour may change in a future version to the equivalent of `getLong(..) == 1L`

[23] This behaviour may change in a future version to use `compareTo` or the equivalent of `getLong(..) == 1L` instead

[24] The `DECFLOAT` decimal format stores values as sign, integral number with 16 or 34 digits, and an exponent. This is similar to `java.math.BigDecimal`, but instead of an exponent, that uses the concept `scale`, where `scale = -1 * exponent`.

[25] In practice, values with precision 19 are possible up to the maximum value of the `BIGINT` backing the value.

# Appendices

# Appendix A: Extended connection properties

Jaybird has a number of connection properties that can be used to configure a connection.

This appendix provides a list of most connection properties and a short explanation to each of them. The properties listed below are usable as JDBC connection properties.

The properties marked as *boolean property* can be included in the JDBC URL with values `true`, but also without a value, or with an empty value. The default for these properties is always `false`. For readability we suggest that you only specify these properties explicitly when you want to enable them, and if you do, to use explicit value `true`.

A subset of these properties is also exposed in the `javax.sql.DataSource` implementations in Jaybird. When using data sources, unexposed properties can be set using `setNonStandardProperty`.

## A.1. Authentication and security properties

| Connection property (+ aliases) | Explanation |
|---|---|
| `user`<br>`userName`, `user_name`, `isc_dpb_user_name` | Name of the user for the connection. |
| `password`<br>`isc_dpb_password` | Password corresponding to the specified user. |
| `roleName`<br>`sqlRole`, `sql_role_name`, `isc_dpb_sql_role_name` | Name of the SQL role for the specified connection. |
| `authPlugins`<br>`auth_plugin_list` | Jaybird specific property (*Jaybird 4*). The comma-separated list of authentication plugins to try. See Authentication plugins for more information.<br>Default: `Srp256,Srp` |
| `wireCrypt` | Jaybird specific property (*Jaybird 3.0.4*). Allowed values: `DEFAULT`, `ENABLED`, `REQUIRED`, `DISABLED` (case insensitive). Configures Firebird 3 wire encryption behaviour. See Wire encryption support for more information.<br>Default: `DEFAULT`. |
| `dbCryptConfig` | Jaybird specific property (*Jaybird 3.0.4*). Configures Firebird 3 database encryption support. See Database encryption support for more information. |

| Connection property (+ aliases) | Explanation |
|---|---|
| `process_id`<br>`processId` (*Jaybird 3*), `isc_dpb_process_id` | Specifies the process id reported to Firebird. See Process information for more information. |
| `process_name`<br>`processName` (*Jaybird 3*), `isc_dpb_process_name` | Specifies the process name reported to Firebird. See Process information for more information. |

# A.2. Other properties

| Connection property (+ aliases) | Explanation |
|---|---|
| `encoding`<br>`lc_ctype`, `isc_dpb_lc_ctype` | Character encoding for the connection using the Firebird character set name. This property tells the database server the encoding in which it expects character content. For a list of the available encodings see Available Encodings.<br>Default: `NONE` (unless `charSet` is specified). |
| `charSet`<br>`localEncoding` | Jaybird specific property. Character set for the connection using Java character set name. Similar to the previous property, however instead of Firebird-specific name allows using a Java character set name. |
| `sqlDialect`<br>`dialect` (*Jaybird 3*), `sql_dialect`, `isc_dpb_sql_dialect` | SQL dialect, can be 1, 2 and 3.<br>Default: 3 (in Jaybird 2.2 and earlier the default is a combination of dialect 3 and the actual database dialect) |
| `defaultHoldable`<br>`defaultResultSetHoldable` | Jaybird specific property. Boolean property. Set result sets to be holdable by default. A workaround for applications with incorrect assumptions on result sets in auto-commit mode. See Default holdable result sets for more information. |
| `useFirebirdAutocommit` | Jaybird specific property (*Jaybird 2.2.9*). Boolean property. Enable experimental feature to use Firebird auto-commit for JDBC auto-commit mode. See Firebird auto commit mode (experimental) for more information. |
| `generatedKeysEnabled` | Jaybird specific property (*Jaybird 4*). Configure generated keys support behaviour. See Configuring generated keys support for more information. |

| Connection property (+ aliases) | Explanation |
|---|---|
| `isolation`<br>`defaultIsolation` | Jaybird specific property. Specify the default transaction isolation level. Accepted values are: `TRANSACTION_NONE`, `TRANSACTION_READ_UNCOMMITTED`, `TRANSACTION_READ_COMMITTED`, `TRANSACTION_REPEATABLE_READ`, `TRANSACTION_SERIALIZABLE` (case sensitive). Although `TRANSACTION_NONE` and `TRANSACTION_READ_UNCOMMITTED` are allowed values, these behave the same as `TRANSACTION_READ_COMMITTED`.<br>Default: `TRANSACTION_READ_COMMITTED` |
| `socketBufferSize`<br>`socket_buffer_size` | Jaybird specific property. Tells Jaybird Type 4 driver the size of the socket buffer. Should be used on the systems where default socket buffer provided by JVM is not correct. |
| `blobBufferSize`<br>`blob_buffer_size` | Jaybird specific property. Tells the driver the size of the buffer that is used to transfer BLOB content. It is recommended to keep the value equal to `n * <database page size>` (and preferably also socket buffer size). |
| `soTimeout` | Jaybird specific property. Socket blocking timeout in milliseconds. Only has effect on Type 4 (pure Java) connections. |
| `connectTimeout`<br>`connect_timeout`,<br>`isc_dpb_connect_timeout` | Connect timeout in seconds (*Jaybird 2.2.2*). For the Java wire protocol the connect timeout will detect unreachable hosts. In the JNI/JNA implementation (native protocol) the connect timeout works as the DPB item `isc_dpb_connect_timeout` which only works after connecting to the server for the `op_accept` phase of the protocol. This means that – for the native protocol – the connect timeout will not detect unreachable hosts within the timeout. |
| `columnLabelForName` | Jaybird specific property (*Jaybird 2.2.1*). Boolean property. Set property to `true` for backwards compatible behaviour (`getColumnName()` returns the column label). Don't set the property or set it to `false` for JDBC-compliant behaviour (recommended). |
| `useStreamBlobs`<br>`use_stream_blobs` | Jaybird specific property. Boolean property. Tells the driver to create stream BLOBs. By default segmented BLOBs are created. |
| `ignoreProcedureType` | Jaybird specific property (*Jaybird 3.0.6*). Boolean property. Set property to `true` to disable usage of procedure type metadata to decide to use `SELECT` for selectable procedure instead of `EXECUTE PROCEDURE`. See Connection property ignoreProcedureType for more information. |

| Connection property (+ aliases) | Explanation |
|---|---|
| `octetsAsBytes` | Jaybird specific property. (*Jaybird 2.1.1 Jaybird 3*) Boolean property. Unused since Jaybird 3, removed in Jaybird 4. Makes `ResultSet.getObject` for `(VAR)CHAR CHARACTER SET OCTETS` return `byte[]` instead of `String`. Since Jaybird 3, this is the default behaviour (with further enhancements to identify these columns as JDBC `(VAR)BINARY`). See Binary types BINARY/VARBINARY for more information. |
| `decfloatBind` | Instructs Firebird to convert columns and parameters of type `DECFLOAT` to a different type. (*Jaybird 3.0.6 Firebird 4*) Accepted values are: `native`, `char` or `character`, `double precision`, `bigint, n` where `n` is the target scale to convert to a `NUMERIC(18,n)`. See Connection property decfloatBind for more information. |
| `timeZoneBind` | Instructs Firebird to convert columns and parameters of type `TIME WITH TIME ZONE` and `TIMESTAMP WITH TIME ZONE` to a different type. (*Jaybird 3.0.6 Firebird 4*) Accepted values are: `native`, `legacy`. See Time zone bind configuration for more information. |
| `sessionTimeZone` | Configures the session time zone. (*Jaybird 3.0.6*) In Jaybird 3, only configures the Firebird 4 server-side session time zone. In Jaybird 4, also configures the time zone used for legacy datetime conversion on all Firebird versions. See Connection property sessionTimeZone for more information.<br>Default: not set (Jaybird 3 and earlier), or the JVM default time zone (*Jaybird 4*) |
| `useStandarUdf`<br>`use_standard_udf` | Jaybird specific property. Boolean property. Tells the JDBC driver to assume that standard UDF library is registered in the database when converting escaped function calls. With recent versions of Firebird, it is advisable to not specify this property and rely on the built-in functions instead. See Supported JDBC Scalar Functions for more information. |
| `timestampUsesLocalTimezone` | Jaybird specific property. Boolean property. Changes how `getTime`/`getTimestamp` methods accepting a `java.util.Calendar` apply the calendar offset in calculations. **TODO**: Improve documentation on exact effect |
| `num_buffers`<br>`isc_dpb_num_buffers` | Number of database pages that will be cached. Overrides server or database default for this specific connection. Use with care to avoid using an excessive amount of memory. |
| `set_db_readonly`<br>`isc_dpb_set_db_readonly` | Boolean property. Set the database into read-only state. |
| `set_db_sql_dialect`<br>`isc_dpb_set_db_sql_dialect` | Set the SQL dialect of the database. |

| Connection property (+ aliases) | Explanation |
|---|---|
| `set_db_charset`<br>`isc_dpb_set_db_charset` | Set the default character set of the database. |
| `paranoia_mode` | Jaybird specific property. Boolean property. Unused since Jaybird 2.2. Tells the driver to throw exception in situations not covered by the specification. |
| `noResultSetTracking` | Jaybird specific property. Boolean property. Unused since at least Jaybird 2.2, will be removed in Jaybird 4. |
| `useTranslation`<br>`mapping_path` | Jaybird specific property. Deprecated, will be removed in Jaybird 4. This allows mapping of characters to be overridden (see `translation/hpux.properties` in Jaybird jar for example). Value is the path to a mapping properties file. |

In addition, Jaybird allows using arbitrary Database Parameters Block entries as connection properties (provided they are defined in Jaybird's `org.firebirdsql.gds.ISCConstants`). The current Firebird API has almost 90 DPB parameters, however only few of them are interesting for regular users. If a DPB item called `isc_dpb_XXX` exists, then Jaybird allows these to be specified as `isc_dpb_XXX` and `XXX`. By default properties are mapped as string DPB items. If a DPB item requires another type, it will need to be explicitly defined in Jaybird.

## A.3. Transaction isolation levels

It is possible to redefine the transaction isolation levels through connection properties.

| Connection property | Explanation |
|---|---|
| `TRANSACTION_READ_COMMITTED` | Specify the definition of transaction isolation level `READ_COMMITTED`.<br>Default: `isc_tpb_read_committed,isc_tpb_rec_version,isc_tpb_write,isc_tpb_wait` |
| `TRANSACTION_REPEATABLE_READ` | Specify the definition of transaction isolation level `REPEATABLE_READ`.<br>Default: `isc_tpb_concurrency,isc_tpb_write,isc_tpb_wait` |
| `TRANSACTION_SERIALIZABLE` | Specify the definition of transaction isolation level `TRANSACTION_SERIALIZABLE`.<br>Default: `isc_tpb_consistency,isc_tpb_write,isc_tpb_wait` |

For data sources, this feature is exposed using a definition properties file and the `setTpbMapping` property. See Transaction Isolation Levels for more information.

# Appendix B: System properties

Jaybird provides a number of system properties to control global behaviour of Jaybird.

## B.1. Logging

To configure logging, the following system properties are available. See Logging for details.

`org.firebirdsql.jdbc.fallbackConsoleLogger`
: *Jaybird 2.2.8 Jaybird 3* Set to true for fallback to log to console if log4j is not used or not available

`FBLog4j`
: *Jaybird 3* Set to true to attempt to use log4j (if on classpath) for logging

`org.firebirdsql.jdbc.useLog4j`
: *Jaybird 2.2.8 Jaybird 3* Alias for `FBLog4j`

`org.firebirdsql.jdbc.forceConsoleLogger`
: *Jaybird 3* Set to true to force logging to console (`System.out` for info, `System.err` for warn, error and fatal) instead of default `java.util.logging`

`org.firebirdsql.jdbc.disableLogging`
: *Jaybird 3* Set to true to disable logging

`org.firebirdsql.jdbc.loggerImplementation`
: *Jaybird 3* Fully-qualified name of `org.firebirdsql.logging.Logger` implementation to use for logging

These properties need to be set before Jaybird is loaded and used.

## B.2. Process information

For Firebird 2.1 and higher, Jaybird can provide Firebird with process information. This information can be specified in connection properties, or globally using the following system properties.

`org.firebirdsql.jdbc.processName`
: *Jaybird 2.2* Process name to send to Firebird

`org.firebirdsql.jdbc.pid`
: *Jaybird 2.2* PID to send to Firebird (must be a valid integer)

The property values are read for each connect, so the value can be changed at any time.

## B.3. Character set defaults

The following system properties control character set behaviour for connections.

`org.firebirdsql.jdbc.defaultConnectionEncoding`

*Jaybird 3* Firebird character set name to use as connection character set when no explicit connection character set is configured (defaults to `NONE` when not set)

`org.firebirdsql.jdbc.requireConnectionEncoding`

*Jaybird 3.0.2* Set to true to disallow connections without an explicit connection character set. This property will have no effect if `org.firebirdsql.jdbc.defaultConnectionEncoding` has been set.

The property values are read for each connect, so the value can be changed at any time.

## B.4. Other properties

The following system properties control other global behaviour of Jaybird.

`org.firebirdsql.jna.syncWrapNativeLibrary`

*Jaybird 3* Set to true to add a synchronization proxy around the native client library.

`org.firebirdsql.datatypeCoderCacheSize`

*Jaybird 4* Integer value for the number of encoding specific data type coders cached (default and minimum is 1). Setting to a higher value may improve performance, most common use case is connection character set `NONE` with a database that uses more than one character set for its columns. Jaybird will log a warning (*"Cleared encoding specific datatype coder cache [..]"*) when the cache size was exceeded.

`org.firebirdsql.nativeResourceShutdownDisabled`

*Jaybird 4* Set to true to disable automatic shutdown and unload of native libraries and other native resources. Normally you should only use this if the automatic shutdown misbehaves and causes application errors. If you need to set this to true, we'd appreciate it if you post a message to the Firebird-Java list with details on why you needed to enable this, so we can improve or fix this feature.

These properties need to be set before Jaybird is loaded and used. Technically, `org.firebirdsql.jna.syncWrapNativeLibrary` is dynamic, but a native library will usually be loaded once.

## B.5. Useful Java system properties

The following Java system properties are relevant for Jaybird.

`jdk.net.useFastTcpLoopback`

*Firebird 3.0.2 Jave 8 update 60 Windows 8 / Windows Server 2012* Set to true on Windows to enable "TCP Loopback Fast Path" (`SIO_LOOPBACK_FAST_PATH` socket option). "TCP Loopback Fast Path" can improve performance for localhost connections.

Java only has an 'all-or-nothing' support for the "TCP Loopback Fast Path", so Jaybird cannot enable this for you: you must specify this property on JVM startup. This has the benefit that this works for all Jaybird versions, as long as you use Java 8 update 60 or higher (and Firebird 3.0.2 or higher).

# Appendix C: Data Type Conversion Table

## C.1. Mapping between JDBC, Firebird and Java Types

The below table describes a mapping of the JDBC data types defined in `java.sql.Types` class to the Firebird data types. Also, for each JDBC data type a class instance of which is returned by `ResultSet.getObject` method is provided.

| JDBC Type | Firebird Type | Java Object Type |
|---|---|---|
| `CHAR` | `CHAR` | `String` |
| `VARCHAR` | `VARCHAR` | `String` |
| `LONGVARCHAR` | `BLOB SUB_TYPE TEXT` | `String` |
| `NUMERIC` | `NUMERIC` | `java.math.BigDecimal` |
| `DECIMAL` | `DECIMAL` | `java.math.BigDecimal` |
| `SMALLINT` | `SMALLINT` | `java.lang.Short` |
| `INTEGER` | `INTEGER` | `java.lang.Integer` |
| `BIGINT` | `BIGINT` | `java.lang.Long` |
| `REAL` | [26] | `java.lang.Float` |
| `FLOAT` | `FLOAT` | `java.lang.Double` |
| `DOUBLE` | `DOUBLE PRECISION` | `java.lang.Double` |
| `LONGVARBINARY` | `BLOB SUB_TYPE BINARY` | `byte[]` |
| `DATE` | `DATE` | `java.sql.Date` |
| `TIME` | `TIME` | `java.sql.Time` |
| `TIMESTAMP` | `TIMESTAMP` | `java.sql.Timestamp` |
| `BLOB` | `BLOB SUB_TYPE < 0` | `java.sql.Blob` |
| `BOOLEAN` (*Jaybird 2.2*) | `BOOLEAN` (*Firebird 3*) | `java.lang.Boolean` |
| `JaybirdTypeCodes.DECFLOAT` (*Jaybird 4*)[27] | `DECFLOAT` (*Firebird 4*) | `java.math.BigDecimal` |

## C.2. Data Type Conversions

This table specifies the compatible conversions between the Firebird and Java types.

| | SMALLINT | INTEGER | BIGINT | REAL | FLOAT | DOUBLE | DECIMAL | NUMERIC | CHAR | VARCHAR | BLOB SUB_TYPE 1 | BLOB SUB_TYPE 0 | BLOB SUB_TYPE < 0 | DATE | TIME | TIMESTAMP | BOOLEAN | DECFLOAT |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| `String` | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X |
| `BigDecimal` | X | X | X | X | X | X | X | X | X | X | | | | | | | X | X |
| `Boolean` | X | X | X | X | X | X | X | X | X | X | | | | | | | X | X |

| | SMALLINT | INTEGER | BIGINT | REAL | FLOAT | DOUBLE | DECIMAL | NUMERIC | CHAR | VARCHAR | BLOB SUB_TYPE 1 | BLOB SUB_TYPE 0 | BLOB SUB_TYPE < 0 | DATE | TIME | TIMESTAMP | BOOLEAN | DECFLOAT |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Integer | X | X | X | X | X | X | X | X | X | X | | | | | | | X | X |
| Long | X | X | X | X | X | X | X | X | X | X | | | | | | | X | X |
| Float | X | X | X | X | X | X | X | X | X | X | | | | | | | X | X |
| Double | X | X | X | X | X | X | X | X | X | X | | | | | | | X | X |
| byte[] | | | | | | | | | | | X | X | X | | | | | |
| Blob | | | | | | | | | | | X | X | X | | | | | |
| Date | | | | | | | | | | | | | | X | | X | | |
| Time | | | | | | | | | | | | | | | X | | | |
| Timestamp | | | | | | | | | | | | | | X | | X | | |

[26] A Firebird `REAL` is an alias for `FLOAT` and by default handled as `java.sql.Types.FLOAT`

[27] JDBC does not yet define a `java.sql.Types` code for `DECFLOAT`

# Appendix D: Connection Pool Properties

This chapter contains the list of properties of the `ConnectionPoolDataSource`, `DataSource` and `XADataSource` interface implementations.

The documentation in this section is only valid for Jaybird 2.2 and earlier.

Connection pooling was removed from Jaybird 3.0. Either use the connection pool provided by your application server, or use a third-party connection pool like c3p0 [https://www.mchange.com/projects/c3p0/], Apache DBCP [https://commons.apache.org/proper/commons-dbcp/] or HikariCP [https://brettwooldridge.github.io/HikariCP/].

## D.1. Standard JDBC Properties

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

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

## D.2. Pool Properties

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

| Property | Description |
|---|---|
| `blockingTimeout` | Maximum time in milliseconds during which application can be blocked waiting for a connection from the pool. If no free connection can be obtained, an exception is thrown. |
| `retryInterval` | Period in which the pool will try to obtain a new connection while blocking the application. |
| `pooling` | Allows to switch connection pooling off. |

| Property | Description |
|---|---|
| `statementPooling` | Allows to switch statement pooling off. |
| `pingStatement` | Statement that will be used to "ping" the JDBC connection, in other words, to check if it is still alive. This statement must always succeed. he default SQL statement for the Firebird database is `"SELECT CAST(1 AS INTEGER) FROM rdb$database"`. |
| `pingInterval` | Time during which connection is believed to be valid in any case. The pool "pings" the connection before giving it to the application only if more than specified amount of time passed since last "ping". |
| `isolation` | Default transaction isolation level. All connections returned from the pool will have this isolation level. One of:<br>• `TRANSACTION_READ_COMMITTED`<br>• `TRANSACTION_REPEATABLE_READ`<br>• `TRANSACTION_SERIALIZABLE` |
| `transactionIsolationLevel` | Integer value from `java.sql.Connection` interface corresponding to the transaction isolation level specified in isolation property. |

## D.3. Runtime Pool Properties

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

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

## D.4. Firebird-specific Properties

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

| Property | Description |
|---|---|
| `database` | Path to the database in the format `[host/port:]/path/to/database.fdb` |

| Property | Description |
|---|---|
| `type` | Type of the driver to use. Possible values are:<br>• `PURE_JAVA` or `TYPE4` for type 4 JDBC driver<br>• `NATIVE` or `TYPE2` for type 2 JDBC driver<br>• `EMBEDDED` for using embedded version of the Firebird<br>• `ORACLE` for accessing Oracle-mode Firebird |
| `blobBufferSize` | Size of the buffer used to transfer BLOB content. Maximum value is 64k-1. |
| `socketBufferSize` | Size of the socket buffer. Needed on some Linux machines to fix performance degradation. |
| `charSet` | Character set for the connection. Similar to `encoding` property, but accepts Java names instead of Firebird ones. |
| `encoding` | Character encoding for the connection. See Firebird documentation for more information. |
| `userName` | Name of the user that will be used by default. |
| `password` | Corresponding password. |
| `roleName` | SQL role to use. |
| `tpbMapping` | TPB mapping for different transaction isolation modes. |

## D.5. Non-standard parameters

Many of the above mentioned Firebird parameters have have a corresponding DPB entry. However, not every DPB entry has a corresponding getter/setter. This was done intentionally, Firebird provides almost 70 DPB parameters, but for most of the applications only few are needed. The remaining parameters are used by specialized applications (usually server or database management software) for setting some default values of the database, controlling the cache buffers on the server, etc. Creating a corresponding getter/setter for each of them simply does not make sense.

For those Java applications that still need non-standard connectivity parameters, `DataSource` and `ConnectionPoolDataSource` implementations provides a getter and two setters:

- `getNonStandardProperty(String name)` method returns a non-standard property specified by `name` parameter. If this property was not previously set, `null` is returned.
- `setNonStandardProperty(String name, String value)` method sets the property specified by the first parameter to a value contained in the second parameter.
- `setNonStandardProperty(String nameValuePair)` method provides a possibility to set a DPB parameter using following syntax:

```
dataSource.setNonStandardProperty("isc_dpb_sql_dialect=3");
```

The parameter syntax of the last method is not very common in Java code, it would be much more natural to use two-parameter setter. However, it has a specialized use, because there's no possibility to use two-parameter setter method in configuration files. Usually, when setting a configuration parameter of a data source, web-containers use the Java reflection API and consider only those setters that take one parameter. For instance, in the Tomcat server the configuration parameter would look like this:

```
<parameter>
  <name>nonStandardProperty</name>
  <value>sql_dialect=3</value>
</parameter>
```

Syntax of the parameter is

```
<name>[<whitespace>][{=|:|<whitespace>}[<whitespace>]<value>]
```

where `<name>` is the name of the DPB parameter, and `<value>` is its value. The two are separated by any combination of whitespace and either whitespace or "=" (equal sign) or ":" (colon) characters. Considering the aliases described in Extended connection properties. For example following values are equivalent:

```
isc_dpb_sql_dialect   3
isc_dpb_sql_dialect : 3
sql_dialect         : 3
sql_dialect=3
```

# Appendix E: Character Encodings

Character encodings and their correct use with Firebird from Java is an important topic, that initially seems to be complex, but in fact can be formulated by just a few rules. This appendix provides information on this topic.

## E.1. Encodings Types

Firebird uses character encodings in two different areas:

- The database encoding defines the character set in which `CHAR`, `VARCHAR` and `BLOB SUB_TYPE TEXT` fields are physically stored on the disk. There is a default database encoding that is specified during database creation. It is also possible to specify character sets on a per column basis.
- The client connection encoding defines a character set in which client will send and expects to receive character data. This encoding might or might not match the database default encoding.

Firebird performs translation between character sets of the client connection and the character set of the content. The list of allowed character sets as well as the allowed translations between them are specified in the `fbintl` shared library located in the `intl/` directory of the Firebird installation.[28] There is also a special character set NONE that tells Firebird not to interpret the contents of the character field.

Firebird uses the following algorithm when performing translations:

- If source and target character sets match, send the content unchanged.
- If the target character set is `NONE`, send source content unchanged.
- If the source character set is `NONE`, send source content unchanged.
- If there is a direct translation rule between source and target character sets, use that rule.
- If there is no direct translation rule, check if there is rule to translate the source character set into the UTF8 character set and a rule to translate from `UTF8` into the target character set. If yes, use these two rules for translation.
- If no suitable translation rule can be found, throw an exception.

## E.2. Encodings in Java

The Java programming language is based on the Unicode character set and uses the UTF-16 encoding, in which each character is represented by one or two 16-bit units. Firebird, on the other side, is not based on Unicode and allows different character sets to be assigned to different database objects. Additionally, Firebird requires a connection character set to be specified in connection options, which forces Firebird to convert data from the character set of the database object into the character set of the client application.

There are two boundary cases that we will consider here, one when Firebird database was created with default[29] character set `UTF8`, another when the Firebird database was created without specifying the character set (i.e. character set `NONE`).

The character set UTF8 in Firebird 2.0 and higher is a Unicode character set that uses UTF-8 encoding and occupies from one to four 8-bit units. Firebird has supported Unicode character set for a long time, however its implementation was deficient in Firebird 1.5 and earleir – it did not support proper uppercasing and correct sorting. These issues were addressed in Firebird 2.0 and at the moment nothing prevents developers from using Unicode in the database and on the client side, which greatly simplifies the internationalization and localization of the applications.

### E.2.1. The UTF8 character set

A developer must ensure two things to enable use of Unicode characters in the database and the application:

1. The database objects must be defined with the UTF8 character set; this can be done by either creating database with default UTF8 character set or by adding CHARACTER SET UTF8 clause to the column or domain definitions.
2. The encoding connection property in the JDBC driver has to be set to UTF8; this can be done in several ways: the easiest one is to add the appropriate parameter to the JDBC URL (see the first example), another possibility is to use appropriate method of the DriverManager class (see the second example). Applications that use DataSource interface to obtain the database connections also have access to the encoding property.[30]

*Specifying the connection encoding in JDBC URL*

```
Connection connection = DriverManager.getConnection(
    "jdbc:firebirdsql:localhost/3050:employee?encoding=UTF8",
    "SYSDBA", "masterkey");
```

*Specifying connection encoding in the connection properties*

```
Properties props = new Properties();

props.setProperty("user", "SYSDBA");
props.setProperty("password", "masterkey");
props.setProperty("encoding", "UTF8");

Connection connection = DriverManager.getConnection(
    "jdbc:firebirdsql:localhost/3050:employee", props);
```

There are a few limitations related to using the UTF8 character set:

- It is not possible to create Unicode columns longer than 8191 Unicode characters; this limitation is caused by the fact that the longest possible VARCHAR column can occupy 32765 bytes (32767 for CHAR columns) and a single UTF8 character can occupy up to four bytes.
- It is not possible to index Unicode columns longer than 1023 characters;[31] this limitation is caused by the fact that the longest index key cannot be longer than a quarter of the database page, which in Firebird 2.0 and higher can be maximum 16k and the before mentioned fact that each UTF8 character can occupy up to four bytes.

It should be mentioned that using Unicode character set might cause noticeable performance degradation when the database is accessed over wide-area networks. This mainly applies to the cases when non-latin characters are stored in the database, as those characters will require two or more bytes, which in turn might cause additional roudtrips to the server to fetch data.

### E.2.2. The NONE character set

Java introduces additional complexity when the `NONE` character set is used. The reason for this is that Java internally stores all strings in Unicode format, and the application must define the character encoding for the byte contents to the JVM. When the `NONE` character set is used, Jaybird does not know how to interpret the received data. The only choice that is left to Jaybird is to construct a string using the default character set of the JVM, which usually matches the regional settings of the operating system and can be accessed from within the JVM through the `file.encoding` system property.

(*Jaybird 3*) Starting with Jaybird 3, with connection character set `NONE`, Jaybird will use the explicit character set of `CHAR`, `VARCHAR` and `BLOB SUB_TYPE TEXT` columns for the conversion. This addresses most of the problems described in this paragraph, except for columns without an explicit character set (ie their character set is `NONE`).

It is clear that a conversion using default character set that happens inside the JVM can lead to errors when the same content is accessed from two or more different Java Virtual Machines that have different configuration. One application running on the computer with, for example, Russian regional settings saves the Russian text (the default character set of the JVM is Cp1251) and another application running on computer with German regional settings (default character set is Cp1252) will read in such case some special or accented characters. However, when all client applications run same OS with the same regional settings in most cases will not have any severe consequences (except probably wrong sorting order or uppercasing on the server side).

On Linux and other Unix platforms it might have more severe consequences as it is very common that regional settings are not configured and that the default "C" locale is used and the non-ASCII characters will be replaced with question marks ("?").

Therefore, application should use `NONE` character encoding as an encoding for a database and a connection only when at least one of the following is met:

- Database will contain only ASCII characters,
- It is guaranteed that all Java Virtual Machines accessing the database will have the same default encoding that can correctly handle all characters stored in the database,
- (*Jaybird 3*) All columns have an explicit character set. When columns have an explicit character set (other than `NONE`) and connection character set `NONE` is used, Firebird will send an identifier of the character set of each column, and Jaybird will use that character set for the conversion.

As a partial workaround, you can specify the encoding that should be used to interpret bytes coming from the server in the `charSet` connection property. The following rules are used when interpreting the `encoding` and `charSet` properties:

- When only `encoding` property specified, Jaybird uses the default mapping between server and Java encodings. When `encoding` property is not set or set to `NONE` and `charSet` property is not set,

the default JVM encoding is used to interpret bytes coming from the server.

- When only `charSet` property is specified, Jaybird uses the reverse mapping to specify the connection encoding for the server and interprets byte stream according to the value of the property.
- When both `encoding` and `charSet` property are specified, Jaybird sets the connection encoding according to the value of the `encoding` property, but interprets the byte stream according to the `charSet` property. (*Jaybird 3*) With Jaybird 3 and higher, this option has limitations when `encoding=NONE`: the conversion using `charSet` will only be applied for columns that don't have an explicit character set, otherwise that explicit character set is used for the conversion.

The last case is most powerful, but also is the most dangerous in use. When used properly, it can solve the problems with the legacy databases; when used incorrectly, one can easily trash the content of the database.

## E.3. Available Encodings

The below table lists the available character encodings in the default Firebird distribution and their mapping to the Java ones:

| **Firebird encoding (`encoding` property)** | **Java encoding (`charSet` property)** | **Size in bytes** | **Comments** |
|---|---|---|---|
| NONE | - | 1 | Raw bytes, no interpretation of the content is possible. |
| ASCII | ASCII | 1 | - |
| BIG_5 | Big5 | 2 | Traditional Chinese |
| DOS437 | Cp437 | 1 | MS-DOS: United States, Australia, New Zeland, South Africa |
| DOS737 | Cp737 | 1 | MS-DOS: Greek |
| DOS775 | Cp775 | 1 | MS-DOS: Baltic |
| DOS850 | Cp850 | 1 | MS-DOS: Latin-1 |
| DOS852 | Cp852 | 1 | MS-DOS: Latin-2 |
| DOS857 | Cp857 | 1 | IBM: Turkish |
| DOS858 | Cp858 | 1 | IBM: Latin-1 + Euro |
| DOS860 | Cp860 | 1 | MS-DOS: Portuguese |
| DOS861 | Cp861 | 1 | MS-DOS: Icelandic |
| DOS862 | Cp862 | 1 | IBM: Hebrew |
| DOS863 | Cp863 | 1 | MS-DOS: Canadian French |
| DOS864 | Cp864 | 1 | IBM: Arabic |
| DOS865 | Cp865 | 1 | MS-DOS: Nordic |

| Firebird encoding (`encoding` property) | Java encoding (`charSet` property) | Size in bytes | Comments |
|---|---|---|---|
| DOS866 | Cp866 | 1 | IBM: Cyrillic |
| DOS869 | Cp869 | 1 | IBM: Modern Greek |
| EUCJ_0208 | EUC_JP | 2 | JIS X 0201, 0208, 0212, EUC encoding, Japanese |
| GB_2312 | EUC_CN | 2 | GB2312, EUC encoding, Simplified Chinese |
| ISO8859_1 | ISO-8859-1 | 1 | ISO 8859-1, Latin alphabet No. 1 |
| ISO8859_2 | ISO-8859-2 | 1 | ISO 8859-2 |
| ISO8859_3 | ISO-8859-3 | 1 | ISO 8859-3 |
| ISO8859_4 | ISO-8859-4 | 1 | ISO 8859-4 |
| ISO8859_5 | ISO-8859-5 | 1 | ISO 8859-5 |
| ISO8859_6 | ISO-8859-6 | 1 | ISO 8859-6 |
| ISO8859_7 | ISO-8859-7 | 1 | ISO 8859-7 |
| ISO8859_8 | ISO-8859-8 | 1 | ISO 8859-8 |
| ISO8859_9 | ISO-8859-9 | 1 | ISO 8859-9 |
| ISO8859_13 | ISO-8859-13 | 1 | ISO 8859-13 |
| KSC_5601 | MS949 | 2 | Windows Korean |
| UNICODE_FSS | UTF-8 | 3 | 8-bit Unicode Transformation Format (deprecated since FB 2.0) |
| UTF8 | UTF-8 | 4 | 8-bit Unicode Transformation Format (FB 2.0+) |
| WIN1250 | Cp1250 | 1 | Windows Eastern European |
| WIN1251 | Cp1251 | 1 | Windows Cyrillic |
| WIN1252 | Cp1252 | 1 | Windows Latin-1 |
| WIN1253 | Cp1253 | 1 | Windows Greek |
| WIN1254 | Cp1254 | 1 | Windows Turkish |
| WIN1255 | Cp1255 | 1 | - |
| WIN1256 | Cp1256 | 1 | - |
| WIN1257 | Cp1257 | 1 | - |

[28] On Windows this library is represented by `fbintl.dll`, on Linux – `libfbintl.so`

[29] The default character set simplifies the explanation, since we do not have to consider the cases when different columns with different character sets are used within the same connection. The statements made here, obviously, can be applied to those cases as well.

[30] See https://github.com/FirebirdSQL/jaybird/wiki for configuration examples of the most popular application servers.

[31] 2047 characters in Firebird 4 with a page size of 32 kilobytes

# Appendix F: Supported JDBC Scalar Functions

The JDBC API has an escaped syntax for numeric, string, time, date, system and conversion functions. Jaybird will try to provide an equivalent of the JDBC function using the built-in capabilities of the Firebird database. When no equivalent is available, Jaybird will pass the function call "as is" to the database assuming that it contains the necessary UDF, UDR or stored function declaration.

Not all functions described in the JDBC specification have corresponding built-in functions in Firebird, but some are available in the standard UDF library `ib_udf`[32] shipped with Firebird. Jaybird provides a connection parameter `use_standard_udf` to configure the driver to assume that functions from that UDF are available in the database. In this case Jaybird will convert all JDBC function calls into the corresponding calls of the UDF functions.

(*Jaybird 3*) In recent Firebird versions, the number of built-in functions has been greatly increased, and Jaybird 3 and higher can now map almost all JDBC escapes to those built-in functions. Using the `use_standard_udf` is no longer advisable, especially as UDFs are now deprecated and will be removed in a future Firebird version.

Below you will find the list of JDBC functions and whether they have a corresponding equivalent in the "built-in" and in the "UDF" modes.

## F.1. Numeric Functions

| JDBC | built-in | UDF mode | Description |
|---|---|---|---|
| `ABS(number)` | X | X | Absolute value of `number` |
| `ACOS(float)` | X | X | Arccosine, in radians, of `float` |
| `ASIN(float)` | X | X | Arcsine, in radians, of `float` |
| `ATAN(float)` | X | X | Arctangent, in radians, of `float` |
| `ATAN2(float1, float2)` | X | X | Arctangent, in radians, of `float2` / `float1` |
| `CEILING(number)` | X | X | Smallest integer >= `number` |
| `COS(float)` | X | X | Cosine of `float` radians |
| `COT(float)` | X | X | Cotangent of `float` radians |
| `DEGREES(number)` | X | | Degrees in `number` radians (*Jaybird 4*) |
| `EXP(float)` | X | | Exponential function of `float` |
| `FLOOR(number)` | X | X | Largest integer ⇐ `number` |
| `LOG(float)` | X | X | Base e logarithm of `float` |
| `LOG10(float)` | X | X | Base 10 logarithm of `float` |
| `MOD(integer1, integer2)` | X | X | Remainder for `integer1` / `integer2` |

| JDBC | built-in | UDF mode | Description |
|---|---|---|---|
| `PI()` | X | X | The constant pi |
| `POWER(number, power)` | X | | `number` raised to (integer) `power` |
| `RADIANS(number)` | X | | Radians in `number` degrees (*Jaybird 4*) |
| `RAND(integer)` | | X[33] | Random floating point for seed `integer` |
| `ROUND(number, places)` | X | | `number` rounded to `places` places |
| `SIGN(number)` | X | X | -1 to indicate `number` is < 0; 0 to indicate `number` is = 0; 1 to indicate `number` is > 0 |
| `SIN(float)` | X | X | Sine of `float` radians |
| `SQRT(float)` | X | X | Square root of float |
| `TAN(float)` | X | X | Tangent of `float` radians |
| `TRUNCATE(number, places)` | X | | `number` truncated to `places` places (*Jaybird 3*) |

Legend: X – available in this mode.

# F.2. String Functions

| JDBC | built-in | UDF mode | Description |
|---|---|---|---|
| `ASCII(string)` | X | X | Integer representing the ASCII code value of the leftmost character in `string` |
| `CHAR(code)` | X | X | Character with ASCII code value `code`, where `code` is between 0 and 255 |
| `CHAR_LENGTH(string [,CHARACTERS])` | X[34] | | Returns the length in characters of the string expression |
| `CHAR_LENGTH(string, OCTETS)` | X | | Returns the length in bytes of the string expression whose result is the smallest integer not less than the number of bits divided by 8, alias for `OCTET_LENGTH` (*Jaybird 4*) |
| `CHARACTER_LENGTH( string [,CHARACTERS])` | X[35] | | Alias for `CHAR_LENGTH` (*Jaybird 3*) |
| `CHARACTER_LENGTH(string, OCTETS)` | X | | Alias for `CHAR_LENGTH`, `OCTET_LENGTH` (*Jaybird 4*) |
| `CONCAT(string1, string2)` | X | X | Character string formed by appending `string2` to `string1` |
| `DIFFERENCE(string1, string2)` | | | Integer indicating the difference between the values returned by the function `SOUNDEX` for `string1` and `string2` |

| JDBC | built-in | UDF mode | Description |
|---|---|---|---|
| `INSERT(string1, start, length, string2)` | X | | A character string formed by deleting `length` characters from `string1` beginning at `start`, and inserting `string2` into `string1` at `start` (*Jaybird 3*) |
| `LCASE(string)` | X | X | Converts all uppercase characters in `string` to lowercase (*Jaybird 3*) |
| `LEFT(string, count)` | X | X | The `count` leftmost characters from `string` |
| `LENGTH(string [,CHARACTERS])` | X[36] | X[37] | Number of characters in `string`, excluding trailing blanks (built-in *Jaybird 3*) |
| `LENGTH(string, OCTETS])` | X[38] | | Number of characters in `string`, excluding trailing blanks (*Jaybird 4*) |
| `LOCATE(string1, string2 [,start])` | X[39] | | Position in `string2` of the first occurrence of `string1`, searching from the beginning of `string2`; if `start` is specified, the search begins from position `start`. 0 is returned if `string2` does not contain `string1`. Position 1 is the first character in `string2` (*Jaybird 3* with `start` required, *Jaybird 4* with `start` optional) |
| `LTRIM(string)` | X | X | Characters of `string` with leading blank spaces removed (*Jaybird 3*) |
| `OCTET_LENGTH(string)` | X | | Returns the length in bytes of the string expression whose result is the smallest integer not less than the number of bits divided by 8 |
| `POSITION(substring IN string [,CHARACTERS])` | X[40] | | Returns the position of first occurrence of `substr` in `string` (*Jaybird 3* without `CHARACTERS`, *Jaybird 4* with optional `CHARACTERS`) |
| `POSITION(substring IN string, OCTETS)` | X[41] | | Returns the position of first occurrence of `substr` in `string` (*Jaybird 4* with caveat) |
| `REPEAT(string, count)` | X | | A character string formed by repeating `string` `count` times (*Jaybird 3*) |
| `REPLACE(string1, string2, string3)` | X | | Replaces all occurrences of `string2` in `string1` with `string3` |
| `RIGHT(string, count)` | X | | The `count` rightmost characters in `string` |
| `RTRIM(string)` | X | X | The characters of `string` with no trailing blanks (*Jaybird 3*) |
| `SOUNDEX(string)` | | | A character string, which is data source-dependent, representing the sound of the words in `string`; this could be a four-digit SOUNDEX code, a phonetic representation of each word, etc |

| JDBC | built-in | UDF mode | Description |
|---|---|---|---|
| `SPACE(count)` | X | | A character string consisting of `count` spaces (*Jaybird 3*) |
| `SUBSTRING(string, start, length)` | X | X | A character string formed by extracting `length` characters from `string` beginning at `start` |
| `UCASE(string)` | X | X | Converts all lowercase characters in `string` to uppercase |

Legend: X – available in this mode.

## F.3. Time and Date Functions

| JDBC | built-in | UDF mode | Description |
|---|---|---|---|
| `CURRENT_DATE[()]` | X | | Synonym for `CURDATE()` |
| `CURRENT_TIME[()]` | X | | Synonym for `CURTIME()` |
| `CURRENT_TIMESTAMP[()]` | X | | Synonym for `NOW()` |
| `CURDATE()` | X | X | The current date as a date value |
| `CURTIME()` | X | X | The current local time as a time value |
| `DAYNAME(date)` | | | A character string representing the day component of `date`; the name for the day is specific to the data source |
| `DAYOFMONTH(date)` | X | X | An integer from 1 to 31 representing the day of the month in `date` |
| `DAYOFWEEK(date)` | X | | An integer from 1 to 7 representing the day of the week in `date`; 1 represents Sunday (*Jaybird 3*) |
| `DAYOFYEAR(date)` | X | | An integer from 1 to 366 representing the day of the year in `date` (*Jaybird 3*) |
| `EXTRACT(field FROM source)` | X | | Extract the field portion from the source. The source is a datetime value. The value for field may be one of the following: `YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, `SECOND` |
| `HOUR(time)` | X | X | An integer from 0 to 23 representing the hour component of `time` |
| `MINUTE(time)` | X | X | An integer from 0 to 59 representing the minute component of `time` |
| `MONTH(date)` | X | X | An integer from 1 to 12 representing the month component of `date` |

| JDBC | built-in | UDF mode | Description |
|---|---|---|---|
| `MONTHNAME(date)` | | | A character string representing the month component of `date`; the name for the month is specific to the data source |
| `NOW()` | X | X | A timestamp value representing the current date and time |
| `QUARTER(date)` | X | | An integer from 1 to 4 representing the quarter in `date`; 1 represents January 1 through March 31 (*Jaybird 4*) |
| `SECOND(time)` | X | X | An integer from 0 to 59 representing the second component of `time` |
| `TIMESTAMPADD( interval, count, timestamp)` | X | | A timestamp calculated by adding `count` number of `interval`(s) to `timestamp` (*Jaybird 4*) |
| `TIMESTAMPDIFF( interval, timestamp1, timestamp2)` | X | | An integer representing the number of `interval` by which `timestamp2` is greater than `timestamp1` (*Jaybird 4*) |
| `WEEK(date)` | X | X | An integer from 1 to 53 representing the week of the year in `date` |
| `YEAR(date)` | X | X | An integer representing the year component of `date` |

Legend: X – available in this mode.

## F.4. System Functions

| JDBC | built-in | UDF mode | Description |
|---|---|---|---|
| `DATABASE()` | | | Name of the database |
| `IFNULL(expression, value)` | X | X | `value` if `expression` is null; `expression` if `expression` is not null |
| `USER()` | X | | User name in the DBMS (*Jaybird 3*) |

Legend: X – available in this mode.

## F.5. Conversion Functions

| JDBC | built-in | UDF mode | Description |
|---|---|---|---|
| `CONVERT(value, SQLtype)` | X | X | `value` converted to `SQLtype` where `SQLtype` may be one of the following SQL types:<br>• `BIGINT`<br>• `BINARY` (*Jaybird 4*)<br>• `BLOB` (*Jaybird 4*)<br>• `CHAR`<br>• `CLOB` (*Jaybird 4*)<br>• `DATE`<br>• `DECFLOAT` (*Jaybird 4*)<br>• `DECIMAL`<br>• `DOUBLE` (*Jaybird 4*)<br>• `DOUBLE PRECISION`<br>• `FLOAT`<br>• `INTEGER`<br>• `LONGNVARCHAR` (*Jaybird 4*)<br>• `LONGVARBINARY` (*Jaybird 4*)<br>• `LONGVARCHAR` (*Jaybird 4*)<br>• `NCHAR` (*Jaybird 4*)<br>• `NCLOB` (*Jaybird 4*)<br>• `NVARCHAR` (*Jaybird 4*)<br>• `REAL`<br>• `SMALLINT`<br>• `TIME`<br>• `TIME_WITH_TIMEZONE` (*Jaybird 4*)<br>• `TIME_WITH_TIME_ZONE` (*Jaybird 4*)<br>• `TIMESTAMP`<br>• `TIMESTAMP_WITH_TIMEZONE` (*Jaybird 4*)<br>• `TIMESTAMP_WITH_TIME_ZONE` (*Jaybird 4*)<br>• `TINYINT` (*Jaybird 4*)<br>• `VARBINARY` (*Jaybird 4*)<br>• `VARCHAR`<br>Since Jaybird 4, these type names can also be prefixed with `SQL_`. |

Legend: X – available in this mode.

(*Jaybird 4*) The following improvements where added to `CONVERT` support in Jaybird 4:

- Both the `SQL_<datatype>` and `<datatype>` mapping is now supported
- Contrary to the JDBC specification, we allow explicit length or precision and scale parameters
- `(SQL_)VARCHAR`, `(SQL_)NVARCHAR` (and *value* not a parameter (`?`)) without explicit length is converted using `TRIM(TRAILING FROM value)`, which means the result is `VARCHAR` except for blobs where this will result in a blob; national character set will be lost. If *value* is a parameter (`?`), and no length is specified, then a length of 50 will be applied (cast to `(N)VARCHAR(50)`).
- `(SQL_)CHAR`, `(SQL_)NCHAR` without explicit length will be cast to `(N)CHAR(50)`
- `(SQL_)BINARY`, and `(SQL_)VARBINARY` without explicit length will be cast to `(VAR)CHAR(50) CHARACTER SET OCTETS`. With explicit length, `CHARACTER SET OCTETS` is appended.
- `(SQL_)LONGVARCHAR`, `(SQL_)LONGNVARCHAR`, `(SQL_)CLOB`, `(SQL_)NCLOB` will be cast to `BLOB SUB_TYPE TEXT`, national character set will be lost
- `(SQL_)LONGVARBINARY`, `(SQL_)BLOB` will be cast to `BLOB SUB_TYPE BINARY`
- `(SQL_)TINYINT` is mapped to `SMALLINT`
- `(SQL_)ROWID` is not supported as length of `DB_KEY` values depend on the context
- `(SQL_)DECIMAL` and `(SQL_)NUMERIC` without precision and scale are passed as is, in current Firebird versions, this means the value will be equivalent to `DECIMAL(9,0)` (which is equivalent to `INTEGER`)
- Unsupported/unknown *SQLtype* values (or invalid length or precision and scale) are passed as is to cast, resulting in an error from the Firebird engine if the resulting cast is invalid

[32] On Windows platform it is represented by the `ib_udf.dll`, on Linux it is represented by the `libib_udf.so`.

[33] Maps to UDF `RAND()` taking no parameters. The random number generator is seeded by the current time. There is no function where the seed can be specified.

[34] Second parameter is ignored in Jaybird 3 and earlier, supported in Jaybird 4 and higher

[35] Second parameter ignored in Jaybird 3 and earlier, supported in Jaybird 4 and higher

[36] In Jaybird 3, the second parameter is ignored, in Jaybird 4 the `CHARACTERS` parameter only determines that characters are counted, the ignored blanks (space (0x20) or NUL (0x00)) are not determined by the parameter but by the underlying type

[37] The trailing blanks are also counted, only works if second parameter is omitted

[38] The `OCTETS` parameter only determines that bytes are counted, the ignored blanks (space (0x20) or NUL (0x00)) are not determined by the parameter but by the underlying type

[39] In Jaybird 3, start is required, start is optional since Jaybird 4

[40] In Jaybird 3 and earlier only supported without the `CHARACTERS` parameter

[41] Parameter `OCTETS` is ignored

# Appendix G: Jaybird versions

This appendix lists the distribution files and supported specifications of recent Jaybird versions.

## G.1. Jaybird 4

Jaybird 4 is not yet released

### G.1.1. Java support

Jaybird 4 supports Java 7 (JDBC 4.1), Java 8 (JDBC 4.2), and Java 9 and higher (JDBC 4.3).

Given the limited support period for Java 9 and higher versions, we will limit support on those versions to the most recent LTS version and the latest release. Currently that means we support Java 11 and Java 12.

Jaybird 4 provides libraries for Java 7, Java 8 and Java 11. The Java 8 builds have the same source and all JDBC 4.3 related functionality and can be used on Java 9 and higher as well.

Jaybird 4 is not modularized, but all versions declare the automatic module name `org.firebirdsql.jaybird`.

### G.1.2. Firebird support

Jaybird 4 supports Firebird 2.5 and higher.

Formal support for Firebird 2.0 and 2.1 has been dropped (although in general we expect the driver to work). The Type 2 and embedded server JDBC drivers use JNA to access the Firebird client or embedded library.

#### Notes on Firebird 3 support

Jaybird 4 does not (yet) support the Firebird 3 zlib compression.

#### Notes on Firebird 4 support

Jaybird 4 does not support the protocol improvements of Firebird 4 like statement and session timeouts. Nor does it implement the new batch protocol.

Jaybird time zone support uses functionality added after Firebird 4 beta 1 (4.0.0.1436), you will need version 4.0.0.1481 or later for the `timeZoneBind` connection property.

### G.1.3. Supported Specifications

Jaybird supports the following specifications:

| Specification | Details |
|---|---|
| JDBC 4.3 | Jaybird supports most of JDBC 4.3, inasfar the features are required or supported by Firebird. It is not officially JDBC compliant, because we currently don't have access to the TCK. |
| JCA 1.5 | Jaybird provides an implementation of `javax.resource.spi.ManagedConnectionFactory` and related interfaces. CCI interfaces are not supported. |
| JTA | The driver provides an implementation of the `javax.transaction.xa.XAResource` interface via the JCA framework and a `javax.sql.XADataSource` implementation. |

## G.1.4. Distribution

The Jaybird driver has compile-time and run-time dependencies to JCA 1.5. Additionally, if the antlr-runtime classes are found in the class path, it is possible to use generated key retrieval.

### Distribution package

The latest version of Jaybird can be downloaded from https://firebirdsql.org/en/jdbc-driver/

The following files can be found in the distribution package:

| File name | Description |
|---|---|
| `jaybird-<java>-4.0.0-beta-1.jar` | An archive containing the JDBC driver, the JCA connection manager, the Services API and event management classes. Where `<java>` is either `jdk17`, `jdk18` or `java11`. |
| `jaybird-full-<java>-4.0.0-beta-1.jar` | Same as above, but including the JCA 1.5 dependency. Where `<java>` is either `jdk17`, `jdk18` or `java11`. |
| `lib/antlr-runtime-4.7.2.jar` | Optional dependency, required if you want to use `getGeneratedKeys` support |
| `lib/connector-api-1.5.jar` | Required dependency; part of `jaybird-full`, not necessary when deploying to a Java EE application server |
| `lib/jna-5.3.0.jar` | Optional dependency, required if you want to use Type 2 native, local or embedded protocols |

### Maven

Alternatively, you can use maven to automatically download Jaybird and its dependencies.

Jaybird 4 is available from Maven central:

Groupid: `org.firebirdsql.jdbc`,
Artifactid: `jaybird-XX` (where `XX` is `jdk17`, `jdk18` or `java11`).
Version: `4.0.0-beta-1`

For example:

```
<dependency>
  <groupId>org.firebirdsql.jdbc</groupId>
  <artifactId>jaybird-jdk18</artifactId>
  <version>4.0.0-beta-1</version>
</dependency>
```

The Maven definition of Jaybird depends on antlr-runtime by default.

If your application is deployed to a Java EE application server, you will need to exclude the `javax.resource:connector-api` dependency, and add it as a provided dependency:

```
<dependency>
  <groupId>org.firebirdsql.jdbc</groupId>
  <artifactId>jaybird-jdk18</artifactId>
  <version>4.0.0-beta-1</version>
  <exclusions>
    <exclusion>
      <groupId>javax.resource</groupId>
      <artifactId>connector-api</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>javax.resource</groupId>
  <artifactId>connector-api</artifactId>
  <version>1.5</version>
  <scope>provided</scope>
</dependency>
```

If you want to use Type 2 support (native, local or embedded), you need to explicitly include JNA as a dependency:

```
<dependency>
  <groupId>net.java.dev.jna</groupId>
  <artifactId>jna</artifactId>
  <version>5.3.0</version>
</dependency>
```

For native and local you can use the `org.firebirdsql.jdbc:fbclient` dependency to provide the client library. See Maven dependency for native client for details.

# G.2. Jaybird 3

## G.2.1. Java support

Jaybird 3 supports Java 7 (JDBC 4.1), Java 8 (JDBC 4.2), and Java 9 - 11 (JDBC 4.3).

There are no Java 9+ specific builds, the Java 8 builds have the same source and all JDBC 4.3 related functionality.

Given the limited support period for Java 9 and higher versions, we may limit support on those versions to the most recent LTS version and the latest release.

Jaybird 3.0 is not modularized, but since Jaybird 3.0.3, it declares the automatic module name `org.firebirdsql.jaybird`.

## G.2.2. Firebird support

Jaybird 3 supports Firebird 2.0 and higher, but is only tested with Firebird 2.5, 3.0 and 4.0.

Formal support for Firebird 1.x has been dropped (although in general we expect the driver to work). The Type 2 and embedded server JDBC drivers use JNA to access the Firebird client or embedded library.

#### Notes on Firebird 3 support

Jaybird 3.0.4 added support for wire protocol encryption and database encryption.

Jaybird 3.0 does not support the Firebird 3 zlib compression.

#### Notes on Firebird 4 support

Jaybird 3.0 can connect and query Firebird 4. Longer object names are supported.

The new data types introduced in Firebird 4 are not supported. Support for data types like `DECFLOAT` and `NUMERIC`/`DECIMAL` with precision higher than 18 will be introduced in Jaybird 4.

The Srp256 authentication plugin is supported (*Jaybird 3.0.5*), but the other SrpNNN plugins are not.

Jaybird 3 does not support the Firebird 4 zlib compression.

## G.2.3. Supported Specifications

Jaybird supports the following specifications:

| Specification | Details |
|---|---|
| JDBC 4.3 | Jaybird supports most of JDBC 4.3, inasfar the features are required or supported by Firebird. It is not officially JDBC compliant, because we currently don't have access to the TCK. |
| JCA 1.5 | Jaybird provides an implementation of `javax.resource.spi.ManagedConnectionFactory` and related interfaces. CCI interfaces are not supported. |
| JTA | The driver provides an implementation of the `javax.transaction.xa.XAResource` interface via the JCA framework and a `javax.sql.XADataSource` implementation. |

## G.2.4. Distribution

The Jaybird driver has compile-time and run-time dependencies to JCA 1.5. Additionally, if the antlr-runtime classes are found in the class path, it is possible to use generated key retrieval.

### Distribution package

The latest version of Jaybird can be downloaded from https://firebirdsql.org/en/jdbc-driver/

The following files can be found in the distribution package:

| File name | Description |
|---|---|
| `jaybird-3.0.6.jar` | An archive containing the JDBC driver, the JCA connection manager, the Services API and event management classes. |
| `jaybird-full-3.0.6.jar` | Same as above, but including the JCA 1.5 dependency. |
| `lib/antlr-runtime-4.7.jar` | Optional dependency, required if you want to use `getGeneratedKeys` support |
| `lib/connector-api-1.5.jar` | Required dependency; part of `jaybird-full`, not necessary when deploying to a Java EE application server |
| `lib/jna-4.4.0.jar` | Optional dependency, required if you want to use Type 2 native, local or embedded protocols |

### Maven

Alternatively, you can use maven to automatically download Jaybird and its dependencies.

Jaybird 3 is available from Maven central:

Groupid: `org.firebirdsql.jdbc`,
Artifactid: `jaybird-jdkXX` (where `XX` is `17` or `18`).
Version: `3.0.6`

For example:

```
<dependency>
  <groupId>org.firebirdsql.jdbc</groupId>
  <artifactId>jaybird-jdk18</artifactId>
  <version>3.0.6</version>
</dependency>
```

The Maven definition of Jaybird depends on antlr-runtime by default.

If your application is deployed to a Java EE application server, you will need to exclude the `javax.resource:connector-api` dependency, and add it as a provided dependency:

```
<dependency>
  <groupId>org.firebirdsql.jdbc</groupId>
  <artifactId>jaybird-jdk18</artifactId>
  <version>3.0.6</version>
  <exclusions>
    <exclusion>
      <groupId>javax.resource</groupId>
      <artifactId>connector-api</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>javax.resource</groupId>
  <artifactId>connector-api</artifactId>
  <version>1.5</version>
  <scope>provided</scope>
</dependency>
```

If you want to use Type 2 support (native, local or embedded), you need to explicitly include JNA as a dependency:

```
<dependency>
  <groupId>net.java.dev.jna</groupId>
  <artifactId>jna</artifactId>
  <version>4.4.0</version>
</dependency>
```

For native and local you can use the `org.firebirdsql.jdbc:fbclient` dependency to provide the client library. See Maven dependency for native client for details.

# G.3. Jaybird 2.2

Jaybird 2.2 is end-of-life and will not receive further updates. We recommend to upgrade to Jaybird 3.

## G.3.1. Java support

Jaybird 2.2 supports Java 6 (JDBC 4.0), Java 7 (JDBC 4.1) and Java 8 (JDBC 4.2). Java 5 support was dropped in Jaybird 2.2.8.

For compatibility with Java 9 modules, version 2.2.14 introduced the automatic module name `org.firebirdsql.jaybird`. This guarantees a stable module name for Jaybird, and allows for future modularization of Jaybird.

### G.3.2. Firebird support

Jaybird 2.2 supports Firebird 1.0 and higher, but is only tested with Firebird 2.5 and 3.0.

Connecting to Firebird 3 requires some additional configuration, see Jaybird and Firebird 3.0 [https://github.com/FirebirdSQL/jaybird/wiki/Jaybird-and-Firebird-3] for details.

Firebird 4 is not formally supported in Jaybird 2.2.x, although connecting and most functionality will work. We suggest that you use Jaybird 3.x or higher for Firebird 4. Support for newer data types like `DECFLOAT` and `NUMERIC`/`DECIMAL` with precision higher than 18 will be introduced in Jaybird 4.

The Type 2 and embedded server JDBC drivers require the appropriate JNI library. Precompiled JNI binaries for Windows and Linux platforms are shipped in the default installation, other platforms require porting/building the JNI library for that platform.

### G.3.3. Supported Specifications

Jaybird 2.2 supports the following specifications:

| Specification | Details |
|---|---|
| JDBC 4.2 | Driver does not fully support JDBC 4.2 features, but implements large update count methods by calling the normal update count methods, and methods with `SQLType` by calling methods accepting the `java.sql.Types` integer value. Supports new `java.time` classes with some caveats. |
| JDBC 4.1 | Driver implements all JDBC 4.1 methods added to existing interfaces. The driver explicitly supports `closeOnCompletion`, most other methods introduced with JDBC 4.1 throw `SQLFeatureNotSupportedException`. |
| JDBC 4.0 | Driver implements all JDBC 4.0 interfaces and supports exception chaining. |
| JCA 1.0 | Jaybird provides an implementation of `javax.resource.spi.ManagedConnectionFactory` and related interfaces. CCI interfaces are not supported. Although Jaybird 2.2 depends on the JCA 1.5 classes, JCA 1.5 compatibility is currently not guaranteed. |
| JTA | The driver provides an implementation of the `javax.transaction.xa.XAResource` interface via the JCA framework and a `javax.sql.XADataSource` implementation. |

### G.3.4. Distribution

The Jaybird driver has compile-time and run-time dependencies to JCA 1.5. Additionally, if the antlr-runtime classes are found in the class path, it is possible to use generated key retrieval.

#### Distribution package

The latest version of Jaybird can be downloaded from https://firebirdsql.org/en/jdbc-driver/

The following files can be found in the distribution package:

| File name | Description |
|---|---|
| `jaybird-2.2.15.jar` | An archive containing the JDBC driver, the JCA connection manager, the Services API and event management classes. |
| `jaybird-full-2.2.15.jar` | Same as above, but including the JCA 1.5 dependency. |
| `lib/antlr-runtime-3.4.jar` | Optional dependency, required if you want to use `getGeneratedKeys` support |
| `lib/connector-api-1.5.jar` | Required dependency; part of `jaybird-full`, not necessary when deploying to a Java EE application server |
| `lib/log4j-core.jar` | Optional dependency, core Log4J classes that provide logging. |

Jaybird 2.2 has compile-time and run-time dependencies on the JCA 1.5 classes. Additionally, if Log4J classes are found in the class path, it is possible to enable extensive logging inside the driver. If the ANTLR runtime classes are absent, the generated keys functionality will not be available.

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

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

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

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

#### Maven

Alternatively, you can use maven to automatically download Jaybird and its dependencies.

Jaybird 2.2 is available from Maven central:

Groupid: `org.firebirdsql.jdbc`,
Artifactid: `jaybird-jdkXX` (where `XX` is `16`, `17` or `18`).
Version: `2.2.15`

For example:

```
<dependency>
  <groupId>org.firebirdsql.jdbc</groupId>
  <artifactId>jaybird-jdk18</artifactId>
  <version>2.2.15</version>
</dependency>
```

The Maven definition of Jaybird depends on antlr-runtime by default.

If your application is deployed to a Java EE application server, you will need to exclude the `javax.resource:connector-api` dependency, and add it as a provided dependency:

```
<dependency>
  <groupId>org.firebirdsql.jdbc</groupId>
  <artifactId>jaybird-jdk18</artifactId>
  <version>2.2.15</version>
  <exclusions>
    <exclusion>
      <groupId>javax.resource</groupId>
      <artifactId>connector-api</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>javax.resource</groupId>
  <artifactId>connector-api</artifactId>
  <version>1.5</version>
  <scope>provided</scope>
</dependency>
```

# Appendix H: License