jxDBCon-quick start guide

Keve Müller

18th October 2001

Abstract

This mini tutorial is intended to be used by developers, who want a quick start into the functionality of the PostgreSQL driver part of the jxDBCon distribution. It will not explain things generic to JDBC or Java. Instead it assumes the reader has already read or reading in parallel the excellent documentation provided by Sun for Java database connectivity.

1 Preface

The drivers tries to deploy the functionality available in PostgreSQL by exploiting the features defined in the JDBC API without creating new, incompatible, interfaces. Thus making it easier for developers to use the same code with different JDBC compliant drivers for different DBMS. Thus you should be able to use the driver without knowing about its internal structure and working. This quick start guide will give You an overview of the facilities of the driver and especially show You its limits.

2 Connecting

There are two ways to connect to the back-end. Via the Type 4 driver or the Type 2 driver. The Type2 drivers delegates communication to libPQ, the Type 4 driver handles communication autonomously. The parameters/features differ slightly:

	Type2	Type4
URL prefix	jdbc:postgresql:lib	jdbc:postgresql:net
Standalone	no, requires libpq	yes, pure Java
Unix domain sockets	yes	no
TCP/IP	yes	yes
SSL	yes	yes
Authentication types	all	plain and crypt
...	...	...

Generally You should use the Type4 driver unless You need the domain sockets functionality.

2.1 Using DriverManager

The standard way to connect with JDBC is using the DriverManager. The URL syntax for the driver is:

```
    jdbc:driver:subdriver[//[[user][:password]@][host][:port][/catalog][?parame
```

Almost everything can be left out and will default to default values. You can add parameters to both the URL and the info properties used when connecting. The parameters given in the URL override those given in the properties.

```
import java.sql.Connection;
import java.sql.DriverManager;

public final class SimpleConn {
static final String driverClass="org.sourceforge.jxdbcon.JXDBConDriver";
static final String dbURL="jdbc:postgresql:net//server/cat";
public static void main(String[] argv) {
try {
Class driver=Class.forName(driverClass);

Connection conn=DriverManager.getConnection(dbURL,"username","password");
//...
} catch (Exception ex) {
ex.printStackTrace();
}
}
}
```

Algorithm 1: Simple Connection example.

2.2 Using DataSource

With JDBC 3.0 a new connection method was introduced: `javax.sql.DataSource`. The driver implements this with the `PGDataSource` class. You can set parameters with the setter functions of this class. The datasource instance can then be serialized to a JNDI directory.

2.3 Parameters

The parameters supported are explained in detail in the API docs for InfoKeys and PGInfoKeys. Only the most prominent are listed here.

Catalog PostgreSQL has the notion of SQL catalogs which are sometimes referred to as databases. In the driver the database refers to the postmaster identified by the host name and port number. The database runs different catalogs, which for now are not inter-operable. You always have to be connected to one catalog. Once PostgreSQL will be enabled to access multiple catalogs the driver can use this with minor changes. Until then a call the `setCatalog` will re-connect to the postmaster, like `\c` in `pgsql` does.

UseSSL To use SSL you have to download and install the JSSE package from Sun. The packages need to be available at compile time, otherwise the SSL functionality is commented out. If you use a self signed certificate for your database, you need to import it. First issue `openssl x509 -in server.crt -outform DER -out server.xxx` to make it readable to Java routines, then import it with `keytool` into your keystore. If You specify “force” as a value to this key, then the driver will refuse to connect without SSL. Otherwise it will fall back to standard unencrypted communication.

UseAsync There are two ways how the Frontend/Backend protocol communication can be implemented. Synchronous and Asynchronous. In synchronous mode incoming data will be buffered by the operating system and only read by the driver when it is engaged in some communication. In asynchronous mode a separate thread handles the communication and gets incoming data immediately off the stream. This feature is important if You want to use PostgreSQL listen/notify constructs. With the async protocol You can immediately react to an incoming notification. Async is not fully tested, nor are speed implications examined.

UseClientEnc If this key is set, its value is used as the Java encoding name for converting text data in the protocol. No attempt is made to find out the client encoding the backend thinks it is using.

ForceClientEnc If this key is set, its value will be transmitted in a SET CLIENT_ENCODING TO command right after the communication channel is set up. You should normally not use it.

UseDBEnc After the communication channel is set up, the driver queries some properties of the backend these include its Endianness, the database’s encoding and the communication channel’s encoding. When this property is set, it will override the queried value for the database encoding. You should normally not use it.

2.4 Encodings

Since PostgreSQL6 the backend can be compiled with multibyte support. This also adds the notion of ’encodings’ to the backend. The driver queries the encodings in use at startup. You can override this in the connection parameters. If the backend is compiled without multibyte support, then there is no way to tell the encoding in use. In this case, you must specify which encoding to use.

3 Compliance

The driver tries to be as compliant as possible. Lack of compliance is due to features missing in PostgreSQL.

- Entry Level SQL92 is not supported by the backend
- The driver supports escape syntax, either by evaluation in the driver itself (all functions supported) or by mapping to the appropriate PGSQL function (limited set of functions supported)
- All features of PostgreSQL are made accessible by the driver
- Great care is taken in the driver to implement all DatabaseMetaData functions.

- There is no support in the driver for RowSets, 3rd party RowSet implementations can be layered on top of the driver.

3.1 JDBC 1.0 API Compliance

The driver is fully JDBC 1.0 API compliant

3.2 JDBC 2.0 API Compliance

The driver is JDBC 2.0 API compliant

3.3 JDBC 3.0 API Compliance

The driver does not support Savepoints because the underlying database has no support for it.

4 Meta data

The meta data for the database tries to be as accurate as possible but is far from being complete. The philosophy of PostgreSQL sometimes hinders the implementation of some functions. Nevertheless you should use the information provided and not rely on hard coded constants. All functions will give a reasonable result, sometimes this result is not what could be returned through excessive queries to the backend.

5 Exceptions

The driver has preliminary support for internationalized exceptions. All exceptions thrown by the driver extend I18nSQLException. They have an identifier part and a variable parameter part. The identifiers a looked up against a Message catalog which can be localized. No localization has been done yet (not even english...).

The database has no support for SQLState or vendorCodes. The driver uses a mapping table to virtualize the error strings sent by the backend into error identifiers, which then can be mapped to localized messages, SQLStates, etc.

Support for SQLState is begun, but not yet in the shipping driver.

SQLWarnings are supported.

DataTruncation is supported, if the truncation can be detected at the driver level. E.g. setFloat(3.14) to an int column.

BatchUpdateException is supported in the context of batch updates. The data embedded in the exception is accurate and valid.

6 Transactions

A Connection is initially in Auto-Commit mode. You can change this by issuing a setAutoCommit(false).

Savepoints are not supported by PostgreSQL.

7 Connection Pooling, Distributed Transactions

The driver has no implementation of a connection pool or distributed transactions. You can use 3rd party implementations with the driver.

8 Statements

8.1 ResultSets

There are currently two ResultSet implementations. PGPlainResultSet encapsulates data retrieved by a non-cursored query. It can be used in both Auto-Commit mode and Transactions. The query result is returned as a whole. The ResultSet is scrollable.

If Auto-Commit is disabled, then You can use CURSOR-ed queries. To do this simply give the Statement a cursor name via setCursorName. By default binary cursors are used, thus the data is more accurate and faster to access. The ResultSet returned is PGCursorResultSet which is scrollable but currently does not prefetch rows. There is space for improvements here.

Updatable ResultSets are not supported, as the backend has no support for it. The preliminary support which was available in v0.2 of the driver is removed. It will probably come back as part of jxUtil's support for RowSets.

Multiple ResultSets are supported, if the statement contains multiple commands separated by semicolon, then You can use the ResultSets getMoreResults() call to switch to the next result.

8.2 PreparedStatements

PostgresSQL's protocol has no support for PreparedStatements. Every statement has to be submitted with all the parameters converted to literals. This is a major drawback when implementing JDBC PreparedStatements.

The current implementation uses a multi-level approach to doing things in a compatible fashion.

Consider the following simple preparedStatement example: INSERT INTO testTable VALUES ('test',?); For this to work properly the type of the positional parameter must be known to the driver. This is important, because there is no trivial way to convert any type into its literal representation. The simplest example is the boolean type. If in the above example testTable's second column is of boolean type, then the following functions must be supported when setting its value. setBoolean, setByte, setInt, setLong, setFloat, setDouble, setBigDecimal, setString, setObject. Now, how can You code setInt without knowing the underlying type it has to convert-to.

Some of the conversions are caught by the backend and that's why most things work, but not all.

Until the backend will support PreparedStatements the driver has to parse the statement and do the necessary lookups to be able to type the positional parameters. In this case this would involve a SELECT FROM pg_class. As this is a performance issue, a backdoor allows the developer to type the parameters via a JDBC escape: INSERT INTO testTable VALUES('test',{cast(?,bool)});

8.2.1 Type conversion

If the underlying database type of a positional parameter is known to the driver, then all type conversions required by the JDBC specification are performed.

8.2.2 I/O of PreparedStatement

There is no way to implement PreparedStatement.getMetaData before the statement is actually executed, because of the lack of support in PostgreSQL. PreparedStatement.getParameterMetaData is supported by the driver, but the general contract on positional parameters apply here, too. Thus is the driver could not deduce the underlying database type, the information supplied here is not accurate.

8.3 CallableStatement

There is full support for the JDBC escape syntax for calling functions. There are two ways to specify a function, by name and by oid. When using the function name, be aware that PostgreSQL supports function overloading. Prominent example is the function text. If the function's name and parameter list length is not sufficient to uniquely identify a function You need to specify the oid or cast the parameters.

PostgreSQL functions may return a ResultSet set, but this is currently not supported in the FE/BE protocol. If you want to call a function that returns a ResultSet you have to do it in a query: stmt.executeQuery("SELECT setfunc()");

The protocol uses binary representation for the parameters and the result, which is spiced by some endianness conversion. See the comments in PGCallableStatement for this.

PostgreSQL does not support OUT or INOUT parameters.

9 Data types

The driver has a strong type system. Every type is represented by an instance of AbstractType class. This makes its behaviour consistent and compliant to the specs.

In PostgreSQL there are 3 ways to data can be exchanged via the FE/BE protocol. Text literals, binary data, and function call endian swapped data. The driver supports all of them with some deficiencies when decoding the binaries.

Literals are convenient but slow and error prone. The backend has to convert into literal form, which usually is a longer representation, and the backend has to convert back in order to work with the data. Binary data transmission does not have this limitations and its interpretation is easier, too. Consider the array types or the geometric types. Parsing them is much slower than just advancing a byte pointer.

The type system allows for extended features like the JDBC2 Array types or JDBC2 SQLData. These are already in the driver for the most common types.

The type mappings table shows the mapping used by the driver for the different PostgreSQL types. There are three different tables: for base types2, PGSQL special types4 and PGSQL structured types6. Please refer to the JDBC reference on how to access data of a given type. Additionally all types support (get|set)Bytes access with no processing. But be aware there are many traps when using these functions.

PostgreSQL type	JDBC type	Comments
int2	SMALLINT	
int4	INTEGER	
int8	BIGINT	Cannot insert Long.MIN_VALUE (-9223372036854775808), due to a bug in PostgreSQL.
float4	REAL	Cannot insert values smaller than 3.4028235E-38 and Float.MAX_VALUE (3.4028235E38), due to bugs in PostgreSQL. Values are truncated when using text transfer.
float8	DOUBLE	When inserting value smaller than Double.MIN_VALUE PostgreSQL silently truncates. Values are truncated when using text transfer.
numeric	NUMERIC	Only partial binary support for now.
bool	BOOLEAN	
char	OTHER	CHAR(1) type, type reporting may change.
name	OTHER	VARCHAR(32) type with null byte padding, type reporting may change
bpchar	CHAR	
varchar	VARCHAR	
text	LONGVARCHAR	
bytea	LONGVARBINARY	
date	DATE	
time	TIME	
abstime	TIMESTAMP	
timestamp	TIMESTAMP	Higher resolution.
oid	BLOB	Characteristics of this BLOB type are different on PostgreSQL compared to other vendors BLOB implementation.
<any>[]	ARRAY	Arrays of any known element type are supported.

Table 2: Mapping of base types

PostgreSQL type	JDBC type	Comments
unknown	OTHER	
money	NUMERIC	Under construction, don't use!
aclitem	LONGVARCHAR	Only read support, may change.
regproc	BIGINT,CHAR	Mixed type, not supported yet
reltime	OTHER	No full support, yet
bit	OTHER	
varbit	OTHER	
macaddress	OTHER	
inet	OTHER	No full support yet.
cidr	OTHER	No full support yet
interval	OTHER	No support.
int2vector	ARRAY	
oidvector	ARRAY	
oid	BIGINT	
tid	OTHER	No support.
cid	OTHER	No support.
xid	OTHER	No support
SET	OTHER	No support.
smgr	OTHER	No support

Table 4: Mappings for PostgreSQL special types.

PostgreSQL type	Element types	Element data	Comments
tinterval	abstime,abstime	from,to	
timetz	time,int4	time,zone offset	
point	float8,float8	x,y	
line	point,point	start,end	No support in the backend!
lseg	point,point	p1,p2	
box	point,point	p1,p2	
cicle	point,float8	center,radius	
path	bool,point[]	closed,points	Does not work for now.
polygon	bool,point[]	closed,points	Does not work for now.

Table 6: Mapping of structured/user definable types

9.1 Blob

Blob support has many problems with PostgreSQL as the support for Blobs is quite good, but has some small differences when compared to other DBMS. Blobs in JDBC are locator types, this fits the PostgreSQL notion of using oid to point to blobs. You can obtain a Blob by calling getBlob on a column of oid type which contains a valid large object oid. The Blob interface's functions are supported to modify it. For writing blobs, You can use the setBinaryStream() functions.

Note that bytea types cannot be retrieved as Blobs. This type maps to a LONG-VARBINARY and can be controlled with the getBytes/setBytes, getBinaryStream/setBinaryStream functions.

9.2 Array

Arrays on any basic type are supported an an instance of java.sql.Array is returned by getArray or getObject calls.

9.3 Structured and user defined types

Only known structured types (listed in table above) are supported, as PostgreSQL has no way to describe the internal storage or the external textual representation of a composite type. Only types for that a handler exsists in the driver can be worked with.

For these type full there is full support in the driver, including a java.sql.Struct representation and custom type mapping with the java.sql.SQLData interface.

Unknown/unhandled types are mapped to the PGTypes.unknown type, which has only access via the getBytes/setBytes function. You can use this to implement your own type handlers external to the driver.

10 Escape syntax

JDBC escape syntax is supported. Some of the function mappings are not yet done, some cannot be mapped to PostgreSQL functions, because of lack of support in the database.

See EscapeFuncs/PGEscapeFuncs for details.

11 Batch Updates

Batch updates are supported by the driver. The statements in a batch are concatenated and sent as on query to the backend. The multiple results are collected and transformed into the representation required by JDBC.

BatchUpdateExceptions are supported.

BatchUpdates for CallableStatements are not yet supported by the driver.