

# Chapter 1

# DBXTalk

*with the participation of:*

**Guillermo Polito** ([guillermopolito@gmail.com](mailto:guillermopolito@gmail.com))

DBXTalk is an open-source suite of tools to manage relational databases on top of Pharo. Its backbone is the DBXTalk database driver. The DBXTalk database driver allows the interaction with the major relational database engines through the SQL language. All other tools in the suite make use of this driver to fulfill their duties. Currently, the DBXTalk database driver supports the interaction with the following database engines: Oracle, MSSQL, PostgreSQL, MySQL, SQLite, SQLite 3, Sybase ASE, Firebird, and Interbase.

The DBXTalk suite, within its set of tools, provides GLORP, an Object-Relational-Mapper (ORM) framework; Phoseydon, a tool providing scaffolding capabilities for easy applications; and Neptuno, a browser to analyze our database structure.

In this chapter we will present DBXTalk core ideas, the tools it provides, and finally use them to build step by step a little application.

## 1.1 DBXTalk Architecture

The DBXTalk Driver relies on several components in order to connect to different relational databases:

- The OpenDBXDriver package talks to the OpenDBX library and handles the engines differences.
- OpenDBX is a C library which stands as an adapter between the different database engines and our Pharo image, and provides a common interface to interact with through foreign function interface (FFI).

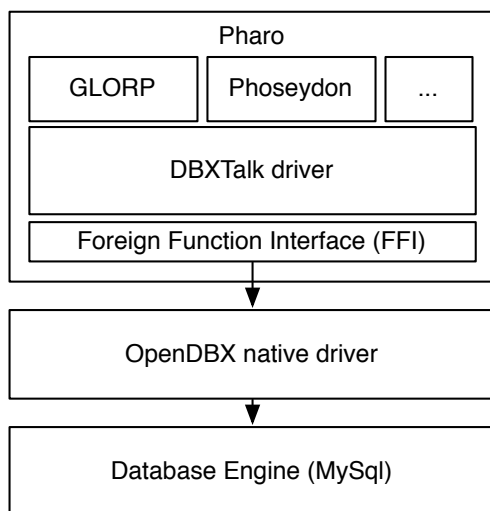


Figure 1.1: Architecture of the DBXTalk suite of tools

- We will need the corresponding client database library that OpenDBX will talk to.

## 1.2 DBXTalk Installation

To install DBXTalk library, we need to install the previously detailed components: the OpenDBXDriver as well as some databases.

### Install OpenDBX Driver

As we already introduced, an important part of DBXTalk architecture is the OpenDBX Driver, which allows us to communicate with different relational database engines using a common approach. OpenDBX is an open-source library, licensed under LGPL. Its installation instructions for different engines and operating systems can be found on <http://www.linuxnetworks.de/doc/index.php/OpenDBX>.

**Stéf** ► Here is a typical installation on MacOSX ◀

You can also participate in the Issue Tracker and mailing list of this project to ask questions and contribute. The issue tracker is available

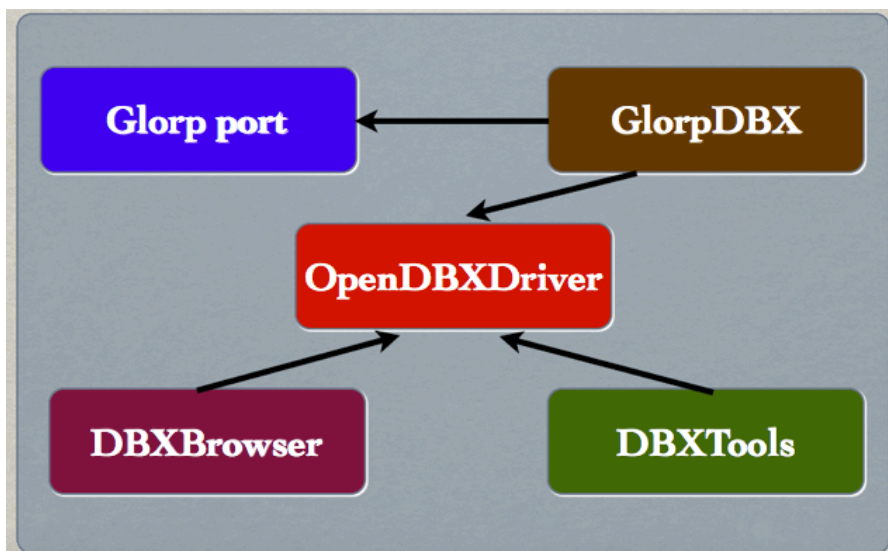


Figure 1.2: To redo and introduce

at <http://bugs.linuxnetworks.de/index.php?project=3>. The mailing-list is <https://lists.sourceforge.net/lists/listinfo/libopendbx-devel>

## Install Smalltalk OpenDBXDriver

Once you have installed the OpenDBX Driver, its Smalltalk counterpart is also needed to use the DBXTalk suite. The easiest way to install it using its metacello configuration. It will ensure that all its dependencies are loaded, such as FFI. This can be performed executing in a workspace the following script:

```
Gofer it
  squeaksource: 'DBXTalk';
  package: 'ConfigurationOfOpenDBXDriver';
  load.
```

```
((Smalltalk at: #ConfigurationOfOpenDBXDriver) perform: #project) perform: #version:
  with: #stable) load
```

## Ensuring everything was ok

The OpenDBXDriver package comes along with a large set of tests you can use to test the your installation. But before running the tests, you may want to configure the database connection settings in order to match your actual configuration. To do that, just go to the corresponding DBX\*Engine\*Facility class in the OpenDBXDriverTests package, and modify the createConnection method to suite your needs.

For example, if you want to configure the tests to run for a MySQL database, we should go to DBXMySQLFacility>>createConnection, to see the following:

```
DBXMySQLFacility>>createConnection
self connectionSettings: (DBXConnectionSettings
    host: 'localhost'
    port: '3306'
    database: 'sodbxtest'
    userName: 'sodbxtest'
    userPassword: 'sodbxtest').
self platform: DBXMySQLBackend new.
```

There you can change the host, port, database, username and password to connect to your own database.

**Stéf** ► *Once you have modify the settings you should be able to runs the tests* ◀

## 1.3 Getting down to business with DBXTalk

Now you have installed your database driver, you are ready to use it and build applications with it! But you need to know the basics. In the following sections you will be introduced in creating connections, execute SQL statements and handle transactions.

## 1.4 Creating a connection

The first step to execute a SQL statement in a database engine is to create a connection to that database. The DBXTalk OpenDBXDriver uses 3 objects to fulfill this objective: a connection object, which uses a platform/backend object and a connection settings object.

For example, to open a connection to a MySQL database called sodbxtest, located in localhost you can use the following code:

```
settings := DBXConnectionSettings
    host: 'localhost'
```

```
port: '3306'  
database: 'sodbxtest'  
userName: 'sodbxtest'  
userPassword: 'sodbxtest'.  
platform := DBXMySQLBackend new.  
connection := DBXConnection platform: platform settings: settings.  
  
connection connect.  
connection open.
```

As you can see, after creating the connection object, you have to send it first the connect message to let OpenDBX create all the needed internal structures. After that, you can send it the open message to associate the connection to the desired database, verify the credentials and enable us to start sending queries.


The main reason to separate these two operations is to configure some extra options before the connection is open. If you do not want to specify any of these options, you can send the connectAndOpen message to the connection instead:

```
connection connectAndOpen.
```

## Connection special options

As we told you in the last section, you can specify some special options to the connection before it is opened. All of these options can be enabled through the following helper methods of the connection. What they do is to try to enable the desired option, and return a boolean indicating if the option succeeded or not.

Each option may or not succeed because it depends on your database engine support for it.

- `enableCompression`. It tries to enable the Compression option.
- `enableEncryption: aEncryptionOption`. It tries to enable the Encryption option. The possible encryption option values are never, try and always.
  - `DBXEncryptionValues never`
  - `DBXEncryptionValues try`
  - `DBXEncryptionValues always`
- `enableMultipleStatements`. It tries to enable Multiple Statements option.  
 ► *no idea what it is* ◀
- `enablePagedResults`. It tries to enable paged results.

- `enableSpecialModes`: modes. It tries to enable specific modes. For example MySQL special modes are described at <http://dev.mysql.com/doc/refman/5.0/en/server-sql-mode.html>.

**Stéf** ► *How do I create the tables?* ◀

**Stéf** ► *would be good to describe the object model of the example.* ◀

**Stéf** ► *we should have stuff and stuff container: for example one Comix, a eerie of comix, and authors or one card, edition, author or a todo, lists of todo, associated persons* ◀

## 1.5 Executing SQL statements

The execution of a SQL statement in the database of your choice is performed sending the `execute`: message –or `executeMultiStatement`: if enabled- to the connection object. So, once you have your connection established and open, evaluate the following code:

**Create a table** . The first action is to create a table.

*"we create a table to store our trading card game cards in our MySQL database"*

```
connection execute: 'CREATE TABLE CARD(ID INT AUTO_INCREMENT PRIMARY
KEY, NAME VARCHAR(100))'
```

*"we insert some cards into our database"*

```
connection execute: 'INSERT INTO CARDS (NAME) VALUES("Giant Growth")'.
```

```
connection execute: 'INSERT INTO CARDS (NAME) VALUES("Llanowar Elves")'.
```

*"If we did enable multistatements before opening the connection we can do some inserts at the same time"*

```
connection executeMultiStatement: 'INSERT INTO CARDS (NAME) VALUES("Rancor");
INSERT INTO CARDS (NAME) VALUES("Counterspell")'.
```

*"we execute an invalid SQL statement to see how it raises a DBXError"*

```
connection execute: 'some invalid sql statement'.
```

## Fetching results

So far we have only executed statements without looking at their results. Each statement execution has a result, which may be one of `DBXResult`, `DBXResultSet` or `DBXMultiStatementResultSetIterator` classes.

A `DBXResult` is obtained when executing a SQL statement which does not have a set of rows as a result –such as a `CREATE TABLE`, `INSERT` or `UPDATE`

operation. You can ask it for its `rowsAffected` to know how many rows were affected in the operation.

```
result := connection execute: 'UPDATE CARDS SET NAME="CounterSpell" where
    NAME="Counterspell" '.
Transcript show: result rowsAffected.
```

A SQL statement with rows as a result –such as a *SELECT* statement–brings a `DBXResultSet` as result. A `DBXResultSet` is a consumable set of rows. This means that once you’ve consumed all the rows in the `DBXResultSet`, they will not be available any more. You can work with a `DBXResultSet` in several ways:

**Stéf** ► *stef stopped here* ◀

- Delegate row iteration to the result set:

```
cardsResultSet := connection execute: 'SELECT * FROM CARDS'.
cardsResultSet rowsDo: [ :row | Transcript show: (row valueAt: 1) ]
```

- Getting all rows from a result set:

```
cardsResultSet := connection execute: 'SELECT * FROM CARDS'.
rows := cardsResultSet rows
rows do: [ :row | Transcript show: (row valueAt: 1) ]
```

- Fetching one row at a time:

```
cardsResultSet := connection execute: 'SELECT * FROM CARDS'.
[ row := cardsResultSet nextRow.
  row notNil ]
  whileTrue: [ Transcript show: (row valueAt: 1); cr ]
```

When accessing a `ResultSet` rows, we get some `DBXRow` instances to play with. A `DBXRow` is the model of a database row, and each of the values for its columns can be accessed in two formats: the raw format, which is the string representation of the value, sent by the database, and a converted format, which is the Pharo object representation for that object. The `DBXRow` objects understand some of the following messages:

- `#valueAt:` retrieves the converted value at the column in the given index.
- `#values` retrieves a sequenceable collection with all the converted values of the row.
- `#rawValueAt:` retrieves the raw value at the column in the given index.

- `#rawValues` retrieves a sequenceable collection with all the raw values of the row.

Some examples of the behavior of a `DBXRow` are the following:

```
myRow valueAt: 1. -> 1      "First column is INTEGER and contains a 1. It retrieves a
                             Pharo SmallInteger"
myRow rawValueAt: 1. -> '1'  "First column is INTEGER and contains a 1. It retrieves
                             an string"

myRow values. -> #(1 'Llanowar Elves') "A collection with every row value converted
                                         to Pharo objects."
myRow rawValues. -> #('1' 'Llanowar Elves') "A collection with every row raw value."
```

Finally, the other kind of result we can get from a SQL statement is a `DBXMultiStatementResultSetIterator`. It stands as a `DBXResultSet` container, for multi-statement queries. You can browse the results using the following convenience methods.

```
myMultistatementResult allResultsDo: [ :aResult | "doSomething here with the result"]. "
                                         internal iteration of the results."

[result := myMultistatementResult next. result notNil]
whileTrue: [ :aResult | "doSomething here with the result"]. "iterating the results one by
one."
```

««««< HEAD =====

## 1.6 Tools

## 1.7 Glorp

»»»»> e7c68e168df735d2cdbf57b2e29c6bab2ec0dc66



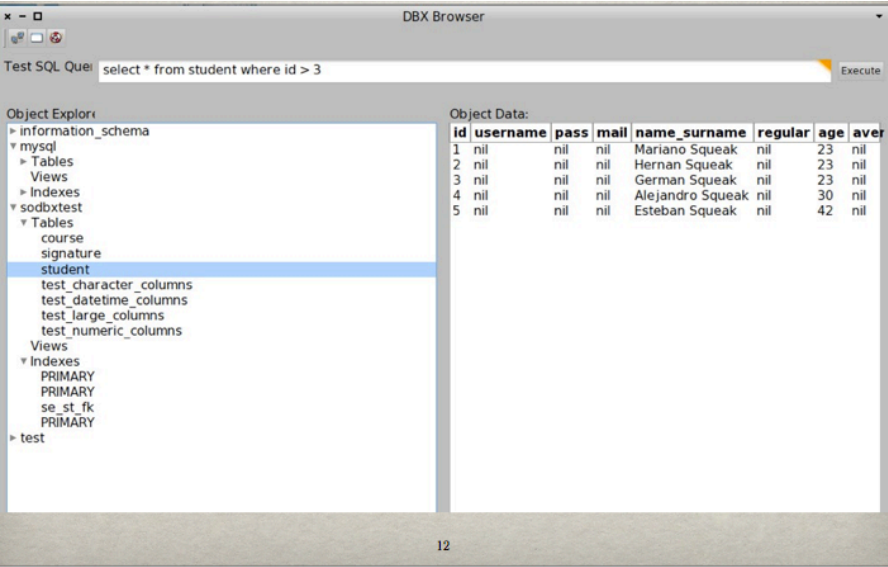


Figure 1.3: To redo