# PostgreSQL Cocoa Framework

**version 0.5**

**June 2003**

**by P3 Consulting**

# Preface

## *License*



WARNING

PRELIMINARY VERSION OF DOCUMENTATION.
MAY BE INCOMPLETE, IN CASE OF DIFFERENCES BETWEEN THIS DOCUMENT AND HeaderDoc -GENERATED DOCUMENTATION, THE LATEST SHOULD PREVAIL.

# PostgreSQL Cocoa Framework Classes

## *PostgreSQL*

The PostgreSQL class holds the connection to a PostgreSQL database. This class maintains a pool of instances of connections automatically reused when needed.

To create a instance of the class:

```
PostgreSQL *pgconn = [PostgreSQL newPostgreSQLConnection] ;
```

To return an instance to the pool:

```
+ (void)releasePostgreSQLConnection:(PostgreSQL *)inConn;
```

### Methods to connect to a database

```
- (BOOL)connectToDatabase:(const char *)dbName onHost:(const char
    *)hostName login:(const char *)loginName password:(const char
    *)password return:(int *)returnCode;
```

Connects to a database using parameters and returns result code in last parameter. Parameters must be valid address (parameters cannot be nil), empty string parameters are valid.

```
- (BOOL)connectToDatabase:(const char *)connectInfo return:(int
    *)returnCode;
```

Connects to a database using connection string and returns result code in last parameter. Connection string cannot be nil, empty string is valid but unlikeley to result in a valid connection. Each parameter setting is in the form keyword = value. (To write an empty value or a value containing spaces, surround it with single quotes, e.g., keyword = 'a value'. Single quotes and backslashes within the value must be escaped with a backslash, e.g., \' or \\.) Spaces around the equal sign are optional. The currently recognized parameter keywords are:

`host`

Name of host to connect to. If this begins with a slash, it specifies Unix-domain communication rather than TCP/IP communication; the value is the name of the directory in which the socket file is stored. The default is to connect to a Unix-domain socket in /tmp.

`hostaddr`

IP address of host to connect to. This should be in standard numbers-and-dots form, as used by the BSD functions inet_aton et al. If a nonzero-length string is specified, TCP/IP communication is used. Using hostaddr instead of host allows the application to avoid a host name look-up, which may be important in applications with time constraints. However, Kerberos authentication requires the host name. The following therefore applies. If host is specified without hostaddr, a host name lookup is forced. If hostaddr is specified without host, the value for hostaddr gives the remote address; if Kerberos is used, this causes a reverse name query. If both host and hostaddr are specified, the value for hostaddr gives the remote address; the value for host is ignored, unless Kerberos is used, in which case that value is used for Kerberos authentication. Note that authentication is likely to fail if libpq is passed a host name that is not the name of the machine at hostaddr.

`port`
: Port number to connect to at the server host, or socket file name extension for Unix-domain connections.

`dbname`
: The database name.

`user`
: User name to connect as.

`password`
: Password to be used if the server demands password authentication. options

`tty`
: A file or tty for optional debug output from the backend.

`requiressl`
: Set to 1 to require SSL connection to the backend. Libpq will then refuse to connect if the server does not support SSL. Set to 0 (default) to negotiate with server.See postgreSQL reference manual for more information

```
- (BOOL)connectToDatabaseWithDictionary:(NSDictionary *)connectInfo
    return:(int *)returnCode;
```

Connects to a database using dictionary keys :
`hostname`, `port`, `dbname`, `user`, `password`, `tty` and `options`.

```
- (BOOL)connectAskingUser:(int *)returnCode forWindow:(NSWindow
    *)hostWindow;
```

Presents a connection dialog to the user. If forWindow parameter is not nil, the dialog is a pane attached to the specified window.

```
- (BOOL)connectAskingUserWithDefaults:(NSDictionary *)defaults return:
    (int *)returnCode forWindow:(NSWindow *)hostWindow;
```

Presents a connection dialog to the user. If forWindow parameter is not nil, the dialog is a pane attached to the specified window. The defaults dictionnary is used to prefill the fields of the dialog. Keys of the default dictionnary are: hostname, port, dbname, user, password.

```
- (void)disconnect;
```

Disconnect from the host database. A disconnected PosgreSQL object can be reused for another connection.s

## Methods to help debugging host application

```
+ (void)setDebug:(BOOL)yn;
```

Class method to turn on|off global debugging mode. When a new PostgreSQL object is allocated its debugging mode is initialized with the current global debugging mode value. When a connection is returned to the pool, its debug flag is reset to `NO`.

```
+ (NSString *)frameworkVersion;
```

Class method returning the framework version extracted from its bundle.

+ (int)makeCommandBufferSize;

  Returns the size of the internal character buffer used for temporary operation. Avoid calling makeCommand methods with parameter generating a buffer larger than the value returned by this method: results would be probably incorrected. Internally, functions of the snprintf-family are used to avoid buffer overflow.

- (void)setDebugMode:(BOOL)turnOn;

  To turn on|off connection debugging mode.

- (BOOL)debugMode;

  Returns current debugging mode of the connection.

-(BOOL)startTracing:(const char *)traceFileName;

  Equivalent to startTracing:(const char *)traceFileName append:NO;

-(BOOL)startTracing:(const char *)traceFileName append:(BOOL)inMode;

  Turns on Postmaster tracing. TraceFileName should points on a valid pathname for which running process have write access right. Added to standard PostgreSQL tracing (PQtrace), the framework adds tracing of executeCommand and resulting server message:

- (const char *) serverMessage;

  Returns a pointer on a C string containing the server message resulting of the latest operation executed.

- (BOOL)startTracing:(const char *)traceFileName append:(BOOL)append;

  Start tracing of command in the specified file. Lines are added to the trace file on each executeCommand.
  If append is YES, the file is opened with mode «a», if append is NO, the file is opened with mode «w».

-(void)stopTracing;

  Turns off tracing.

- (FILE *)tracingFile;

  Returns current tracing stream, or NULL if tracing is currently off. used in combination with setTracingFile to use one tracing file for several PostgreSQL connections.

- (void)setTracingFile:(FILE *)inStream;

  If inStream is not NULL, turns on tracing using the specified stream.
  If inStream is NULL, turns off tracing.

- (NSDictionary *)getVariables;

  Returns connection's variables. Varaiables are retrieved from framework cached variable or by executing a SHOW ALL statement if cache is empty.

- (NSDictionary *)loadVariables;

  Returns connection's variables always by executing a SHOW ALL statement (cached copy is updated).

- (NSStringEncoding) showClientEncoding;

  Returns client encoding by converting "Current client encoding" variable into a NSStringEncoding.

- `(NSStringEncoding) showServerEncoding;`

  Returns server encoding by converting "`Current server encoding`" variable into a `NSStringEncoding`.

- `(void)description;`

  Returns a string describing the PostgreSQL object (for debuging purpose).

- `(NSString *)postgreSQLVersion;`

  Returns the version string of the server (the one returned by "`select version()`").

- `(NSString *)frameworkVersion;`

  Returns the version of the framework (from the bundle's info string).

- `(const char *)databaseName;`

  Returns a pointer on the connected database name.

- `(const char *)hostName;`

  Returns a pointer on the connected host name.

- `(const char *)loginName;`

  Returns a pointer on the connected database login name.

- `(const char *)password;`

  Returns a pointer on the connected database login password.

- `(const char *)port;`

  Returns a pointer on the connected database port (PQport).

- `(const char *)tty;`

  Returns a pointer on the connected database tty(PQtty).

- `(const char *)options;`

  Returns a pointer on the connected database options (PQoptions).

- `(int)socket;`

  Returns the connected database socket (PQsocket).

- `(int)backendPID;`

  Returns the connected database cakend process id (PQbackendPID).

- `(PGconn *)getPGconn;`

  Returns a pointer on the connected database connection decriptor, in case you need to implement calls not available in the framework.

- `(NSString *)uniqueID;`

  Returns an unique string generated by calling `CFUUIDCreateString`.

- `(const char *)commandCBuffer;`

  Returns a pointer on the internal command buffer as a C string.

- `(const char *)commandUTF8Buffer;`

  Returns a pointer on the internal command buffer as a UTF-8 string.

`- (NSString *)commandBufferString;`

Returns a copy (autorelased) of the internal command buffer.

## Methods to get meta data information

This set of methods is just a short writing of statements querying the pg_* metadata tables. See the PostgreSQL developer documentation for more information over system tables. Once understood the structure of system tables and looking at source code of the framework, you should be able to create your own specific queries if needed.

`- (NSDictionary *)databases;`

Returns of dictionary describing the databases hosted at same host as the current connection. The dictionary is generated by executing a `SELECT * FROM pg_database` statement, followed by `resultAsDictionary`.

`- (NSDictionary *)tables;`

Execute the query `SELECT * FROM pg_tables WHERE schemaname = 'public'` followed by `resultAsDictionary`.

`- (NSDictionary *)pgTables;`

Execute the query `SELECT * FROM pg_tables WHERE schemaname = 'pg_catalog'` followed by `resultAsDictionary`.

`- (NSDictionary *)views;`

Execute the query `SELECT * FROM pg_views WHERE schemaname = 'public'` followed by `resultAsDictionary`.

`- (NSDictionary *)pgViews;`

Execute the query `SELECT * FROM pg_tables WHERE schemaname = 'pg_catalog'` followed by `resultAsDictionary`.

`- (NSDictionary *)users;`

Execute the query `SELECT * FROM pg_user` followed by `resultAsDictionary`.

`- (NSDictionary *)sequences;`

Creates a temporary view containing the schema name, the sequence name and the owner name, executes a `SELECT *` on this view followed by `resultAsDictionary`.

`- (NSDictionary *)indexes;`

Execute the query `SELECT * FROM pg_indexes` followed by `resultAsDictionary`.

`- (NSDictionary *)triggers;`

Execute the query `SELECT * FROM pg_trigger` followed by `resultAsDictionary`.

`- (NSDictionary *)rules;`

Execute the query `SELECT * FROM pg_rules` followed by `resultAsDictionary`.

`- (NSDictionary *)groups;`

Execute the query `SELECT * FROM pg_group` followed by `resultAsDictionary`.

- `(NSDictionary *)constraints;`

  Execute the query `SELECT * FROM pg_constraint` followed by `resultAsDictionary`.

- `(NSDictionary *)languages;`

  Execute the query `SELECT * FROM pg_language` followed by `resultAsDictionary`.

- `(NSDictionary *)settings;`

  Execute the query `SELECT * FROM pg_settings` followed by `resultAsDictionary`.

## Methods to help integration in Cocoa-style programming

PostgreSQL init method registers the receiver as an observer for the `NSApplicationWillTerminateNotification` and creates a `NSTimer` to listen for asynchronous notifications (with a time interval of 1.0).
PostgreSQL class implements the `NSCoder` protocol.

- `(void)setDelegate:(id)newDelegate;`

  Sets the delegate object (retained by the PosgreSQL object).
  The connection delegate is invoked by the framework on special circumstances, most methods implemented by the delegate accepts only one parameter: the PostgreSQL object generating the event:
  when result from an asynchronous query is available:
  `postgreSQLAsyncResultAvailable:(PostgreSQL *)inSource`.

  when a time-out occurs:
  `postgreSQLAsyncResultTimedOut:(PostgreSQL *)inSource`.

  when a notification from the backend is received:
  `postgreSQL:(PostgreSQL *)inSource notification:(NSDictionary *)info.`

  when application terminates:
  `postgreSQLAppWillTerminate:(PostgreSQL *)inSource.`

- `(id)delegate;`

  Returns the connection delegate.

- `(PostgreSQL *)clone:(int *)resultCode;`

  Duplicates the receiver and open a new connection to the server using the same parameters as the receiver. Returns `nil` in case of errror and the error code in buffer pointed by resultCode parameter.

- `(int)clientEncoding;`

  Returns the current client encoding.

- `(int)setClientEncoding:(const char *)encoding;`

  Set the client encoding.

- `(NSString *)escapeString:(const char *)inSource;`

  Escape the C string for inclusion in SQL queries. (Look at the Cocoa Extensions Framework for more powerful utilities regarding escpaing of string for various usages).

- `(NSData *)escapeBinary:(const unsigned char *)inSource length:(size_t)len;`

  Escape the C string using the PosgreSQL utility. THIS DOESN'T WORK FOR UTF-8 encoded strings.

- `(NSData *)unescapeBinary:(const unsigned char *)inSource length:(size_t)len;`

  Available only with libpq version 7.3 above.

- `(const char *)serverMessage;`

  Returns the server message resulting from the latest executed command `(PQerrorMessage)`.

- `(const char *)connectErrorMessage:(int)errorCode;`

  Returns the status message from the specified error code `(PQresStatus)`.

### Methods for handling server notifications

- `(PgSQLListener *)startNotificationFor:(const char *)inTableName delegate:(id)notificatonDelegate userInfo:(id)userInfo;`

  Register a delegate handler to be notified on server notifications for the sepcified table. userInfo is passed back in the userInfo dictionary of the notification parameter of the delegates's method.
  `- (void)pgSQLNotify:(NSNotification *)notification;`
  The `userInfo` dictionary of the notification contains the following key:
  `userInfo`: the original userInfo passed when registering the notification handler,
  `condition`: the table name generating the server notification,
  `PostgreSQL`: the receiver of the original startNotificationFor.

- `(void)removeNotificationFor:(PgSQLListener *)listener;`

  Unregister a listener.

### Methods for transaction processing

- `(BOOL)beginTransaction;`

  Starts a transaction by executing a `BEGIN` statement. You can test if a transaction is in progress by calling `transactionInProgress`. Returns YES if successful.

- `(BOOL)commitTransaction;`

  Commits the transaction by executing an `END` statement.

- `(BOOL)rollbackTransaction;`

  Rolls back the current transaction.

- `(BOOL)transactionInProgress;`

  Returns YES if a transaction is in progress.

## Methods for executing commands

The PosgreSQL object maintains an internal buffer to hold the command string to be executed. The programmer clears, adds string to the buffer, then execute it and eventually checks for result using the following methods. Result of query can be extracted as a whole dictionary, row by row as an array of fiels or by automatic assignation of column values to binded variables.

- `(void)clearCommands;`

  Clears the command buffer of the connection. You should explicitly clear the command buffer every time before starting definition of a new query. No other method will clear the command buffer as as side effect.

- `(void)makeCommand:(const char *)cmd;`

  Append the command string to the current command buffer (using `NSString stringWithCString`). Internally the command buffer is maintained as a NSString. If you want to append a command string containing UTF- characters use makeCommandWithString instead.

- `(void)makeCommandWithString:(NSString *)cmd;`

  Append the command string to the current command buffer.

- `(void)makeCommandf:(const char *)format, ...;`

  Append the command string using `printf` formatting.

- `(void)vmakeCommandf:(const char *)format args:(va_list)ap;`

  Append the command string using `vprintf` formatting.

- `(BOOL)executeCommand;`

  Pass the `current` command buffer string to the server for execution. The classic approach for a `SELECT` query:

```
[conn clearCommands];
[conn makeCommand:"SELECT …"]
if ([conn executeCommand] && ([conn rowsAffected] > 0)) {
    [conn bind…]   ;
    …
    while ([conn nextRow])    {
        …
    }
}
```

  ExecuteCommand clears the internal binding array. If you want to keep the same variables binding between command execution, use `executeCommandKeepBinding` instead.
  ExecuteCommand will fail if an asynchronous command is running (race condition on internal result data structure).
  After executing a buffer command, the internal cursor is positioned BEFORE the first row. To retrieve data, call `nextRow`.

- `(BOOL)executeCommandKeepBinding;`

  Same as `executeCommand` but without clearing the internal binding structure. It is useful if you have several queries returning the same set of columns in the same scope.

- `(BOOL)executeCommandWithCursor:(const char *)cursorName binary: (BOOL)binary;`

  Executes the command inside a cursor. You could also use makeCommand family of methods to achieve manually the same affect. Example:
  `[conn makeCommand:"DECLARE CURSOR myCursor FOR"] ;`

- `(BOOL)executeAsyncCommand;`

  Sends the command buffer string to the server for asynchronous execution. Checking for result may be done by polling with asyncResultAvailable or by registering an observer for `PostgreSQLResultAvailable` notification. In the later case you should also check for `PostgreSQLTimeOut` notification. Asynchronous queries are done from another thread, each PosgreSQL object has an asynchronous thread associated, this is why internally we use a connection pool: to reuse efficiently the created threads.

- `(BOOL)executeAsyncCommandWithCursor:(const char *)cursorName binary: (BOOL)binary;`

  Execute the command inside a CURSOR context. You may retrieve result by executing FETCH sql commands.

- `(BOOL)closeCursor:(const char *)cursorName;`

- `(BOOL)cancelRequest;`

  Cancel the latest asynchronous query done.

- `(BOOL)asyncResultAvailable;`

  Returns YES if result is available from lastest asynchronous query.

- `(long)getTimeOut;`

  Returns the time-out value used for asynchronous queries.

- `(void)setTimeOut:(long)inTimeOut;`

  Set the time-out value for asynchronous queries (0 for no time-out).

- `(id)curRowField:(int)fieldIndex;`

  Fetch the value of the field at index fieldIndex of the current result row.

- `(BOOL)bufferHasCommands;`

  Returns YES if internal command buffer is not empty.

- `(PgSQLResult *)getResultSet;`

  Returns the latest query result as a PgSQLResult object.

- `(BOOL)resultReturned;`

  Returns YES if latest command executed returns a result set (rowsAffected > 0).

- `(void)goRow:(int)rowIndex;`

  Position the internal row cursor on the specified row index. (Don't fetch any data).

- `(void)goRowRelative:(int)rowIndex;`

  Position the internal row cursor on the specified row index relatively to the current row. (Don't fetch any data).

- `(const char *)uniqueRowIdForTable:(const char *)tableName;`

  This methos assumes you have a `_rowid bigserial` field in your tables to emulate the OpenBase functionality. Returns the next value the _rowid will take on the next `INSERT` statement. Be aware of the fact that this is subject to race condition in multi-user environment and in consequence you should LOCK the table and use transaction to avoid any problem.

- `(const char *)uniqueRowIdForTable:(const char *)tableName column:(const char *)colName;`

  Same as previous method, but the column name is here a parameter for more general use if you don't use _rowid style of database design.

- `(long long)uniqueRowIdForSequence:(const char *)tableName;`

  This methos assumes you have a `_rowid bigserial` field in your tables to emulate the OpenBase functionality. Returns the value of the next `_rowid` using nextval() PostgreSQL function.

- `(long long)uniqueRowIdForSequence:(const char *)tableName column:(const char *)colName;`

  Same as previous method but column name is a parameter.

- `(NSData *)retrieveBinary:(Oid)inOid;`

  Retrieve binary data from an Oid using lo_XXX PostgreSQL internal functions.

- `(NSData *)retrieveBinaryFromStartLength:(Oid)inOid start:(int)inStart length:(int)length;`

  Retrieve partial binary data from an Oid using lo_XXX PostgreSQL internal functions.

- `(Oid)insertBinary:(unsigned char *)inData size:(int)size;`

- `(Oid)insertBinaryFromFile:(NSFileHandle *)inFile;`

- `(Oid)insertBinaryFromData:(NSData *)inData;`

- `(BOOL)overWriteBinaryWithDataFromStart:(Oid)inOid data:(NSData *)inData start:(int)inStart;`

- `(int)exportBinaryToFile:(Oid)inOid pathName:(const char *)inPathName;`

- `(Oid)importBinaryFromFile:(const char *)inPathName;`

- `(int)unlinkBinary:(Oid)inOid;`

### Methods to export data in XML format

*See PgSQLresultSetProtocol.*

## *PgSQLBindingProtocol*

The binding protocol defines the methods to associate program memory location with fields value when fetching query results.

Memory associated with query results should be in the same scope as the call that actually fetch result. There are a `bind`*`Type`*`:(`*`Type`*` *)var` and a `bind`*`Type`*`:(`*`Type`*` *)var column:(int)col` for each supported type. PostgreSQL native types (int, float, char *, bingint, TEXT, POINT, Polygon, …) and major Cocoa types (NSString, NSNumber, NSArray, NSData, … ) are supported.
One special case are
`- (void)bindNSCalendarDate:(NSCalendarDate **)var format:(NSString *)dateFormat;`
`- (void)bindNSCalendarDate:(NSCalendarDate **)var column:(int)col format:(NSString *)dateFormat;`
Both accept a parameter to specify with which format date returned from queries should be scanned.

PgSQLBindingProtocol is implemented by PostgreSQL and PgSQLResult classes.

## *PgSQLResultSetProtocol*

The result set protocol defines method for retrieving data from queries.
PgSQLResultSetProtocol is implemented by PostgreSQL and PgSQLResult classes.

`- (int)rowsAffected;`

Returns the number of rows affected by latest command. To be used with SELECT queries.

`- (int)rowsAffectedByCommand;`

Returns the number of rows affected by latest command. To be used with UPDATE, DELETE and INSERT queries.

`- (BOOL)isColumnNULL:(int)fieldIndex;`

Returns if column at index fieldIndex of the current result row is NULL.

`- (BOOL)nextRow;`

Position the internal row cursor on the next row and returns YES if data is available (e.g. we don't reach the end of the result set) and fetch data in the binded variables.

`- (BOOL)previousRow;`

Position the internal row cursor on the previous row and returns YES if data is available (e.g. we don't reach «before first row» position) and fetch data in the binded variables.

`- (void)firstRow;`

Position the internal row cursor on the first row. (Don’t fetch any data).

`- (void)lastRow;`

Position the internal row cursor on the last row. (Don’t fetch any data).

`- (void)beforeFirstRow;`

Position the internal row cursor on the last row. (Don’t fetch any data).

`- (void)afterLastRow;`

Position the internal row cursor after the last row. (Don’t fetch any data).

`- (void)goRow:(int)rowIndex;`

Positions the internal row index to the specified one (absolute position from 0 to N-1).

- `(void)goRowRelative:(int)rowIndex;`

  Positions the internal row index to the specified one elatively to te current position (resulting position is clipped to [0; N-1] interval.

- `(int)resultColumnCount;`

  Returns the number of columns in the current result set.

- `(const char *)resultColumnName:(int)col;`

  Returns the name of the column at index `col` in the current result set.

- `(const char *)resultColumnTypeName:(int)col;`

  Returns the type name of the column at index `col` in the current result set.

- `(Oid)resultColumnTypeName:(int)col;`

  Returns the type Oid of the column at index `col` in the current result set.

- `(id)curRowField:(int)fieldIndex;`

  Returns field at fieldIndex of current row as an Obj-C object (NSString, NSNumber, etc.) according to the type of the column in the database.

- `(NSArray *)resultRowAsArray;`

  Returns the current result's row as an array of objects.

- `(NSDictionary*)resultAsDictionary;`

  Returns the current result as a dictionary of objects:
  «columns» contains an array of dictionary : { «name», «typeOid» «typeName»}.
  «rows» contains the actual data as an array of array of objects.

- `(void)xmlExport:(NSString *)inFilePath root:(NSString *)inRootName;`

  Calls the `xmlExport:root:exportQuery` with exportQuery set to `YES`.

- `(void)xmlExport:(NSString *)inFilePath root:(NSString *)inRootName exportQuery:(BOOL)inExportQuery;`

  Exports the data in xml format:

```
<inRootName>
  <select query=commandBuffer>
       <row>
            <field_name>
                 field_value
            </field_name>
       </row>
  </select>
</inRootName>
```

  If `inExportQuery` is `NO`, the `<select>` tags are commented out by xml comment tags (`<!-- ... -->`).

- `(const char *)resultTableName:(int)col;`

  This funciton is provided for source coded compatibility with other frameworks (e.g. OpenBase) supporting this functionality. PostgreSQL currently doesn't provide a way to return the table name of a column in a result set.

### PgSQLResult

### PgSQLBindItem

### PgSQLListener

### PgAsyncThread

PgAsyncThread class manages the launching of an asynchronous thread to handle asynchronous queries on the specified connection.

```
+ (NSConnection *)startASyncThreadForConnection:(PostgreSQL *)inConn;
```

```
- (id)initForConnection:(PostgreSQL *)theConnection;
```

```
- (void)executeAsyncCommand:(NSString *)cmdBuffer;
```

### LoginWindowController - LoginPanelController

The 2 controllers classes to manage the entry of connection parameters by the user.

```
- (id)initWithDefaults:(NSDictionary *)defaults;
```

Keys in defaults dictionary:
hostname, port, dbname, user, password, options

```
- (IBAction)doCancel:(id)sender;
```

IBAction for cancel button. Just stop the modal loop.

```
- (IBAction)doConnect:(id)sender;
```

IBAction for Connect button. Try to connect and if successful stops the modal loop, if not display an error message and lets the user takes appropriate action.

```
-(void)setConnection:(PostgreSQL *)inConnection;
```

After initialisation of the controller, you have to pass a PosgreSQL object to it, before going to modal loop, to be used to connect to the database.

```
-(BOOL)connectionOK;
```

Returns if we successfuly connect to the database specified by the user.

```
-(int)errorCode;
```

Returns the error code returned by the backend.

```
-(NSDictionary *)makeDefaultsFromConnection;
```

If successfully connected, returns a dictionary with the hostname, port, dbname, user, password, options keys set according to the user specified parameters.
Returns nil if connection was not successful.