

dbConnect API
Version 0.3
Revision 1

Programmers Guide

Table of Contents

Chapter 1: Overview	4
Introduction	4
History	4
About the Author.....	4
Chapter 2: Compiling dbConnect	5
Compiling for Unix	5
Compiling for Windows	6
Chapter 3: C++ API	7
DbConnect API Classes	7
Public Types.....	7
DbConnection Class	8
BaseQuery Class.....	12
BaseValue Class	17
BaseFieldDescription Class.....	23
Utility API Classes	27
JDate Class	27
ConfigFile / Config Section Class	27
DLoader Class	27
HexDigest Namespace	27
Exceptions	28
BaseException Class	28
dbConnect Exceptions.....	29
Chapter 4: Drivers	30
IBM DB2	30
Overview	30
Unix Configuration Parameters.....	30
MySQL	30
Overview	30
Unix Configuration Parameters.....	30
Windows Configuration	30
PostgreSQL-7	31
Overview	31
Unix Configuration Parameters.....	31
Windows Configuration	31
mSQL	31
Overview	31
Unix Configuration Parameters.....	31

dbConnect 0.3 API Programmers Guide

by Johnathan Ingram, Rogueware Software

Copyright © 1999-2005 by Johnathan Ingram, Rogueware.

Legal Notice

The dbConnect API is Copyright © by Johnathan Ingram, Rogueware.

The software and associated documentation is released under LGPL license.

Chapter 1: Overview

Introduction

dbConnect API is an open source project aimed at developing an easy to use and uniform C++ API to access multiple databases on the Windows and Unix platforms.

dbConnect API uses the native API's of the Database Management Systems (DBMS) to achieve maximum speed and the lowest level of integration.

The dbConnect API is designed as a modular framework allowing database drivers to be loaded and unloaded dynamically during runtime. The modular framework also allows for the redeployment of the dbConnect API components without recompiling of the application in the event of upgrading the API.

History

The dbConnect API was started by Johnathan Ingram as a simple wrapper to the Oracle Call Interface (OCI) in 1999.

The wrapper soon evolved into an API as support for MySQL and other databases where added.

In 2001 the API was formally registered as a SourceForge project and the 0.2.x series was released to the open source community.

In the beginning of 2002 it was realised that the 0.2.x series had some "short comings" in the future roadmap of the project.

The entire 2002 year was devoted to the redesign on the dbConnect API and in December 2002 the first release of the 0.3.x series became available.

About the Author

I, Johnathan Ingram, am a Solutions Architect focused predominantly in the Internet Service Provider (ISP) industry.

I am based in Johannesburg, South Africa.

The projects I am involved in range from large mail installations to full ISP deployments and integration with an OSS system called Evolutionware.

I have a keen interest, amongst others, in databases and there applications. Utilising databases in nearly all the development I have been involved in has led me to believe that database access and utilisation should be kept as simple and to the point as possible as it normally constitutes a major component of all software application development.

I hope you find the dbConnect project useful and enjoy using it as much as I have enjoyed bringing it to the open source community.

Chapter 2: Compiling dbConnect

Obtain the latest 0.3.x distribution form <http://www.sourceforge.net>.

Before compiling, ensure that you have the required database libraries installed for the dbConnect API drivers you would like to compile.

See the drivers chapter for more details.

Compiling for Unix

The GNU C Compiler "gcc" is required for compiling dbConnect on the Unix platforms.

1. Extract the tar ball into a directory of your choice
tar -xvfz dbconnect-0.3.x.tar.gz
cd dbconnect-0.3.x
2. Run the configure script passing the required configure parameters for the database drivers you would like to compile. See the drivers chapter for more details on driver configuration script options. Run *./configure --help* for help.
./configure --enable-mysql
3. Build the API
make
4. Build the examples
make example

Once the dbConnect API is compiled the following files are available:

dbconnect-0.3.x/include	Directory contains all required header files for application development. Can copy to /usr/include if required.
dbconnect-0.3.x/lib	Directory contains the main dbConnect library and a sub directory containing the driver .so files. Can copy the main dbConnect library to /usr/lib if required or add the lib directory to your LD_LIBRARY_PATH. <i>Export LD_LIBRARY_PATH=\$ LD_LIBRARY_PATH:[path to dbConnect dir]/dbconnect-0.3.x/lib</i>
dbconnect-0.3.x/lib/drivers	Directory contains all the compiled database drivers. The dbconnect.cfg file references these libraries. These libraries are shipped with the application once complete and can be upgraded by newer versions without having to recompile your application.

Compiling for Windows

Microsoft Visual Studio is required for compiling dbConnect on the Windows platforms.

1. Extract the zip file into a directory of your choice using winzip or an appropriate alternative.
2. Edit the dbconnect-0.3.x/windows.conf file. Uncomment the sections for the drivers you wish to compile.
3. Open a command shell.
4. Run the vsvars32.bat file in the command shell to export the required Visual Studio environment.
Example location for vsvars32.bat: C:\Program Files\Microsoft Visual Studio .NET 2003\Common7\Tools\vsvars32.bat
5. Change to the dbconnect-0.3.x directory on the command shell.
6. Build the API
nmake /F Makefile.win
7. Build the examples
nmake /F Makefile.win example

Once the dbConnect API is compiled the following files are available:

dbconnect-0.3.x\include	Directory contains all required header files for application development. Need to add this as a include directory in your application.
dbconnect-0.3.x\lib	Directory contains the main dbConnect library and a sub directory containing the driver .dll files. Can copy the main dbConnect library to windows\system32 if required or add the lib directory to your PATH. <i>dbConnect dir]/dbconnect-0.3.x/lib</i>
dbconnect-0.3.x\lib\drivers	Directory contains all the compiled database drivers. The dbconnect.ini file references these libraries. These libraries are shipped with the application once complete and can be upgraded by newer versions without having to recompile your application.

Chapter 3: C++ API

DbConnect API Classes

Public Types

- DbConnectionDriverInfo
- FieldType

DbConnectionDriverInfo

```
struct DbConnectionDriverInfo
{
    string  author;
    string  vendor;
    string  copyright;
    string  driverType;
    string  driverName;
    string  driverDescription;
    string  dbConnectVersion;
};
```

Holds information returned by the currently loaded driver.

FieldType

```
enum FieldType
{
    FT_UNKNOWN,
    FT_NULL,
    FT_STRING,
    FT_WSTRING, //Wide String for unicode.
    FT_BLOB,
    FT_CLOB,
    FT_DATETIME,
    FT_DOUBLE,
    FT_BIT,
    FT_SHORT,
    FT_LONG,
    FT_UNSIGNED_SHORT,
    FT_UNSIGNED_LONG,
    FT_BOOLEAN
};
```

Defines the internally allocated field type for parameters field values and field information.

DbConnection Class

The DbConnection class is constructed and is normally assigned to a smart pointer. It can be constructed statically if required.

The class manages the database connection pool and thread safe obtaining of a query object.

Public Types and Properties

- Driver

Driver

```
enum Driver
{
    NONE,
    MYSQL,
    MSQL,
    POSTGRESQL,
    DB2,
    ORACLE,
    ODBC
};
```

Constructor

```
DbConnection(
    Driver driver,
    const string &configFile="");
```

Creates an instance of the DbConnection class bound to a specified database driver. The driver .so / .dll for the specified database driver is loaded and all reference are resolved including the references to the database libraries. The .so / .dll to load is obtained from the .cfg / .ini file specified.

If no file is specified, the file "dbconnect.ini" on Windows / "dbconnect.cfg" on Unix is searched for in the following order:

- Searches the current directory
- Searches the system config directory. (c:\windows\system32 on Windows, /etc/dbconn on Unix)

If no config file can be found, the default library name for the database driver will be used if found in the library path of the operating system.

Once the .so / .dll database driver is loaded, it is checked for version compatibility with the DBConnect API.

Parameters

driver	Specific database driver to use. E.g DbConnection::MYSQL
configFile	Full path to .cfg / .ini file to specifically use for its configuration. Leave blank for default searching of configuration file to occur.

Destructor

```
~DbConnection();
```

When the destructor is called, all active connections to the database are terminated including any queries attached to the connection.

Public Methods

- checkCompatibility
- connect
- disconnect
- getDriverInformation
- requestQueryConnection
- setPingInterval

checkCompatibility

```
bool  
checkCompatibility(  
    const string &ver);
```

Checks if a library version is compatible with the current version of dbConnect API.

Note: You can run older versions of driver .so / .dll files with newer versions of the API as long as they are compatible.

Parameters

ver String representing the version of the library. E.g. "0.3.5"

Return

Returns true if version compatible or false if not.

connect

```
void  
connect (  
    const string &username,  
    const string &password="",  
    const string &databaseName="",  
    const string &host="localhost",  
    int maxConnections=1,  
    int minConnections=1,  
    const string &optParam1="",  
    const string &optParam2="");
```

Create the connection pool and connect required connections to the database. A minimum of [minConnections](#) connections will be connected to the database. A maximum of [maxConnections](#) can be connected on demand.

Parameters

Username	The username to authenticate to the database with
password	The password to authenticate to the database with
databaseName	Name of the database connecting to for the driver
host	Host the database resides on. Default "localhost"
maxConnections	Maximum number of connections the database pool can connect. Default 1
minConnections	Maximum number of connections the database pool will hold open. Default 1
optParam1	Driver specific optional parameter that can be used by the driver.
optParam2	Driver specific optional parameter that can be used by the driver.

disconnect

```
void  
disconnect(  
    time_t timeout=120);
```

Disconnect all active database connections.

Parameters

[timeout](#) Timeout in seconds to wait for connections to become idle. Default 120.

getDriverInformation

```
DbConnectionDriverInfo*  
getDriverInformation();
```

Obtains the driver information by querying the loaded .so / .dll driver.

Return

Returns a pointer to the DbConnectionDriverInfo driver information structure.

requestQueryConnection

```
BaseQuery*  
requestQueryConnection();
```

Attaches a connection from the database connection pool to a query object establishing a connection to the database if one is not available and the maximum connections in the database pool has not been exceeded.

The query object can be utilized to perform queries against the database. The connection will be released when the BaseQuery Class object goes out of scope making the database connection available again.

Return

Returns a void pointer to a BaseQuery Class object

setPingInterval

```
void  
setPingInterval(  
    time_t interval=120);
```

```
time_t pingInterval);
```

Set the time interval in seconds that a connection will be checked to make sure it is alive. If the connection is not alive, it will automatically be connected to the database if the database is still available.

Parameters

[pingInterval](#) Interval in seconds. Minimum of 60 seconds

BaseQuery Class

The BaseQuery class is constructed and is normally assigned to a smart pointer. The connection is allocated back to the database connection pool when the object goes out of scope.

The base query class represents a database connection form the database connection pool that has been allocated for querying.

The class provides methods for processing queries against the database. The BaseQuery class represents the common functionality of the actual database driver class instance. Hence all driver classes inherit from the BaseQuery class.

Constructor

```
BaseQuery();
```

Used internally by the DbConnection Class.

Do not construct the class explicitly.

Destructor

```
virtual  
~BaseQuery();
```

Releases the connection back to the database connection pool. If there is a transaction outstanding on the connection, it will automatically be rolled back unless commit was called.

The destructor is normally called when the assigned smart pointer goes out of scope.

Public Methods

- bindParam
- clearBindParams
- command
- commit
- execute
- eof
- fetchNext
- fieldCount
- getFieldByColumn
- getFieldByName
- getFieldInfoByColumn
- getFieldInfoByName
- rollback
- transBegin

bindParam

```
virtual BaseValue*  
    bindParam(  
        const string& paramName) = 0;
```

Returns a reference to the bind parameter that can be used to set or read the value. This is performed after "command" is called and before "execute" is called. An exception will be generated if the bind parameter cannot be found in the SQL specified to the "command" function.

Parameters

paramName Name of the bind parameter specified in the SQL query when using the "command" function.

Return

Returns the BaseValue Class parameter object for the parameter name specified

clearBindParams

```
virtual void  
    clearBindParams() = 0;
```

Clears all bind parameters.

command

```
virtual void  
    command(  
        const string& sqlStatement);
```

Sets the statement. Parameters in the statement can then be set by "bindParam" and the statement executed by "execute".

Parameters

sqlStatement Statement representing any SQL / DLL etc that can be executed against the specific database driver.

SQL/DLL Format: SQL and DDL is specified according to the ENBF of the database it is to be executed against.
E.g. "SELECT myfield FROM mytable"

Parameters: Parameters are specified in the form ":paramname"
E.g. "SELECT myfield FROM mytable WHERE myfield = :myparam"

Functions / Stored Procedures: Specified by name in capital letters.
The parameters to the functions are worked out by the driver, and still need to be bound or a missing parameter error will occur on the "execute"
E.g. "MYFUNCTION"

commit

```
virtual void
```

```
commit() = 0;
```

Commits the current transaction on the connection.

execute

```
virtual void  
execute();
```

Bind parameters are bound to the SQL statement that was specified by "command" and then executed against the database on the connection. Any results from a previous "execute" are lost.

If no transaction has been created, a transaction is automatically created.

eof

```
virtual bool  
eof();
```

If no data was returned from the "execute" then this function will always return true. Otherwise eof will always be false while there is still available records in the record set to be fetched.

The "execute" and "fetchNext" commands will set the value returned by "eof".

Return

Returns true if the end of result set has been reached, else false.

fetchNext

```
virtual void  
fetchNext() = 0;
```

Fetches the next available record in the record set. After an "execute" the record set pointer is positioned before the first record. The data from the first row in a record set can only be retrieved once "fetchNext" has been called.

Note that an exception will be raised if you try to fetch data on an empty result set or a result set that has reached the end of all available rows.

fieldCount

```
virtual unsigned int  
fieldCount();
```

Obtains the number of fields present in the result set after an "execute" command.

Return

Returns the number of fields present in the result set.

getFieldByColumn

```
virtual BaseValue*  
    getFieldByColumn(  
        int index) = 0;
```

Returns a reference to the value that can be used to read the value.
The `index` starts at 0 and ends at `fieldCount - 1`. An exception will be raised in the event an index is specified for a field value that is not present.

Parameters

`index` Index to the field. 0 based.

Return

Returns BaseValue Class in order to read the value from.

getFieldByName

```
virtual BaseValue*  
    getFieldByName(  
        const string& fieldName) = 0;
```

Returns a reference to the value that can be used to read the value.
The `fieldName` must have been present in the SQL query, otherwise an exception is raised.

Parameters

`fieldName` Name of the field.

Return

Returns BaseValue Class in order to read the value from.

getFieldInfoByColumn

```
virtual BaseFieldDescription*  
    getFieldInfoByColumn(  
        int index) = 0;
```

Returns a reference to the value that can be used to query the field details of a field in a record set. An exception will be raised in the event an index is specified for a field value that is not present.

Parameters

`index` Index to the field. 0 based.

Return

Returns a BaseFieldDescription Class in order to read the field information from.

getFieldInfoByName

```
virtual BaseFieldDescription*  
    getFieldInfoByName(  
        const string& fieldName) = 0;
```

Returns a reference to the value that can be used to query the field details of a field in a record set. The `fieldName` must have been present in the SQL query, otherwise an exception is raised.

Parameters

`fieldName` Name of the field.

Return

Returns a `BaseFieldDescription` Class in order to read the field information from.

rollback

```
virtual void  
    rollback() = 0;
```

Rolls back the current transaction on the connection.

transBegin

```
virtual void  
    transBegin() = 0;
```

Creates a new transaction on the connection.

BaseValue Class

The BaseFieldDescription class is constructed and managed by the driver for parameters and result field values.

The base value class provides methods to set and read values as well as the ability to convert between value types such as representing an integer as a string etc.

Constructor

```
BaseValue(  
    const string& fieldName);
```

Used internally by the DbConnection Class.

You can use this class to represent data for applications other than database values if required.

Destructor

```
virtual  
    ~BaseValue();
```

Releases any memory associated with the stored value.

Public Methods

- asBinary
- asBoolean
- asDateTime
- asFloat
- asLong
- asString
- asUnsignedLong
- getSize
- isNull
- name
- setBinary
- setBoolean
- setDateTime
- setDate
- setFloat
- setLong
- setNull
- setString
- setTime
- setUnsignedLong

asBinary

```
virtual void*  
    asBinary();
```

Obtain a pointer to the data. The pointer returned points to a block of data inside the class. It is un-typed and the size of the data can be determined by using the "getSize" function.

Return

Returns a pointer to the data.

asBoolean

```
virtual bool  
    asBoolean();
```

Obtain the data as a Boolean value.

Note: If the original value was not set as a Boolean value, the class will try and convert the internal data to a Boolean type. E.g. The string "true" will return the Boolean value true.

Return

Returns the internal data as a Boolean converting if required.

asDateTime

```
virtual JDate  
    asDateTime();
```

Obtain the data as a date representation.

Note: If the original value was not set as a date/time value, the class will try and convert the internal data a date type. E.g. The long value will be interpreted as a Unix timestamp.

Return

Returns the internal data as a JDate Class representation.

asFloat

```
virtual double  
    asFloat();
```

Obtain the data as a floating point representation.

Note: If the original value was not set as a floating point value, the class will try and convert the internal data a floating point type. E.g. The long value will be interpreted as a floating point value.

Return

Returns the internal data as a floating point representation.

asLong

```
virtual DBLONG  
asLong();
```

Obtain the data as a signed long representation.

Note: If the original value was not set as a signed long value, the class will try and convert the internal data a signed long type. E.g. the date value will be interpreted as a Unix timestamp long value.

Return

Returns the internal data as a signed long representation.

asString

```
virtual const char*  
asString();
```

Obtain the data as string representation.

Note: If the original value was not set as a string value, the class will try and convert the internal data a string type. E.g. the date value will be interpreted as a string in ISO format.

Return

Returns the internal data as a string representation.

asUnsignedLong

```
virtual DBULONG  
asUnsignedLong();
```

Obtain the data as an unsigned long representation.

Note: If the original value was not set as an unsigned long value, the class will try and convert the internal data an unsigned long type. E.g. the date value will be interpreted as a Unix timestamp long value.

Return

Returns the internal data as an unsigned long representation.

getSize

```
virtual DBULONG  
getSize();
```

Obtain the data size of the internally stored data.

E.g. For a string, it will store the string length, for a binary data, it will return the size of the binary data in memory. For the special NULL value, the size will be 0.

Return

Returns the memory size of the data in the value.

isNull

```
virtual bool  
    isNULL();
```

Determine if the internal value is the special NULL value.

Return

Returns true if the internal value represents the special NULL value, otherwise false.

name

```
virtual string  
    name();
```

Obtain the field name.

Return

Returns the name of the field.

setBinary

```
virtual void  
    setBinary(  
        void *bindVar,  
        unsigned long bindVarSize);
```

Set the value as a binary data value.

Note: This function makes a deep copy of the value.

Parameters

bindVar Pointer containing the raw binary data
bvarsize Size of the raw binary data

setBoolean

```
virtual void  
    setBoolean(  
        bool bindVar);
```

Set the value as a Boolean value.

Note: This function makes a deep copy of the value.

Parameters

bindVar Boolean value to set.

setDateTime

```
virtual void  
    setDateTime(  
        JDate &bindVar);
```

Set the value as a date time value.

Note: This function makes a deep copy of the value.

Parameters

bindVar JDate Class value to set.

setDate

```
virtual void  
    setDate(  
        JDate &bindVar);
```

Set the value as a date value.

Note: This function makes a deep copy of the value.

Parameters

bindVar JDate Class value to set.

setFloat

```
virtual void  
    setFloat(  
        double bindVar);
```

Set the value as a floating point value.

Note: This function makes a deep copy of the value.

Parameters

bindVar Floating point value to set.

setLong

```
virtual void  
    setLong(  
        DBLONG bindVar);
```

Set the value as a long value.

Note: This function makes a deep copy of the value.

Parameters

bindVar Long value to set.

setNull

```
virtual void  
    setNULL();
```

Set the value as the special NULL value.

setString

```
virtual void  
    setString(  
        const string &bindValue);
```

Set the value as a string value.

Note: This function makes a deep copy of the value.

Parameters

bindValue String value to set.

setTime

```
virtual void  
    setTime(  
        JDate &bindValue);
```

Set the value as a time value.

Note: This function makes a deep copy of the value.

Parameters

bindValue JDate Class value to set.

setUnsignedLong

```
virtual void  
    setUnsignedLong(  
        DBULONG bindValue);
```

Set the value as an unsigned long value.

Note: This function makes a deep copy of the value.

Parameters

bindValue Unsigned long value to set.

BaseFieldDescription Class

The BaseFieldDescription class is constructed and managed by the driver.

The base field description class represents contains all the information for a field in a result set.

Constructor(s)

```
BaseFieldDescription();
```

```
BaseFieldDescription(  
    string&    name,  
    long int   position,  
    FieldType type,  
    bool       isIncrement = false,  
    bool       isPriKey    = false,  
    bool       isUnique    = false,  
    bool       isNotNull   = false,  
    long int   precision   = 0,  
    long int   scale       = 0);
```

Used internally by the database driver.

Do not construct the class explicitly.

Destructor

```
virtual  
~BaseFieldDescription();
```

Release the class and information associated by the field.

Used internally by the database driver.

Public Methods

- isIncrement
- isNotNull
- isPriKey
- isUnique
- name
- position
- precision
- scale
- type

isIncrement

```
bool
```

```
isIncrement();
```

Determine if the field is an auto-increment field.
E.g. The SERIAL field in PostgreSQL will return true here.

Note: Not all databases support this functionality and in the event it is not supported will return false. See the driver specific information for more details.

Return

Returns true if the field is an auto increment field, otherwise false

isNotNull

```
bool  
isNotNull();
```

Determine if the field has a NOT NULL constraint.

Note: Not all databases support this functionality and in the event it is not supported will return false. See the driver specific information for more details.

Return

Returns true if the field is an has a NOT NULL constraint, otherwise false

isPriKey

```
bool  
isPriKey();
```

Determine if the field is part of the primary key.

Note: Not all databases support this functionality and in the event it is not supported will return false. See the driver specific information for more details.

Return

Returns true if the field is part of the table primary key, otherwise false

isUnique

```
bool  
isUnique();
```

Determine if the field is part of a unique constraint.

Note: Not all databases support this functionality and in the event it is not supported will return false. See the driver specific information for more details.

Return

Returns true if the field is part of a unique constraint, otherwise false

name

```
const string&  
name();
```

Obtain the name of the field.

Return

Returns the name of the field.

position

```
long int  
position();
```

Obtain the field position in the result set.

Return

Returns the field position in the result set.

precision

```
long int  
precision();
```

Obtain the precision of the field.

Note: Not all databases support this functionality and in the event it is not supported will return 0. See the driver specific information for more details.

Return

Returns the precision of the field.

scale

```
long int  
scale();
```

Obtain the scale of the field.

Note: Not all databases support this functionality and in the event it is not supported will return 0. See the driver specific information for more details.

Return

Returns the scale of the field.

type

```
FieldType  
type();
```

Obtain the internal field type mapping for the result set field.

Return

Returns the FieldType enumeration to determine the internal type the field has been mapped to.

Utility API Classes

The utility classes are classes that dbConnect depend on. The classes however are available for use in your application as required.

JDate Class

ConfigFile / Config Section Class

DILoader Class

HexDigest Namespace

Exceptions

All exceptions are inherited from the BaseException Class. This allows for the trapping of a specific exception or the trapping of a generic exception obtaining the details from the properties of the exception in a uniform manner.

BaseException Class

The base exception class provides basic information on what errors where generated. It is recommended to use the base exception class for all other custom exceptions as it provides an easy to use and common method of exception handling allowing the trapping of specific exceptions or the generic handling of exceptions.

The class overloads the char*, const char* and string operators and returns the error description.

Public Types and Properties

```
Int      code;  
string   name;  
string   description;  
time_t   when;
```

Properties

code	Error code of the exception. Each exception has its own unique error code.
name	Unique exception name.
description	Description of the error generated.
when	The Unix timestamp of when the error occurred.

Constructor

```
BaseException(  
    const int _code=-1,  
    const string &_name="Unknown",  
    const string &_description="",  
    const time_t _when=time(NULL));
```

Constructs a new exception.

Parameters

<code>_code</code>	The error code of the exception
<code>_name</code>	The unique exception name
<code>_description</code>	The description for the error being generated
<code>_when</code>	The timestamp of when the error occurred

Destructor

```
~BaseException();
```

Destructor of the object.

dbConnect Exceptions

All exceptions are inherited from the base exception class.

- Error
- ErrorQuerying
- DriverError
- NotImplemented
- UnknownException
- NotConnected
- AlreadyConnected
- ErrorConnecting
- QueryConnectionTimeout
- ErrorPingingConnection
- IndexOutOfRange
- NameNotFound
- EndOfResultSet
- BindParameterError
- BindParameterNotPresent
- BindParameterNotSet
- TransactionError
- CommandError
- ResultSetError

Chapter 4: Drivers

IBM DB2

Overview

IBM DB2 is a commercial database. It is a great alternative to Oracle if you find Oracle over complex and difficult to maintain. DB2 is easy to install, configure and maintain and is one of my personal favourite databases from a configuration and performance perspective.

The IBM DB2 database driver supports version 8.x of the IBM DB2 database.

Unix Configuration Parameters

The following configuration parameters are available when configuring the dbConnect API for compiling:

<code>--enable-db2</code>	Enables the building of IBM DB2 database driver
<code>--with-db2_include</code>	IBM DB2 include files (Default /home/db2inst1/sqllib/include)
<code>--with-db2_lib</code>	IBM DB2 library files (Default /home/db2inst1/sqllib/lib)

MySQL

Overview

MySQL is an open source database with optional commercial support. It is light weight, performs well but trades speed for lack of functionality compared to commercial databases.

The MySQL database driver supports version 3.x (no transactions) and version 4.x (transactions if using INNODB tables) of the MySQL database.

Unix Configuration Parameters

The following configuration parameters are available when configuring the dbConnect API for compiling:

<code>--enable-mysql</code>	Enables the building of MySQL database driver
<code>--with-mysql_include</code>	MySQL include files (Default /usr/include/mysql)
<code>--with-mysql_lib</code>	MySQL library files (Default /usr/lib/mysql/lib)

Windows Configuration

The following parameters are available in the windows.conf file:

<code>MYSQL</code>	Uncomment to enable the MySQL driver
<code>MYSQL_HOME</code>	The directory where MySQL is installed
<code>MYSQL_INCLUDE</code>	MySQL include files
<code>MYSQL_LIB</code>	MySQL lib file to link against

PostgreSQL-7

Overview

PostgreSQL, often referred to as the "poor mans Oracle" has great functionality and adequate performance. Not as fast as MySQL, it offers all the functionality you would expect from a commercial database.

The PostgreSQL-7 database driver supports version 7.x of the PostgreSQL database.

Unix Configuration Parameters

The following configuration parameters are available when configuring the dbConnect API for compiling:

--enable-pgsql	Enables the building of PostgreSQL database driver
--with-pgsql_include	PostgreSQL include files (Default /usr/include)
--with-pgsql_lib	PostgreSQL library files (Default /usr/lib)

Windows Configuration

The following parameters are available in the windows.conf file:

PGSQL	Uncomment to enable the PostgreSQL driver
PGSQL_HOME	The directory where PostgreSQL client library is installed
PGSQL_INCLUDE	PostgreSQL include files
PGSQL_LIB	PostgreSQL lib file to link against

mSQL

Overview

mSQL, which is apparently where MySQL was derived from, is one of the fastest performing databases with the least amount of functionality.

The mSQL database driver supports version 3.x of the mSQL database.

Unix Configuration Parameters

The following configuration parameters are available when configuring the dbConnect API for compiling:

--enable-mSQL	Enables the building of mSQL database driver
--with-msql_include	mSQL include files (Default /usr/local/msql3/include)
--with-msql_lib	mSQL library files (Default /usr/local/msql3/lib)