The NCBI C++ Toolkit Book
The NCBI C++ Toolkit Book

The NCBI C++ Toolkit

10: Database Access - SQL, Berkley DB

Last Update: July 31, 2013.

Overview

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

Introduction

Database Access [Library dbapi: include | src]

The DBAPI library provides the underlying user-layer and driver API for the NCBI database connectivity project. The project's goal is to provide access to various relational database management systems (RDBMS) with a single uniform user interface. Consult the detailed documentation for details of the supported DBAPI drivers.

The BDB library is part of the NCBI C++ Toolkit and serves as a high-level interface to the Berkeley DB. The primary purpose of the library is to provide tools for work with flatfile, federated databases. The BDB library incorporates a number of Berkeley DB services; for a detailed understanding of how it works, study the original Berkeley DB documentation from http://www.oracle.com/database/berkeley-db/db/. The BDB library is compatible with Berkeley DB v. 4.1 and higher. The BDB library, as it is right now, is architecturally different from the dbapi library and does not follow its design. The BDB is intended for use by software developers who need small-footprint, high-performance database capabilities with zero administration. The database in this case becomes tightly integrated with the application code.

Chapter Outline

The following is an outline of the topics presented in this chapter:

DBAPI Overview

DBAPI is a consistent, object-oriented programming interface to multiple back-end databases. It encapsulates leading relational database vendors' APIs and is universal for all applications regardless of which database is used. It frees developers from dealing with the low-level details of a particular database vendor's API, allowing them to concentrate on domain-specific issues and build appropriate data models. It allows developers to write programs that are reusable with many different types of relational databases and to drill down to the native database APIs for added control when needed.

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

DBAPI has open SQL interface. It takes advantage of database-specific features to maximize performance and allows tight control over statements and their binding and execution semantics.

DBAPI has "Native" Access Modules for Sybase, Microsoft SQL Server, SQLITE, and ODBC. It provides native, high-performance implementations for supported vendor databases. It allows porting to other databases with minimal code changes.

DBAPI is split into low-layer and user-layer.

In addition, a simplified C++ API (SDBAPI) layer is provided for cases where the full DBAPI feature set is not required.

See the DBAPI configuration parameters reference for details on configuring the DBAPI library.

See the DBAPI sample programs for example code:

- http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/src/sample/app/dbapi/

Security

Preventing SQL Injection

Much has been written about networked database security - in particular, SQL injection. Please see the common resources for more information.

When using DBAPI or SDBAPI, the two most important rules for protecting against SQL injection are:

1. Never construct a SQL statement from user-supplied input if the same functionality can be achieved by passing the user input to stored procedures or parameterized SQL.
2. If constructing a SQL statement from user-supplied input cannot be avoided, then you MUST sanitize the user input.

The following sample programs illustrates how to protect against SQL injection for basic SQL statements using SDBAPI and DBAPI:

- http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/src/sample/app/sdbapi/sdbapi_simple.cpp
- http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/src/sample/app/dbapi/dbapi_simple.cpp

See the Security FAQ for more information.

Using Kerberos with DBAPI

Individual users (i.e. not service accounts) within NCBI can use Kerberos with DBAPI, provided the following conditions are met:

1. The database must allow them to connect using Kerberos. (Email dbhelp@ncbi.nlm.nih.gov if you need help with this.)
2. DBAPI must be configured to enable Kerberos.
    a. Either the NCBI_CONFIG__DBAPI__CAN_USE_KERBEROS environment variable must be set to true; or
    b. the can_use_kerberos entry in the dbapi section of the application configuration file must be set to true.

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

3 Their Kerberos ticket must not be expired.

4 They must pass an empty string for the user name.

This is also covered in the DBAPI section of the Library Configuration chapter.

Simple Database Access via C++

This section shows how to execute a simple static SQL query using the simplified database API (SDBAPI). Note that database load-balancing is performed automatically and transparenty when using SDBAPI.

C++ source files using SDBAPI should contain:

```
#include <dbapi/simple/sdbapi.hpp>
```

Application makefiles should contain:

```
LIB = $(SDBAPI_LIB) xconnect xutil xncbi
LIBS = $(SDBAPI_LIBS) $(NETWORK_LIBS) $(DL_LIBS) $(ORIG_LIBS)
```

This example connects to a load-balanced service:

```
CDatabase db("dbapi://" + my_username + ":" + my_password +
 "@" + my_servicename + "/" + my_dbname);
db.Connect();
CQuery query = db.NewQuery();
query.SetSql("select title from Journal");
query.Execute();
ITERATE(CQuery, it, query.MultiSet()) {
 string col1 = it[1].AsString(); // Note: uses 1-based index !
 NcbiCout << col1 << NcbiEndl;
}
```

Note: SDBAPI always uses load balancing - you don't have to call DBLB_INSTALL_DEFAULT().

See the SDBAPI sample programs for more example code:

- http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/src/sample/app/sdbapi/

Database Load-Balancing (DBLB)

Setting up Load-Balancing of Database Servers

For the following to be clear, it is important to distinguish between a database name, an underlying (actual) server name (e.g. MSSQL17), which hosts a variety of databases, a database server alias, and a service name. A server alias may be moved to a different underlying server. The server alias is often used with sqsh, and the GUI tools, such as SQL Management studio. The service name is used by the load-balancer to look up the underlying server to use, and is the name that should be used by an application. The server aliases and service names often share a common prefix and would look similar, and in fact for reasons presented below, there should be at least one server alias that is identical to the service name.

The following steps must be done prior to database load-balancing:

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

1. Ask the DBAs to add your service name (e.g. YOURSERVICE) to the load-balancer configuration database. Typically, the names are clear, for example, there are server aliases YOURSERVICE1, and YOURSERVICE2 that already exist, and databases that have "YOURSERVICE" as an embedded string, but if not, the databases providing the service and the server aliases involved should be given. Note that if databases are moved to different underlying servers, both the server aliases, and the load-balancer configuration which points to those servers are both moved, synchronously.
2. Tell the DBAs which of the server aliases point to the server that should be used, if the load-balancer is unavailable, as the DBAPI will look for a server alias with the same name as the service, in that case.
3. The DBAs will also ask for a DNS name to match the service name as a backup connection method, should everything else fail.

Using Database Load-Balancing from C++

For simplest access, see the section on using SDBAPI above. SDBAPI uses the database load-balancing by default.

If more flexibility is required, and you want to activate the database load-balancing for the more general NCBI DBAPI:

1. Before the very first DBAPI connection attempt, call:

   #include <dbapi/driver/dbapi_svc_mapper.hpp>
   DBLB_INSTALL_DEFAULT();
2. Link '$(XCONNEXT)' and 'xconnect' libraries to your application.

If steps (1) and (2) above are done then the DBAPI connection methods (such as Connect() or ConnectValidated()) will attempt to resolve the passed server name as a load-balanced service name.

Note: If steps (1) and (2) above are not done, or if DBLB library is not available (such as in the publicly distributed code base), or if the passed server name cannot be resolved as a load-balanced service name, then the regular database server name resolution will be used – i.e. the passed name will first be interpreted as a server alias (using the "interfaces" file), and if that fails, it will be interpreted as a DNS name. Note however that by default if the service name resolves (exists), then the regular database server name resolution will not be used as a fallback, even if DBAPI can't connect (for whatever reason) to the servers that the service resolves to.

Example:

```
#include <dbapi/driver/dbapi_svc_mapper.hpp>

DBLB_INSTALL_DEFAULT();
IDataSource* ds = dm.CreateDs("ftds");
IConnection* conn = ds->CreateConnection();

// (Use of validator here is optional but generally encouraged.)
CTrivialConnValidator my_validator(my_databasename);

conn->ConnectValidated(my_validator, my_username, my_password,
my_servicename);
```

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

Load-Balanced Database Access via Python and Perl

Load-balanced database access can be achieved from scripts by using the utility ncbi_dblb, which is located under /opt/machine/lbsm/bin:

Here are some sample scripts that demonstrate using ncbi_dblb to retrieve the server name for a given load-balanced name:

From Python:

```
#!/usr/bin/env python

import subprocess, sys

if len(sys.argv) > 1:
 # Use the -q option to fetch only the server name.
 cmd = ['/opt/machine/lbsm/bin/ncbi_dblb', '-q', sys.argv[1]]
 srv = subprocess.Popen(cmd, stdout=subprocess.PIPE).communicate()[0].strip()
 # Do whatever is needed with the server name...
 print 'Server: "' + srv + '"'
```

From Perl:

```
#!/usr/bin/env perl -w

use strict;

if (@ARGV) {
 # Use the -q option to fetch only the server name.
 my $cmd = '/opt/machine/lbsm/bin/ncbi_dblb -q ' . $ARGV[0];
 my $srv = `$cmd`; chomp($srv);
 # Do whatever is needed with the server name...
 print 'Server: "' . $srv . '"'
}
```

Advantages of using DBLB

C++ Specific

- A database-level verification mechanism.
- Latch onto the same database server for the life of your process. It's often useful to avoid possible inter-server data discrepancy. The "latch-on" mechanism can be relaxed or turned off if needed.
- Automatic connection retries. If a connection to the selected server cannot be established, the API will try again with other servers (unless it is against the chosen "latch-on" strategy).
- The default connection strategy is *configurable*. You can change its parameters using a configuration file, environment variables, and/or programmatically. You can also configure locally for your application ad-hoc mappings to the database servers (this is usually not recommended but can come in handy in emergency cases or for debugging).
- If needed, you can implement your own customized mapper. Components of the default connection strategy can be used separately, or in combination with each other and with the user-created strategies, if necessary.

General

- Connecting to the database servers by server name and/or "interfaces" file based aliases still works the same as it used to.
- Automatic avoidance of unresponsive database servers. This prevents your application from hanging for up to 30 seconds on the network timeout.
- Independence from the database "interfaces" file. A centrally maintained service directory is used instead, which is accessible locally and/or via network. It also dynamically checks database servers' availability and excludes unresponsive servers.

How it works (by default)

The following steps are performed each time a request is made to establish a load-balanced connection to a named database service:

1. The requests will first go through the DBLB mechanism that tries to match the requested service name against the services known to the NCBI Load Balancer and/or those described in the application's configuration file.
2. If the requested service name is unknown to the load balancer then this name will be used "as is".
3. However, if this service name is known to the DBLB then the DBLB will try to establish a connection to the database server that it deems the most suitable. If the service is handled by the NCBI load-balancer, then the unresponsive servers will be weeded out, and a load on the machines that run the servers may be taken into account too.
4. C++ only: If the connection cannot be established, then DBLB will automatically retry the connection, now using another suitable database server.
5. This procedure may be repeated several times, during which there will be only one attempt to connect to each database.
6. C++ only: Once a database connection is successfully established it will be "latched-on". This means that when you will try to connect to the same service or alias within the same application again then you will be connected to the same database server (this can be relaxed or turned off completely).
7. For example, you can connect to the "PMC" service which is currently mapped to two servers. The server names are provided dynamically by the NCBI load-balancer, so you never have to change your configuration or recompile your application if either a service configuration or an "interfaces" file get changed.
8. C++ only: If ConnectValidated() is used to connect to a database, then requests to establish database connections will first go through the server-level load-balancing mechanism. On successful login to server, the database connection will be validated against the validator. If the validator does not "approve" the connection, then DBAPI will automatically close this connection and repeat this login/validate attempt with the next server, and so on, until a "good" (successful login + successful validation) connection is found. If you want to validate a connection against more than one validator/database, then you can combine validators. Class CConnValidatorCoR was developed to allow combining of other validators into a chain.

NCBI DBAPI User-Layer Reference

Object hierarchy

See Figure 1.

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

Includes

For most purposes it is sufficient to include one file in the user source file: dbapi.hpp.

```
#include <dbapi/dbapi.hpp>
```

For static linkage the following include file is also necessary:

```
#include <dbapi/driver/drivers.hpp>
```

Objects

All objects are returned as pointers to their respective interfaces. The null (0) value is valid, meaning that no object was returned.

Object Life Cycle

In general, any child object is valid only in the scope of its parent object. This is because most of the objects share the same internal structures. There is no need to delete every object explicitly, as all created objects will be deleted upon program exit. Specifically, all objects are derived from the static CDriverManager object, and will be destroyed when CDriverManager is destroyed. It is possible to delete any object from the framework and it is deleted along with all derived objects. For example, when an IConnection object is deleted, all derived IStatement, ICallableStatement and IResultSet objects will be deleted too. If the number of the objects (for instance IResultSet) is very high, it is recommended to delete them explicitly or enclose in the auto_ptr<...> template. For each object a Close() method is provided. It disposes of internal resources, required for the proper library cleanup, but leaves the framework intact. After calling Close() the object becomes invalid. This method may be necessary when the database cleanup and framework cleanup are performed in different places of the code.

CVariant Type

The CVariant type is used to represent any database data type (except BLOBs). It is an object, not a pointer, so it behaves like a primitive C++ type. Basic comparison operators are supported (==, !=, <) for identical internal types. If types are not identical, CVariantException is thrown. CVariant has a set of getters to extract a value of a particular type, e.g. GetInt4(), GetByte(), GetString(), etc. If GetString() is called for a different type, like DateTime or integer it tries to convert it to a string. If it doesn't succeed, CVariantException is thrown. There is a set of factory methods (static functions) for creating CVariant objects of a particular type, such as CVariant::BigInt(), CVariant::SmallDateTime(), CVariant::VarBinary() etc. For more details please see the comments in variant.hpp file.

Related sample code:

- http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/src/dbapi/test/dbapi_unit_test_object.cpp

Choosing the Driver

There are several drivers for working with different SQL servers on different platforms. The ones presently implemented are "ctlib" (Sybase), "dblib"(MS SQL, Sybase), "ftds" (MS SQL, Sybase, cross platform). For static linkage these drivers should be registered manually; for dynamic linkage this is not necessary. The CDriverManager object maintains all registered drivers.

```
DBAPI_RegisterDriver_CTLIB();
DBAPI_RegisterDriver_DBLIB();
```

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

Related sample code:

- http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/src/dbapi/test/dbapi_unit_test_context.cpp

Data Source and Connections

The IDataSource interface defines the database platform. To create an object implementing this interface, use the method CreateDs(const string& driver). An IDataSource can create objects represented by an IConnection interface, which is responsible for the connection to the database. It is highly recommended to specify the database name as an argument to the CreateConnection() method, or use the SetDatabase() method of a CConnection object instead of using a regular SQL statement. In the latter case, the library won't be able to track the current database.

```
IDataSource *ds = dm.CreateDs("ctlib");
IConnection *conn = ds->CreateConnection();
conn->Connect("user", "password", "server", "database");
IStatement *stmt = conn->CreateStatement();
```

Every additional call to IConnection::CreateStatement() results in cloning the connection for each statement. These connections inherit the same default database, which was specified in the Connect() or SetDatabase() method. Thus if the default database was changed by calling SetDatabase(), all subsequent cloned connections created by CreateStatement() will inherit this particular default database.

Related sample code:

- http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/src/sample/app/dbapi/dbapi_simple.cpp
- http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/src/dbapi/test/dbapi_unit_test_connection.cpp

Main Loop

The library simulates the main result-retrieving loop of the Sybase client library by using the IStatement::HasMoreResults() method:

```
stmt->Execute("select * from MyTable");
while( stmt->HasMoreResults() ) {
 if( stmt->HasRows() ) {
 IResultSet *rs = stmt->GetResultset();

 // Retrieve results, if any

 while( rs->Next() ) {
 int col1 = rs->GetVariant(1).GetInt4();
 ...
 }
 }
}
```

This method should be called until it returns false, which means that no more results are available. It returns as soon as a result is ready. The type of the result can be obtained by calling the IResultSet::GetResultType() method. Supported result types are eDB_RowResult,

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

eDB_ParamResult, eDB_ComputeResult, eDB_StatusResult, eDB_CursorResult. The method IStatement::GetRowCount() returns the number of updated or deleted rows.

The IStatement::ExecuteUpdate() method is used for SQL statements that do not return rows:

```
stmt->ExecuteUpdate("update...");
int rows = stmt->GetRowCount();
```

The method IStatement::GetResultSet() returns an IResultSet object. The method IResultSet::Next() actually does the fetching, so it should be always called first. It returns false when no more fetch data is available. All column data, except Image and Text is represented by a single CVariant object. The method IResultSet::GetVariant() takes one parameter, the column number. Column numbers start with 1.

Related sample code:

- http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/src/sample/app/dbapi/dbapi_simple.cpp
- http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/src/dbapi/test/dbapi_unit_test_stmt.cpp

Input and Output Parameters

The method ICallableStatement::SetParam(const CVariant& v, const string& name) is used to pass parameters to stored procedures and dynamic SQL statements. To ensure the correct parameter type it is recommended to use CVariant type factories (static methods) to create a CVariant of the required internal type. There is no internal representation for the BIT parameter type, please use TinyInt of Int types with 0 for false and 1 for true respectively. Here are a few examples: CVariant::Int4(Int4 *p), CVariant::TinyInt(UInt1 *p), CVariant::VarChar(const char *p, size_t len) etc.

There are also corresponding constructors, like CVariant::CVariant(Int4 v), CVariant::CVariant(const string& s), ..., but the user must ensure the proper type conversion in the arguments, and not all internal types can be created using constructors.

Output parameters are set by the ICallableStatement::SetOutputParam(const CVariant& v, const string& name) method, where the first argument is a null CVariant of a particular type, e.g. SetOutputParam(CVariant(eDB_SmallInt),"@arg").

Related sample code:

- http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/src/sample/app/dbapi/dbapi_simple.cpp

Stored Procedures

The ICallableStatement object is used for calling stored procedures. First get the object itself by calling IConnection::PrepareCall(). Then set any parameters. If the parameter name is empty, the calls to SetParam() should be in the exact order of the actual parameters. Retrieve all results in the main loop. Get the status of the stored procedure using the ICallableStatement::GetReturnStatus() method.

```
ICallableStatement *cstmt = conn->PrepareCall("ProcName");
Uint1 byte = 1;
cstmt->SetParam(CVariant("test"), "@test_input");
cstmt->SetParam(CVariant::TinyInt(&byte), "@byte");
```

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

```
cstmt->SetOutputParam(CVariant(eDB_Int), "@result");
cstmt->Execute();
while(cstmt->HasMoreResults()) {
 if( cstmt->HasRows() ) {
 IResultSet *rs = cstmt->GetResultSet();
 switch( rs->GetResultType() ) {
 case eDB_RowResult:
 while(rs->Next()) {

 // retrieve row results

 }
 break;
 case eDB_ParamResult:
 while(rs->Next()) {

 // Retrieve parameter row

 }
 break;
 }
 }
}

// Get status
int status = cstmt->GetReturnStatus();
```

It is also possible to use IStatement interface to call stored procedures using standard SQL language call. The difference from ICallableStatement is that there is no SetOutputParam() call. The output parameter is passed with a regular SetParam() call having a *non-null* CVariant argument. There is no GetReturnStatus() call in IStatement, so use the result type filter to get it - although note that result sets with type eDB_StatusResult are not always guaranteed to be returned when using the IStatement interface.

```
sql = "exec SampleProc @id, @f, @o output";
stmt->SetParam(CVariant(5), "@id");
stmt->SetParam(CVariant::Float(&f), "@f");
stmt->SetParam(CVariant(5), "@o");
stmt->Execute(sql);
while(stmt->HasMoreResults()) {
 IResultSet *rs = stmt->GetResultSet();

 if( rs == 0 )
 continue;

 switch( rs->GetResultType() ) {
 case eDB_ParamResult:
 while( rs->Next() ) {
 NcbiCout << "Output param: "
 << rs->GetVariant(1).GetInt4()
 << NcbiEndl;
 }
```

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

```
 break;
 case eDB_StatusResult:
 while( rs->Next() ) {
 NcbiCout << "Return status: "
 << rs->GetVariant(1).GetInt4()
 << NcbiEndl;
 }
 break;
 case eDB_RowResult:
 while( rs->Next() ) {
 if( rs->GetVariant(1).GetInt4() == 2121 ) {
 NcbiCout << rs->GetVariant(2).GetString() << "|"
 << rs->GetVariant(3).GetString() << "|"
 << rs->GetVariant(4).GetString() << "|"
 << rs->GetVariant(5).GetString() << "|"
 << rs->GetVariant(6).GetString() << "|"
 << rs->GetVariant(7).GetString() << "|"
 << NcbiEndl;
 } else {
 NcbiCout << rs->GetVariant(1).GetInt4() << "|"
 << rs->GetVariant(2).GetFloat() << "|"
 << rs->GetVariant("date_val").GetString() << "|"
 << NcbiEndl;
 }
 }
 break;
 }
}
stmt->ClearParamList();
```

Related sample code:

- http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/src/sample/app/dbapi/dbapi_simple.cpp
- http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/src/dbapi/test/dbapi_unit_test_proc.cpp

Cursors

The library currently supports basic cursor features such as setting parameters and cursor update and delete operations.

```
ICursor *cur = conn->CreateCursor("table_cur",
 "select ... for update of ...");
IResultSet *rs = cur->Open();
while(rs->Next()) {
 cur->Update(table, sql_statement_for_update);
}
cur->Close();
```

Related sample code:

- http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/src/dbapi/test/dbapi_unit_test_cursor.cpp

Working with BLOBs

Due to the possibly very large size, reading and writing BLOBs requires special treatment. During the fetch the contents of the whole column must be read before advancing to the next one. That's why the columns of type IMAGE and TEXT are not bound to the corresponding variables in the resultset and all subsequent columns are not bound either. So it is recommended to put the BLOB columns at the end of the column list. There are several ways to read BLOBs, using IResultSet::Read(), IResultSet::GetBlobIStream(), and IResultSet::GetBlobReader() methods. The first is the most efficient; it reads data into a supplied buffer until it returns 0 bytes read. The next call will read from the next column. The second method implements the STL istream interface. After each successful column read you should get another istream for the next column. The third implements the C++ Toolkit IReader interface. If the data size is small and double buffering is not a performance issue, the BLOB columns can be bound to CVariant variables by calling IResultSet::BindBlobToVariant(true). In this case the data should be read using CVariant::Read() and CVariant::GetBlobSize(). To write BLOBs there are also several options. To pass a BLOB as a SQL parameter you should store it in a CVariant using CVariant::Append() and CVariant::Truncate() methods. To store a BLOB in the database you should initialize this column first by writing a zero value (0x0) for an IMAGE type or a space value (' ') for a TEXT type. After that you can open a regular IResultSet or ICursor and for each required row update the BLOB using IResultSet::GetBlobOStream(). NOTE: this call opens an additional connection to the database.

Related sample code:

- http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/src/dbapi/test/dbapi_unit_test_lob.cpp

Updating BLOBs Using Cursors

It is recommended to update BLOBs using cursors, because no additional connections are opened and this is the only way to work with ODBC drivers.

```
ICursor *blobCur = conn->CreateCursor("test",
 "select id, blob from BlobSample for update of blob");
IResultSet *blobRs = blobCur->Open();
while(blobRs->Next()) {
 ostream& out = blobCur->GetBlobOStream(2, blob.size());
 out.write(buf, blob.size());
 out.flush();
}
```

Note that GetBlobOStream() takes the column number as the first argument and this call is invalid until the cursor is open.

Using Bulk Insert

Bulk insert is useful when it is necessary to insert big amounts of data. The IConnection::CreateBulkInsert() takes one parameter, the table name. The number of columns is determined by the number of Bind() calls. The CVariant::Truncate(size_t len) method truncates the internal buffer of CDB_Text and CDB_Image object from the end of the buffer. If no parameter specified, it erases the whole buffer.

```
NcbiCout << "Initializing BlobSample table..." << NcbiEndl;
IBulkInsert *bi = conn->CreateBulkInsert(tbl_name);
CVariant col1 = CVariant(eDB_Int);
CVariant col2 = CVariant(eDB_Text);
bi->Bind(1, &col1);
bi->Bind(2, &col2);
for(int i = 0; i < ROWCOUNT; ++i ) {
 string im = "BLOB data " + NStr::IntToString(i);
 col1 = i;
 col2.Truncate();
 col2.Append(im.c_str(), im.size());
 bi->AddRow();
}
bi->Complete();
```

Related sample code:

- http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/src/dbapi/test/dbapi_unit_test_bcp.cpp

Diagnostic Messages

The DBAPI library is integrated with the C++ Toolkit diagnostic and tracing facility. By default all client and server messages are handled by the Toolkit's standard message handler. However it is possible to redirect the DBAPI-specific messages to a single CDB_MultiEx object and retrieve them later at any time. There are two types of redirection, per data source and per connection. The redirection from a data source is enabled by calling IDataSource::SetLogStream(0). After the call all client- and context-specific messages will be stored in the IDataSource object. The IDataSource::GetErrorInfo() method will return the string representation of all accumulated messages and clean up the storage. The IDataSource::GetErrorAsEx() will return a pointer to the underlying CDB_MultiEx object. Retrieving information and cleaning up is left to the developer. Do NOT delete this object. The connection-specific redirection is controlled by calling IConnection::MsgToEx(boolean enable) method. This redirection is useful; for instance, to temporarily disable default messages from the database server. The IConnection::GetErrorInfo() and IConnection::GetErrorAsEx() methods work in the same manner as for the IDataSource

Related sample code:

- http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/src/dbapi/test/dbapi_unit_test_msg.cpp

Trace Output

The DBAPI library uses the Toolkit-wide DIAG_TRACE environment variable to do the debug output. To enable it set it to any value. If you have any problems with the DBAPI please include the trace output into your email.

NCBI DBAPI Driver Reference

(Low-level access to the various RDBMSs.)

- NCBI DBAPI Driver Reference
  - Overview
  - The driver architecture

Overview

The NCBI DBAPI driver library describes and implements a set of objects needed to provide a uniform low-level access to the various relational database management systems (RDBMS). The basic driver functionality is the same as in most other RDBMS client APIs. It allows opening a connection to a server, executing a command (query) on this connection and getting the results back. The main advantage of using the driver is that you don't have to change your own upper-level code if you need to move from one RDBMS client API to another.

The driver can use two different methods to access the particular RDBMS. If the RDBMS provides a client library for the given computer system (e.g. Sun/Solaris), then the driver uses that library. If no such client library exists, then the driver connects to an RDBMS through a special gateway server which is running on a computer system where such a library does exist.

The driver architecture

There are two major groups of the driver's objects: the RDBMS-independent objects, and the objects which are specific to a particular RDBMS. The only RDBMS-specific object which user should be aware of is a "Driver Context". The "Driver Context" is effectively a "Connection" factory. The only way to make a connection to the server is to call the Connect () method of a "Driver Context" object. So, before doing anything with an RDBMS, you need to create at least one driver context object. All driver contexts implement the same interface defined in I_DriverContext class. If you are working on a library which could be used with more than one RDBMS, the driver context should not be created by the library. Instead, the library API should include a pointer to I_DriverContext so an existing driver context can be passed in.

There is no "real" factory for driver contexts because it's not always possible to statically link the RDBMS libraries from different vendors into the same binary. Most of them are written in C and name collisions do exist. The Driver Manager helps to overcome this problem. It allows creating a mixture of statically linked and dynamically loaded drivers and using them together in one executable.

The driver context creates the connection which is RDBMS-specific, but before returning it to the caller it puts it into an RDBMS-independent "envelope", CDB_Connection. The same is true for the commands and for the results - the user gets the pointer to the RDBMS-independent "envelope object" instead of the real one. It is the caller's responsibility to delete those objects. The life spans of the real object and the envelope object are not necessarily the same.

Once you have the connection object, you could use it as a factory for the different types of commands. The command object in turn serves as a factory for the results. The connection is always single threaded, that means that you have to execute the commands and process their results sequentially one by one. If you need to execute the several commands in parallel, you can do it using multiple connections.

Another important part of the driver is error and message handling. There are two different mechanisms implemented. The first one is exceptions. All exceptions which could be thrown by the driver are inherited from the single base class CDB_Exception. Drivers use the exception mechanism whenever possible, but in many cases the underlying client library uses callbacks or handlers to report error messages rather than throwing exceptions. The driver supplies a handler's stack mechanism to manage these cases.

To send and to receive the data through the driver you have to use the driver provided datatypes. The collection of the datatypes includes: one, two, four and eight byte integers; float and double; numeric; char, varchar, binary, varbinary; datetime and smalldatetime; text and image. All datatypes are derived from a single base class CDB_Object.

Sample program

This program opens one connection to the server and selects the database names and the date when each database was created (assuming that table "sysdatabases" does exist). In this example the string "XXX" should be replaced with the real driver name.

```
#include <iostream>
#include <dbapi/driver/public.hpp>
#include <dbapi/driver/exception.hpp>
/* Here, XXXlib has to be replaced with the real name, e.g. "ctlib" */
#include <dbapi/driver/XXXlib/interfaces.hpp>
USING_NCBI_SCOPE;
int main()
{
 try { // to be sure that we are catching all driver related exceptions
 // We need to create a driver context first
 // In real program we have to replace CXXXContext with something real
 CXXXContext my_context;
 // connecting to server "MyServer"
 // with user name "my_user_name" and password "my_password"
 CDB_Connection* con = my_context.Connect("MyServer", "my_user_name",
 "my_password", 0);
 // Preparing a SQL query
 CDB_LangCmd* lcmd =
 con->LangCmd("select name, crdate from sysdatabases");
 // Sending this query to a server
 lcmd->Send();
 CDB_Char dbname(64);
 CDB_DateTime crdate;
 // the result loop
 while(lcmd->HasMoreResults()) {
 CDB_Result* r= lcmd->Result();
 // skip all but row result
 if (r == 0 || r->ResultType() != eDB_RowResult) {
 delete r;
 continue;
 }
 // printing the names of selected columns
 NcbiCout << r->ItemName(0) << " \t\t\t"
 << r->ItemName(1) << NcbiEndl;
 // fetching the rows
```

```
while ( r->Fetch() ) {
r->GetItem(&dbname); // get the database name
r->GetItem(&crdate); // get the creation date
NcbiCout << dbname.Value() << ' '
<< crdate.Value().AsString("M/D/Y h:m")
<< NcbiEndl;
}
delete r; // we don't need this result anymore
}
delete lcmd; // delete the command
delete con; // delete the connection
}
catch (CDB_Exception& e) { // printing the error messages
CDB_UserHandler_Stream myExHandler(&cerr);
myExHandler.HandleIt(&e);
}
}
```

Error handling

Error handling is almost always a pain when you are working with an RDBMS because different systems implement different approaches. Depending on the system, you can get error messages through return codes, callbacks, handlers, and/or exceptions. These messages could have different formats. It could be just an integer (error code), a structure, or a set of callback's arguments. The NCBI DBAPI driver intercepts all those error messages in all different formats and converts them into various types of objects derived from CDB_Exception.

CDB_Exception provides the following methods for all exceptions:

- GetDBErrCode() - returns the integer code for this message (assigned by SQL server).
- SeverityString(void) - returns the severity string of this message (assigned by SQL server).
- GetErrCodeString() - returns the name for this error code (e.g. "eSQL").
- Type() - returns the type value for this exception type (e.g. eSQL).
- TypeString() - returns the type string for this exception type (e.g. "eSQL"). This is a pass-through to CException::GetType().
- ErrCode() - alias for GetDBErrCode().
- Message() - returns the error message itself. This is a pass-through to CException::GetMsg().
- OriginatedFrom() - returns the SQL server name. This is a pass-through to CException::GetModule().
- SetServerName() - sets the SQL server name.
- GetServerName() - returns the SQL server name.
- SetUserName() - sets the SQL user name.
- GetUserName() - returns the SQL user name.
- SetExtraMsg() - sets extra message text to be included in the message output.
- GetExtraMsg() - gets the extra message text.
- SetSybaseSeverity() - sets the severity value for a Sybase exception - N.B. Sybase severity values can be provided for Sybase/FreeTDS ctlib/dblib drivers only.

- GetSybaseSeverity() - gets the severity value for a Sybase exception - N.B. Sybase severity values can be provided by Sybase/FreeTDS ctlib/dblib drivers only.
- ReportExtra() - outputs any extra text to the supplied stream.
- Clone() - creates a new exception based on this one.

N.B. The following CDB_Exception methods are deprecated:

- Severity() - returns the severity value of this message (assigned by SQL server).
- SeverityString(EDB_Severity sev) - returns the severity string of this message (assigned by SQL server).

The DBAPI driver may throw any of the following types derived from CDB_Exception:

- CDB_SQLEx This type is used if an error message has come from a SQL server and indicates an error in a SQL query. It could be a wrong table or column name or a SQL syntax error. This type provides the additional methods:
  - BatchLine() - returns the line number in the SQL batch that generated the error.
  - SqlState() - returns a byte string describing an error (it's not useful most of the time).
- CDB_RPCEx An error message has come while executing an RPC or stored procedure. This type provides the additional methods:
  - ProcName() - returns the procedure name where the exception originated.
  - ProcLine() - returns the line number within the procedure where the exception originated.
- CDB_DeadlockEx An error message has come as a result of a deadlock.
- CDB_DSEx An error has come from an RDBMS and is not related to a SQL query or RPC.
- CDB_TimeoutEx An error message has come due to a timeout.
- CDB_ClientEx An error message has come from the client side.

Drivers use two ways to deliver an error message object to an application. If it is possible to throw an exception, then the driver throws the error message object. If not, then the driver calls the user's error handler with a pointer to the error message object as an argument. It's not always convenient to process all types of error messages in one error handler. Some users may want to use a special error message handler inside some function or block and a default error handler outside. To accommodate these cases the driver provides a handler stack mechanism. The top handler in the stack gets the error message object first. If it knows how to deal with this message, then it processes the message and returns true. If handler wants to pass this message to the other handlers, then it returns false. So, the driver pushes the error message object through the stack until it gets true from the handler. The default driver's error handler, which just prints the error message to stderr, is always on the bottom of the stack.

Another tool which users may want to use for error handling is the CDB_MultiEx object. This tool allows collecting multiple CDB_Exception objects into one container and then throwing the container as one exception object.

Driver context and connections

Every program which is going to work with an NCBI DBAPI driver should create at least one Driver Context object first. The main purpose of this object is to be a Connection factory, but it's a good idea to customize this object prior to opening a connection. The first step is to setup two message handler stacks. The first one is for error messages which are not bound to some particular connection or could occur inside the Connect() method. Use PushCntxMsgHandler

() to populate it. The other stack serves as an initial message handler stack for all connections which will be derived from this context. Use PushDefConnMsgHandler() method to populate this stack. The second step of customization is setting timeouts. The SetLoginTimeout() and SetTimeout() methods do the job. If you are going to work with text or image objects in your program, you need to call SetMaxTextImageSize() to define the maximum size for such objects. Objects which exceed this limit could be truncated.

```
class CMyHandlerForConnectionBoundErrors : public CDB_UserHandler
{
 virtual bool HandleIt(CDB_Exception* ex);
 ...
};
class CMyHandlerForOtherErrors : public CDB_UserHandler
{
 virtual bool HandleIt(CDB_Exception* ex);
 ...
};
...
int main()
{
 CMyHandlerForConnectionBoundErrors conn_handler;
 CMyHandlerForOtherErrors other_handler;
 ...
 try { // to be sure that we are catching all driver related exceptions
 // We need to create a driver context first
 // In real program we have to replace CXXXContext with something real
 CXXXContext my_context;
 my_context.PushCntxMsgHandler(&other_handler);
 my_context.PushDefConnMsgHandler(&conn_handler);
 // set timeouts (in seconds) and size limits (in bytes):
 my_context.SetLoginTimeout(10); // for logins
 my_context.SetTimeout(15); // for client/server communications
 my_context.SetMaxTextImageSize(0x7FFFFFFF); // text/image size limit
 ...
 CDB_Connection* my_con =
 my_context.Connect("MyServer", "my_user_name", "my_password",
 I_DriverContext::fBcpIn);
 ...
 }
 catch (CDB_Exception& e) {
 other_handler.HandleIt(&e);
 }
}
```

The only way to get a connection to a server in an NCBI DBAPI driver is through a Connect () method in driver context. The first three arguments: server name, user name and password are obvious. Values for mode are constructed by a bitwise-inclusive-OR of flags defined in EConnectionMode. If reusable is false, then driver creates a new connection which will be destroyed as soon as user delete the correspondent CDB_Connection (the pool_name is ignored in this case).

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

Opening a connection to a server is an expensive operation. If program opens and closes connections to the same server multiple times it worth calling the Connect() method with reusable set to true. In this case driver does not close the connection when the correspondent CDB_Connection is deleted, but keeps it around in a "recycle bin". Every time an application calls the Connect() method with reusable set to true, driver tries to satisfy the request from a "recycle bin" first and opens a new connection only if necessary.

The pool_name argument is just an arbitrary string. An application could use this argument to assign a name to one or more connections (to create a connection pool) or to invoke a connection by name from this pool.

```
...
// Create a pool of four connections (two to one server and two to another)
// with the default database "DatabaseA"
CDB_Connection* con[4];
int i;
for (i = 4; i--; ) {
 con[i]= my_context.Connect((i%2 == 0) ? "MyServer1" : "MyServer2",
 "my_user_name", "my_password", 0, true,
 "ConnectionPoolA");
 CDB_LangCmd* lcmd= con[i]->LangCmd("use DatabaseA");
 lcmd->Send();
 while(lcmd->HasMoreResults()) {
 CDB_Result* r = lcmd->Result();
 delete r;
 }
 delete lcmd;
}
// return all connections to a "recycle bin"
for(i= 0; i < 4; delete con_array[i++]);
...
// in some other part of the program
// we want to get a connection from "ConnectionPoolA"
// but we don't want driver to open a new connection if pool is empty
try {
 CDB_Connection* my_con= my_context.Connect("", "", "", 0, true,
 "ConnectionPoolA");
 // Note that server name, user name and password are empty
 ...
}
catch (CDB_Exception& e) {
 // the pool is empty
 ...
}
```

An application could combine in one pool the connections to the different servers. This mechanism could also be used to group together the connections with some particular settings (default database, transaction isolation level, etc.).

Driver Manager

It's not always known which NCBI DBAPI driver will be used by a particular program. Sometimes you want a driver to be a parameter in your program. Sometimes you need to use

two different drivers in one binary but can not link them statically because of name collisions. Sometimes you just need the driver contexts factory. The Driver Manager is intended to solve these problems.

Let's rewrite our Sample program using the Driver Manager. The original text was.

```
#include <iostream>
#include <dbapi/driver/public.hpp>
#include <dbapi/driver/exception.hpp>
/* Here, XXXlib has to be replaced with the real name, e.g. "ctlib" */
#include <dbapi/driver/XXXlib/interfaces.hpp>
USING_NCBI_SCOPE;
int main()
{
 try { // to be sure that we are catching all driver related exceptions
 // We need to create a driver context first
 // In real program we have to replace CXXXContext with something real
 CXXXContext my_context;
 // connecting to server "MyServer"
 // with user name "my_user_name" and password "my_password"
 CDB_Connection* con = my_context.Connect("MyServer", "my_user_name",
 "my_password", 0);
 ...
```

If we use the Driver Manager we could allow the driver name to be a program argument.

```
#include <iostream>
#include <dbapi/driver/public.hpp>
#include <dbapi/driver/exception.hpp>
#include <dbapi/driver/driver_mgr.hpp> // this is a new header
USING_NCBI_SCOPE;
int main(int argc, const char* argv[])
{
 try { // to be sure that we are catching all driver related exceptions
 C_DriverMgr drv_mgr;
 // We need to create a driver context first
 I_DriverContext* my_context= drv_mgr.GetDriverContext(
 (argc > 1)? argv[1] : "ctlib");
 // connecting to server "MyServer"
 // with user name "my_user_name" and password "my_password"
 CDB_Connection* con = my_context->Connect("MyServer", "my_user_name",
 "my_password", 0);
 ...
```

This fragment creates an instance of the Driver Manager, dynamically loads the driver's library, implicitly registers this driver, creates the driver context and makes a connection to a server. If you don't want to load some drivers dynamically for any reason, but want to use the Driver Manager as a driver contexts factory, then you need to statically link your program with those libraries and explicitly register those using functions from dbapi/driver/drivers.hpp header.

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

Text and Image Data Handling

text and image are SQL datatypes and can hold up to 2Gb of data. Because they could be huge, the RDBMS keeps these values separately from the other data in the table. In most cases the table itself keeps just a special pointer to a text/image value and the actual value is stored separately. This creates some difficulties for text/image data handling.

When you retrieve a large text/image value, you often prefer to "stream" it into your program and process it chunk by chunk rather than get it as one piece. Some RDBMS clients allow to stream the text/image values only if a corresponding column is the only column in a select statement.

Let's suppose that you have a table T (i_val int, t_val text) and you need to select all i_val, t_val where i_val > 0. The simplest way is to use a query:

```
select i_val, t_val from T where i_val > 0
```

But it could be expensive. Because two columns are selected, some clients will put the whole row in a buffer prior to giving access to it to the user. The better way to do this is to use two selects:

```
select i_val from T where i_val > 0
select t_val from T where i_val > 0
```

Looks ugly, but could save you a lot of memory.

Updating and inserting the text/image data is also not a straightforward process. For small texts and images it is possible to use just SQL insert and update statements, but it will be inefficient (if possible at all) for the large ones. The better way to insert and update text and image columns is to use the SendData() method of the CDB_Connection object or to use the CDB_SendDataCmd object.

The recommended algorithm for inserting text/image data is:

- Use a SQL insert statement to insert a new row into the table. Use a space value (' ') for each text column and a zero value (0x0) for each image column you are going to populate. Use NULL only if the value will remain NULL.
- Use a SQL select statement to select all text/image columns from this row.
- Fetch the row result and get an I_ITDescriptor for each column.
- Finish the results loop.
- Use the SendData() method or CDB_SendDataCmd object to populate the columns.

Example

Let's suppose that we want to insert a new row into table T as described above.

```
CDB_Connection* con;
...
// preparing the query
CDB_LangCmd* lcmd= con->LangCmd("insert T (i_val, t_val) values(100, ' ')
\n");
lcmd->More("select t_val from T where i_val = 100");
// Sending this query to a server
lcmd->Send();
```

```
I_ITDescriptor* my_descr;
// the result loop
while(lcmd->HasMoreResults()) {
 CDB_Result* r= lcmd->Result();
 // skip all but row result
 if (r == 0 || r->ResultType() != eDB_RowResult) {
 delete r;
 continue;
 }
 // fetching the row
 while(r->Fetch()) {
 // read 0 bytes from the text (some clients need this trick)
 r->ReadItem(0, 0);
 my_deskr = r->GetImageOrTextDescriptor();
 }
 delete r; // we don't need this result anymore
}
delete lcmd; // delete the command
CDB_Text my_text;
my_text.Append("This is a text I want to insert");
//sending the text
con->SendData(my_descr, my_text);
delete my_descr; // we don't need this descriptor anymore
...
```

The recommended algorithm for updating the text/image data is:

- Use a SQL update statement to replace the current value with a space value (' ') for a text column and a zero value (0x0) for an image column.
- Use a SQL select statement to select all text/image columns you want to update in this row.
- Fetch the row result and get an I_ITDescriptor for each column.
- Finish the results loop.
- Use the SendData() method or the CDB_SendDataCmd object to populate the columns.

Example

```
CDB_Connection* con;
...
// preparing the query
CDB_LangCmd* lcmd= con->LangCmd("update T set t_val= ' ' where i_val = 100");
lcmd->More("select t_val from T where i_val = 100");
// Sending this query to a server
lcmd->Send();
I_ITDescriptor* my_descr;
// the result loop
while(lcmd->HasMoreResults()) {
 CDB_Result* r= lcmd->Result();
 // skip all but row result
 if (r == 0 || r->ResultType() != eDB_RowResult) {
 delete r;
 continue;
```

```
 }
 // fetching the row
 while(r->Fetch()) {
 // read 0 bytes from the text (some clients need this trick)
 r->ReadItem(0, 0);
 my_deskr = r->GetImageOrTextDescriptor();
 }
 delete r; // we don't need this result anymore
}
delete lcmd; // delete the command
CDB_Text my_text;
my_text.Append("This is a text I want to see as an update");
//sending the text
con->SendData(my_descr, my_text);
delete my_descr; // we don't need this descriptor anymore
...
```

Results loop

Each connection in the NCBI DBAPI driver is always single threaded. Therefore, applications have to retrieve all the results from a current command prior to executing a new one. Not all results are meaningful (i.e. an RPC always returns a status result regardless of whether or not a procedure has a return statement), but all results need to be retrieved. The following loop is recommended for retrieving results from all types of commands:

```
CDB_XXXCmd* cmd; // XXX could be Lang, RPC, etc.
...
while (cmd->HasMoreResults()) {
 // HasMoreResults() method returns true // if the Result() method needs to
be called.
 // It doesn't guarantee that Result() will return not NULL result
 CDB_Result* res = cmd->Result();
 if (res == 0)
 continue; // a NULL res doesn't mean that there is no more results
 switch(res->ResultType()) {
 case eDB_RowResult: // row result
 while(res->Fetch()) {
 ...
 }
 break;
 case eDB_ParamResult: // Output parameters
 while(res->Fetch()) {
 ...
 }
 break;
 case eDB_ComputeResult: // Compute result
 while(res->Fetch()) {
 ...
 }
 break;
 case eDB_StatusResult: // Status result
 while(res->Fetch()) {
```

```
 ...
 }
 break;
 case eDB_CursorResult: // Cursor result
 while(res->Fetch()) {
 ...
 }
 break;
 }
 delete res;
}
```

If you don't want to process some particular type of result, just skip the while (res->Fetch()) {...} in the corresponding case.

Supported DBAPI drivers

- FreeTDS (TDS ver. 7.0)
- Sybase CTLIB
- Sybase DBLIB
- ODBC
- MySQL Driver

FreeTDS (TDS ver. 7.0)

This driver is the most recommended, built-in, and portable.

- Registration function (for the manual, static registration) DBAPI_RegisterDriver_FTDS()
- Driver default name (for the run-time loading from a DLL) "ftds".
- Driver library ncbi_xdbapi_ftds
- FreeTDS libraries and headers used by the driver $(FTDS_LIBS) $(FTDS_INCLUDE)
- FreeTDS-specific driver context attributes "version", default = <DBVERSION_UNKNOWN> (also allowed: "42", "46", "70", "100")
- FreeTDS works on UNIX and Windows platforms.
- This driver supports Windows Domain Authentication using protocol NTLMv2, which is a default authentication protocol for Windows at NCBI.
- This driver supports TDS protocol version auto-detection. TDS protocol version cannot be detected when connecting against Sybase Open Server.
- Caveats:
  - Default version of the TDS protocol (<DBVERSION_UNKNOWN>) will work with MS SQL Server and Sybase SQL Server. If you want to work with Sybase Open server you should use TDS protocol version 4.6 or 5.0. This can be done either by using a driver parameter "version" equal either to "46" or to "50" or by setting an environment variable TDSVER either to "46" or to "50".
  - Although a slightly modified version of FreeTDS is now part of the C++ Toolkit, it retains its own license: the GNU Library General Public License.
  - TDS protocol version 4.2 should not be used with MS SQL server.

Sybase CTLIB

- Registration function (for the manual, static registration) DBAPI_RegisterDriver_CTLIB()
- Driver default name (for the run-time loading from a DLL) "ctlib"
- Driver library ncbi_xdbapi_ctlib
- Sybase CTLIB libraries and headers used by the driver (UNIX) $(SYBASE_LIBS) $(SYBASE_INCLUDE)
- Sybase CTLIB libraries and headers used by the driver (MS Windows). You will need the Sybase OpenClient package installed on your PC. In MSVC++, set the "C/C++".General."Additional Include Directories" and Linker.General."Additional Library Directories" properties to the Sybase OpenClient headers and libraries (for example "C:\Sybase\include" and "C:\Sybase\lib" respectively). Also set the Linker.Input."Additional Dependencies" property to include the needed Sybase OpenClient libraries: LIBCT.LIB LIBCS.LIB LIBBLK.LIB. To run the application, you must set environment variable %SYBASE% to the Sybase OpenClient root directory (e.g. "C:\Sybase"), and also to have your "interfaces" file there, in INI/sql.ini. In NCBI, we have the Sybase OpenClient libs installed in \\snowman\win-coremake\Lib\ThirdParty\sybase.
- CTLIB-specific header (contains non-portable extensions) dbapi/driver/ctlib/interfaces.hpp
- CTLIB-specific driver context attributes "reuse_context" (default value is "true"), "version" (default value is "125", also allowed "100" and "110")
- Caveats:
  - Cannot communicate with MS SQL server using any TDS version.

Sybase DBLIB

- Registration function (for the manual, static registration) DBAPI_RegisterDriver_DBLIB()
- Driver default name (for the run-time loading from a DLL) "dblib"
- Driver library dbapi_driver_dblib
- Sybase DBLIB libraries and headers used by the driver (UNIX) $(SYBASE_DBLIBS) $(SYBASE_INCLUDE)
- Sybase DBLIB libraries and headers used by the driver (MS Windows) Libraries: LIBSYBDB.LIB. See Sybase OpenClient installation and usage instructions in the Sybase CTLIB section (just above).
- DBLIB-specific header (contains non-portable extensions) dbapi/driver/dblib/interfaces.hpp
- DBLIB-specific driver context attributes "version" (default value is "46", also allowed "100")
- Caveats:
  - Text/image operations fail when working with MS SQL server, because MS SQL server sends text/image length in the reverse byte order, and this cannot be fixed (as it was fixed for FreeTDS) as we do not have access to the DBLIB source code.

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

- DB Library version level "100" is recommended for communication with Sybase server 12.5, because the default version level ("46") is not working correctly with this server.

ODBC

- Registration function (for the manual, static registration) DBAPI_RegisterDriver_ODBC()
- Driver default name (for the run-time loading from a DLL) "odbc"
- Driver library dbapi_driver_odbc
- ODBC libraries and headers used by the driver (MS Windows) ODBC32.LIB ODBCCP32.LIB ODBCBCP.LIB
- ODBC libraries and headers used by the driver (UNIX) $(ODBC_LIBS)$ (ODBC_INCLUDE)
- ODBC-specific header (contains non-portable extensions) dbapi/driver/odbc/interfaces.hpp
- ODBC-specific driver context attributes "version" (default value is "3", also allowed "2"), "use_dsn" (default value is false, if you have set this attribute to true, you need to define your data source using "Control Panel"/"Administrative Tools"/"Data Sources (ODBC)")
- Caveats:
  - The CDB_Result::GetImageOrTextDescriptor() does not work for ODBC driver. You need to use CDB_ITDescriptor instead. The other way to deal with texts/images in ODBC is through the CDB_CursorCmd methods: UpdateTextImage and SendDataCmd.
  - On most NCBI PCs, there is an old header odbcss.h (from 4/24/1998) installed. The symptom is that although everything compiles just fine, however in the linking stage there are dozens of unresolved symbol errors for ODBC functions. Ask "pc.systems" to fix this for your PC.
  - On UNIX, it's only known to work with Merant's implementation of ODBC, and it has not been thoroughly tested or widely used, so surprises are possible.

MySQL Driver

There is a direct (without ODBC) MySQL driver in the NCBI C++ Toolkit DBAPI. However, the driver implements a very minimal functionality and does not support the following:

- Working with images by chunks (images can be accessed as string fields though)
- RPC
- BCP
- SendData functionality
- Connection pools
- Parameter binding
- Canceling results
- ReadItem
- IsAlive
- Refresh functions
- Setting timeouts

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

Major Features of the BDB Library

The BDB library provides tools for the development of specialized data storage in applications not having access to a centralized RDBMS.

- **C++ wrapper on top of Berkeley DB.** The BDB library takes care of many of the ultra low-level details for C programmers using the Berkeley DB. The BDB implements B-Tree file access (both keyed and sequential), environments, cursors, and transactions.
- **Error checking.** All error codes coming from the Berkeley DB are analyzed and processed in a manner common to all other components of the C++ Toolkit. When an error situation is detected, the BDB library sends an exception that is reported by the diagnostic services and can be handled by the calling application, similar to any other Toolkit exception.
- **Support for relational table structure and different data types.** The Berkeley DB itself is "type agnostic" and provides no means to manipulate data types. But for many cases, clear data type support can save a lot of work. The Toolkit implements all major scalar data types so it can be used like a regular database.
- **Cross platform compatibility.** The BDB databases can be transferred across platforms without reconverting the data. The BDB tracks the fact that the database was created as big-endian or little-endian and does the conversion transparently when the database migrates.
- **Easy BLOBs.** The BDB library supports keyed BLOB storage. BLOBs can be streamed to and from the database. A set of additional interfaces has been written to simplify the BLOB access in comparison with the original Berkeley DB C library.
- **Disk-based cache interface.** The BDB library implements a cache disk cache service used by other Toolkit components to minimize client-server traffic and to store parts of the data locally. Different cache management and data expiration policies have been put in place.
- **Database maps.** The BDB library includes template classes similar to STL map and multimap but persistently stores the map content in the Berkeley DB files.
- **Simple queries.** The BDB library includes implementation of a simple query language to search records in flat files.

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

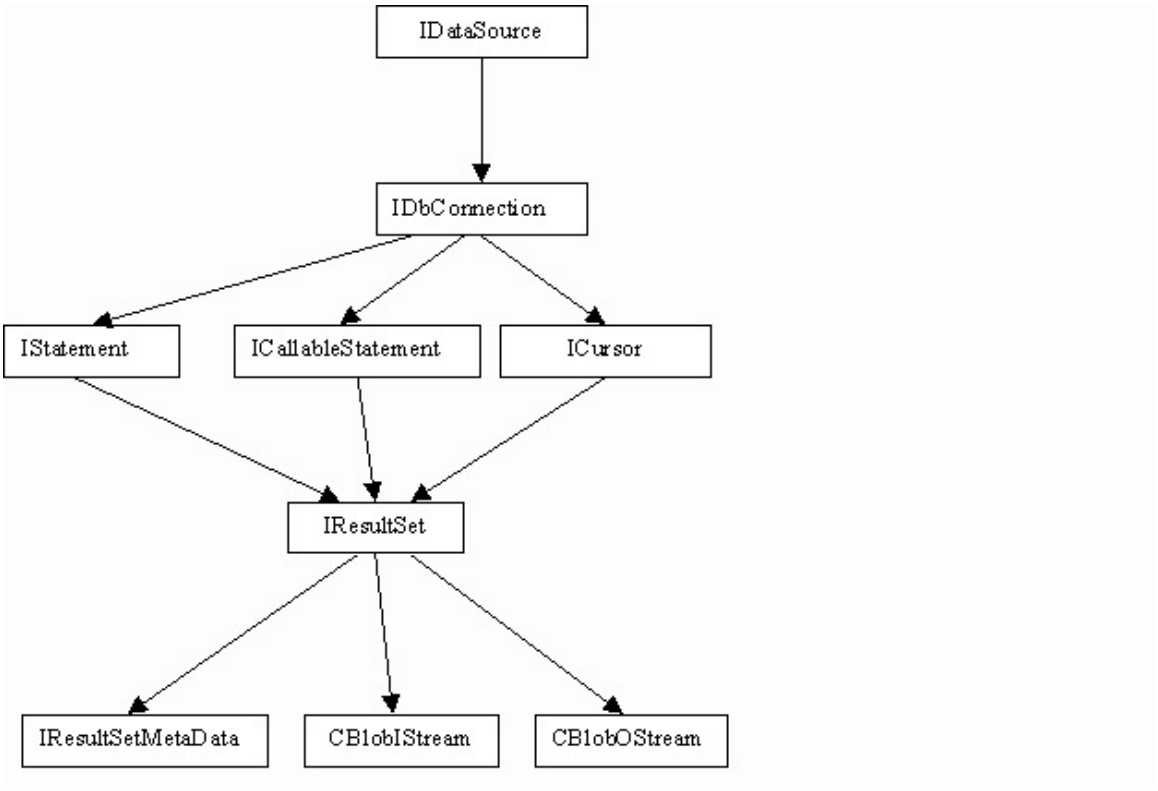

Figure 1. Object Hierarchy