Delphi + ZeosLib + Firebird

Instalación, comentarios y ejemplos de uso

A ZEOS basics tutorial not only for Firebird ...

Author Michael Seeger (ZeosLib Development Team)
Date 26.03.2008, 16:01 **Views** 6143 at 22,03,2010
Description A ZEOS basics tutorial not only for Firebird ...
Category Firebird / Interbase **Type**

The ZeosLib DBOs 6.1.5 - With Delphi 7 and Firebird 1.5 This little article shows how to access Firebird databases by using the ZEOS component Library in version 6.1.5 (including Patches 1&2) and how to use these components in database applications. It does not matter if you use the "real" SQL-Server or the embedded version which is restricted to local held databases. A couple of examples (also migrated Delphi-BDE demos) shall explain how to use the ZEOS components.

Although this article describes the usage of the ZEOS Library using Firebird, all the basics can be used with other SQL servers / databases that are supported by ZEOS.

Note: The Firebird Server can be downloaded from download section of
http://www.ibphoenix.com
http://www.firebirdsql.org

The ZEOS Library

The name "ZEOS" has no special meaning. The founders of ZEOS found that this name just sounded good. Since that time the Library is called "ZEOS".

Generally we can say about the ZEOS Library that the developers are inteneded to copy the functions and the behaviour of the corresponding BDE components as good as possible. The intension is to minimize the learning courve for developers who migrate from BDE to ZEOS. Of course there must have been made some compromises so that they are not a hundred percent compatible because the ZEOS components shall be applicable universally.

The ZEOS Library in version 6.1.5 consists of the following nine components which shall be introduced in the following:

- TZConnection
- TZQuery
- TZReadOnlyQuery
- TZUpdateSQL
- TZTable
- TZStoredProc
- TZSQLProcessor
- TZSQLMonitor
- TZSQLMetadata

Installing the ZEOS Library and additional stuff

The installation of the ZEOS Library under Delphi 7 professional is not that complicated. Once the current ZEOS version and all the patches (while writing this article it was library version 6.1.5 and the corresponding patches 1&2) are downloaded and unzipped into a directory of your choice completely you only have to follow the installation instructions (note: Firebird does not need any additional DLL's, here!):

Open the delphi project group **ZeosDbo.bpg** from subdirectory **packages\delphi7 ZeosDbo.bpg** and install the following components in given order:

- ZCore.bpl
- ZParseSql.bpl
- ZPlain.bpl
- ZDbc.bpl
- ZComponent.bpl

Note: If there occur some errors while compiling that say that a certain dcu file could not be found then just add the subdirectory **packages\delphi7\build** to delphis library path. All dcu files that are created while compilation are located here.

Attention: The client library of Firebird Server version 1.5.1 (not embedded!) was delivered as "gds32.dll" and not "fbclient.dll". This causes trouble while accessing via ZEOS because the protocol "firebird-1.5" aassumes a DLL named "fbclient.dll". A workaround is to copy the "gds32.dll" and rename this copy to "fbclient.dll".

Additive: Event Alerter component (TZIBEventAlerter)

An additional component, that only exists for Firebird (and also Interbase) intercepts all (registered) events, that are triggered by Stored Procedures of a database, and makes it possible to react upon them. This component was posted by Alexey Gilev (aka "GAF") into the ZEOS-Forum and made available to the ZEOS-Team. The original version can be downloaded here:
http://sourceforge.net/tracker/index.php?func=detail&aid=911233&group_id=35994&atid=415826

The event alerter is only tested for ZEOS version 6.1.4 but runs with some modifications without any problems in version 6.1.5 (with Patch 1&2). The event alerter that was ported to ZEOS version 6.1.5 will be installed as follows (please consider the ReadMe file! I'm not going to give any warrenty that this solution is error free! You use it at your own risk!):

- Download patch for ZIBEventAlerter (integrated) here
- unzip the files and distribute them to the according ZEOS directories
- if necessary recompile and reinstall ZEOS (see above)

Basics: Transactions

Some mandatory basics about transactions have to be told in order to understand how the TZConnection component works internally and how to work with it.

In general: You only can have access to a database within the context of a valid ("running") transaction. A transaction has to fulfill the following four characteristics that are known as ACID characteristics of a transaction.

Atomicity
All actions performed on a database have to be executed successfully. If only one error occurs then the original state of the database has to be restored. According to the principle "all or nothing".

Consitency
A transaktion transfers a database from one consitent state to an other consistent state. If an error occurs then the original state of the database has to be restored.

Isolation
A transaction has to be handled by the server as if it was the only one running. This means it has to run indepently from other transactions. A user must not notice the changes done by other users.

Durability
Changes in the dataset of a database that are caused by SQL statements that are executed between the start of a transaction and a COMMIT have to be fixed irrevocably.

Transactions encapsulate consecutive accesses on a database. A database access may be of reading or writing nature (INSERT, UPDATE, DELETE) or it may change the structure of a database. Transactions are terminated by COMMIT or ROLLBACK. COMMIT confirms all changes in a database since the start of a transaction. ROLLBACK resets all changes in a database since the start of a transaction.

Transactions running on the server have to be isolated from each other. So they may run independently. A transaction has to be handled by the server as it was the only one that is currently running. This means for the user that he must never see the changes of other users while he is in a running transaction because the other changes have nothing to do with his transaction. This is called the isolation of a transaction. By using different transaction isolation levels (TILs) the developer may protect the data of an SQL resultset from access by other transactions. The behaviour described above is called the standard isolationlevel known as SERIALIZABLE. The standard isolationlevel of Firebird is called SNAPSHOT and comes very close to SERIALIZABLE.

TZConnection

The TZConnection component is a combination of a BDE TDatabase like component a component that handles a transaction. This combination makes sense because all access to a Firebird database (and also other databases) is always made in a running transaction. Such a transaction is startet by the ZEOS Library whenever a connection (method Connect of TZConnection) to a database is opened. This causes that each database access is done within the context of a running transaction, automatically. The so called AutoCommit mode is always on (set to "True"). This is also the standard behaviour of the corresponding BDE component. If AutoCommit is activated then every change of an SQL statement will be confirmed in the database by COMMIT ater its successfull execution. If this behaviour shall be turned off and an explicit transaction shall be started then the method StartTransaction has to be called. Within this explicit transaction it is possible to execute a couple of SQL statements that make changes to the database, in succession. These statements then can be confirmed as a "group" by COMMIT. If an explicit transaction is active then AutoCommit is always turned off. By Calling the Method Commit all the changes made within this explicit transaction are confirmed. Calling the method Rollback resets these changes. In both cases AutoCommit will be set to True when the method call (Commit or Rollback) is done. The explicit transaction has ended.

Retaining

After confirming the chanes made in a transaction by COMMIT or resetting them by ROLLLBACK the transaction normally is going to be ended and an existing resultset of a query or stored procedure will be discarded. These COMMITs and ROLLBACKs are called "hard" commit or "hard" rollback. By using the ZEOS library this will become a little bit different. ZEOS keeps the resultset alive. This is achieved by closing transaction with "soft" commits or "soft" rollbacks. All this is done by the TZConnection object. This method is

called retaining. The COMMIT and ROLLBACK commands are executed with the addition RETAINING. Retaining causes the closing of the current transaction and immediately opening a new transaction with all the data and resources (especially the resultset) of the "old" transaction.

Retaining becomes a problem if it is uses for huge tables. It constrains the internal cleanup mechanism of firebird (garbage collection). This leads (because of the versioning and the multigenerational architecture of Firebird) to a lot of old records that have to be kept but will not be needed anymore. This influences the server's performanced in a negative way. A so called sweep would discard these old versions and improve the performance. This sweep will only be executed by sending a "hard" COMMIT or ROLLBACK. The ZEOS Library only executes these "hard" commands when ending the database connection (closing connection). It is not possible to send them while a database connection is active. So the database connection should be deactivated and immediately activated occasionally to achieve this performance improvement.

Transaction Isolation Levels of TZConnection

The TZConnection component provides four useful and predefined Transaction Isolation Levels (TIL):

tiRepeatableRead
It corresponds to the TIL "SNAPSHOT" which is the standard of Firebird servers. It is a combination of the trasaction parameters "concurrency" and "nowait". A snapshot of the current database is made. Other users are only influenced (constrained) if two transactions work on one record simultaneously. If conflicts arise while accessing data an error message will be returned. Changes within other transactions will not be noticed. This TIL covers the requirement of the SQL standard (SERIALIZABLE) widely.

tiReadCommitted
It corresponds to the TIL "READ COMMITTED". It is a combination of the transaction parameters "read_committed", "rec_version" and "nowait". This TIL recognizes all changes in other transaction that have been confirmed by COMMIT. The parameter "rec_version" is responsible for the behaviour that the most current values that were committed by other users will be considered. the parameter "nowait" is resposible for the behaviour that there is no waiting for the release of a locked record. So the server is more stressed than in TIL tiRepeatableRead because it has to do all the refreshes to get these values again and again.

tiSerializable
It corresponds to the TIL "SNAPSHOT TABLE STABILITY". It is used to get an exclusive access to the result set. Realized by transaction parameter "consistency" it prevents that "foreign" transaction may access the written data. Only the transaction which has written the data may access them. This prevents also a multi user access to the written data. Because this TIL is very restrictive by accessing written data it should be applied with caution and care.

tiNone
No TIL is used to isolate the transaction.

The TIL tiReadUncommitted is not supported by Firebird. If this TIL is used, an error will be triggerded and the transaction will not be isolated (like using tiNone).

Recommendation

It is advisable to isolate transactions with transaction isolation level tiRepeatableRead (the Firebird standard). This TIL covers the requirement of the SQL standard (SERIALIZABLE) widely. It prevents all problems concerning consistency that may arise by using transactions. Second choice would be tiReadCommitted but this depends on the application and the necessity if the reseult set always has to be current.

Customizing TILs

If you want to customize your TILs or expand a given TIL then you can do this by using the parameters of TZConnection. The most important thing about this is that the TIL that should be expanded has to be set in property IsolationLevel. If it is set the TIL parameters (see IB/FB API reference) you want to add may be added in sourcecode. The following example shows the expansion of a TIL preset to tiNone:

```
:
ZConnection.TransactIsolationLevel := tiNone;
ZConnection.Properties.Add('isc_tpb_concurrency');
ZConnection.Properties.Add('isc_tpb_wait');
ZConnection.Connect;
:
```

Protocol

The most important setting in a TZConnection object is the server protocol it is set in property Protocol. It determines which protocol shall be used and thus which SQL server will be accessed. This method makes ZEOS so flexible. You don't have to install special components for each database you want to access as it was in Versions 5.x and earlier. The components will be installed once. This is enough. You only choose the protocol for the supported SQL server you want to access and you are done. So you set protocol "firebird-1.5" to access Firebird 1.5 server.

Read-Only-Connection

The database connection maintained by a TZConnection object is set to read only by default (ReadOnly = True). This means that no writing access to the connected database is allowed. To get writing access to the database you have to set ReadOnly to False.

Codepages

Codepages will be determined by setting parameter "lc_ctype" or "Codepage" in TZConnection. This parameter must be added to the property Properties. E. g.:

```
ZConection.Properties.Add ('lc_ctype=ISO8859_1');

or

ZConnection.Properties.Add ('Codepage=ISO8859_1');
```

Note: The Codepage support of Firebird (also embedded version) in version 1.5 with ZEOS is a little bit buggy. These bugs are corrected in Firebird version 1.5.1.

Features of the Firebird embedded server

Normally the server name or IP address of the server is given in property HostName. By using a Firebird embedded server you may leave this property empty. Only property Database has to be determined. Here you have to specify drive path and name of the database including extension.

An other feature of the embedded server is that you may specify any login name with a pasword of your choice. It doesn't matter what you choose you will get connected.

Setting the TZConnection object property Connected to True in designtime is extremely bad if you don't reset

it to False before compiling. If you then start the compiled application with IDE running you will get an error that says that the database cannot be opened because it is already in use. So you should establish the connection to the database when starting the application (e. g. in OnCreate event of the main form) and then open the needed queries and tables. Deactivating the connection should also be done in main form (e. g. in OnDestroy). It is not necessary to close all open queries and tables. This will be done when closing the connection of the TZConnection object. If you use datamodel forms you have to take care that the datamodel form is created before the main form is created (set in the IDE's project options)..

Useful TZConnection parameters

Additional parameters for establishing connections to Firebird databases are::

CreateNewDataBase:
A new database will be created based on the specified CREATE DATABASE statements. When the database is created the connection will be established immediately. All this happens by calling the Connect method of TZConnection.

```
:
ZConnection1.Database := 'd:\db1.fdb';
ZConnection1.Protocol := 'firebird-1.5';
ZConnection1.Properties.Add ('CreateNewDatabase=CREATE DATABASE ' +
QuotedStr ('d:\db1.fdb') + ' USER ' +
QuotedStr ('sysdba') + ' PASSWORD ' + QuotedStr ('masterkey') +
' PAGE_SIZE 4096 DEFAULT CHARACTER SET ISO8859_1');
ZConnection1.Connect;
:
```

To execute this correctly you have to set the Database and Protocol properties at minimum (also possible in objectinspector).

Dialect:
This parameter sets Firebird's SQL dialiect. To set dialect "1" you have to use the following code:

```
ZConnection.Properties.Add ('Dialect=1');
```

The dialect of Firebird 1.5.x is set to "3" by default.

Rolename:
This parameter sets a rolename. A logged in user then works in the context of the role's rights but before the user has to be assigned to this role. The Firebird embedded server does not support this feature.

TZQuery

The usage of TZQuery is similar to the usage of BDE's TQuery component.

Recommandation: RequestLive and TZUpdateSQL

If an SQL dataset shall be updatable then RequestLive has to be set to true and you should generally use according update SQL statements that will be defined in TZUpdateSQL. If this is done just assign TZUpdateSQL to the TZQuery object. Now all changes that will be made in the result set will be done to the database by using the defined statements of TZUpdateSQL. According to experience RequestLive mode runs more smoothly by using TZUpdateSQL.

Usage of parameters in SQL statements

Using parameters in SELECT statments is as easy as using them with BDE's TQuery. If TZQuery has a result set then you have to use the Open method. If you want to execute an SQL statement which has no result set (e. g.: INSERT or UPDATE) you have to use ExecSQL (see also: TZStoredProc).

TZReadOnlyQuery

This is a Query component that is quite similar to the TZQuery component. There is just one difference: The result set is read only. There is no possibility to assign a TZUpdateSQL object.

TZUpdateSQL

A TZUpdateSQL object provides statements to modify the data of a result set that is retrieved by a TZQuery object. The TZUpdateSQL component is comparable to BDE's TUpdateSQL component. Here is an example how to define the statements of an SQL statement with corresponding update statements (based on a dialect 3 database):

- **SQL.Sql:**

```
SELECT * FROM names
```

- **UpdateSQL.InsertSql:**

```
INSERT INTO names (recno, name)
VALUES (:recno, :name)
```

- **UpdateSQL.ModifySql:**

```
UPDATE names
SET recno = :RecNo,
name = :name
WHERE recno = :old_recno
```

- **UpdateSQL.DeleteSql:**

```
DELETE FROM names
WHERE recno = :old_recno
```

The "OLD_" parameter prefix for SQL statements

The "old_" prefix is handled according to the handling with BDE components. By using "OLD_" as prefix for a fieldname you are able to access the value of the field before it was changed. This is very helpful if you have to compare fieldvalues in a WHERE clause.

Queries with read only resultsets

In geneal a TZUpdateSQL object is assigned to a TZQuery object that has a read only resultset. This makes it possible to change its data. Such read only queries are queries that join multiple tables. But also with "normal" "RequestLive" resultsets you may use TZUpdateSQL (see: TZQuery).

Multiple statements in TZQuery and TZUpdateSQL

The components TZQuery and TZUpdateSql provide the possibility to execute multiple statements, internally. So it is possible to place multiple SQL statements (even with parameters) for execution in SQL property. They only have to be separated by semicolon. Here an example:

```
:
With Query do Begin
Sql.Clear;
Sql.Add('DELETE FROM table1;');
Sql.Add('INSERT INTO table1 VALUES (:Val1, :Val2);');
Sql.Add('INSERT INTO table2 VALUES (:Val3, :Val2);');
Sql.Add('UPDATE table3 SET field1 = :Val4;');
Params.ParamByName('Val1').AsInteger := 123;
:
ExecSql;
End;
:
```

The statements will be executed in given order. It is also possible to execute multiple statements if they are grouped in this manner inside multiple TZUpdateSql-Objects in order to update multiple tables.

TZTable

TZTable acts like BDE's TTable. As a principle you only should use TZTable in a C/S application if you have very small tables because all records of the table will be transferred from server into client's memory by opening the TZTable. This is a bahaviour similar to a "SELECT * FROM XYZ" statement. You should even prevent a statement like this in a C/S application. The intension is to keep the resultset that has to be transferred from server to client as small as possible (perferably onle one record).

TZStoredProc

TZStoredProc provides the possiblity to execute stored procedures that are saved in a database. There are two kinds of stored procedures: Procedures that return a resultset and procedures that do not return a resultset. TZStoredProc works similar to BDE's TStoredProc. The only difference between them is that you don't have to call Prepare before you call the ExecProc method.

Stored Procedures with Resultsets

If a stored procedure returns a result set then it will be activated by calling the Open method (when all existing parameters have got their values):

```
:
With spSumByName do Begin
Close;
ParamByName ('Name').Value := 'DontKnowHow';
Open;
End;
:
```

The resultset can be worked on like a resultset of a TZQuery.

Stored Procedures without Resultsets

If a stored procedure has no resultset then it will be executed by calling the ExecProc method (when all existing parameters have got their values). Here is an example (conConnection.AutoCommit = True):

```
:
With spDeleteByName do Begin
ParamByName ('Name').Value := 'DontKnowHow';
conConnection.StartTransaction
Try
// execute StoredProc
ExecProc;
Except
conConnection.Rollback;
End;
conConnection.Commit;
End;
:
```

Attention: Bug in ZEOS Library V 6.1.5!

If a stored procedure was executed via ExecProc at least one time, an exception will be triggered when disconnecting TZConnection (used by TZStoredProc) from database. Exception's Message is : "invalid statement handle". SQL error number is -901. This Exception will not be triggered if the stored procedure is executed using a TZQuery or TZReadOnlyQuery. Execution of the stored procedure will be done by calling the procedure using the following SQL command (put into the SQL property):
[syntax="sql"]EXECUTE PROCEDURE DeleteByName (:Name)[/color][/font][/list]
Executing a stored procedure this way is similar to execution in TZStoredProc (conConnection.AutoCommit = True):

```
:
// using a TZQuery object
With qryDeleteByName do Begin
ParamByName ('Name').Value := 'DontKnowHow';
conConnection.StartTransaction
Try
ExecSQL; // execute SQL statement in TZQuery
Except
conConnection.Rollback;
End;
conConnection.Commit;
End;
:
```

Info: Even with a dialect 3 database it is not needed to put the name of the stored procedure into quotation marks Names of stored procedures in Firebird are not case sensitive.

Problems with TZStoredProc when using dialect 1 databases

The problem when using dialect 1 databases is that the metadata of stored proecedure is not properly transferred to the TZStoredProc objects when choosing the stored procedure in object inspector. All parapeter data is not properly interpreted. An update from dialect 1 to dialect 3 solves this problem. Is the dialect 1 database vital for the application system then a TZQuery or TZReadOnlyQuery has to be used as workaround.

TZSQLProcessor

This component provides the function to process SQL scripts that can be loaded by calling the methods LoadFromStream() or LoadFromFile(). The loaded SQL-Script is put into an according property called Script. Importan is that the correct delimiter for the script is set (Property is also Delimiter). By default ";" will be set as delimiter. This is sufficient for the most scripts. But if you want to create stored procedures or triggers via script you should set delimiter according to the setting of the script's "SET TERM >newDelimiter< >oldDelimiter<" command (normally "^" is used for this). In addition to this the property DelimiterType has to be set to dtSetTerm. Here some lines of code that show how to process an SQL script:

```
:
sqlScript.Script.Clear;
sqlScript.LoadFromFile('c:\temp\createdb.sql');
conConnection.StartTransaction;
Try
sqlScript.Execute;
Except
conConnection.Rollback;
End;
conConnection.Commit;
:
```

The SQL script is processed within an explicit transaktion (AutoCommit is turned on). If execution succeeds the changes will be committed otherwise they will be rolled back.

TZSQLMonitor

Using the TZSQLMonitor component you may log certain actions or events of the ZEOS database components. The journal may be written as file or collected in a TMemo object or something like that.

Writing the actions or events to a logfile only needs a few settings:

```
:
sqlMonitor.FileName := 'C:\Log\MyAppLog.log';
sqlMonitor.Active := True;
sqlMonitor.AutoSave := True;
:
```

To collect the logged actions or events in a TMemo object you have to implement the OnLogTrace event as follows:

```
Procedure Tfrm_MyApp.sqlMonitorLogTrace (Sender: TObject;
Event: TZLoggingEvent);
Begin
If Trim (Event.Error) > '' Then
memMontor.Lines.Add (DateTimeToStr (Event.Timestamp) + ': ' +
Event.Message + #13#10 + ' Error: ' + Event.Error)
Else
memMontor.Lines.Add (DateTimeToStr (Event.Timestamp) + ': ' +
Event.Message);
End; // sqlMonitorLogTrace
```

The OnLogTrace event is always triggered when an action or event was logged by sqlMonitor.LogEvent (oEvent). The oEvent parameter stands for an instance of a TZLoggingEvent class object.

Properties of TZLoggingEvent

Category (TZLoggingCategory): Stands for the category of the logged action or event (lcConnect, lcDisconnect, lcTransaction, lcExecute or lcOther)

Protocol (String): The protocol that is used to access the database (see: TZConnection)

Message (String): The text that is logged.

ErrorCode (Integer): The error code in case of an error.

Error (String): The error text in case of an error.

Timestamp (TDateTime): Date and time of the logged action or event.

TZSQLMetadata

With this special TDataSet component it is possible to access the metadata of a database like tables, columns, etc. (This chapter is still to be expanded!)

TZIBEventAlerter (Add On only for Firebird and Interbase)

By using this component you are able to intercept events triggered by stored procedures of a Firebird database and react to them. You only have to register the event's text (string) you want to react to in property Events which is a stringlist. You can do this using objectinspector or inside the sourcecode. The event alerter is activated by registering the listed events. To do this you should call the Registerevents method because the Properties AutoRegister and Registered do not work properly. If you once have registered the events the component is able to react to them. All events are unregistered (deleted) by calling method UnregisterEvents and the event alertet is turned off.

Registration of events and "activation" of the event alerter:

```
:
EventAlerter.Events.Add ('Minimum Stock Level Reached');
EventAlerter.Events.Add ('Credit Limit Exceeded');
EventAlerter.RegisterEvents;
:
```

Master/Detail with ZEOS Library

ZEOS DataSet components come with two kinds of master/detail connections: those with a server sided filter and those with a client sided filter. Both kinds and one kind in additioin that is independent from ZEOS (and thus without any comfort) will be described here.

Note: If we talk about "master" or "detail" then a TDataSet descendant (TZQuery/TZReadOnlyQuery, TZTable or a TZStoredProc) is meant that accesses either a master resultset or a detail resultset of a master/detail connection.

Master/Detail with server sided filters

This method is the default behaviour of the BDE's TQuery component. A master/detail connection of two DataSets is established as follows:

- The master's DataSource is assigned to the DataSource of the detail.
- All primary key fields of the master have to be compared with the foreign key fields of the detail in the detail SQL statement.

This is an example for a simple master/detail queries. Requirement: We use TZQuery or TZReadOnlyQuery to establish the master/detail connection:

- **Master SQL:**

```
SELECT id, feld1, feld2, feld3
FROM master
```

[*]**Detail-SQL:**

```
SELECT feld1, feld2, master_id
FROM detail
WHERE master_id = :id
```

Parameter :id stands for the content of master's "id" field (is the primary key) because the master's DataSource is assigned to the property DataSource of the detail. So this parameter references to the "id" field of the master. The rield "master_id" in detail query is the foreign key field of the detail table which references the primary key of the master.

If the cursor of the master changes its position while server sided filters are used, the SQL statement of the detail is executed using the current key values. So the result set of the detail is automatically refreshed.

Master/Detail with client sided filters

This is the default behaviour of a BDE TTable component. Here a master/detail connection between two DataSets is established as follows:

- The DataSource of the master is assigned to the property MasterDataSource of the detail.
- The primary key fields of the master are assigned to property MasterField of the detail.
- The foreign key of the detail which references the primary key of the master is assigned to propety IndexFieldNames.

This is an example for a simple master/detail queries. Requirement: We use TZQuery or TZReadOnlyQuery to establish the master/detail connection:

- **Master SQL:**

```
SELECT id, feld1, feld2, feld3
FROM master
```

- **Detail-SQL:**

```
SELECT feld1, feld2, master_id
FROM detail
WHERE master_id = :id
```

Parameter :id stands for the content of master's "id" field (is the primary key) because the master's DataSource is assigned to the property DataSource of the detail. So this parameter references to the "id"

field of the master. The rield "master_id" in detail query is the foreign key field of the detail table which references the primary key of the master.

If the cursor of the master changes its position while server sided filters are used, the SQL statement of the detail is executed using the current key values. So the result set of the detail is automatically refreshed.

Master/Detail with client sided filters

This is the default behaviour of a BDE TTable component. Here a master/detail connection between two DataSets is established as follows:

- The DataSource of the master is assigned to the property MasterDataSource of the detail.
- The primary key fields of the master are assigned to property MasterField of the detail.
- The foreign key of the detail which references the primary key of the master is assigned to propety IndexFieldNames.

With client sided filters both DataSets first transfer all table rows from server to client. The detail then sets a filter (on client side) to get the details according to the current master record.

In case of creating a new detail record for a master/detail connection with a client sided filter there is a kind of automatism: The Foreign key fields of the detail (set in property IndexFieldNames of the detail) will be filled automatically with the according (current) primary key data of the master (set in proptery MasterField of the detail). Note: With server sided filters you have to care about this functionality, manually in your program's code. This can be achieved by implementing the OnNewRecord event of the detail. This event is always triggered when an new record is to be created (see: Delphi online help for TDataSet). According to the SQL statements, defined above you only have to implement the following:

```
Procedure dmMasterDetail.qryDetailNewRecord (DataSet: TDataSet);
Begin
qryDetailMASTER_ID.Value := qryMasterID.Value;
End;
```

Corresponding TFields were created for the fields "master_id" of the detail and "id" of the master using the fieldeditor.

For master/detail connections in ZEOS there is an additional option that is set in property Options: It is doAlwaysResyncDetail. If this option is set then the resultset of the detail is only refreshed when post is called or a record changes (both within master DataSet).

Master/Detail - "by hand"

Normally you implement a master/detail connection according to the method used by server sided filters. The SQL statements for this look exactly like that. Only the properties that are set in master and detail (see above) will not be set here. Both TZQueris are working independently. This means: The detail DataSet does not recognize any changes in master DataSet. Its synchronzation has to be implemented, manually. This will be done in the OnChange event of the master. OnChange is triggered when changing to a new record or field data has been changed (see: Delphi online help for TDataSource). Synchronization of the detail (according to the example above) would be implemented like this:

```
Procedure dmMasterDetail.dsMasterDataChange (
Sender: TObject; Field: TField);
Begin
With qryDetail do Begin
Close;
ParamByName('id').Value := qryMasterID.Value;
Open;
End;
End;
```

This is the same function that ZEOS executes automatically when using server sided filters: The detail query is executed (agin) with the current ID value of the master which will be assigned to parameter ":id" in the detail query. So the resultset of the detail will be refreshed according to the current master record.

If a new record shall be created in detail DataSet then you have to act the same way as with server sided filters (see above).

Cached Updates

The developers of the ZEOS Library are intended to implement the functionality of the BDE components as good as possible. This is why there is also the possibility of cached updates with ZEOS. You only have to set the property CachedUpdates of a DataSet descendant (TZTable, TZQuery oder TZStoredProc) to true. From this time on all changes in the result set will be cached and they can be easily committed to the database by calling ApplyUpdates and CommitUpdates one after the other either automatically in your program code or triggered manually by a user. CancelUpdates causes that the canges will not be committed to the database. In this case all cached changes will be reset (like using a ROLLBACK). Here you will find a little code snippet that tries to show you how cached updates are implemented (don't argue about the sense in this...). AutoCommit mode of TZConnectin is turned on (True):

```
:
With DataSet do Begin

bEverythingOK := True;
CachedUpdates := True;
DataSet.First;

While (not DataSet.EOF) and (bEverythingOK) do Begin
DataSet.Edit;
:
// process record
:
DataSet.Post;
DataSet.Next;
:
bEverythingOK := AFunctionForValidation;
End;

If bEverythingOK Then Begin
ApplyUpdates;
CommitUpdates;
End
Else
CancelUpdates;

CachedUpdates := False;
End;
:
```

BLOB Fields

According to BDE's components the components of the ZEOS Library are capable of handling BLOB fields. Here is an example how a new record with a BLOB field is created. The BLOB field is filled with a bitmap. To achieve this we have to use a Stream:

```
:
Var TheStream : TMemoryStream;
Begin
TheStream := TMemoryStream.Create;
Try
Image1.Picture.Bitmap.Savetostream(TheStream);
With qryBlobInsert do Begin
Sql.Text := 'INSERT INTO EVENTS (EventNo,EVENT_PHOTO) ' +
'VALUES (100,:ThePicture)';
Params.Clear;
Params.CreateParam(ftBlob,'ThePicture', ptInput);
ParamByName('ThePicture').LoadfromStream(TheStream,ftBlob);
ExecSQL;
End;
Finally
TheStream.Free;
End;
End;
:
```

This section about blob fields is not completed, yet. Please feel free to send me a private message or eMail if you have any issues about blobs in order to expand this section.

Sample project: "EasyQuery"

To prevent being too theoretical now we will create a small sample project tho demonstrate how the ZEOS components are used in general. We will create a small application that accesses the tables "Customer" and "Country" of the Firebird sample datbase "Employee". You should be able to navigate in table "Customer" and edit its data. And not bein too boring we will implement a DBLookupCombobox for field "Country" in table "Customer". This field is a foreign key that references to field "Country" in table "Country".

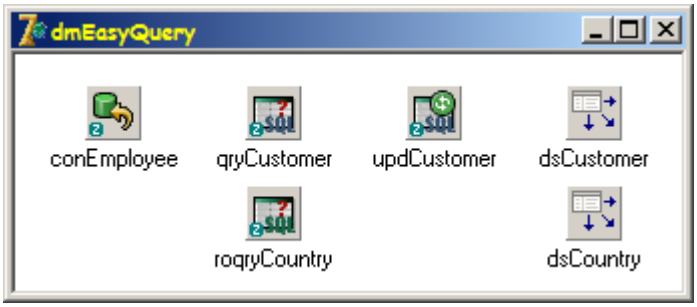
So let's get started!

First of all we have to create a new project in Delphi. Additionaly to the default form we have to create a DataModule.

The following properties of the DataModule have to be set:

Name: dmEasyQuery

Components for the DataModule



TZConnection

Database: Employee.fdb
Name: conEmployee
Password: "password"
Protocol: firebird-1.5
ReadOnly: False
TransactionIsolationLevel: tiReadCommitted
User: "username"

TZQuery

Connection: conEmployee
Name: qryCustomer
RequestLive: True
SQL: SELECT * FROM customer ORDER BY customer
UpdateObject: updCustomer

Note: You have to create persistent TFields for all tablefields using the fieldeditor!

The **OnAfterPost** event of qryCustomer will be implemented like this:

```
procedure TdmEasyQuery.qryCustomerAfterPost(DataSet: TDataSet);
begin
// Refresh of resultset to actualize (sort) data shown in DBGrid.
qryCustomer.Refresh;
end;
```

TZReadOnlyQuery

Connection: conEmployee
Name: roqryCountry
SQL: SELECT country FROM country ORDER BY 1

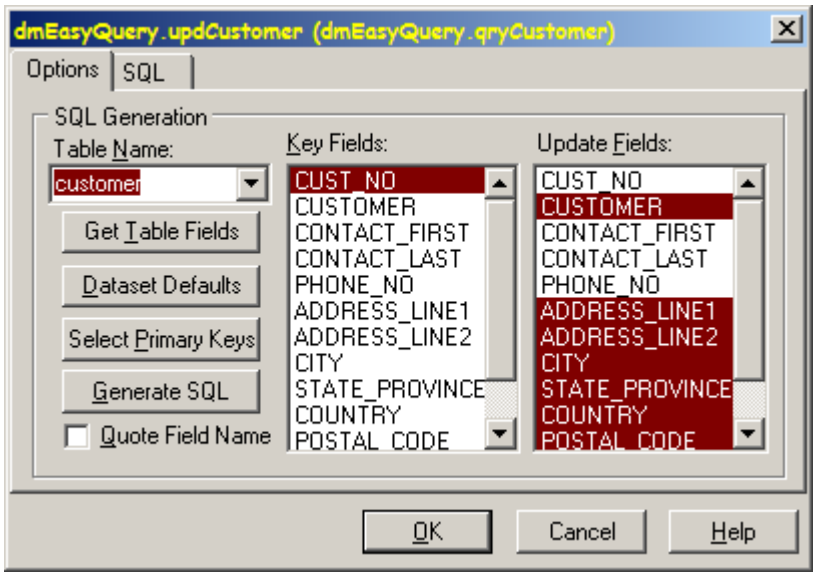
Note: You have to create persistent TFields for all tablefields using the fieldeditor!

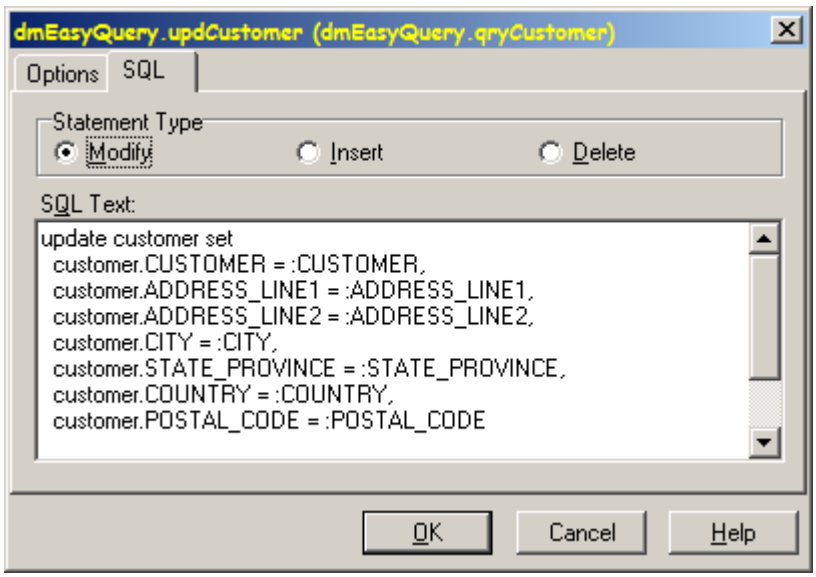
TZUpdateSQL

DeleteSQL: created by UpdateSql editor
InsertSQL: created by UpdateSql editor
ModifySQL: created by UpdateSql editor
Name: updCustomer

The delete-, insert- and modify-Statements for the TZUpdateSQL object are created with the UpdateSQL editor, automatically. The editor will be activated by double clicking on the TZUpdateSQL component. As key field you have to select the field CUST_NO. The fields ADDRESS_LINE1, ADDRESS_LINE2, CITY,STATE_PROVINCE, COUNTRY und POSTAL_CODE will selected as update fields in list "Update

Fields". Now the statements will be generated by clicking button "Generate SQL". If you only have such easy queries you can save a lot of typing by using the UpdateSQL editor. If it gets a little mor complex you should create the needed statements "by hand".





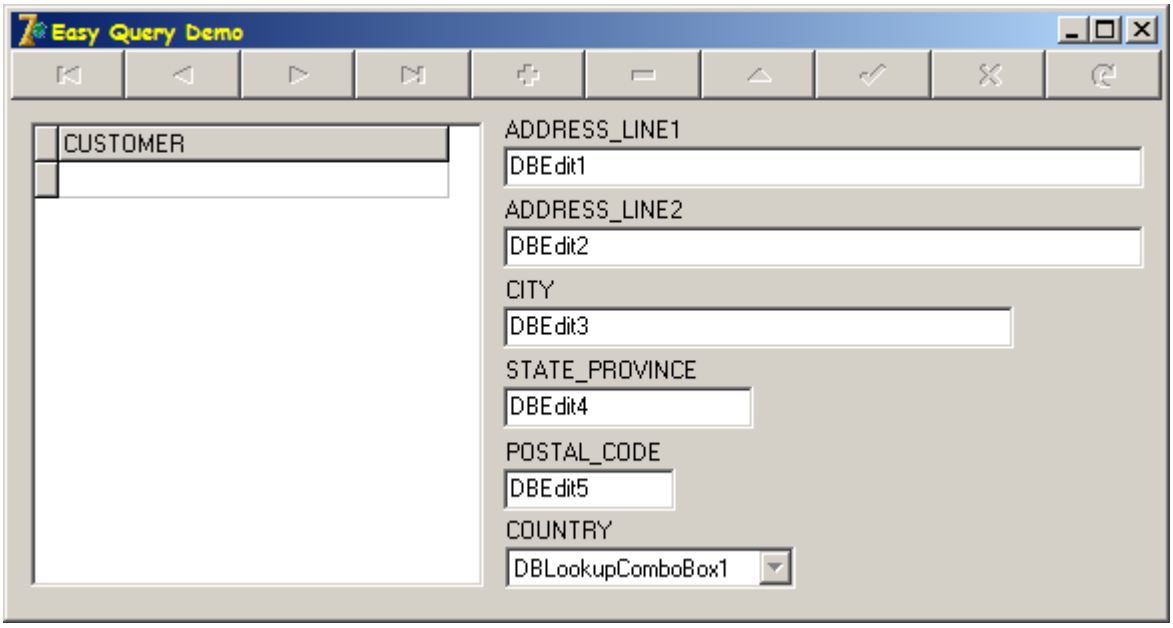
TDataSource

DataSet: sqlCustomer
Name: dsCustomer

TDataSource

DataSet: rosqlCountry
Name: dsCountry

Now the created DataModule will be saved as **dm_EasyQuery.pas**.[/list]
Components for the form



The main form will have the following components and is initialized as follows:

Properties

> **Caption**: Easy Query Demo
> **Name**: frmEasyQuery

In the Unit's interface you have to add dm_EasyQuery to the uses clause to get access to the database components.

The following events of the main form have to be implemented::

OnCreate
When creating the form the connection to the database will be established. After connecting to the database the queries will be opened:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
dmEasyQuery.conEmployee.Connect;
dmEasyQuery.qryCustomer.Open;
dmEasyQuery.roqryCountry.Open;
end;
```

OnDestroy
When closing the application (destroying the main form) the database connection will be cut. All queries will be closed automatically before disconnecting.

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
dmEasyQuery.conEmployee.Disconnect;
end;
```

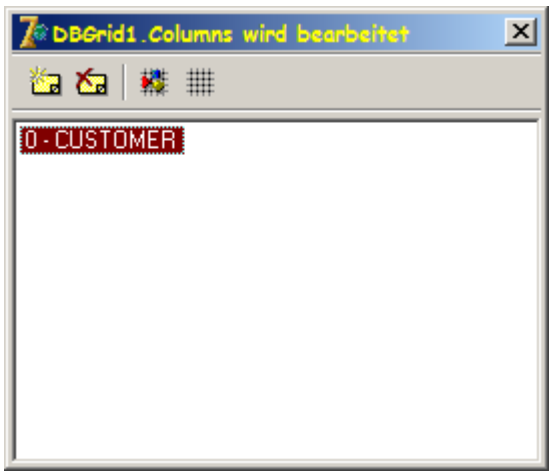
TLabel

> **Caption**: CUSTOMER

TDBGrid

DataSource: dmEasyQuery.dsCustomer
Options.dgTabs: False

In columneditor we will create a TColumn setting its property FieldName to "CUSTOMER". After that the according column in DBGrid1 will be enlarged.



TDBEdit

(5x)

TLabel]

(5x)

These objects will be created by using the columneditor of qryCustomer: Select columns ADDRESS_LINE1, ADDRESS_LINE2, CITY, STATE_PROVINCE and POSTAL_CODE and drag and drop them onto then main form. Align them and adapt them to the layout you see in the screenshot.

TLabel

Caption: COUNTRY

TDBLookUpComboBox

DataField: COUNTRY
DataSource: dmEasyQuery.dsCustomer
KeyField: COUNTRY
ListField: COUNTRY
ListSource: dmEasyQuery.dsCountry

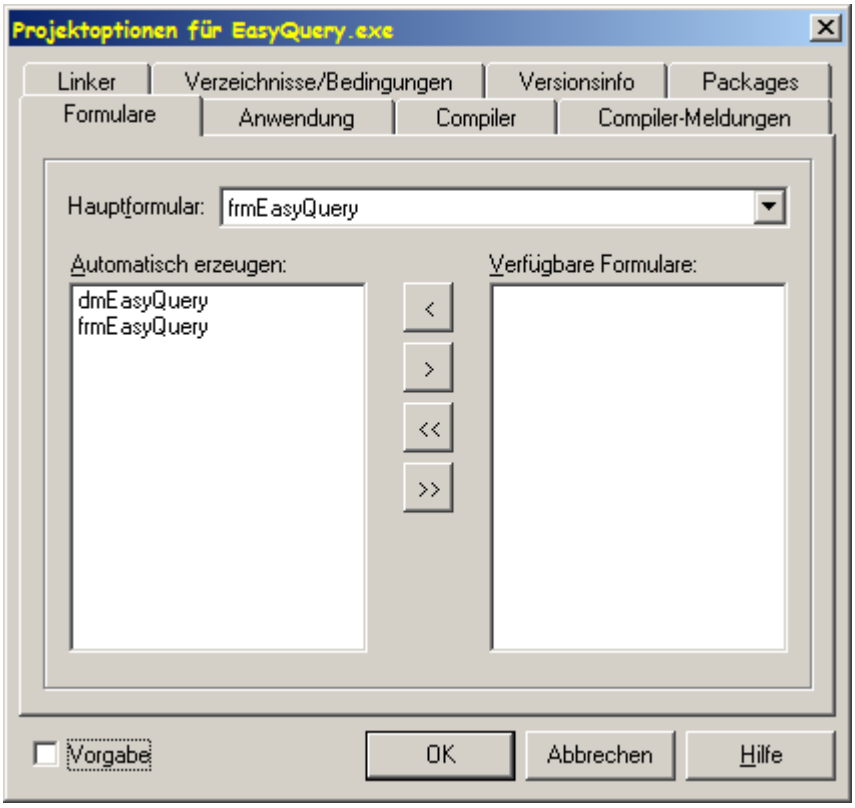
TDBNavigator

DataSource: dmEasyQuery.dsCustomer

Now the created form will be saved as **frm_EasyQuery.pas**.

Note: The following project options have to be changed:

dmEasyQuery has to be placed on top of the creation order in list "Create automatically". This ensures that all objects of **frmEasyQuery** may access the database objects. **frmEasyQuery** is still the main form.



Additional Examples

FishFact
This is the most popular database demo for Delphi. It was migrated to use ZEOS components.

Transactions
A sample application concerning "transactions with ZEOS". It uses a small self made test database.

StoredProc
This is a sample applikation that shows how to use stored proecedures. The database is the Firebird Employee sample database.

MasterDetail
A small application that shows how master/detail connections (server and client sided) will be implemented.

EventDemo
Also a Delphi database sample that was migrated from IBX to ZEOS. It uses the component TIBEventAlerter. This application needs the database Events (shipped with Delphi) that had to be migrated to dialect 3 to get this sample running.

You will find these Examples here for download.
A downloadable PDF-Version of this Tutorial is also on its way!

Michael Seeger
ZeosLib Development Team