FIBPlus Developer's Guide
Part I

Database connection

To connect to a database (DB) you should use the TpFIBDatabase component. For more details about its properties and methods read FIBPlus help file.

Connection parameters

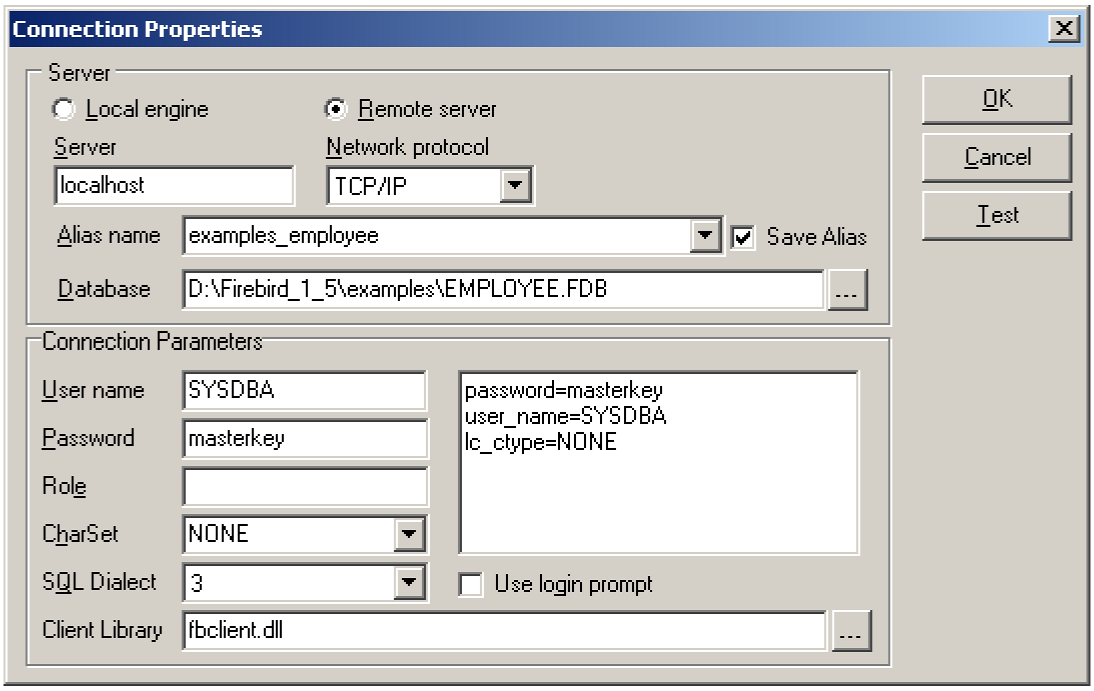
Connection parameters are typical for InterBase/Firebird server:

- path to a database file;
- user name and password;
- user role;
- charset;
- dialect;
- client library (gds32.dll for InterBase and fbclient.dll for Firebird).

To set all the properties at once you can use a built-in connection setting dialog (see picture 1).

The dialog «Database Editor» can be invoked from the component context menu (right click on the component) at design-time.

Here you may set all necessary parameters, including getting them from/saving them to Alias. You may also check whether the parameters are correct by using a test connection.



Server
Local engine [✓ Remote server]
Server: localhost
Network protocol: TCP/IP
Alias name: examples_employee
Save Alias ✓
Database: D:\Firebird_1_5\examples\EMPLOYEE.FDB

Connection Parameters
User name: SYSDBA
Password: masterkey
Role:
CharSet: NONE
SQL Dialect: 3
Client Library: fbclient.dll

```
password=masterkey
user_name=SYSDBA
lc_ctype=NONE
```

☐ Use login prompt

Figure 1. Connection Properties TpFIBDatabase

Similar to the actions in the dialog, you can do the same in the application code.

To connect to a database you should call the Open method or set the Connected property to True. It’s also possible to use this code to connect to a database:

```
function Login(DataBase: TpFIBDatabase; dbpath, uname, upass, urole: string):
Boolean;

begin
  if DataBase.Connected then DataBase.Connected := False;
  with FDataBase.ConnectParams do begin
    UserName := uname;
    Password := upass;
    RoleName := urole;
  end;
  DataBase.DBName  := dbpath;
  try DataBase.Connected := True;
  except
    on e: Exception do
      ShowMessage(e.Message);
  end;
  Result := DataBase.Connected;
end;
```

To close the connection either call the Close method or set the Connected property to False. You can also close all datasets and connected transactions at once:

```
procedure Logout(DataBase: TpFIBDatabase);
var i: Integer;
begin
  if not DataBase.Connected then
    Exit;
  for i := 0 to DataBase.TransactionCount - 1 do
    if TpFIBTransaction(DataBase.Transactions[i]).InTransaction then
      TpFIBTransaction(DataBase.Transactions[i]).Rollback
  DataBase.CloseDataSets;
  DataBase.Close;
   end;
```

How to create and drop database

It’s very easy to create a new DB. You need to set DB parameters and call the CreateDatabase method:

Delphi
```
with Database1 do begin
  DBParams.Clear;
  DBParams.Add('USER ''SYSDBA'' PASSWORD ''masterkey''');
  DBParams.Add('PAGE_SIZE = 2048');
  DBParams.Add('DEFAULT CHARACTER SET WIN1251');
  DBName := 'SERV_DB:C:\DB\TEST.IB';
  SQLDialect := 3;
end;

try
  Database1.CreateDataBase;
except
// Error handling
end;
```

C++
```
Database1->DBParams->Clear();
Database1->DBParams->Add("USER 'SYSDBA' PASSWORD 'masterkey'");
Database1->DBParams->Add("PAGE_SIZE = 2048");
Database1->DBParams->Add("DEFAULT CHARACTER SET WIN1251");
Database1->DBName = "SERV_DB:C:\\DB\\TEST.GDB";
Database1->SQLDialect = 3;

try
  { Database1->CreateDatabase(); }
catch (...)
{ // Error
}
```

To drop a database, use the DropDatabase method. Note: you should be connected to the database when using this method.

Metadata caching

FIBPlus enables developers to get system information about field tables automatically, set in TpFIBDataSet such field properties as Required (for NOT NULL fields), ReadOnly (for calculated fields) and DefaultExpression (for fields with default database values). This feature is very useful for both programmers and users, because programmers do not need to set property values manually when writing client applications, and users get clearer messages when working with the programme e.g. if any field is NOT NULL, and the user attempts to leave it empty, he will see a message «Field ‘…’ must have a value.». This is more understandable than a system InterBase/Firebird PRIMARY KEY violation error. The same with calculated fields, it is not possible to edit such fields, so FIBPlus will set the ReadOnly property to True for all calculated fields, and the user will not get a vague message on trying to change the field values in TDBGrid.

This feature has one disadvantage, which is revealed on low speed connections. To get information on fields, FIBPlus components execute additional “internal queries” on InterBase/Firebird system tables. If there are many tables in the application, or many fields in these tables, the application work can slow down anв net traffic increase. This becomes especially obvious on first query opening, as every open query is followed by the internal queires. On subsequent opening of the query, FIBPlus uses the information, which has already been obtained, but users may notice a slight work slowdown when the application starts.

This is where Metadata caching comes in. TpFIBDatabase enables developers to save metadata information at the client machine and use it during the applications execution and subsequent executions. The *TCacheSchemaOptions.*property is responsible for this process:

```
 TCacheSchemaOptions = class(TPersistent)
   property LocalCacheFile: string;
   property AutoSaveToFile: Boolean .. default False;
   property AutoLoadFromFile: Boolean .. default False;
   property ValidateAfterLoad: Boolean .. default True;
 end;
```

The *LocalCacheFile* property sets the name of the local cache file where this information will be saved. *AutoSaveToFile* helps to save cache to the file automatically on closing the application. *AutoLoadFromFile* loads cache from the file. And *ValidateAfterLoad* defines whether it's necessary to check the saved cache after its loading. Besides there is an *OnAcceptCacheSchema* event, where you may define objects, for which you don't need to load the saved information.

This property is also very easy-to-use.

```
with pFIBDatabase1.CacheSchemaOptions do begin
  LocalCacheFile := 'fibplus.cache';
  AutoSaveToFile := True;
  AutoLoadFromFile := True;
  ValidateAfterLoad:= True;
end;
```

To summarize, FIBPlus gathers information about fields accessed in a TpFIBDataset. This information is used to set properties on TField's. To reduce the overhead this introduces, the TpFIBDatabase can be set to save and load this cache between program executions.

BLOB field caching

BLOB field caching at the client is one more unique FIBPlus feature. Blobs are unique in InterBase/Firebird compared to other datatypes. When returned in a query, what is actually returned is a BLOB ID. When the field value is required, FIBPlus automatically asks the server for the data that relates to the BLOB ID, i.e. generating at least one more round trip to the server. This can create a performance bottleneck when the same BLOB is retrieved a number of times. BLOB field Caching helps reduce this performance hit.

If BlobSwapSupport.Active := True, FIBPlus will automatically save fetched BLOB fields in the defined directory (the SwapDir property).By default the SwapDir property is equal to {APP_PATH}, that is, it sets the directory with the executed application. You can also set the directory where to save BLOB fields. I.e., SwapDir := '{APP_PATH}' + '\BLOB_FILES\'

There are four events enabling work with this property in TpFIBDatabase:

```
property BeforeSaveBlobToSwap: TBeforeSaveBlobToSwap;
property AfterSaveBlobToSwap: TAfterSaveLoadBlobSwap;
property AfterLoadBlobFromSwap: TAfterSaveLoadBlobSwap;
property BeforeLoadBlobFromSwap: TBeforeLoadBlobFromSwap;
```

where

```
TBeforeSaveBlobToSwap = procedure(const TableName, FieldName: string;
RecordKeyValues: array of variant; Stream: TStream; var FileName: string; var
CanSave: boolean) of object;
TAfterSaveLoadBlobSwap = procedure(const TableName, FieldName: string;
RecordKeyValues: array of variant; const FileName: string) of object;
TBeforeLoadBlobFromSwap = procedure(const TableName, FieldName: string;
RecordKeyValues: array of variant; var FileName: string; var CanLoad:
boolean) of object;
```

You do not need to worry about these event handlers unless you want more control over how the BLOB field cache works. Event handlers help to manage BLOB field saving and reading from the disc. In particular you can forbid saving some BLOB field in event handlers depending on the field name and the table name, on other field values, free disc space, etc.

You can also save BLOB fields by using the MinBlobSizeToSwap property. There you can set a minimal BLOB field size for saving at disk..

This technology has a number of limitations:

1. The table must have a primary key.
2. BLOB fields must be read by the TpFIBDataSet component.
3. Your application must monitor the free disk space. For these purposes you can use the even handler BeforeSaveBlobToSwap.
4. Unfortunately all BLOB_ID's are changed after database backup/restore, so local cache becomes useless and is automatically cleared. After each application connection to the database, FIBPlus automatically opens a special thread, which checks in a separate connection the whole disk cache on BLOB fields. If there are no BLOB fields, the corresponding files are immediately deleted.

Client BLOB filters

These are user functions which help to handle (encrypt, pack, unpack, etc) BLOB fields at the client transparently for the user application. This feature helps you to pack or code blob fields in a database without changing the client application. FIBPlus has a mechanism of client BLOB filters similar to the one built in InterBase/Firebird. An advantage of a local blob-filter is an ability to decrease network traffic of the application considerably if you pack blob-fields before sending them to and then unpack them after getting to the client. This is done by means of registering two procedures for reading and writing blob-fields in TpFIBDatabase. As a result FIBPlus will automatically use these procedures to handle all blob-fields of the set type in all TpFIBDataSets using one TpFIBDatabase instance.

To illustrate this technology we will write an example of handlers which will pack and unpack each BLOB field and register these handlers. Note: Remember that user BLOB field subtype must be negative, the positive values allocated to InterBase/Firebird itself.

If you need to pack BLOB fields, write these two methods:

```
procedure PackBuffer(var Buffer: PChar; var BufSize: LongInt);
var srcStream, dstStream: TStream;
begin
  srcStream := TMemoryStream.Create;
  dstStream := TMemoryStream.Create;
  try
    srcStream.WriteBuffer(Buffer^, BufSize);
    srcStream.Position := 0;
    GZipStream(srcStream, dstStream, 6);
    srcStream.Free;
    srcStream := nil;
    BufSize := dstStream.Size;
    dstStream.Position := 0;
    ReallocMem(Buffer, BufSize);
    dstStream.ReadBuffer(Buffer^, BufSize);
  finally
    if Assigned(srcStream) then srcStream.Free;
    dstStream.Free;
  end;
end;
```

```
procedure UnpackBuffer(var Buffer: PChar; var BufSize: LongInt);
var srcStream,dstStream: TStream;
begin
  srcStream := TMemoryStream.Create;
  dstStream := TMemoryStream.Create;
  try
    srcStream.WriteBuffer(Buffer^, BufSize);
    srcStream.Position := 0;
    GunZipStream(srcStream, dstStream);
    srcStream.Free;
    srcStream:=nil;
    BufSize := dstStream.Size;
    dstStream.Position := 0;
    ReallocMem(Buffer, BufSize);
    dstStream.ReadBuffer(Buffer^, BufSize);
  finally
    if assigned(srcStream) then srcStream.Free;
    dstStream.Free;
  end;
end;
```

Now we need to register the two methods before connecting to a database. Call the RegisterBlobFilter function. The first parameter value is BLOB field type (equals to –15), the second and third are packing and unpacking functions:

pFIBDatabase1.RegisterBlobFilter(-15, @PackBuffer, @UnpackBuffer);

You can also see the demo example BlobFilters for more details.

Handling lost connections

FIBPlus provides the developers with a unique feature of handling lost connections between the client and server. TpFIBDatabase and TpFIBErrorHandler components are responsible for handling this.

You can see the example ConnectionLost for demonstration. The notes below will explain how it works. The TpFIBDatabase has three special events

AfterRestoreConnect – fires if the connection was restored.

OnLostConnect – fires on the lost connection if any operation with the database caused an error. Here you can specify one of the three following actions (see the `TOnLostConnectActions description`) - close the application, ignore the error report or try to restore the connection.

OnErrorRestoreConnect – fires if the connection was not restored.

In the example when the connection is lost, the user has a number of choices. If the connection has been restored successfully, the corresponding message is shown. If the error occurs, you can count the number of restore attempts or do any other necessary actions.

We will provide you with more details about TpFIBErrorHandler in the corresponding event. If the connection is lost, the event handler suppresses the standard exception message.

```
procedure TForm1.dbAfterRestoreConnect(Database: TFIBDatabase);
begin
  MessageDlg('Connection restored', mtInformation, [mbOk], 0);
end;

procedure TForm1.dbErrorRestoreConnect(Database: TFIBDatabase;
  E: EFIBError; var Actions: TOnLostConnectActions);
begin
  Inc(AttemptRest);
  Label4.Caption:=IntToStr(AttemptRest);
  Label4.Refresh
end;

procedure TForm1.dbLostConnect(Database: TFIBDatabase; E: EFIBError;
  var Actions: TOnLostConnectActions);
begin
  case cmbKindOnLost.ItemIndex of
   0: begin
       Actions := laCloseConnect;
       MessageDlg('Connection lost. TpFIBDatabase will be closed!',
        mtInformation, [mbOk], 0);
      end;
   1:begin
      Actions := laTerminateApp;
      MessageDlg('Connection lost. Application will be closed!',
       mtInformation, [mbOk], 0
      );
     end;
   2:Actions := laWaitRestore;
  end;
end;

procedure TForm1.pFibErrorHandler1FIBErrorEvent(Sender: TObject;
  ErrorValue: EFIBError; KindIBError: TKindIBError; var DoRaise: Boolean);
begin
  if KindIBError = keLostConnect then begin
    DoRaise := false;
    Abort;
  end;
end;
```

Other useful methods

The TpFIBDatabase component has many useful methods. We will consider the most common of them.

How to execute simple SQL- queries

If you need to execute a simple SQL query in order to get or set some application parameters, use the following methods:

```
function Execute(const SQL: string): boolean;
```

- executes an SQL-query, transferred in the SQL parameter, and returns True if the query was a success.

```
function QueryValue(const aSQL: string; FieldNo:integer; ParamValues:array of
variant; aTransaction:TFIBTransaction=nil):Variant; overload;
```

- it gets a field value with the FieldNo index as a result of executing aSQL in aTransaction. If you do not set the transaction, DefaultTransaction will be used. You can send parameters to the query. The value will be returned as a Variant variable. Use QueryValueAsStr to get a value as a string, and QueryValues – as an array. Remember that in this case the SQL must return not more than one string.

How to get generator values

Use the following method to get generator values:

```
function Gen_Id(const GeneratorName: string; Step: Int64; aTransaction:
TFIBTransaction = nil): Int64;
```

How to get information about tables and fields

```
procedure GetTableNames(TableNames: TStrings; WithSystem: Boolean);
procedure GetFieldNames(const TableName: string; FieldNames: TStrings;
WithComputedFields: Boolean = True);
```

The first method gets all table names and fills the TableNames list. The WithSystem parameter indicates whether to show system table names.

The second method takes TableName and fills FieldNames with the field names. The WithComputedFields parameter indicates whether to include COMPUTED BY fields.

Working with transactions

A transaction is an operation of database transfer from one consistent state to another.

All operations with the dataset (data/metadata changes) are done in the context of a transaction. To understand special FIBPlus features completely you need to know about InterBase / FIBPlus transactions. Please read the topic «Working with Transaction» in ApiGuide.pdf for InterBase.

All the changes done in the transaction can be either committed (in case there are no errors) by Commit or rolled back (Rollback). Besides these basic methods TpFIBTransaction has their context saving analogues: CommitRetaining and RollbackRetaining, i.e. on the client side, these will not close a TpFibQuery or TpFibDataset.

To start the transaction you should call the StartTransaction method or set the Active property to True. To commit the transaction call Commit/CommitRetaing, to roll it back - Rollback/RollbackRetaining.

TpFIBQuery and TpFIBDataSet components have some properties which help to control transactions automatically. In particular they are: the TpFIBDataSet.AutoCommit property; the poStartTransaction parameter in TpFIBDataSet.Options; qoStartTransaction and qoCommitTransaction in TpFIBQuery.Options.

How to set transaction parameters

Transaction parameters are not a trivial topic and require much explanation, so FIBPlus DevGuide won't cover the subject in detail. We highly recommend you to read InterBase ApiGuide.pdf to understand how transactions work.

Nevertheless in most cases you do not need to know about all peculiarities of transaction control at the API level. FIBPlus has a number of mechanisms which help developers' work easier. I.e. TpFIBTransaction has three basic transaction types: tpbDefault, tpbReadCommited, tpbRepeatableRead. At design time you can also create special types of your own in the TpFIBTransaction editor and use them as internal ones. Set the transaction type to set its parameters::

TpbDefault – parameters must be set in TRParams

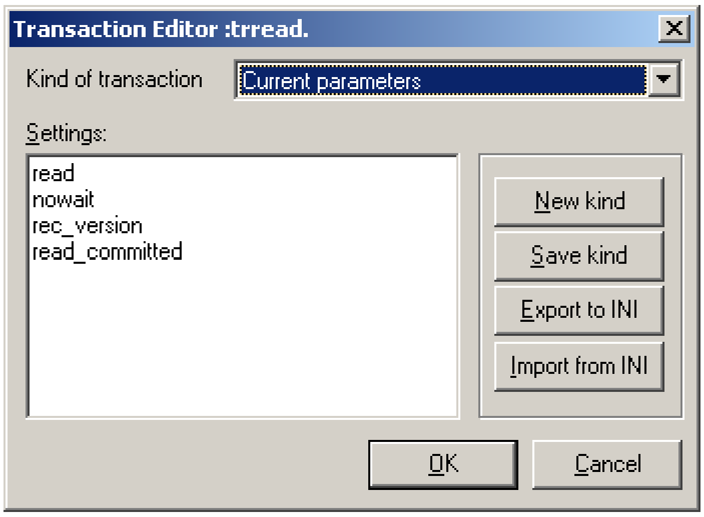
tbpReadCommited – shows the ReadCommited isolation level

tbpRepeatableRead – shows the RepeatableRead isolation level

Planning to use transactions in the application

Efficient InterBase/Firebird applications depend heavily on correct transaction use. In a multi-generation architecture (record versioning) Update transactions retain record versions.

So in general try to make the Update transactions as short as possible. Read-only transactions can remain open because they do not retain versions.



```
read
nowait
rec_version
read_committed
```

Figure 2. Transaction Editor

How to use SavePoints

InterBase/Firebird servers do not support nested transactions. But InterBase 7.X and Firebird 1.5 support SavePoints. FIBPlus realizes this functionality by three methods:

```
procedure SetSavePoint(const SavePointName:string);
procedure RollBackToSavePoint(const SavePointName:string);
procedure ReleaseSavePoint(const SavePointName:string);
```

The first method sets a save point with the SavePointName name. The second rolls the transaction back to SavePointName. The third releases SavePointName server resources.

SQL-query execution

An application works with a database by issuing SQL instructions. They are used to get and modify data\metadata. FIBPlus has a special TpFIBQuery component responsible for SQL operator execution. This robust, light and powerful component can perform any actions with the database.

TpFIBQuery is very easy-to-use: just set the TpFIBDatabase component, fill in the SQL property and call any ExecQuery method (ExecQueryWP, ExecQueryWPS).

NOTE: The tpFIBQuery is not a TDataset descendant, so it does not act in exactly the same way or exhibit the same methods / properties as you would expect to find in a dataset. For the TDataset descendant, please refer to the TpFIBDataset.

The example below will show how to create TpFIBQuery dynamically at run-time and thus get data about clients.

```
var sql: TpFIBQuery;

sql := TpFIBQuery.Create(nil);
with sql do
try
  Database := db;
  Transaction := db.DefaultTransaction;
  SQL.Text := 'select first_name, last_name from customer';
  ExecQuery;
  while not Eof do begin
    Memo1.Lines.Add(
      FldByName['FIRST_NAME'].AsString+' '+
      FldByName['LASTST_NAME'].AsString);
    Next;
  end;
  sql.Close;
finally
  sql.Free;
end;
```

How to transfer parameters

Very often you need to use parameters in SQL queries. To this end FIBPlus has the Params property and TpFIBQuery methods ParamsCount, ParamByName. Besides there are some ExecWP methods (execute with parameters), which execute queries with preset parameters. It's really easy to use parameters as you can see from the code samples below:

```
sql.SQL.Text :=
  'select first_name, last_name from customer'+
  'where first_name starting with :first_name';

{ variant 1 }
sql.ParamByName('first_name').AsString := 'A';
sql.ExecQuery;

{ variant 2 }
sql.ExecWP('first_name', ['A']);

{ variant 3 }
sql.ExecWP(['A']);
```

SQL- sections

FIBPlus provides developers with a wide range of SQL query control capabilities. In particular they are SQL sections: a list of fields, conditions, grouping and sorting order, query execution plan. Effectively FIBPlus parses your SQL and identifies the following elements of an SQL statement. These simple string properties can be read and modified:

FieldsClause – has a field list;

MainWhereClause – has the main clause WHERE (see the details below);

OrderClause – has the clause «order by»;

GroupByClause – has the clause «group by»;

PlanClause – has the clause «plan».

FIBPlus also has unique features, such as macros and conditions – an extended mechanism of work with the WHERE clause. The sections below will provide you with more details about these features.

Macros

Macros help you to operate with the variable parts of your queries. This allows you to change and specify the queries without rewriting the query code.

The macro syntax is **@@<MACROS_NAME>[%<DEFAULT_VALUE>][#]@**

So macro is a specific order of symbols between the marks @@ and @. The parameter <MACROS_NAME> is obligatory after @@. You can also set the default macro value after the symbol "%". Besides you can set the parameter # (not obligatory), it will make FIBPlus write parameter names in inverted commas.

Macros are used similar to parameters. This code example will demonstrate you this similarity:

```
Sql.SQL.Text := 'select * from @@table_clause@ where @@where_clause% 1=1@';
Sql.ExecWP(['CUSTOMER','FIRST_NAME STARTING WITH ''A''']);
```

Call the SetDefaultMacroValue method of the object parameter to set the default macro value.

Macro can also have a parameter. To search for it use the FindParam function, to set the parameter use the ParamByName method:

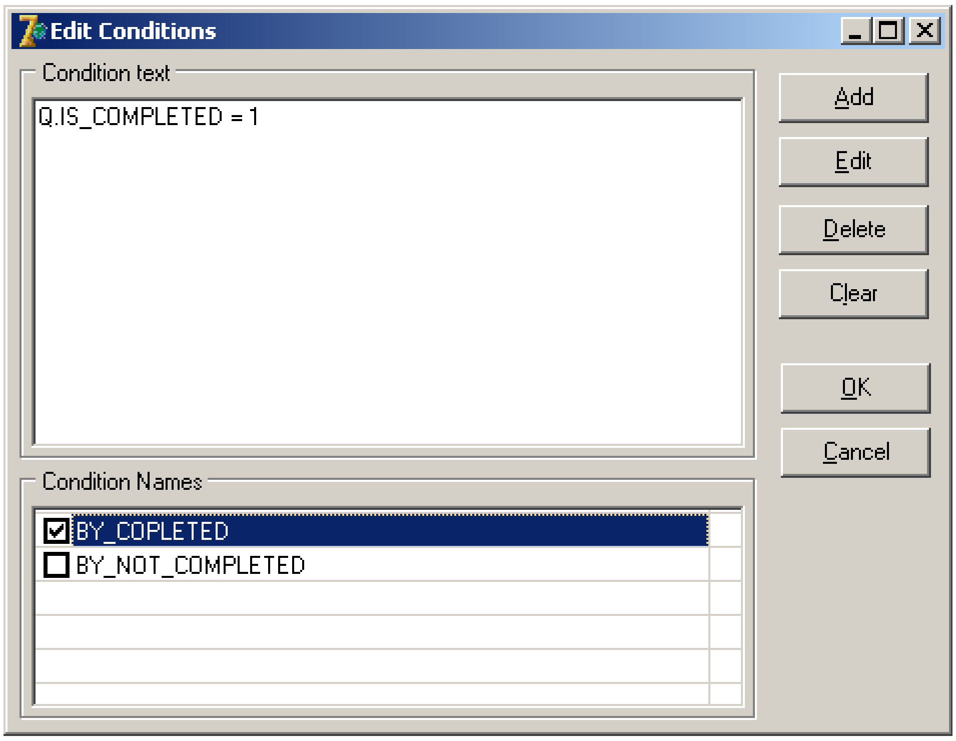
```
Sql.SQL.Text := 'select * from @@table_clause@ where @@where_clause% 1=1@';
Sql.Params[0].AsString := 'CUSTOMER';
Sql.Params[1].AsString := 'CUST_NO = :CUST_NO';
if Assigned(Sql.FindParam('CUST_NO')) then
  Sql.ParamByName('CUST_NO').AsInteger := 1001;
Sql.ExecQuery;
```

You can also see the code example ServerFilterMarcoses to get to know how to use macros for TpFIBDataSet

Conditions

The mechanism of conditions is another option to change the variable part of your SQL-queries

You can set one or more parameter conditions for any SQL at design time or runtime. The built-in dialog shown in picture 3 is very convenient for these purposes.



Q.IS_COMPLETED = 1

☑ BY_COPLETED
☐ BY_NOT_COMPLETED

Figure 3. Edit Conditions.

To make the condition active you should set the Active property to True.

pFIBQuery1.Conditions[0].Active := True;

or

pFIBQuery1.Conditions.ByName('by_customer').Active := True;

This is a sample code showing how to work with Conditions:

```
if pFIBQuery1.Open then pFIBQuery1.Close;
pFIBQuery1.Conditions.CancelApply;
```

```
  pFIBQuery1.Conditions.Clear;
  if byCustomerFlag then
    pFIBQuery1.Conditions.AddCondition('by_customer', 'cust_no = 1001',
True);
  pFIBQuery1.Conditons.Apply;
  pFIBQuery1.Open;
```

TpFIBDataSet has two additional methods CancelConditions and ApplyConditions, which call correspondingly Conditions.Cancel and Conditions.Apply. This code sample for TpFIBDataSet is simpler than the previous.

```
with pFIBDataSet1 do begin
  if Active then Close;
  CancelConditioins;
  Conditions.Clear;
  if byCustomerFlag then Conditions.AddCondition('by_customer', 'cust_no =
1001', True);
  ApplyConditions;
  Open;
end;
```

You can also see the example ServerFilterConditions to get to know how to set conditions for TpFIBDataSet.

Batch processing

FIBPlus has internal methods for batch processing, called Batch methods. These methods can be helpful for replication between databases and import/export operations.

```
function BatchInput(InputObject: TFIBBatchInputStream) :boolean;
function BatchOutput(OutputObject: TFIBBatchOutputStream):boolean;
procedure BatchInputRawFile(const FileName:string);
procedure BatchOutputRawFile(const FileName:string;Version:integer=1);
procedure BatchToQuery(ToQuery:TFIBQuery; Mappings:TStrings);
```

The Version parameter is responsible for format compatibility with old file versions, created by FIBPlus BatchOutputXXX method. If Version = 1, FIBPlus uses an old principle of work with file versions: the external file keeps data ordered by SQL query fields. It is supposed that on reading data saved by the BatchInputRawFile method, parameters will have the same order in the reading SQL. The number of TpFIBQuery fields (the source of the data) must coincide to the number of TpFIBQuery parameters which will read the data. For string fields it is important to have the same length for the field being written and for the reading parameter whereas their names can differ.

If Version = 2, FIBPlus uses a new principle of writing data. Besides the data, the file also keeps system information about fields (name, type and length). On reading the data, it will be chosen by similar names. The order and number of fields in the writing TpFIBQuery can differ from those of parameters in the reading TpFIBQuery. Their types and length can also differ. Only names must coincide.

It’s very easy to work with batch methods. We will show this using a simple example. The code below consists of three parts. The first saves data about clients into an external file, the second loads them into a database and the third shows how to change the data.

```
{ I }
pFIBQuery1.SQL := 'select EMP_NO, FIRST_NAME, LAST_NAME from CUSOMER';
pFIBQuery1.BatchOutputRawFile('employee_buffer.fibplus', 1);

{ II }
pFIBQuery1.SQL := 'insert into employees(EMP_NO, FIRST_NAME, LAST_NAME)'+
  ' values(:EMP_NO, :FIRST_NAME, :LAST_NAME)';
```

```
pFIBQuery1.BatchInputRawFile('employee_buffer.fibplus');

{ III }
pFIBQuery1.SQL := 'select EMP_NO, FIRST_NAME, LAST_NAME from CUSOMER';
pFIBQuery2.SQL := 'insert into tmp_employees(EMP_NO, FIRST_NAME, LAST_NAME)'+
  ' values(:EMP_NO, :FIRST_NAME, :LAST_NAME)';

mapStrings.Add('EMP_NO=EMP_NO');
mapStrings.Add('FIRST_NAME=FIRST_NAME');
mapStrings.Add('LAST_NAME=LAST_NAME');

pFIBQuery1.BatchToQuery(pFIBQuery2, mapStrings);
```

We will discuss more about batch processing when talking about TpFIBDataSet as it also has some batch processing methods.

The OnBatchError event occurs in case of incorrect processing. Using the parameter BatchErrorAction (TBatchErrorAction = (beFail, beAbort, beRetry, beIgnore)) in the code of this event you can decide what to do in this case.

Execution of stored procedures

Execution of stored procedures is very similar to query execution. You only need to write 'execute procedure some_proc(:proc_param)' in the SQL text or 'select * from some_proc(:proc_param)' for selectable procedures (i.e. those that return a result set).

If non-selectable procedure returns any results you can get them after the query executing using the Fields property.

```
Sql.SQL.Text := 'execute procedure some_proc(:proc_param)';
Sql.ExecWP([25]);
ResultField1 := Sql.Fields[0].AsInteger;
```

This feature makes FIBPlus different from BDE, ADO and other libraries where input data are also available through parameters.

Besides, FIBPlus enables developers to execute stored procedures by using the TpFIBStoredProc component. TpFIBStoredProc is a direct TpFIBQuery descendant with the StoredProcName property. **It is recommended to use TpFIBStoredProc to execute non-selectable procedures.**

How to execute DDL(Data Definition Language) commands.

DDL is the subset of SQL that allows you to change the structure of the database, e.g. to create tables.

Besides SQL-operators TpFIBQuery helps to execute DDL- commands. In order to be able to execute a DDL-command you need to set the ParamsCheck property to False. New FIBPlus versions support macros for DDL.

Recurrent use of queries

All client libraries including FIBPlus have to transfer the complete query text in order to prepare a query for execution. Once you have a prepared query it is enough to transfer only handle and parameter values. FIBPlus knows when you have changed the SQL in a TpFIBQuery, so it will only prepare when it is required. By TpFIBPLus handling the preparing of queries for you, it ensures re-using the same component for the same query, merely changing the parameters, will offer optimum performance.

If however, your application does not lend itself to using the same query component for the same query, FIBPlus offers a query pool mechanism. If there are numerous similar query requirements in your application you can use methods from pFIBCacheQueries.pas to manage the recurrent use:

```
function GetQueryForUse (aTransaction: TFIBTransaction; const SQLText:
string): TpFIBQuery;
procedure FreeQueryForUse (aFIBQuery: TpFIBQuery);
```

You don't need to create TpFIBQuery instances. Being called for the first time the GetQueryForUse procedure will create a TpFIBQuery instance and then will return a link to the existing component when you execute the same query again and again. As you see, on every recurrent procedure call FIBPlus will use the prepared query and thus transfer the query text to the server only once. When you don't need the recurrent query anymore (when the query results are obtained from the TpFIBQuery component) you should call the FreeQueryForUse method. Such FIBPlus mechanism is used for internal purposes i.e. on calling generators to get primary key values. You can use these methods in your applications to optimize network traffic.

Work with datasets

The TpFIBDataSet component is responsible for work with datasets. It is based on the TpFIBQuery component and helps to cache selection results. TpFIBDataSet is a TDataSet descendant so it supports all TDataSet properties, events and methods. To get more information about TDataSet please read Delphi/C++Builder help manuals.

Basic principles of work with datasets

TpFIBDataSet enables developers to select, insert, update and delete data. All these operations are executed by TpFIBQuery components in TpFIBDataSet.

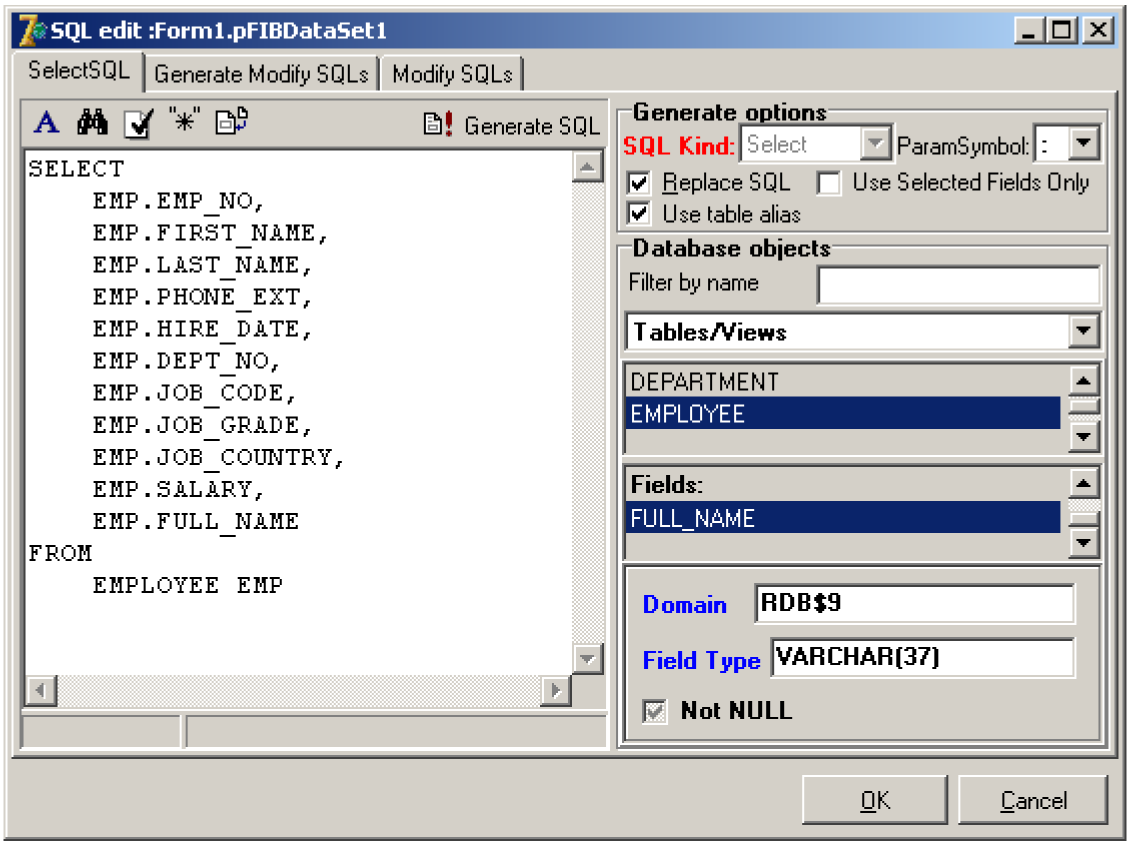
To select data you set the SelectSQL property. It's similar to setting the SQL property of the QSelect component (TpFIBQuery type). Define the InsertSQL.Text property to insert data, UpdateSQL.Text to update, DeleteSQL.Text to delete and RefreshSQL.Text to refresh the data.

We will use a demo database employee.gdb (or .fdb for Firebird) to show how to write Select SQL and get a list of all employees. We will write all queries in InsertSQL, UpdateSQL, etc.

```
with pFIBDataSet1 do begin
  if Active then Close;
  SelectSQL.Text :=
  'select CUST_NO, CUSTOMER, CONTACT_FIRST, CONTACT_LAST from CUSTOMER';

  InsertSQL.Text :=
  'insert into CUSTOMER(CUST_NO, CUSTOMER, CONTACT_FIRST, CONTACT_LAST )'+
  ' values (:CUST_NO, :CUSTOMER, :CONTACT_FIRST, :CONTACT_LAST)';

  UpdateSQL.Text :=
  'update CUSTOMER set CUSTOMER = :CUSTOMER, '+
  'CONTACT_FIRST = :CONTACT_FIRST, CONTACT_LAST = :CONTACT_LAST '+
  'where CUST_NO = :CUST_NO';

  DeleteSQL.Text := 'delete from CUSTOMER where CUST_NO = :CUST_NO';

  RefreshSQL.Text :=
  'select CUST_NO, CUSTOMER, CONTACT_FIRST, CONTACT_LAST ' +
  'from CUSTOMER where CUST_NO = :CUST_NO';
  Open;
end;
```

To open TpFIBDataSet either execute Open/OpenWP methods or set the Active property to True. To close TpFIBDataSet call the Close method.

Don't be concerned by seeing a lot of code lines, all these queries can be automatically created by the TpFIBDataSet editor. You can call it from the component context menu, as shown in picture 4.



```
SELECT
    EMP.EMP_NO,
    EMP.FIRST_NAME,
    EMP.LAST_NAME,
    EMP.PHONE_EXT,
    EMP.HIRE_DATE,
    EMP.DEPT_NO,
    EMP.JOB_CODE,
    EMP.JOB_GRADE,
    EMP.JOB_COUNTRY,
    EMP.SALARY,
    EMP.FULL_NAME
FROM
    EMPLOYEE EMP
```

SQL Kind: Select ParamSymbol: :
[✓] Replace SQL [ ] Use Selected Fields Only
[✓] Use table alias
Filter by name
Tables/Views
DEPARTMENT
EMPLOYEE
Fields:
FULL_NAME
Domain RDB$9
Field Type VARCHAR(37)
[✓] Not NULL

Figure 4. TpFIBDataSet SQL Editor

The example DataSetBasic demonstrates basic use of editable TpFIBDataSet.

Automatic generation of Update queries

Besides TpFIBDataSet SQL editor, FIBPlus can generate all Update queries at run-time in a more effective way than at design-time.

For this purpose use the AutoUpdateOptions property. This group of setting is very important as it makes the coding process simpler and more convenient.

```
TAutoUpdateOptions= class (TPersistent)
  property AutoParamsToFields: Boolean .. default False;
  property AutoReWriteSqls: Boolean .. default False;
  property CanChangeSQLs: Boolean .. default False;
  property GeneratorName: string;
  property GeneratorStep: Integer .. default 1;
  property KeyFields: string;
  property ParamsToFieldsLinks: TStrings;
  property SeparateBlobUpdate: Boolean .. default False;
  property UpdateOnlyModifiedFields: Boolean .. default False;
  property UpdateTableName: string;
  property WhenGetGenID: TWhenGetGenID .. default wgNever;
end;
```

```
TWhenGetGenID=(wgNever,wgOnNewRecord,wgBeforePost);
```

AutoRewriteSQLs - `If there are empty SQLText properties for` InsertSQL, UpdateSQL, DeleteSQL, RefreshSQL they will be automatically generated by the SelectSQL, KeyFields and UpdateTableName properties.

CanChangeSQLs informs that non-empty queries can be rewritten.

GeneratorName sets the generator name and *GeneratorStep* sets the generator step.

KeyFields contains a list of key fields.

SeparateBlobUpdate manages BLOB-field writing in a database. If *SeparateBlobUpdate* is set to True at first a record will be saved without a BLOB-field and then if the operation is a success the BLOB-fields will be also written to the database.

If *UpdateOnlyModifiedFields* is set to True and if *CanChangeSQLs* is set to True, a new SQL query will be automatically created for each modifying operation. This SQL query will contain only fields, which were changed.

UpdateTableName must contain a name of the modified table.

WhenGetGenId enables developers to set a mode of using a generator to form a primary key. Either not to generate the primary key, to generate it on adding a new record, or before Post.

So due to AutoUpdateOptions settings FIBPlus helps not to generate modifying queries at design-time and for this at run-time. To use this feature you just need to write a name of modified table and the key field.

The code below is taken from the example AutoUpdateOptions. You can use this feature both at design-time and run-time:

```
pFIBDataSet1.SelectSQL.Text := 'SELECT * FROM EMPLOYEE';
pFIBDataSet1.AutoUpdateOptions.AutoReWriteSqls := True;
pFIBDataSet1.AutoUpdateOptions.CanChangeSQLs   := True;
pFIBDataSet1.AutoUpdateOptions.UpdateOnlyModifiedFields := True;
pFIBDataSet1.AutoUpdateOptions.UpdateTableName := 'EMPLOYEE';
pFIBDataSet1.AutoUpdateOptions.KeyFields       := 'EMP_NO';
pFIBDataSet1.AutoUpdateOptions.GeneratorName   := 'EMP_NO_GEN';
pFIBDataSet1.AutoUpdateOptions.WhenGetGenID    := wgBeforePost;
pFIBDataSet1.Open;
```

Local sorting

FIBPlus has local sorting methods, which helps you to order TpFIBDataSet buffer in any possible way as well as the ability to remember and restore sorting after TpFIBDataSet has been closed and reopened. In addition FIBPlus enables developers to use the mode where all newly inserted and changed records will be placed to the correct buffer position according to the sorting order. To sort TpFIBDataSet buffer you should call any of three following methods.

```
procedure DoSort(Fields: array of const; Ordering: array of Boolean);
virtual;
procedure DoSortEx(Fields: array of integer; Ordering: array of Boolean);
overload;
procedure DoSortEx(Fields: TStrings; Ordering: array of Boolean); overload;
```

The first parameters defines the list of fields, the second is an array of sorting orders. If you set the sorting order to True it will be ascending, to False – descending.

Here are two examples of the same sorting by two fields in the ascending order:

```
pFIBDataSet1.DoSort(['FIRST_NAME', 'LAST_NAME'], [True, True]);
```
or
```
pFIBDataSet1.DoSortEx([1, 2], [True, True]);
```

You can get the current sorting order by checking the following properties:

```
function SortFieldsCount: integer;
  returns the sorting field number
```

`function SortFieldInfo(OrderIndex:integer): TSortFieldInfo –`
returns information about sorting in the OrderIndex position.

`function SortedFields:string`
returns a string with sorting fields enumerated by ';'

```
TSortFieldInfo = record
  FieldName: string;         //field name
  InDataSetIndex: Integer;   //whether included into index
  InOrderIndex: Integer;     //included ...
  Asc: Boolean;              //True, if ascendant
  NullsFirst: Boolean;       //True, if Null values are the first
end;
```

Set poKeepSorting property to True to put records to the right buffer position on inserting and editing. Use the psGetOrderInfo option if you want to TpFIBDataSet automatically use the local sorting order defined in the ORDER BY query statement.

Set psPersistentSorting option to True in order to keep the sorting order on TpFIBDataSet reopening. Be careful, if you have huge selections this feature will not be effective because at first the buffer will retrieve all records from the server and only then sort them.

Sorting of national symbols

Two parameters are responsible for correct sorting of national symbols: CHARSET (set of symbols) and COLLATION (sorting order). Even if you correctly set these parameters in a database and queries TpFIBDataSet may sort them incorrectly. The point is that default local sorting uses the simplest method without comparing national symbols. As if NONE charset is set.

TpFIBDataSet can sort national symbols according to the national charset. If you use the OnCompareFieldValues event you can compare char fields in an alternative way. There are three standard methods for Ansi-sorting:

```
protected function CompareFieldValues(Field:TField;const
S1,S2:variant):integer; virtual;

public function AnsiCompareString(Field:TField;const val1, val2: variant):
Integer;

public function StdAnsiCompareString(Field:TField;const S1, S2: variant):
Integer;
```

AnsiCompareString is case-sensitive, and StdAnsiCompareString is not.

```
pFIBDataSet1.OnCompareFieldValues := pFIBDataSet1.AnsiCompareString;
pFIBDataSet1.OnCompareFieldValues := pFIBDataSet1.StdAnsiCompareString;
```

You can set any standard methods for CompareFieldValues as default or write your own method if necessary.

Local filtering

In contrast to IBX, TpFIBDataSet has full support of local filtering. It supports the Filter and Filtered properties and the OnFilterRecord event.

The table below shows the additional operators which can be used in the Filter property.

Operation	*Description*
<	Less than.
>	Greater than.
>=	Greater than or equal.
<=	Less than or equal.
=	Equal to
<>	Not equal to
AND	Logical AND
NOT	Logical NOT
OR	Logical OR
IS NULL	Tests that a field value is null
IS NOT NULL	Tests that a field value is not null
+	Adds numbers, concatenates strings, adds number to date/time values
–	Subtracts numbers, subtracts dates, or subtracts a number from a date
*	Multiplies two numbers
/	Divides two numbers
Upper	Upper-cases a string
Lower	Lower-cases a string
Substring	Returns the substring starting at a specified position
Trim	Trims spaces or a specified character from front and back of a string
TrimLeft	Trims spaces or a specified character from front of a string
TrimRight	Trims spaces or a specified character from back of a string
Year	Returns the year from a date/time value
Month	Returns the month from a date/time value
Day	Returns the day from a date/time value
Hour	Returns the hour from a time value

Operation	*Description*
Minute	Returns the minute from a time value
Second	Returns the seconds from a time value
GetDate	Returns the current date
Date	Returns the date part of a date/time value
Time	Returns the time part of a date/time value
Like	Provides pattern matching in string comparisons
In	Tests for set inclusion
*	Wildcard for partial comparisons.

If the Filter property is set to True and is not active, you can use the following methods to navigate on records which satisfy the filtering conditions:

FindFirst

FindLast

FindNext

FindPrior

In case of huge data sets we recommend you to use the server filtering. It can be realized by macros and conditions mechanisms.

To know more about local filtering see the example LocalFiltering.

Important: use the VisibleRecordCount function instead of RecordCount to get the number of records by the Filter condition.

Data search

TpFIBDataSet supports Locate, LocateNext and LocatePrior methods, which are described in a standard Delphi/C++Builder help manual.

Besides FIBPlus has some specific analogues which have some important advantages^

```
function ExtLocate(const KeyFields: String; const KeyValues: Variant;
Options: TExtLocateOptions): Boolean;

function ExtLocateNext(const KeyFields: String; const KeyValues: Variant;
Options: TExtLocateOptions): Boolean;

function ExtLocatePrior(const KeyFields: String; const KeyValues: Variant;
Options: TExtLocateOptions): Boolean;
```

TExtLocateOptions = (eloCaseInsensitive, eloPartialKey, eloWildCards, eloInSortedDS, eloNearest, eloInFetchedRecords)

eloCaseInsensitive	to ignore the case;
eloPartialKey	partial coincidence
eloWildCards	search by wild cards (similar to LIKE operator);
eloInSortedDS	search in a sorted dataset (influences the search speed);
eloNearest	only together with eloInSortedDS. It is placed where “must be”;
eloInFetchedRecords	search only in fetched records.

Master and detail datasets

Besides a standard TDataSet Master-detail mechanism FIBPlus has an additional group of options DetailCondition, described as:

```
TDetailCondition=(dcForceOpen,dcIgnoreMasterClose,dcForceMasterRefresh,
dcWaitEndMasterScroll);
TDetailConditions= set of TDetailCondition;
```

dcForceOpen	if it is active, the detail TpFIBDatSets will be open on opening the master ;
dcIgnoreMasterClose	if it is active, the detail TpFIBDatSet won't close on closing the master;
dcForceMasterRefresh	if it is active, the current master record will be refreshed on refreshing the the detail TpFIBDatSet;
dcWaitEndMasterScroll	if it is active, on scrolling at master TpFIBDataSet will wait a little before reopening the detail. This option helps to avoid useless operations if the master navigation is simple.

Pessimistic locking

Standard record changing behavior of InterBase/Firebird servers is optimistic locking. If two or more users edit the same record at the same time, only the first modification is written to the database, and the second receives an exception error.

As a rule if you need a pessimistic locking in InterBase/Firebird, you use a «dummy update». It means that the record is updated i.e. by the primary key before record editing:

update customer set cust_no = cust_no where cust_no = :cust_no

Then the actual record is automatically fetched from the server. This behaviour helps to guarantee that the record won't be updated from another transaction before the end of the dummy update transaction.

FIBPlus manages this process automatically. You need to activate the psProtectedEdit option or use the the TpFIBDataSet.LockRecord method.

The demo example ProtectedEditing demonstrates how this feature works.

Work in the confined local buffer mode – for huge datasets and random access

The mode was first suggested by Sergey Spirin in gb_Datasets components. Now FIBPlus is also capable of this feature (since version 6.0). It enables navigation of TpFIBDataSet without fetching all the records returned by the query. In fact it is simulation of random access to records by means of supplementary queries. The technology sets a number of query requirements. In particular, one of the obligatory requirements is use of ORDER BY in SelectSQL. First, in ORDER BY it's important to make the combination of field values unique. Second, to speed up data transfer time it's better to have two indices - ascending and descending - for this field combination.

This simple example illustrates the technology. Having such a query in SelectSQL:

```
SELECT * FROM TABLE
ORDER BY FIELD
```

You may get some first records, fetching them successively. To see the last records immediately, you may execute an additional query with descending sorting instead of querying all the records from the server,:

```
SELECT * FROM TABLE
ORDER BY FIELD DESC
```

Obviously successive fetching of several records will result the last records (in relation to the initial query). Similar queries are for exact positioning on any record, as well as on records below and above the current one:

```
SELECT * FROM TABLE
WHERE (FIELD = x)

SELECT * FROM TABLE
WHERE (FIELD < x)
ORDER BY FIELD DESC

SELECT * FROM TABLE
WHERE (FIELD > x)
ORDER BY FIELD
```

To carry out this technology TpFIBDataSet has a new property:

```
property CacheModelOptions:TCacheModelOptions, where

TCacheModelOptions = class(TPersistent)
property BufferChunks: Integer ;
property CacheModelKind: TCacheModelKind ;
property PlanForDescSQLs: string ;
end;
```

BufferChunks replaces the existing property BufferChunks of TpFIBDataSet. The TCacheModelKind type can have a cmkStandard value for the standard local buffer work and a cmkLimitedBufferSize value for the new technology of the confined local buffer. The buffer size is a number of records set in BufferChunks.

The PlanForDescSQLs property enables to set a separate plan for queries with descending sorting.

Note: when using the technology of the confined local buffer,

- You must not activate the CachedUpdate mode;
- The RecNo property will return incorrect values;
- Local filtering will not be supported;
- Work with BLOB-fields may be not stable in the present version;
- You should activate the psGetOrderInfo option in PrepareOptions.

Work with the internal dataset cache

TpFIBDataSet has several special methods for work with its internal record cache (this allows you to excute prior without having to go back to the server). Actually these methods make TpFIBDataSet an analogue of TClientDataSet oriented at InterBase. Its only difference from TClientDataSet is that there must be a connection with the database and SelectSQL must have a correct query. Despite these restrictions the mechanism is very flexible and helps to realize numerous “non standard” things. For example this query will select one Integer field and one String:

```
select cast(0 as integer) some_id, cast('' as varchar(255)) some_name

from RDB$DATABASE.
```

This would, if executed, return one row with an integer field (set to 0) and a string field with an empty string.

You can open this TpFIBDataSet by calling the CacheOpen method. Then you can use the following methods:

```
procedure CacheModify(aFields: array of integer; Values: array of Variant;
KindModify: byte );
procedure CacheEdit(aFields: array of integer; Values: array of Variant);
procedure CacheAppend(aFields: array of integer; Values: array of Variant);
overload;
procedure CacheAppend(Value: Variant; DoRefresh: boolean = False); overload;
procedure CacheInsert(aFields: array of integer; Values: array of Variant);
overload;
procedure CacheInsert(Value: Variant; DoRefresh: boolean = False); overload;
procedure CacheRefresh(FromDataSet: TDataSet; Kind: TCachRefreshKind ;
FieldMap: Tstrings);
procedure CacheRefreshByArrMap( FromDataSet: TDataSet; Kind:
TCachRefreshKind; const SourceFields, DestFields: array of string )
```

to add a record execute:

pFIBDataSet1.CacheInsert([0,1],[255, 'string1'])

to modify:

pFIBDataSet1.CacheModify([0,1],[255, 'string1'])

to delete from cache, call CacheDelete;

The CacheRefresh and CacheRefreshByArrMap methods enable to refresh a record on basis of data from another TpFIBDataSet.

All these operations do not change the database as they are executed in TpFIBDataSet cache.

Sometimes you can also use this technique in a standard mode. For example when you need to insert a record using some complex stored procedure, which returns the code of the inserted record, and then to show the code in TpFIBDataSet. You can insert the code and call the Refresh method:

id := SomeInsertByProc;

pFIBDataSet1.CacheInsert([0], [1]);

pFIBDataSet1.Refresh;

In addition you could also delete some non-existing records from cache without having to refresh the query.

FIBPlus Developers Guide
Part II

Working with BLOB fields

There can be advantages in storing non-structured data in your database, such as images, OLE-objects, sounds, etc. For this you will need to use a special data type - BLOB. SQL queries for BLOB fields do not differ from queries for standard field types; work with these fields in TpFIBDataSet does not differ from TDataSet work. The only difference from an ordinary data type is that you should use streams (special TStream descendants) to set values of the BLOB-parameter.

Before setting the BLOB-parameter value you should put the TpFIBDataSet into edit mode by calling TpFIBDataSet.edit (dsEdit).

Use the TFIBCustomDataset method for work with BLOB fields

```
function CreateBlobStream(Field: TField; Mode: TBlobStreamMode): TStream;
override;
```

If you call CreateBlobStream it creates an instance of TFIBDSBlobStream. It enables data exchange between the BLOB-parameter and the stream, which reads the image from the file. The Field parameter defines a BLOB field, on which a stream will be based. The Mode parameter defines the mode.

```
type TBlobStreamMode = (bmRead, bmWrite, bmReadWrite);
```

bmRead	the stream is used to read a BLOB field;
bmWrite	the stream is used to write a BLOB field;
bmReadWrite	the stream is used to modify a BLOB field.

In this example we are using two procedures, one saves a file into BLOB, and the other loads the BLOB field into the file.

```
procedure FileToBlob(BlobField: TField; FileName: string);
var S: TStream; FileS: TFileStream;
begin
  BlobField.DataSet.Edit;
  S := BlobField.DataSet.CreateBlobStream(BlobField, bmReadWrite);
  try
    FileS := TFileStream.Create(FileName, fmOpenRead);
    S.CopyFrom(FileS, FileS.Size);
  finally
    FileS.Free;
    S.Free;
    BlobField.DataSet.Post;
  end;
end;

procedure BlobToFile(BlobField: TField; FileName: string);
var S: TStream;
    FileS: TFileStream;
begin
  if BlobField.IsNull then Exit;
  S := BlobField.DataSet.CreateBlobStream(BlobField, bmRead);
  try
    if FileExists(FileName)
    then FileS := TFileStream.Create(FileName, fmOpenWrite)
    else FileS := TFileStream.Create(FileName, fmCreate);
    FileS.CopyFrom(S, S.Size);
  finally
    S.Free;
    FileS.Free;
  end;
end;
```

If you use TpFIBQuery with BLOB-fields you still use streams, but in contrast to TpFIBDataSet, you should not create any special streams. TFIBSQLDA has built-in SaveToStream and LoadFromStream methods. I.e. for TpFIBQuery you only need to write:

pFIBQuery1.FN('BLOB_FIELD').SaveToStream(FileS);

Using unique FIBPlus field types

FIBPlus has several unique fields: `TFIBLargeIntField, TFIBWideStringField, TFIBBooleanField, TFIBGuidField.`

`TFIBLargeIntField`

It is a BIGINT field in InterBase/ Firebird.

`TFIBWideStringField`

It is used for string fields in UNICODE_FSS charset. In most cases you cannot usea standard TWideString because of multiple VCL errors.

`TFIBBooleanField`

It emulates a logical field. InterBase and Firebird do not have a standard logical field, so FIBPlus enables you to easily emulate it. For this create a domain (INTEGER or SMALLINT) with a BOOLEAN substring in its name. Set psUseBooleanFlields to True in PrepareOptions in TpFIBDataSet. On creating field objects FIBPlus will check the domain name and if they haveBOOLEAN types, FIBPlus will create TFIBBooleanField instances for these fields.

```
CREATE DOMAIN FIB$BOOLEAN AS SMALLINT
DEFAULT 1 NOT NULL CHECK (VALUE IN (0,1));
```

`TFIBGuidField`

It works similar to TFIBBooleanField: a field should be declared in the domain and the domain name should have GUID. psUseGuidField should be set to True. This example illustrates how to declare a domain:

```
CREATE DOMAIN FIB$GUID AS CHAR(16) CHARACTER SET OCTETS;
```

If AutoGenerateValue is set to True, the field values will be set automatically on inserting field values.

How to work with array fields

Since its early versions InterBase enables to use multidimensional array fields and thus to store specialized data in a convenient way. InterBase array fields are not supported by the SQL standard so it’s very difficult to work with such fields using SQL queries. In practice you can use array fields item by item and only in read-only operations. To change array field values you should use special InterBase API commands. FIBPlus helps you to avoid such difficulties and handles array fields itself.

You can see how to work with array fields in the demo example in demo database EMPLOYEE supplied with the server. The LANGUAGE_REQ field in the JOB table is an array (LANGUAGE_REQ VARCHAR(15) [1:5])

The first example shows how to edit an array field using such special methods as TpFIBDataSet ArrayFieldValue and SetArrayValue, as well as GetArrayValues, SetArrayValue and AsQuad in TFIBXSQLDA. These methods enable you to work with this field as a united structure. TFIBXSQLDA.GetArrayElement helps you to get an array element value by index.

TpFIBDataSet.ArrayFieldValue and TFIBXSQLDA.GetArrayValues methods get a variant array from array field values. E.g. use the following code to get separate elements:

```
var v: Variant;
with ArrayDataSet do begin
  v := ArrayFieldValue(FieldByName('LANGUAGE_REQ'));
  Edit1.Text := VarToStr(v[1]);
end;
```

TpFIBDataSet.SetArrayValue and TFIBXSQLDA.SetArrayValue methods enable you to define all field elements as a variant array:

```
with ArrayDataSet do
  SetArrayValue(FBN('LANGUAGE_REQ'), VarArrayOf([Edit1.Text, ...]));
```

Newest FIBPlus versions have a simple solution: you can set the variant array directly:

```
  FBN('LANGUAGE_REQ').Value := VarArrayOf([Edit1.Text, ...]);
```

It's easy to set the field values in the BeforePost event. If Update or Insert operations are not successful, you need to restore the internal array identifier of the editable record. For this you need to refresh the current record by calling the Refresh method. This rule is set by InterBase API functions, so FIBPlus should take it into consideration. So the event handler is very important for working with array fields. You can place it into OnPostError.

```
procedure ArrayDataSetPostError(DataSet: TDataSet; E: EDatabaseError; var
Action: TDataAction);
begin
  Action := daAbort;
  MessageDlg('Error!', mtError, [mbOk], 0);
  ArrayDataSet.Refresh;
end;
```

The TFIBXSQLDA.AsQuad function for array fields and BLOB's has the field BLOB_ID.

Examples ArrayFields1 and ArrayFields2 demonstrate how to work with array fields. ArrayFields1 shows how to extract an array from a field, ArrayFields2 selects an array directly from SelectSQL. Both examples write a field into a database packing it into an array field

Firebird 1.5.X releases have an error, returning the wrong length for string fields. That's whu you can see a strange last symbol «|» in string array elements.

Using TDataSetsContainer containers

The TDataSetContainer component helps developers to use centralized event handling for different TpFIBDataSet components and to send messages to these components forcing them to execute some additional actions. I.e. before opening all TpFIBDataSets you can set parameters for displaying fields, save and restore sorting, etc. You can get the same results by assigning one handler to several TpFIBDataSets. But it is more convenient to use TDataSetContainer, because you can place it on a separate TDataModule as well as its handlers and thus keep the application code outside visual forms.

Besides you can use TDataSetContainer to set one function for local sorting of all connected TpFIBDataSets.

```
TKindDataSetEvent = (deBeforeOpen, deAfterOpen, deBeforeClose,
  deAfterClose, deBeforeInsert, deAfterInsert, deBeforeEdit, deAfterEdit,
  deBeforePost, deAfterPost, deBeforeCancel, deAfterCancel, deBeforeDelete,
  deAfterDelete, deBeforeScroll, deAfterScroll, deOnNewRecord, deOnCalcFields,
  deBeforeRefresh, deAfterRefresh)
```

In FIBPlus 6.4.2 upwards, the container can be global for all TpFIBDataSet instances. For this you need to set the IsGlobal property to True.

You can also build container chains by using the MasterContainer property. When you "inherit" MasterContainer behaviour you expand container behaviour.

Additional actions for TpFIBUpdateObject data modifying

Besides using standard TpFIBDataSet modifying queries you can set various additional actions by using TpFIBUpdateObject objects. TpFIBUpdateObject is a TpFIBQuery descendant, a "client trigger" which sets additional actions for TpFIBDataSet before or after Insert/Modify/Delete operations. Read more details about TpFIBQuery descendant properties and methods in the Supplement.

I.e. you have a master –detail link master(id, name) and detail(id, name, master_id) and you want to automatically delete all dependant data from detail table on deleting records from master. You should add the pFIBUpdateObject object. Then set the property SQL 'delete from deatail where master_id = :id'; set the DataSet property to the dataset for the master table; set KindUpdate to kuDelete and make it execute before the main DataSet operator ExecuteOrder oeBeforeDefault.

Now an operator from the linked pFIBUpdateObject will be executed before record deleting from MasterDataSet.

Working with shared transactions

As we have already mentioned in "Work with transactions", you should make Update transactions as short as possible. TpFIBDataSet is unique as it can work in the context of two transactions: Transaction and UpdateTransaction. We recommend you to use this method as it is the most correct for work with InterBase/Firebird. Data are read-only in a long-running transaction, whereas all modifying queries are executed in a short-running transaction.

The reading transaction (Transaction), as a rule, is ReadCommited and read-only (in order not to retain record versions). We recommend the following parameters for Transaction: read, nowait, rec_version, read_committed. The updating transaction (UpdateTransaction) is short and concurrent, its recommended parameters are: write, nowait, concurency. If you set AutoCommit = True, each change of TpFIBDataSet will be written to a database and become available to other users.

Note: use the shared transaction mechanism and AutoCommit very carefully, especially in master-detail links. In order not to get errors you should know well when the transactions start and close.

Batch processing

TpFIBDataSet has several methods for batch processing: BatchRecordToQuery and BatchAllRecordToQuery, which execute an SQL query. This query is set in TpFIBQuery, which is transferred as a parameter.

See DatasetBatching example to get to know how to use these methods.

Centralized error handling – TpFIBErrorHandler

FIBPlus provides developers with a mechanism of centralized handling of errors and exceptions which occur with FIBPlus components. For this use the TpFIBErrorHandler component with the OnFIBErrorEvent event:

```
TOnFIBErrorEvent = procedure(Sender: TObject; ErrorValue: EFIBError;
  KindIBError: TKindIBError; var DoRaise: boolean) of object;
```

```
где
TKindIBError = (keNoError, keException, keForeignKey, keLostConnect,
    keSecurity, keCheck, keUniqueViolation, keOther);

EFIBError = class(EDatabaseError)
  //..
  property SQLCode : Long read FSQLCode ;
  property IBErrorCode: Long read FIBErrorCode ;
  property SQLMessage : string read FSQLMessage;
  property IBMessage : string read FIBMessage;
end;
```

The option DoRaise manages FIBPlus behaviour after the handler execution, i.e. it shows whether a standard exception will be generated.

This option can be used for standard handling of different error types described in KindIBError.

FIBPlus versions 6.4.2 upwards have new properties in TpFIBErrorHandler. They enable developers to work with localized server messages correctly. You should set such string properties as Index, Constraint, Exception and At.

Getting TFIBSibEventAlerter events

Use the TFIBSibEventAlerter component to get database events. Define the Database property to show which connection events will be monitored. Then define necessary event names in Events and set Active to True to activate the component.

On getting the component event the OnEventAlert event handler will be executed. It is declared as:

```
procedure (Sender: TObject; EventName: String; EventCount: Integer);
```

where EventName is an event name, EventCount is a number of events executed.

Remember that events will be sent only on committing the transaction in which context it occurred. So several events can occur before ObEventAlert.

The Events example demonstrates how to use events in FIBPlus.

Debugging FIBPlus applications

FIBPlus provides developers with a powerful mechanism for SQL control and debugging. You can use the SQL monitor and a feature for SQL statistics gathering while working with the application.

Monitoring SQL queries

TFIBSQLMonitor is responsible for SQL query monitoring. To use it you should define any information type in TraceFlags and write the OnSQL event handler. This feature is demonstrated in the SQLMonitor demo example.

Registering executable queries

The TFIBSQLLogger component logs SQL query execution and keeps SQL query execution statistics.

Properties:

property	ActiveStatistics:boolean	- shows whether the statistics is active
property	ActiveLogging:boolean	- shows whether the logging is active
property	LogFileName:string	- defines the logging file
property	StatisticsParams :TFIBStatisticsParams	- defines statistics parameters

property LogFlags: TLogFlags - defines which operations are logged

property ForceSaveLog:boolean – defines whether to save a log after every query execution

Methods

function ExistStatisticsTable:boolean; - checks whether the statistics table exists

procedure CreateStatisticsTable; - creates a statistics table

procedure SaveStatisticsToDB(ForMaxExecTime:integer=0); - saves the statistics into a table. This parameter shows statistics of certain queries which we are interested in (you should set the minimal query execution time). Statistics will be saved for queries executed longer than ForMaxExecTime or equal to ForMaxExecTime.

procedure SaveLog; - saves a log into a file (valid if ForceSaveLog is set to True).

The SQLLogger example shows how to work with the TFIBSQLLogger component.

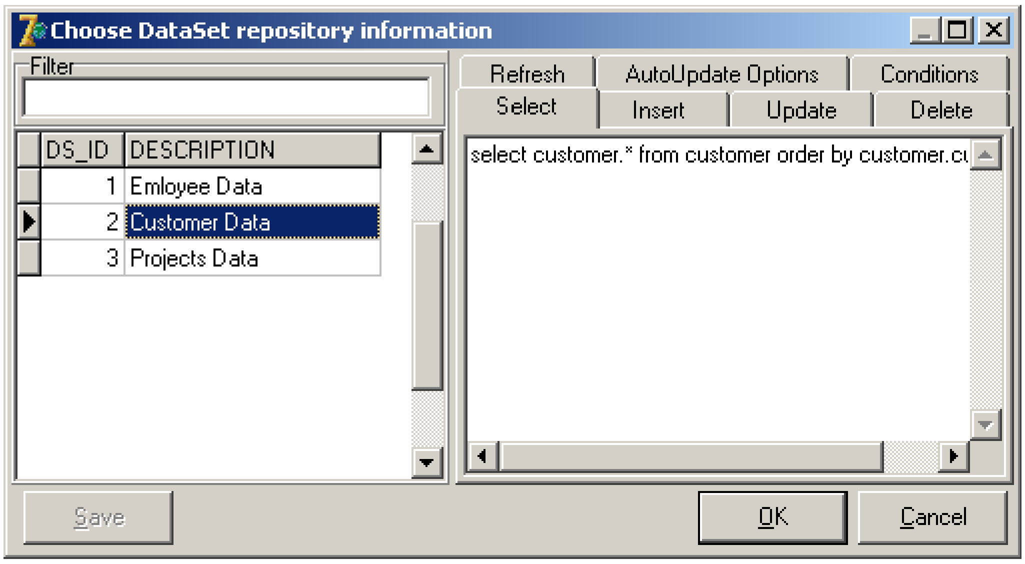
FIBPlus repositories

FIBPlus has three repositories enabling developers to save and use TpFIBDataSet settings, TFields settings and error messages. To use FIBPlus repositories you should set the UseRepositories property (urFieldsInfo, urDataSetInfo, urErrorMessagesInfo) in TpFIBDatabase (defining which repositories you need).

You can learn how to use all repositories from the DataSetRepository, ErrorMessagesRepository and FieldsRepository examples.

Dataset repository

To use the dataset repository in the TpFIBDataSet context menu use «Save to DataSets Repository table» and «Choose from dataSets Repository table». «Save to DataSets Repository table» menu item enables you to save main TpFIBDataSet properties into a special table FIB$DATASETS_INFO, whereas «Choose from dataSets Repository table» helps to load properties from the component already saved in the repository. If the table does not exist in the database, you will be asked to create it. To save the TpFIBDataSet properties into a database you should set a DataSet_ID value (not equal to zero).



DS_ID	DESCRIPTION
1	Emloyee Data
2	Customer Data
3	Projects Data

```
select customer.* from customer order by customer.cu
```

Picture 5. DataSets repository dialog

Then you should set DataSet_ID when running the application and the TpFIBDataSet properties will be loaded from the database table.

As you see in picture 5, FIBPlus saves main SQL queries as well as Conditions and AutoUpdateOptions properties.

Repository mechanism provides you with better application flexibility and enables you to change the program without recompiling it. You should only replicate repository tables.

Field repository

Get access to the field repository by opening the TpFIBDatabase context menu: «Edit field information table».

You can set such properties as DisplayLabel, DisplayWidth, Visible, DisplayFormat and EditFormat for any table field, view and Select procedure. The TRIGGERED value helps to select fields which will be filled in the trigger and which do not require user values even if they are Required (NOT NULL).

You should also set the psApplyRepository parameter in TpFIBDataSet.PrepareOptions in order to get TField settings from the repository on opening the query.

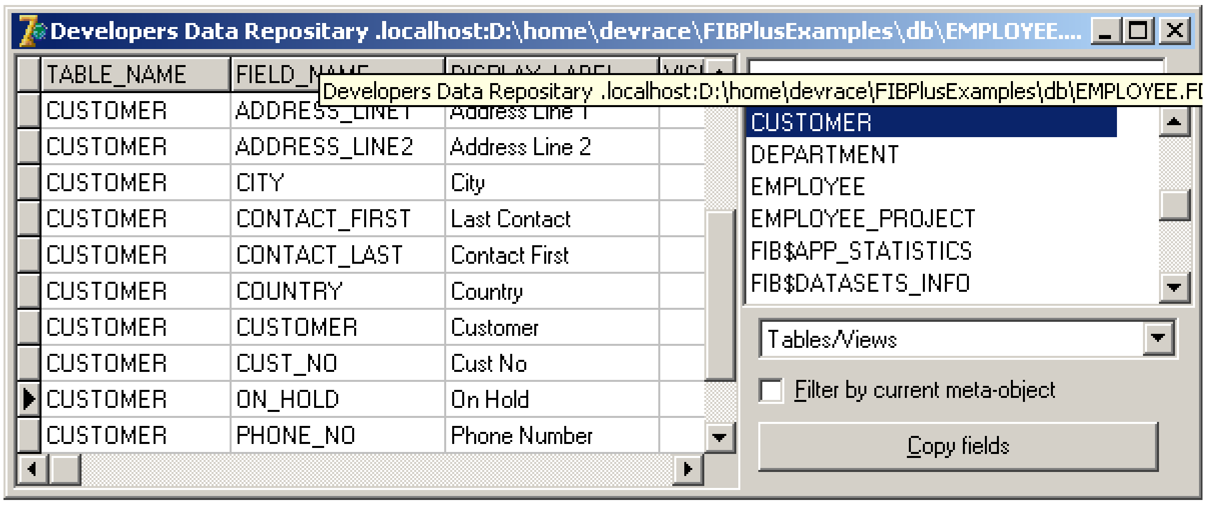
Using field aliases in SQL queries you can see that the settings are not applied for them. This is correct because physical tables do not have aliases. Nevertheless FIBPlus field repository enables you to have settings for such fields with aliases: write ALIAS instead of the table name in the repository.

In version FIBPlus 6.4.2 upwards the dataset has a standard event: TonApplyFieldRepository=**procedure**(DataSet:TDataSet;Field:TField;FieldInfo:TpFIBFieldInfo) **of object**;

It enables developers to use their own settings in the field repository. For example if you want to define the EditMask property, add the EDIT_MASK field to the repository table, define the container in the application and make it global. Then write the following code in the OnApplyFieldRepository event handler:

```
procedure TForm1.DataSetsContainer1ApplyFieldRepository(DataSet: TDataSet;
  Field: TField; FieldInfo: TpFIBFieldInfo);
begin
 Field.EditMask:=FieldInfo.OtherInfo.Values['EDIT_MASK'];
```

end;


Developers Data Repositary .localhost:D:\home\devrace\FIBPlusExamples\db\EMPLOYEE....

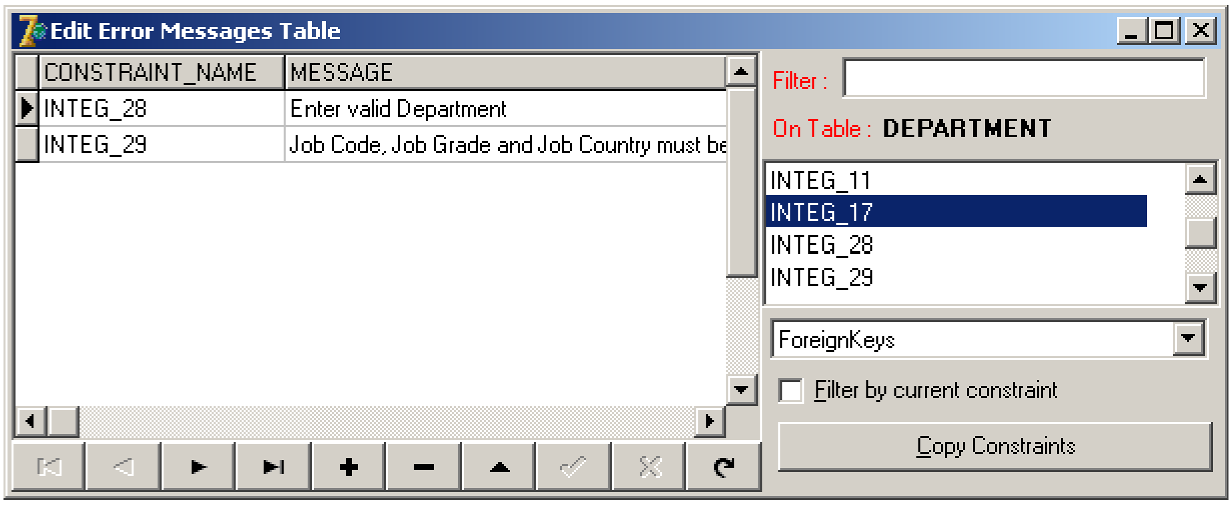
TABLE_NAME	FIELD_NAME	DISPLAY_LABEL
CUSTOMER	ADDRESS_LINE1	Address Line 1
CUSTOMER	ADDRESS_LINE2	Address Line 2
CUSTOMER	CITY	City
CUSTOMER	CONTACT_FIRST	Last Contact
CUSTOMER	CONTACT_LAST	Contact First
CUSTOMER	COUNTRY	Country
CUSTOMER	CUSTOMER	Customer
CUSTOMER	CUST_NO	Cust No
CUSTOMER	ON_HOLD	On Hold
CUSTOMER	PHONE_NO	Phone Number

Picture 6. Field repository

Error message repository

To get access to FIBPlus error message repository use the context menu of the TpFIBDatabase: «Edit error messages table». Using it you can create your own error messages for such cases as primary key (PK) violation, constraints (unique, FK, checks) and unique indices.

To use the error message repository place the TpFIBErrorHandler component on a form or project module and activate the repository. All texts of errors in the repository will be automatically replaced by your own error messages.


Edit Error Messages Table

CONSTRAINT_NAME	MESSAGE
INTEG_28	Enter valid Department
INTEG_29	Job Code, Job Grade and Job Country must be

Picture 7. Error repository

Supporting Firebird 2.0

FIBPlus is compatible with Firebird 2.0, it supports all Firebird 2.0 features.

Now you ca use Execute block statements in SelectSQL.

Now the *poAskRecordCount* option correctly works in all cases because it uses the statement “select count from” (where “select” is your original selection).

The TpFIBDatabase component has two new methods supporting new Firebird 2 commands: RDB$GET_CONTEXT and RDB$SET_CONTEXT:

```
function GetContextVariable (ContextSpace: TFBContextSpace; const VarName:
string): Variant;
```

```
procedure SetContextVariable (ContextSpace: TFBContextSpace; const VarName,
VarValue: string);
```

FIBPlus also supports FB2.0 insert ... into ... returning. Now you should not bother about getting generator values from the client but leave them in the trigger. You can also use RDB$DB_KEY. New possible variants of work with insert returning and RDB$DB_KEY are shown in the example "FB2InsertReturning".

If you have queries joining a table with itself:

```
Select * from Table1 t, Table1 t1
where ....
```

using Firebird 2.0 FIBPlus can understand correctly whether each field was taken from t or t1. This feature helps to generate FIBPlus modifying queries.

Additional features

Full UNICODE_FSS support

FIBPlus version 6.0 upwards correctly works with CHARSET UNICODE_FSS. For this you should use visual components supporting Unicode, e.g. TntControls.

To support Unicode FIBPlus has five new field types:

TFIBWideStringField = class(TWideStringField) – for work with VARCHAR,CHAR;
TFIBMemoField = class(TMemoField, IWideStringField) – for BLOB fields where IWideStringField is an interface for visual TNT components (http://tnt.ccci.org/delphi_unicode_controls)

The psSupportUnicodeBlobs option in TpFIBDataSet.PrepareOptions enables you to work with UNICODE_FSS BLOB fields. By default it is not active, because you need to execute additional query to know a charset for a certain BLOB field. If you do not work with UNICODE, the query will be unnecessary.

TpFIBDatabase has a new method function IsUnicodeCharSet: Boolean, which returns True if the connection uses UNICODE_FSS.

The FIBXSQLVAR class has a new property AsWideString: WideString which returns WideString.

NO_GUI compilation option

FIBPlus 6.0 upwards has a new compilation option: NO_GUI. Using it the library does not refer to standard modules with visual components. This helps you to write applications for system (not user) tasks. See {$DEFINE NO_GUI} in FIBPlus.inc.

Using SynEdit in editors

If you have installed SynEdit components, you can compile editor packages with SynEdit. Then SQL editor will have SQL operator syntax highlighting and the CodeComplete tool. See the define {$DEFINE USE_SYNEDIT} in pFIBPropEd.inc.

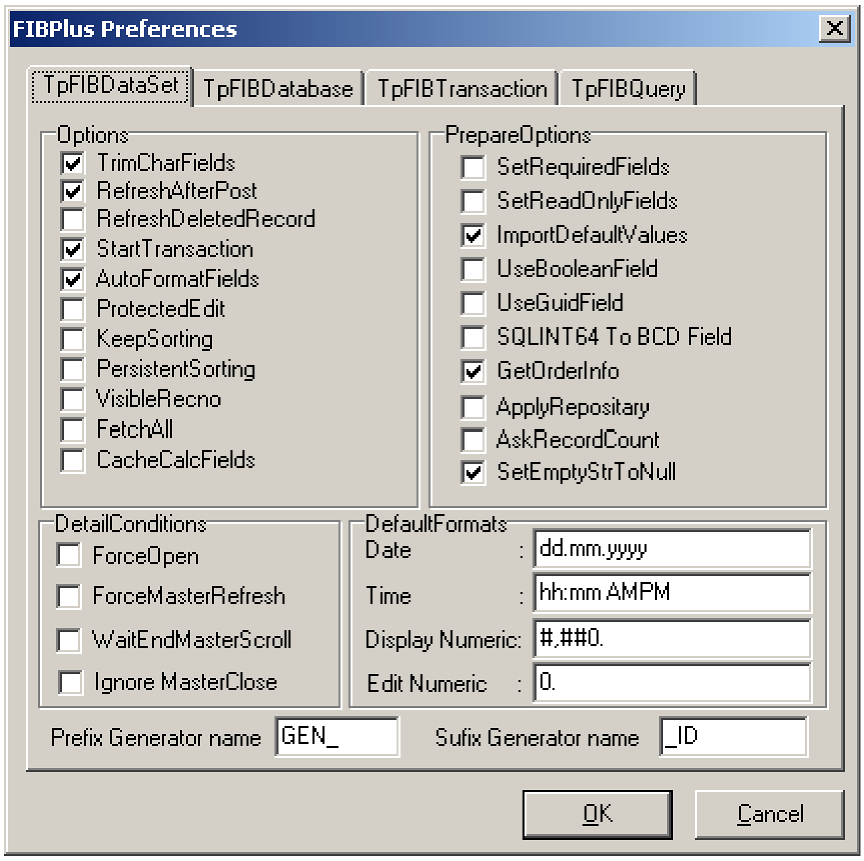
Important: you cannot change compilation defines using the trial FIBPlus version.

Unique tools: FIBPlusTools

Besides components FIBPlus has additional tools FIBPlus Tools, which extend IDE features and enable you to use FIBPlus at design-time more effectively.

Preferences

Preferences help you to set default parameters of the main components. On the first dialog page you can set default values for Options, PrepareOptions and DetailsConditions for all TpFIBDataSet components. You can also set certain keys for these properties. For example if you make SetRequiredFields active, then placing a new TpFIBDataSet component onto the form, you will see that its PrepareOptions property will have pfSetRequiredFields defined. It is important that default settings in FIBPlus Tools Preferences are valid for all new applications that you create. Notice that this concerns only initial defaults: if you change component properties after dropping the component onto the form, Preferences won't have these changes. Besides when changing Preferences, you do not change components which properties were already defined.


FIBPlus Preferences

TpFIBDataSet | TpFIBDatabase | TpFIBTransaction | TpFIBQuery

Options
- [✓] TrimCharFields
- [✓] RefreshAfterPost
- [ ] RefreshDeletedRecord
- [✓] StartTransaction
- [✓] AutoFormatFields
- [ ] ProtectedEdit
- [ ] KeepSorting
- [ ] PersistentSorting
- [ ] VisibleRecno
- [ ] FetchAll
- [ ] CacheCalcFields

PrepareOptions
- [ ] SetRequiredFields
- [ ] SetReadOnlyFields
- [✓] ImportDefaultValues
- [ ] UseBooleanField
- [ ] UseGuidField
- [ ] SQLINT64 To BCD Field
- [✓] GetOrderInfo
- [ ] ApplyRepositary
- [ ] AskRecordCount
- [✓] SetEmptyStrToNull

DetailConditions
- [ ] ForceOpen
- [ ] ForceMasterRefresh
- [ ] WaitEndMasterScroll
- [ ] Ignore MasterClose

DefaultFormats

Date :	dd.mm.yyyy
Time :	hh:mm AMPM
Display Numeric:	#,##0.
Edit Numeric :	0.

Prefix Generator name GEN_ Sufix Generator name _ID

OK Cancel

Picture 8. Tools Preferences.

Prefix Generator name and Suffix Generator name fields are important. Setting them you can form generator names in the AutoUpdateOptions property in TpFIBDataSet. The generator name in AutoUpdateOptions is formed from the table name (UpdateTable), its prefix and suffix.

The following dialog pages help to define main properties for TpFIBDataBase, TpFIBTransaction and TpFIBQuery. In particular if you always work with newest InterBase/Firebird versions (e.g InterBase version 6 upwards), set SQL Dialect=3 in TpFIBDatabase, then you won't need to set it manually all the time.

SQL Navigator

It's the most interesting part of FIBPlus Tools which has no analogues in other products. This tool helps to handle SQL centrally for the whole application.

SQL Navigator enables you to have access to SQL properties of any component from one place.

The button «Scan all forms of active project» scans all application forms and selects those which contain FIBPlus components for work with SQL: TpFIBDataSet, TpFIBQuery, TpFIBUpdateObject and TpFIBStoredProc. Select any of these forms. The list in the right will be filled with components found on this form. Of you click any component you will see their corresponding properties with the SQL code. For TpFIBDataSet there will be shown such properties as SelectSQL, InsertSQL, UpdateSQL, DeleteSQL and RefreshSQL. For TpFIBQuery, TpFIBUpdateObject and TpFIBStoredProc FIBPlus will show an SQL property value.

You can change any property directly from SQLNavigator and its new value will be saved. SQLNavigator helps to work with groups of components. You only need e select corresponding components or forms.

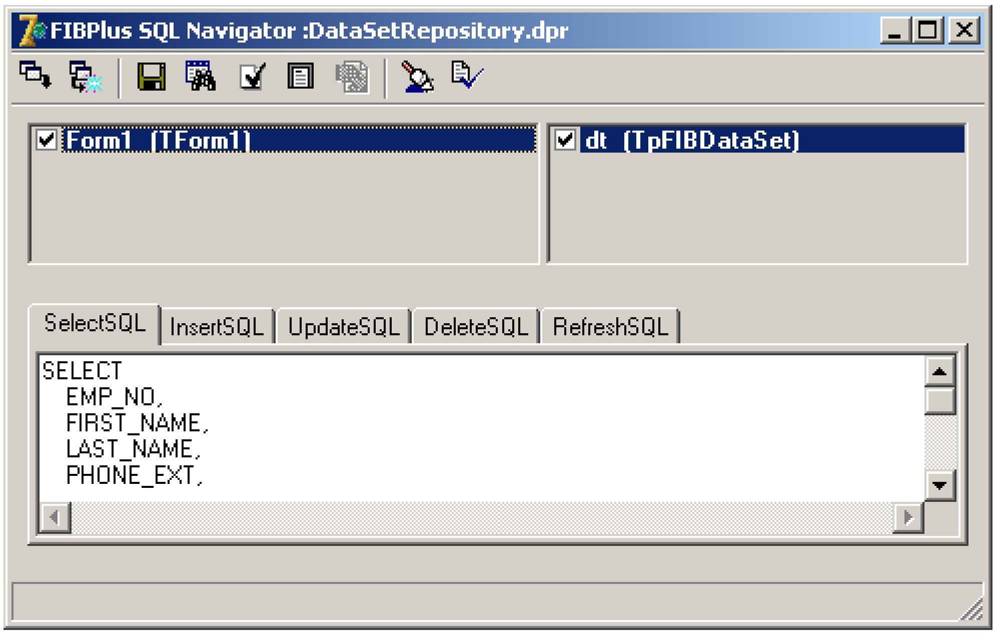
"Save selected SQLs" saves values of selected properties into an external file.

“Check selected SQLs” checks selected SQL queries on correctness in SQLNavigator.

Then you can analyze the file with selected queries by using special tools.

You can also use SQLNavigator for text searching in the SQL of the whole project.

By double clicking any found item you make SQLNavigator select a component and a property and thus can edit SQL.



```
SELECT
  EMP_NO,
  FIRST_NAME,
  LAST_NAME,
  PHONE_EXT,
```

Picture 9. Tools SQL Navigator

How to work with services

Besides the main component palette FIBPlus also has FIBPlus Services. These components are intended for getting information about InterBase/Firebird servers, user control, database parameter setting and database administration.

More details about each issue you can read in InterBase manuals (OpGuide.pdf, parts «Database Security», «Database Configuration and Maintenance», «Database Backup and Restore» and «Database and Server Statistic»).

The Services demonstration example shows how to work with services. Besides you can see a good example in Delphi/C++BuilderL: DelphiX\Demos\Db\IBX\Admin\. Though being written for IBX, this example suits well for FIBPlus.

All services components are TpFIBCustomService descendants and have such properties and methods as: ServerName – a server name used for connection; Protocol – a connection protocol; UserName – a user name; and Password – a user password.

In general work with services is the following: at first you define connection properties (server, protocol, user and password). The you connect to the server (Attach := True), execute necessary operations and disconnect (Attach := False). This sequence of actions is demonstrated in the code below and is necessary for any service.

```
with TpFIBXXXService.Create(nil) do
try
  ServerName := <server_name>;
  Protocol := <net_protocol>;
  UserName := <user_name>;
  Password := <password>;
  Atach;
  try
    //executed operations
  finally
    Deatach;
  end;
finally
  Free;
end;
```

Getting server information

Using the TpFIBServerProperties component you can get information about licenses, server configuration, a number of users connected to the database and a number of connected databases. For more details about properties, events and methods please see the DevGuide Supplement. In general you should connect to the server, define necessary information and get it using the Fetch method.

Parameters and results are described in the module IB_Services.pas. The following record keeps information about the number of server connections, the number of active databases operated by the server and their names:

```
  TDatabaseInfo = record
    NoOfAttachments: Integer;
    NoOfDatabases: Integer;
    DbName: Variant;
  end;
```

These two records have information about InterBase server licenses:

```
  TLicenseInfo = record
```

```
    Key: Variant;
    Id:     Variant;
    Desc:    Variant;
    LicensedUsers: Integer;
  end;

  TLicenseMaskInfo = record
    LicenseMask: Integer;
    CapabilityMask: Integer;
  end;

  TConfigFileData = record
    ConfigFileValue:Variant;
    ConfigFileKey:Variant;
  end;

  TConfigParams = record
    ConfigFileData: TConfigFileData;
    BaseLocation: string;
    LockFileLocation: string;
    MessageFileLocation: string;
    SecurityDatabaseLocation: string;
  end;
```

This record shows information about the server version:

```
  TVersionInfo = record
    ServerVersion: String;
    ServerImplementation: string;
    ServiceVersion: Integer;
  end;

  TPropertyOption = (Database, License, LicenseMask, ConfigParameters, Version);
  TPropertyOptions = set of TPropertyOption;

  TpFIBServerProperties = class(TpFIBCustomService)
    procedure Fetch;
    procedure FetchDatabaseInfo;
    procedure FetchLicenseInfo;
    procedure FetchLicenseMaskInfo;
    procedure FetchConfigParams;
    procedure FetchVersionInfo;
    property DatabaseInfo: TDatabaseInfo
    property LicenseInfo: TLicenseInfo
    property LicenseMaskInfo: TLicenseMaskInfo
    property ConfigParams: TConfigParams
    property Options : TPropertyOptions
  end;
```

The TpFIBLogService component is used for server logging.

Most services return text information using the OnTextNotify event of the TserviceGetTextNotify type. This type is described in IB_Services as:

```
  TServiceGetTextNotify = procedure (Sender: TObject; const Text: string) of
object;
```

TpFIBLogService uses this type of notification.

When working with these services you need to set the reaction on the OnTextNotify event (which starts the service and reads the information). The information is read as:

```
    ServiceStart;
    while not Eof do
      GetNextLine;
```

So the complete code will be the following:

```
with TpFIBXXXService.Create(nil) do
try
  ServerName := <server_name>;
  Protocol := <net_protocol>;
  UserName := <user_name>;
  Password := <password>;
  Atach;
  try
    ServiceStart;
    while not Eof do
      GetNextLine;
  finally
    Deatach;
  end;
finally
  Free;
end;
```

On working with some other similar services you should set different options.

For more details check the Services example.

Managing server users

TpFIBSecurityService is responsible for work with users. It has methods which enable you to get user info, as well as to add, modify and delete users. If you use this component you can get the following information:

A record with user information at the server:

```
  TUserInfo = class
  public
    UserName: string;
    FirstName: string;
    MiddleName: string;
    LastName: string;
    GroupID: Integer;
    UserID: Integer;
  end;
```

Available properties and methods of the component:

```
  TpFIBSecurityService = class(TpFIBControlAndQueryService)
    procedure DisplayUsers;
    procedure DisplayUser(UserName: string);
    procedure AddUser;
    procedure DeleteUser;
    procedure ModifyUser;
    procedure ClearParams;
    property  UserInfo[Index: Integer]: TUserInfo read GetUserInfo;
    property  UserInfoCount: Integer read GetUserInfoCount;

    property SQlRole : string read FSQLRole write FSQLrole;
    property UserName : string read FUserName write FUserName;
    property FirstName : string read FFirstName write SetFirstName;
    property MiddleName : string read FMiddleName write SetMiddleName;
    property LastName : string read FLastName write SetLastName;
    property UserID : Integer read FUserID write SetUserID;
    property GroupID : Integer read FGroupID write SetGroupID;
    property Password : string read FPassword write setPassword;
  end;
```

Use the DisplayUser method to get information about the server user (using the user name as a parameter). DisplayUsers is used to get information about all users. After calling DisplayUser or DisplayUsers you will see the number of users in the UserInfoCount property; the UserInfo index property will return a user record by its index.

To add a new user you should set the following properties: UserName, FirstName, MiddleName, LasName, Password; and execute the AddUser method. DeleteUser and ModifyUser methods work in the same way as AddUser; they require UserName as a minimal obligatory parameter.

Note:

- the Password property is not returned by DisplayUser and DisplayUsers methods.
- The SQLRole property cannot be used to associate a role with a user.

For these operations you should execute queries using TpFIBQuery (GRANT/REVOKE).

The Services example demonstrates all aspects of work with server users.

Configuring databases

TpFIBConfigService helps to set such parameters as:

- set an interval for the automatic garbage collection in the transaction (Sweep Interval);
- set a mode of writing changes on the disk (Async Mode);
- set the DB page size;
- set the reserved space for the DB;
- set the read-only mode;
- activate and deactivate the shadow

It also helps to:

- shutdown the database (shutdown)
- start the database (online).

Before working with this service you should set the property

```
property DatabaseName: string
```

A path to the DB at the server

To set the interval for the automatic garbage collection in the transaction you should use the `SetSweepInterval` method. The only `SetSweepInterval` parameter is the interval, and by default it is equal to 20000.

```
procedure SetSweepInterval (Value: Integer);
```

To set the database dialect you should use the SetDBSqlDialect method; its parameter is the DB dialect. Only three parameters are supported: 1, 2, 3.

```
procedure SetDBSqlDialect (Value: Integer);
```

To set PageBuffers use the SetPageBuffers method; its parameter is the required buffer size.

```
procedure SetPageBuffers (Value: Integer);
```

To activate the shadow use the ActivateShadow method. It doesn't require any parameters.

```
procedure ActivateShadow;
```

To set the asynchronous writing mode you should set the SetAsyncMode method to True; to deactivate this mode – to False.

```
procedure SetAsyncMode (Value: Boolean);
```

To set the read-only mode use the SetReadOnly method. If it is set to True, the read-only mode is active; if it is set to False, the read-only mode is deactivated. This mode is used when you prepare a database for distribution on CD or other mediums

```
procedure SetReadOnly (Value: Boolean);
```

Use SetReserveSpace to set the reserved space for the DB:

```
procedure SetReserveSpace (Value: Boolean);
```

Use the ShutdownDatabase method to shutdown the database. There are three types of shutting the database down: forced, denying new transactions and denying new connections. All the above mentioned operations should be better done under the SYSDBA administrator's connection.

```
procedure ShutdownDatabase (Options: TShutdownMode; Wait: Integer);

TShutdownMode = (Forced, DenyTransaction, DenyAttachment);
```

Use the BringDatabaseOnline method to bring the database online.

```
procedure BringDatabaseOnline;
```

Backing up and restoring databases

TpFIBBackupService and TpFIBRestoreService components help to use database backup and restore functions.

TpFIBValidationService enables you to gather garbage, check the database on errors and repair it if necessary.

Options play a very important role when you backup, restore and check the database. All these options are described in the server manual OpGuide.pdf.

Backup and restore services are simple, and you can work with them in the same way as FIBLogService. The only peculiar thing is the interpretation of their work results:

If you need to be sure there were no errors during the DB backup/restore, you should see the operation log. In case of errors the log will have a string «GBAK: ERROR»

If TpFIBValidationService did not find any errors, its log will contain one empty string.

If the database repairing caused errors (the Mend TpFIBValidationService option), they will be written to the server log.

Remember that TpFIBBackupService creates the backup file at the server; the server does the backup of the DB and keeps the file there. But you can do the backup from another server as well. Write a full path to the database in the connection path:

```
with TpFIBXXXService.Create(nil) do
try
  ServerName := <local_backup_server_name>;
  Protocol := <net_protocol>;
  UserName := <user_name>;
  Password := <password>;
  DatabaseName := <remote_db_name>;
  BackupFile.Add(<local_backup_name>);
  Atach;
  try
    ServiceStart;
    while not Eof do
      GetNextLine;
  finally
    Deatach;
  end;
finally
  Free;
end;
```

These services can work with Embedded server too (remember to set the Local protocol when working with Firebird Embedded Server).

Getting statistics about database

Use the TpFIBStatisticalService component to get important statistics about working database. It is similar to TpFIBLogService.