MWA Software

Writing User Defined Routines (UDRs)

Issue 1.0,
14 February 2022

McCallum Whyman Associates Ltd

EMail: info@ mccallumwhyman.com, http://www.mccallumwhyman.com

Registered in England Registration No. 2624328

COPYRIGHT

Disclaimer

Although our best efforts have been made to ensure that the information contained within is up-to-date and accurate, no warranty whatsover is offered as to its correctness and readers are responsible for ensuring through testing or any other appropriate procedures that the information provided is correct and appropriate for the purpose for which it is used.

CONTENTS **Page**

1

Introduction

As a replacement for the legacy User Defined Functions (UDFs), Firebird 3 introduced the User Defined Routine. As stated in the Firebird documentation, the UDR (User Defined Routines) engine adds a layer on top of the FirebirdExternal engine interface with the purpose of:

- establishing a way to hook external modules into the server and make them available for use
- creating an API so that external modules can register their available routines
- making instances of routines “per attachment”, rather than dependent on the internal implementation details of the engine.

UDRs are implemented in a user provided software library (Windows DLL or Linux shared object) that must be installed on a Firebird Server prior to use. Each UDR has also to be declared as an external function, procedure or trigger in the schema of each database that uses the UDR.

From release 1.4.0 onwards, MWA's Firebird Pascal API package (*fbintf*) comes with a new companion package (*fbudr*). The *fbudr* package provides support for writing your own UDRs in Pascal while making full use of MWA's Firebird Pascal API. It is available for both Free Pascal and Delphi.

UDRs built with this package support:

- UDR Functions, Execute Procedures Select Procedures and Triggers.
- Full use of the *fbintf* package (MWA's Pascal Firebird API).
- Access to input and output parameters by name or position and as Pascal native types
- Exception handling including use of the Firebird status vector to report exceptions to clients. Note that signal based exceptions (e.g. access violations) may result in lost database attachments (see chapter 6).
- Many configurable options (see 4.1).

- A per UDR library log file (by default written to <firebird root directory>/<module name>.log)
- Detailed and configurable logging for library debugging
- User write to log support (see the IFBExternalContext interface).
- A per UDR library configuration file in *ini file* format (by default the UDR library looks for its configuration file in <firebird root directory>/plugins/udr/<module name>.conf). Log options may be configured statically or via the configuration file.
- User sections and configuration parameters supported via the IFBExternalContext interface (see 4.8.1).

A *fbudrtestbed* package is also provided to allow client side debugging of UDRs (see 6.1). This provides an emulator for the Firebird udr engine and allows for program logic to the tested using an IDE (Lazarus or Delphi). For an example see the fbintf/examples/UserManual/testbed directory and 6.2. This example shows how the example UDR library can be tested client side.

1.1 References

1. MWA's Firebird Pascal API (https://mwasoftware.co.uk/downloads/send/5-ibx-current/146-firebirdpascalapiguide)
2. Firebird 4.0 Language Reference (https://firebirdsql.org/file/documentation/pdf/en/refdocs/fblangref40/firebird-40-language-reference.pdf)

2

An Introduction to User Defined Routines

User Defined Routines (UDRs) were introduced in Firebird 3 as an object oriented and are intended to be a better managed alternative to the legacy User Defined Functions (UDFs). UDFs are deprecated from Firebird 4.0 onwards.

The Firebird 4 Language Reference **[2]** states that:

> *UDFs are fundamentally insecure. We recommend avoiding their use whenever possible, and disabling UDFs in your database configuration (UdfAccess = None in firebird.conf; this is the default since Firebird 4). If you do need to call native code from your database, use a UDR external engine instead.*

UDRs are managed using a Firebird “plugin” - the UDR Engine. This is a shared library (DLL under Windows or .so under Linux) that loads and manages user installed shared libraries containing UDRs. The UDR Engine shared library as well as user shared libraries are installed as part of a Firebird server and are located in:

<firebird root directory>/plugins/udr

Note that this location is configurable by modifying Firebird's “plugins.conf” configuration file.

2.1 The UDR Engine

The UDR Engine is a Firebird Plugin and registered in Firebird's “plugins.conf” configuration file. As a plugin, it provides the ***IExternalEngine*** interface[1] to Firebird. Firebird calls this interface's methods to get interfaces to UDR Functions, Procedures and Triggers.

[1]The term interface is used throughout this document and may refer to an interface defined by the Firebird API or a Pascal interface type defined by the *fbintf* or *fbudr* package. In both cases an interface defines a set of functions that may be called via the interface.

The purpose of the UDR Engine is to satisfy these requests by loading and calling user provided shared libraries as determined by an “EntryPoint” name used to identify the UDR.

```
{ CREATE [ OR ALTER ] | RECREATE | ALTER } PROCEDURE <name>

    [ ( <parameter list> ) ]
    [ RETURNS ( <parameter list> ) ]
    EXTERNAL NAME '<external name>' ENGINE <engine>

  { CREATE [ OR ALTER ] | RECREATE | ALTER } FUNCTION <name>
    [ <parameter list> ]
    RETURNS <data type>
    EXTERNAL NAME '<external name>' ENGINE <engine>

  { CREATE [ OR ALTER ] | RECREATE | ALTER } TRIGGER <name>
    ...
    EXTERNAL NAME '<external name>' ENGINE <engine>
```

Figure 1: DLL for User Defined Routines

UDRs are declared as part of a Database schema using a DLL statement as defined in Figure 1 above.

For example

```
create or alter function MyRowCount (
      table_name varchar(31)
    ) returns integer
    external name 'fbudrtests!row_count'
    engine udr;
```

declares a function “MyRowCount”. The declaration is the same as any other function declaration, except that instead of a function body, the function is defined using an external name and engine - the udr engine. In use, the function is called like any other Firebird function. E.g.

```
Select MyRowCount('EMPLOYEE') from RDB$Database;
```

Assuming the above is used with the Firebird example “employee” database, the select query will return 42 (perhaps in a tribute to Lewis Caroll and Douglas Adnams) i.e. the number of rows in the table.

When the select statement is executed, Firebird will call the identified engine - the udr engine - and ask it to return an interface to a UDR with the entry point name “*fbudrtests!row_count*”. It does by:

1. Locating and loading, if possible and if it has not already done so, a shared library called “*fbudrtests*” i.e. determined from the first part of the entry point name (the exclamation mark is the field separator). This library must be located in the Firebird udr directory and must be called “*fbudrtests.dll*” (Windows) or “*libfbudrtests.so*” (Linux).

2. After loading, the shared library's only published entry point **firebird_udr_plugin** is called. The code that handles this entry point must now register each UDR it supports. The registration includes a “routine name”, used to identify the UDR, and a “factory” interface for the UDR. This factory can, on demand, create a new instance of an interface to the UDR.

3. The UDR engine will then look for the routine name “*row_count*” as registered by the library. If found, the corresponding factory is called to create a new instance of the UDR as an interface that is then returned to Firebird. Firebird can now use this interface to satisfy the user request.

2.2 The User Provided UDR Library

There can be as many user provided UDR libraries as is needed, as long as they each have a different name.

A UDR library includes:

- Each UDR function, procedure or trigger that the user library provides.
- An object factory for each such UDR function, procedure or trigger.
- The entry point **firebird_udr_plugin** which is called to register, with the UDR Engine, each UDR function, procedure or trigger factory and the corresponding routine name.

UDR libraries are not restricted to C or C++ and may be written in any programming language capable of generating a shared library and with appropriate language bindings.

When a UDR is executed, it is provided with a context that includes a handle to the current database attachment and transaction. This allows it to invoke database queries on the user's behalf. Otherwise, a UDR is free to access any external code libraries that it needs to use and has access to. This can include mathematical, cryptographic and image processing libraries. It could even access a different database engine (e.g. MySQL).

UDRs are always executed as separate instances and therefore may use their own local variables. However, depending on the server architecture (Superserver, Classic, etc) and implementation details, Firebird may get external engine instances “per database” or “per connection”. Currently, it always gets instances “per database”. Hence any global variables will be shared between UDRs invoked by all users of the same database.

3

Writing UDRs in Pascal

MWA Software's Firebird UDR Support Package (*fbudr*) provides a set of Pascal language bindings and support functions for writing a UDR library in Pascal. Both the Free Pascal Compiler (FPC) and Delphi (from Embarcadero) are supported.

The *fbudr* package is distributed as source code and as part of MWA Software's Firebird Pascal API. The package source may be found in fbudr.lpk (Free Pascal/Lazarus) or fbudr.dpk (Delphi). The package source must be complied and linked into the share library. The *fbudr* package requires the *fbintf* package (e.g. the Firebird Pascal API) and its use includes *fbintf* automatically.

3.1 Creating a UDR Shared Library with fbudr

3.1.1 With Lazarus

Prior to first use, you should first open each of the package files “fbintf.lpk” and “fbudr.lpk” using the menu item Packages->Open Package. This is sufficient for Lazarus to recognise each package.

In the Lazarus IDE, select File->New and click on “Library” as the project type. This will create and show a “library” source code file. In the project inspector, select Add->New Requirement and select the *fbudr* package.

You should now save the library project using the intended module name (shared library name). This will be the first part of the external name used to identify the UDRs provided by the library.

3.1.2 With Delphi

Prior to first use, you should open, in the Delphi IDE, each of the package files “fbintf.dproj” and “fbudr.dproj”, and compile the packages each in turn.

In the Delphi IDE, select File->New->Other and double click on “Dynamic Link Library” when the dialog opens showing each of the options. You should now save the library project using the intended module

name (shared library name). This will be the first part of the external name used to identify the UDRs provided by the library.

The project should now be linked to the *fbudr* package. This is done by

1. Select Project->Options
2. In the Options dialog, select Packages->Run Time Packages.
3. Click on the ellipses in the right hand window at the end of the current list of runtime packages.
4. In the Run Time Packages dialog, select the *fbudr* package by clicking on the yellow folder icon and browsing for the fbudr.dcp file. This is, by default, located in the fbintf\Win32\Debug, or the fbintf\Win64\Debug folders. Select the fbudr.dcp file, click on the “open” button”, click on the “Add” button and finally the “OK” button to close the dialog.
5. In most cases you will also want to select the “link with runtime packages” option.

3.1.3 The Library Source File

```
library myudrlibrary;

uses
  Classes, sysutils,  FBUDRController, <list of user created units>;

exports firebird_udr_plugin;

begin
  with FBUDRControllerOptions do
  begin
    ModuleName := 'myudrlibrary';
    AllowConfigFileOverrides := true;
    LogFileNameTemplate := '$LOGDIR$MODULE.log';
    LogOptions := [loLogFunctions, loLogProcedures, loLogTriggers, loDetails];
  end;
end.
```

Figure 2: Example Library Source

The resulting library source file (.lpr or .dpr) should be the same for both environments and follows a standard template for UDR libraries written using *fbudr* (see Figure 2). The “library” keyword denotes that this is the source of a shared library instead of a program. The “uses” clause should also contain the name of each of your units that defines UDRs.

The “begin..end” block contains statements that are executed when the library is loaded. Here, you should locate overrides for default UDR Controller options. In the above example:

- An “exports” clause declares the **firebird_udr_plugin** entry point. This is defined in the FBUDRController unit and does not have to be defined by the user.
- The “ModuleName” is always set to the library name.
- AllowConfigFileOverrides set to true permits the library's configuration file to contain overrides for controller options.
- The LogFileTemplate defines the location and name of the library's log file. Macros are used here to define the path and file name.

3.2 Defining a UDR

You may now add a unit to the project to provide the source code for a UDR. For example, if the UDR is defined to provide a function declared in the database schema as:

```
create or alter function MyRowCount (
      table_name varchar(31)
    ) returns integer
    external name 'myudrlibrary!row_count'
    engine udr;
```

and with the semantic that it returns the number of rows in the table given as the value of the parameter *table_name*. Figure 3 provides an example implementation of the function.

```
Unit udr_myrowcount;

interface

uses
  Classes, SysUtils, IB, FBUDRController, FBUDRIntf;

type
  TMyRowCountFunction = class(TFBUDRFunction)
  public
    function Execute(context: IFBUDRExternalContext;
                     ProcMetadata: IFBUDRProcMetadata;
                     InputParams: IFBUDRInputParams;
                     ResultSQLType: cardinal): variant; override;
  end;

implementation

function TMyRowCountFunction.Execute(context: IFBUDRExternalContext;
  ProcMetadata: IFBUDRProcMetadata; InputParams: IFBUDRInputParams;
  ResultSQLType: cardinal): variant;
begin
  with context do
  begin
    Result := GetAttachment.OpenCursorAtStart(GetTransaction,
    'Select count(*) from ' + InputParams.ByName('table_name').AsString)[0].AsInteger;
  end;
end;

Initialization
  FBRegisterUDRFunction('row_count',TmyRowCountFunction);
end.
```

Figure 3: Implementation of an example UDR

In Figure 3:

- TMyRowCountFunction sub-classes a TFBUDRFunction class defined in FBUDRController.
- This new sub-class overrides the Execute method which then provides the main body of the UDR.
- The Execute method uses the context provided by Firebird to reference the current database attachment and transaction. The attachment is provided as an *fbintf* ***IAttachment***. This interface provides many methods, including executing SQL statements and returning results. It is used here to invoke a query that counts the number of rows in the requested table.
- Note that the input parameter is provided using a cut down version of the *fbintf* ***IResults*** interface and provides the parameter as a native pascal type with the parameter accessed by name.

- The result is returned as the value of a variant. This must be type compatible with the return type of the function as declared in the database schema.

- The new sub-class is registered with the UDR Controller at unit initialization time and bound to the routine name used in the database schema function declaration.

All that remains to do is to ensure that the unit name is in the list of units in the library source file “uses clause”, and then to compile and link the UDR library and to copy the resulting *udr_myrowcount.dll* or *udr_myrowcount.so* to the Firebird UDR directory. The Firebird Server should be restarted to ensure that it recognises the new library, and the UDR function declaration added to the database schema.

3.3 A Select Procedure

A UDR function is one of the simplest UDRs. An Execute Procedure is almost as simple. The only real difference is that it returns several output values instead of a single result. However, a select procedure is more complex given that it can return multiple rows. In PSQL, a select procedure is distinguished by including at least one “SUSPEND” statement. A SUSPEND statement suspends execution and returns the current output parameter values as the next row.

An example UDR select procedure may be defined as:

```
create or alter procedure MyReadText (
     path varchar(200) not null /*relative to udr directory */
     ) returns (
    text varchar(100) not null
)
external name 'myudrlibrary!read_txt'
engine udr;
```

The semantic of this UDR is that it opens a text file, for which the absolute path is given by the value of the “path” parameter, and then returns each line in the file as successive rows in the output dataset.

In Figure 4:

- TReadTextFile sub-classes TFBUDRSelectProcedure and overrides three methods: open, fetch and close.

- It also uses a local variable FTextFile, defined as a private property of the class. This ensures that it is only available to a single instance of each object of this class.

- FTextFile is used to hold a reference to a stream reader for the text file.

- The *open* method is called first by the Firebird engine and initialises the select procedure. The file path is provided as the input parameter and the method checks to ensure that the file exists. An exception is raised if it does not. Otherwise, the file is opened as a stream reader object.

- If an exception is raised then this is caught by the UDR Controller and returned to Firebird using Firebird's “status” interface. It can then be returned to the client using the Firebird remote protocol.

- The *fetch* method is called repeatedly until it returns false to indicate “no more data”. Here, the fetch method reads the next line from the input file and returns it as the value of the output parameter.

- The *close* method is called after fetch returns false and performs any tidy up needed. In this case, the stream reader is closed.

```
Unit udr_myreadtext;

interface

uses
  Classes, SysUtils, IB, FBUDRController, FBUDRIntf{$IFDEF FPC}, Streamex{$ENDIF};

type
  TReadTextFile  = class(TFBUDRSelectProcedure)
    private
      FTextFile: TStreamReader;
    public
      procedure open(context: IFBUDRExternalContext;
                     ProcMetadata: IFBUDRProcMetadata;
                     InputParams: IFBUDRInputParams);  override;
      function fetch(OutputData: IFBUDROutputData): boolean;  override;
      procedure close; override;
    end;

implementation

procedure TReadTextFile.open(context: IFBUDRExternalContext;
                 ProcMetadata: IFBUDRProcMetadata;
                 InputParams: IFBUDRInputParams);
var aFileName: AnsiString;
    {$IFDEF FPC}F: TFileStream;{$ENDIF}

begin
  aFileName := InputParams.ByName('path').AsString;
  if not FileExists(aFileName) then
    raise Exception.CreateFmt('Unable to find file "%s"',[aFileName]);
  context.WriteToLog('Reading from ' + aFileName);
  {$IFDEF FPC}
  F := TFileStream.Create(aFileName,fmOpenRead);
  FTextFile := TStreamReader.Create(F,8192,true);
  {$ELSE}
  FTextFile := TStreamReader.Create(aFileName, TEncoding.ANSI);
  {$ENDIF}
end;

function TReadTextFile.fetch(OutputData: IFBUDROutputData): boolean;
begin
  Result := not FTextFile.{$IFDEF FPC}EOF {$ELSE}EndOfStream {$ENDIF};
  if Result then
    OutputData.ByName('text').AsString := FTextFile.ReadLine;
end;

procedure TReadTextFile.close;
begin
  if FTextFile <> nil then
    FTextFile.Free;
  FTextFile := nil;
end;

Initialization
  FBRegisterUDRProcedure('read_txt',TReadTextFile);
end.
```

Figure 4: A example select procedure

Note that, as before, the sub-class has to be registered with the UDR Controller at initialisation time, and associated with its routine name.

All that remains to do is to ensure that the unit name is in the list of units in the library source file uses clause, and then to compile and link the UDR library and to copy the resulting *dll* or *so* to the Firebird UDR directory. The Firebird Server should be restarted to ensure that it recognises the new library, and the UDR procedure declaration added to the database schema.

This can be included in the same library as the preceding example UDR function.

3.4 An Example Trigger

The final example is a UDR trigger. UDR Triggers can be defined as “before”, “after” or database triggers. In the former two cases, they may follow an Update, Insert or Delete action.

The example trigger is defined assuming the Firebird example employee database and has the following declaration:

```
Create or Alter Trigger MyEmployeeUpdate Active Before Update On EMPLOYEE
   external name 'myudrlibrary!my_employee_update'
   engine udr;
```

The semantic of this trigger is to identify when the PHONE_EXT field value is updated and to copy the previous value to the field PREVOUS_PHONE_EXT. This requires the following alteration to the EMPLOYEE table:

```
Alter Table EMPLOYEE Add PREVIOUS_PHONE_EXT VarChar(4);
```

In Figure 5:

- TMyEmployeeUpdateTrigger is sub-classed from TUDRTrigger.
- TUDRTrigger provides three methods that can be overridden: BeforeTrigger, AfterTrigger and DatabaseTrigger. In this example, the trigger is defined as “before update” and hence it is the BeforeTrigger method that is overridden.
- The trigger body starts with a “sanity check” to ensure that it is called as a Before Update Trigger. An exception is raised if the check fails.
- The method is provided with two sets of input parameter: one for the “old” values and a second for the “new” values. The former is read/only, while for a “before” trigger, the second is read/write.
- The trigger body compares the old and new values for PHONE_EXT and, if they differ copies the old value to the new value for PREVIOUS_PHONE_EXT.

Note that, as before, the sub-class has to be registered with the UDR Controller at initialisation time, and associated with its routine name.

All that remains to do is to ensure that the unit name is in the list of units in the library source file “uses clause”, and then to compile and link the UDR library and to copy the resulting dll or so to the Firebird UDR directory. The Firebird Server should be restarted to ensure that it recognises the new library, and the UDR trigger declaration added to the database schema.

This can be included in the same library as the preceding example UDRs.

```
Unit udr_mytrigger;

interface

uses
  Classes, SysUtils, IB, FBUDRController, FBUDRIntf;

type
  TMyEmployeeUpdateTrigger = class(TFBUDRTrigger)
  public
     procedure BeforeTrigger(context: IFBUDRExternalContext;
                             TriggerMetaData: IFBUDRTriggerMetaData;
                             action: TFBUDRTriggerAction;
                             OldParams: IFBUDRInputParams;
                             NewParams: IFBUDROutputData); override;
  end;

implementation

procedure TMyEmployeeUpdateTrigger.BeforeTrigger(
  context: IFBUDRExternalContext; TriggerMetaData: IFBUDRTriggerMetaData;
  action: TFBUDRTriggerAction; OldParams: IFBUDRInputParams;
  NewParams: IFBUDROutputData);
begin
  if (TriggerMetaData.getTriggerType <> ttBefore) or (action <> taUpdate) then
    raise Exception.CreateFmt('%s should be an update before trigger',[Name]);

  if OldParams.ByName('PHONE_EXT').AsString <> NewParams.ByName('PHONE_EXT').AsString then
    NewParams.ByName('PREVIOUS_PHONE_EXT').AsString := OldParams.ByName('PHONE_EXT').AsString;
end;

initialization
  FBRegisterUDRTrigger('my_employee_update', TMyEmployeeUpdateTrigger);

end.
```

Figure 5: Example Before Update

4

fbudr Reference

This chapter provides a reference for the *fbudr* package.

4.1 UDR Controller Options

The UDR Controller Options are held in the writeable constant:

```
FBUDRControllerOptions:TFBUDRControllerOptions
```

where

```
TFBUDRControllerOptions = record
    ModuleName: AnsiString;
    AllowConfigFileOverrides: boolean;
    LogFileNameTemplate: AnsiString;
    ConfigFileNameTemplate: AnsiString;
    ForceWriteLogEntries: boolean;
    LogOptions: TFBUDRControllerLogOptions;
    ThreadSafeLogging: boolean;
  end;
```

The options are defined in the following table:

Name	Type	Default	Can Override in Config File	Usage
Module Name	String	None	N	Specifies the module name. Must be set in the library initialisation block.
AllowConfigFileOverrides	Boolean	False	N	Set to true to allow UDR Controller options to be overridden in config file
LogFileNameTemplate	String	(see 4.1.1)	Y	Absolute path to this library's log file. Defined using macros (see below)

Name	Type	Default	Can Override in Config File	Usage
ConfigFileNameTemplate	String	(see 4.1.1)	N	Absolute path to this library's config file.
ForceWriteLogEntries	Boolean	False	Y	If true then the log file is closed and re-opened after each entry is written to the file. This may be necessary to pinpoint the location of a serious error in the library.
LogOptions	set	[]	Y	The set of logging options (see below)
ThreadSafeLogging	Boolean	False	Y	If true then writes to the log file take place in a critical section. This may be needed in some multi-threading environments where multiple simultaneous UDR functions are invoked.

4.1.1 File Name Templates

Both the config and log file path and file name are defined using macros. The macros available are:

- $LOGDIR = Firebird log directory (usually the Firebird root directory)
- $UDRDIR = Firebird UDR directory (usually plugins/udr relative to the Firebird root)
- $TEMP = System temp directory
- $MODULE = Module Name
- $TIMESTAMP = date/time in "yyyymmddhhnnss” format.

Any macro may be used in either file name together with literal text. The defaults are:

ConfigFileNameTemplate: $UDRDIR$MODULE.conf

LogFileNameTemplate: $LOGDIR$TIMESTAMP$MODULE.log

Setting a template value to the empty string implicitly disables the use of a configuration file or log file respectively.

4.1.2 Log File Options

The logging options are:

loLogFunctions,	Log calls to UDR functions and actions
loLogProcedures,	Log calls to UDR procedures and actions
loLogTriggers,	Log calls to UDR triggers and actions
loLogFetches,	Log each fetch in a select procedure
loModifyQueries,	Log each update/insert/delete query called by a UDR routine

loReadOnlyQueries,	Log each select query called by a UDR Routine
loDetails	Log details (e.g. input and output parameters). Used in conjunction with loLogFunctions, loLogProcedures, loLogFetches and loLogTriggers to determine which UDR routines it applies to.

When included in a config file, the syntax of the config file entry uses the logging options as defined above e.g.

LogOptions = [loLogFunctions,loDetails]

4.2 The Configuration File

A UDR Library's configuration file is optional and when present is expected to be found at the location given by the expansion of the ConfigFileNameTemplate Controller Option.

A config file is a text file in *ini file* format. UDR Controller options should be placed in the “Controller” section. e.g.

```
[Controller]
LogOptions = [loLogFunctions, loLogProcedures, loLogTriggers, loDetails,
                                loModifyQueries, loReadOnlyQueries]
LogFileNameTemplate = $LOGDIRMODULE.log
```

Further sections may be added for use by the UDR's themselves.

4.3 The Log File

The primary purpose of the log file is to support debugging of the UDR library. The log always records Controller Options overrides and exceptions caught by the UDR Controller. Otherwise, log file contents depends on the LogOptions setting.

Each log file entry recorded by the UDR library starts with an ampersand, and is followed by a timestamp and the log message itself e.g.

```
@9-1-22 23:40:10.0060:Registering Function row_count
```

The date and time formats use the default shortdate and longtime format settings, expanded to 10th millisecond resolution.

The recording of SQL queries in the log is sourced from the *fbintf* journal and, when present, are formatted as defined by the *fbintf* journal function.

4.4 UDR Functions

UDR Functions are defined by sub-classing the TFBUDRFunction class defined in FBUDRController. A simplified declaration of TFBUDRFunction is given below.

```
  TFBUDRFunction = class(Firebird.IexternalFunctionImpl)
  public
    function getCharSet(context: IFBUDRExternalContext): AnsiString; overload; virtual;
    function Execute(context: IFBUDRExternalContext;
                     ProcMetadata: IFBUDRProcMetadata;
                     InputParams: IFBUDRInputParams;
                     ResultSQLType: cardinal): variant; overload; virtual;
    procedure Execute(context: IFBUDRExternalContext;
                      ProcMetadata: IFBUDRProcMetadata;
                      InputParams: IFBUDRInputParams;
```

```
                    ReturnValue: ISQLParam); overload; virtual;
    class procedure setup(context: IFBUDRExternalContext;
                  metadata: IFBUDRRoutineMetadata;
                  inBuilder: IFBUDRMetadataBuilder;
                  outBuilder: IFBUDRMetadataBuilder); virtual;
     procedure InitFunction; virtual;
     property Name: AnsiString;
     property FirebirdAPI: IFirebirdAPI;
end;
```

A sub-class of this class must override one, but not both, of the overloaded Execute function/procedure.

4.4.1 Properties

Name	Function name given when function is registered with the UDR Controller by a call to FBRegisterUDRFunction.
FirebirdAPI	A reference to the FirebirdAPI interface. This reference is identical to that returned by context.GetFirebirdAPI (see 4.8.1)

4.4.2 GetCharSet Function

This function is called by the Firebird engine to allow the external function an opportunity to declare use of a different character set to the connection default. The function returns a character set name as recorded in the database's RDB$CHARACTER_SETS system table.

The only function parameter provides the execution context (see 4.8.1).

It is unlikely that you will want to use this capability. The AnsiString type includes an attribute identifying the code page used to encode the character string. This is recognised by *fbudr* and any transliteration necessary to or from the connection character is performed automatically.

4.4.3 Execute Function

The Execute function is the version used to implement a UDR function when the return type is a simple scalar type (e.g. integer, string, etc.).

The function parameters are:

- The Execution Context (see 4.8.1).
- The Procedure Metadata (see 4.8.3).
- The Input Parameters (see 4.8.6), and
- The SQL Type of the function result (see list of SQL Type constants in IB.pas in *fbintf*).

The function body should perform whatever algorithm is required by the function definition.

The function result is returned as a variant and must be type compatible with the return value type defined in the database schema Function declaration.

4.4.4 Execute Procedure

The Execute Procedure is identical in use to the Execute Function expect that the result is returned using the ReturnValue parameter. This is set using the ***ISQLParam*** interface which allows for complex types (e.g. Numeric Values, Blobs and Timestamp with time zone).

4.4.5 Setup Procedure

This is a class procedure and is called by the Function Factory when the library is first loaded and before each new instance of an external function object is instantiated from this class. It may perform any processing necessary before the object is instantiated and set class var properties or global variables. It is only rarely expected to be used.

The procedure parameters are:

- The Execution Context (see 4.8.1).
- The Routine Metadata (see 4.8.2), and
- metadata builder interfaces for the input and output parameters.

4.4.6 InitFunction Procedure

This virtual procedure is called from the class constructor and may be overridden to initialise properties defined by the subclass.

Note: this procedure is called by the constructor after initialising the parent class but before writing a log file entry. The class destructor should be overridden in the normal way If there is a need to clean up the class instance when it is destroyed.

4.4.7 Registering a UDR Function

```
TFBUDRFunctionClass = class of TFBUDRFunction;

procedure FBRegisterUDRFunction(aName: AnsiString; aFunction: TFBUDRFunctionClass);
```

A UDR Function subclass must be registered with the UDR Controller at initialisation time using the FBRegisterUDRFunction (for an example see 3.2). The name of the function must be the *routine name* part of the external name of the function in the external function declaration in the database schema.

4.5 UDR Execute Procedures

UDR Execute Procedures are defined by sub-classing the TFBUDRExecuteProcedure class defined in FBUDRController. An Execute Procedure is a procedure that returns, at most, only a single row of output parameters. In PSQL, it is characterised by the lack of a “SUSPEND” statement in the procedure body.

TFBUDRExecuteProcedure is itself sub-classed from TFBUDRProcedure. A simplified declaration of both is given below.

```
  TFBUDRProcedure = class(Firebird.IExternalProcedureImpl)
  public
    function getCharSet(context: IFBUDRExternalContext): AnsiString; overload; virtual;
    class procedure setup(context: IFBUDRExternalContext;
                    metadata: IFBUDRRoutineMetadata;
                    inBuilder: IFBUDRMetadataBuilder;
                    outBuilder: IFBUDRMetadataBuilder); virtual;
```

```
    property Name: AnsiString read Fname;
    property FirebirdAPI: IFirebirdAPI;
  end;

..TFBUDRExecuteProcedure = class(TFBUDRProcedure)
  public
    procedure Execute(context: IFBUDRExternalContext;
                      ProcMetadata: IFBUDRProcMetadata;
                      InputParams: IFBUDRInputParams;
                      OutputData: IFBUDROutputData); virtual; abstract;
  end;
```

A sub-class of TFBUDRExecuteProcedure must override the Execute Procedure, and may override the setup and getCharSet functions inherited from TFBUDRProcedure.

4.5.1 Properties

Name	Function name given when procedure is registered with the UDR Controller by a call to FBRegisterUDRProcedure.
FirebirdAPI	A reference to the FirebirdAPI interface. This reference is identical to that returned by context.GetFirebirdAPI (see 4.8.1)

4.5.2 The getCharSet Function

See 4.8.2.

4.5.3 The Execute Procedure

This procedure is overridden to implement a UDR Execute Procedure and has the following parameters:

- The Execution Context (see 4.8.1).
- The Procedure Metadata (see 4.8.3).
- The Input Parameters (see 4.8.6), and
- The Output Parameters (see 4.8.7).

The Output Parameters are read/write and used to return the singleton row returned by the procedure.

4.5.4 The Setup Procedure

See 4.4.5.

4.5.5 The InitProcedure Procedure

This virtual procedure is called from the class constructor and may be overridden to initialise properties defined by the subclass.

Note: this procedure is called by the constructor after initialising the parent class but before writing a log file entry. The class destructor should be overridden in the normal way If there is a need to clean up the class instance when it is destroyed.

4.5.6 Registering a UDR Execute Procedure

```
TFBUDRProcedureClass = class of TFBUDRProcedure;

procedure FBRegisterUDRProcedure(aName: AnsiString;
  aProcedure: TFBUDRProcedureClass);
```

A UDR procedure subclass must be registered with the UDR Controller at initialisation time using the FBRegisterUDRProcedure (for an example see 3.3). The name of the function must be the *routine name* part of the external name of the procedure in the external procedure declaration in the database schema.

4.6 UDR Select Procedures

UDR Select Procedures are defined by sub-classing the TFBUDRSelectProcedure class defined in FBUDRController. A Select Procedure is a procedure that returns, zero, one or more row of output parameters. In PSQL, it is characterised by including a “SUSPEND” statement in the procedure body.

TFBUDRSelectProcedure is itself sub-classed from TFBUDRProcedure (see 4.5). A simplified declaration of TFBUDRSelectProcedure is given below.

```
  TFBUDRSelectProcedure = class(TFBUDRProcedure)
  public
    procedure open(context: IFBUDRExternalContext;
                   ProcMetadata: IFBUDRProcMetadata;
                   InputParams: IFBUDRInputParams); overload; virtual; abstract;
    function fetch(OutputData: IFBUDROutputData): boolean; virtual;
    procedure close; virtual;
  end;
```

A sub-class of TFBUDRSelectProcedure must override the open procedure and the fetch function. It may override the close procedure. It may also override the getCharset function and the setup procedure inherited from TFBUDRProcedure.

The sub-class will typically include private properties used to pass information from the open procedure to the fetch function.

4.6.1 The getCharSet Function

See 4.8.2.

4.6.2 The Open Procedure

This procedure is used to initialise the select procedure and must perform the work necessary for the subsequent call to the fetch function to return the first data row, if any.

It has the following parameters:

- The Execution Context (see 4.8.1).
- The Procedure Metadata (see 4.8.3), and
- The Input Parameters (see 4.8.6).

4.6.3 The fetch function

The fetch function will be called repetitively by the Firebird engine to return each output row in the dataset returned by a select procedure, until the function returns false in order to indicate no more data.

The output row is returned using the output parameters (see 4.8.7).

4.6.4 The close Procedure

The close procedure is called after the fetch function returns false or otherwise when the object is destroyed. It may be used to perform any tidying up necessary (e.g. releasing resources) after the last row has been returned.

4.6.5 The Setup Procedure

See 4.4.5.

4.6.6 Registering a UDR Select Procedure

See 4.5.6.

4.7 UDR Triggers

UDR Triggers are defined by sub-classing the TFBUDRTrigger class defined in FBUDRController. A simplified declaration of TFBUDRTrigger is given below.

```
  TFBUDRTrigger = class(Firebird.IExternalTriggerImpl)
  public
    function getCharSet(context: IFBUDRExternalContext): AnsiString; overload; virtual;
    procedure AfterTrigger(context: IFBUDRExternalContext;
                           TriggerMetaData: IFBUDRTriggerMetaData;
                           action: TFBUDRTriggerAction;
                           OldParams: IFBUDRInputParams;
                           NewParams: IFBUDRInputParams); virtual;
    procedure BeforeTrigger(context: IFBUDRExternalContext;
                           TriggerMetaData: IFBUDRTriggerMetaData;
                           action: TFBUDRTriggerAction;
                           OldParams: IFBUDRInputParams;
                           NewParams: IFBUDROutputData); virtual;
    procedure DatabaseTrigger(context: IFBUDRExternalContext;
                           TriggerMetaData: IFBUDRTriggerMetaData); virtual;
    class procedure setup(context: IFBUDRExternalContext;
                  metadata: IFBUDRRoutineMetadata;
                  fieldsBuilder: IFBUDRMetadataBuilder); virtual;

    procedure InitTrigger; virtual;
    property Name: AnsiString'
    property FirebirdAPI: IFirebirdAPI;
end;
```

Normally only one of the AfterTrigger, BeforeTrigger and DatabaseTrigger procedures is overridden by a sub-class. Optionally, getCharSet and setup may also be overridden.

4.7.1 Properties

Name	Function name given when trigger is registered with the UDR Controller by a call to FBRegisterUDRTrigger.
FirebirdAPI	A reference to the FirebirdAPI interface. This reference is identical to that returned by context.GetFirebirdAPI (see 4.8.1)

4.7.2 The getCharSet Function

See 4.8.2.

4.7.3 The AfterTrigger Procedure

Override AfterTrigger in order to carry out an *after* trigger's function. Separate interfaces are used to provide the "old" and "new" values of each of the parent dataset's columns. Note that these are both read only for an *after* trigger. The procedure parameters are:

- The Execution Context (see 4.8.1).
- The Trigger Metadata (see 4.8.4)
- “Action” identifying the trigger as an Update, Insert or Delete trigger.
- The OldParams (see 4.8.6) providing the “old” values i.e. prior to the dataset update (Update and Delete triggers only)
- The NewParams (see 4.8.6) providing the “new” value i.e. after the database update (Update and Insert Triggers only).

4.7.4 The Before Trigger Procedure

Override BeforeTrigger in order to carry out a *before* trigger's function. Separate interfaces are used to provide the "old" and "new" values of each of the parent dataset's columns. Note that the new values are writeable for an *after* trigger.

The procedure parameters are:

- The Execution Context (see 4.8.1).
- The Trigger Metadata (see 4.8.4)
- “Action” identifying the trigger as an Update, Insert or Delete trigger.
- The OldParams (see 4.8.6) providing the “old” values i.e. prior to the dataset update. (Update and Delete triggers only)
- The NewParams (Update and Insert Triggers only) (see (4.8.7).

The NewParams may be used to set updated values for the dataset columns prior to the update, or insert action.

4.7.5 The Database Trigger Procedure

Override DatabaseTrigger in order to carry out a database trigger's function. Note that a database trigger does not have any input values, and has only the following parameters:

- The Execution Context (see 4.8.1).
- The Trigger Metadata (see 4.8.4)

4.7.6 The Setup Procedure

This is a class procedure and is called by the Trigger Factory when the library is first loaded and before each new instance is created of a trigger object instantiated using this class. It may perform any processing necessary before the object is instantiated and set class var properties or global variables. It is only rarely expected to be used.

The procedure parameters are:

- The Execution Context (see 4.8.1).
- The Routine Metadata (see 4.8.2), and
- the metadata builder interface for the trigger metadata.

4.7.7 The InitTrigger Procedure

This virtual procedure is called from the class constructor and may be overridden to initialise properties defined by the subclass.

Note: this procedure is called by the constructor after initialising the parent class but before writing a log file entry. The class destructor should be overridden in the normal way If there is a need to clean up the class instance when it is destroyed.

4.7.8 Registering a UDR Trigger

```
TFBUDRTriggerClass = class of TFBUDRTrigger;

procedure FBRegisterUDRTrigger(aName: AnsiString; aTrigger: TFBUDRTriggerClass);
```

A UDR trigger subclass must be registered with the UDR Controller at initialisation time using the FBRegisterUDRTrigger (for an example see 3.4). The name of the trigger must be the *routine name* part of the external name of the trigger in the external trigger declaration in the database schema.

4.8 Support Interfaces

4.8.1 External Context

The External Context interface is provided as a parameter to most UDR function, procedure and trigger methods. It provides access to context information provided by Firebird and to the log and configuration files.

```
 IFBUDRExternalContext = interface
    ['{00b2616d-12e0-436a-8c2c-58670a2be805}']
    function GetFirebirdAPI: IFirebirdAPI;
    function GetAttachment: IAttachment;
    function GetTransaction: ITransaction;
    function GetUserName: AnsiString;
    function GetDatabaseName: AnsiString;
    function GetClientCharSet: AnsiString;
    function obtainInfoCode: Integer;
    function getInfo(code: Integer): Pointer;
    function setInfo(code: Integer; value: Pointer): Pointer;
    function cloneAttachment: IAttachment;
    function GetServiceManager: IServiceManager;    function getStatus: Firebird.IStatus;
    procedure CheckStatus;
    function HasConfigFile: boolean;
    function ReadConfigString(Section, Ident, DefaultValue: AnsiString): AnsiString;
    function ReadConfigInteger(Section, Ident: AnsiString; DefaultValue: integer): integer;
    function ReadConfigBool(Section, Ident: AnsiString; DefaultValue: boolean): boolean;
    procedure WriteToLog(Msg: AnsiString);
```

GetFirebirdAPI	Returns the Firebird Pascal API Interface provided by the *fbintf* package. This is an encapsulation of the Firebird ***IMaster*** interface. Note that this instance of the ***IFirebirdAPI*** interface does not provide any information about the underlying Firebird Client library.
GetAttachment	Returns the *fbintf* ***IAttachment*** interface for the calling user's database attachment.
GetTransaction	Returns the *fbintf* ***ITransaction*** interface for the transaction under which the UDR is invoked.
GetUserName	Returns the logged in user name for the user invoking the UDR.
GetDatabaseName	Returns the path to the database file on the server.
GetClientCharSet	Returns the name of the connection character set.
obtainInfoCode	Guaranteed to return a unique integer allocated from a monotonically increasing series.
getInfo	Returns a pointer to a memory block previously saved using a unique code.
setInfo	Used to save a pointer to a memory block using a unique code and for later retrieval by getInfo. The unique code is usually allocated by a call to obtainInforCode.
cloneAttachment	Clones the attachment returned by IAttachment to create another (concurrent) attachment to the same database. This can be used in multi-threaded UDRs which need an attachment per thread.
GetServiceManager	Returns an IServiceManager interface to the local Firebird Server.
getStatus	Returns a Firebird ***IStatus*** interface. This can be used in calls to the Firebird API in order to receive any error information returned by the API call.
CheckStatus	Checks the status interface returned by a call to getStatus and raises and exception if an error is detected. A call to CheckStatus should follow each call to the Firebird API that uses an ***IStatus*** interface provided by getStatus.
HasConfigFile	Returns true if a configuration file for this UDR library has been located and loaded.
ReadConfigString	Used to return a configuration file entry of type string in the identified section and with the given name.
ReadConfigInteger	Used to return a configuration file entry of type integer in the identified section and with the given name.
ReadConfigBool	Used to return a configuration file entry of type boolean in the identified section and with the given name.
WriteToLog	Writes the provided message to the UDR library log. This message is silently discarded if the log file is disabled.

4.8.2 Routine Metadata

Routine metadata is provided by the Firebird engine with most calls to UDR functions, procedures and triggers. The interface is generic to each type of UDR. Filtered versions i.e. ***IFBUDRProcMetadata*** and ***IFBUDRTriggerMetaData*** are used in most instances.

```
IFBUDRRoutineMetadata = interface
   ['{28a03226-e8df-40e8-b67f-d3dc27886e9f}']
   function getPackage: AnsiString;
   function getName: AnsiString;
   function getEntryPoint: AnsiString; {response is parsed into the following three components}
   function getModuleName: AnsiString;
   function getRoutineName: AnsiString;
   function getInfo: AnsiString;
   function getBody: AnsiString;
   function HasInputMetadata: boolean;
   function HasOutputMetadata: boolean;
   function HasTriggerMetadata: boolean;
   function getFBInputMetadata: IFBUDRMessageMetadata;
   function getFBOutputMetadata: IFBUDRMessageMetadata;
   function getFBTriggerMetadata: IFBUDRMessageMetadata;
   function getTriggerTable: AnsiString;
   function getTriggerType: TFBUDRTriggerType;
 end;
```

getPackage	Returns the name of the “package”, if any, in which the function or procedure has been defined as part of in database schema.
getName	Returns the function, procedure or trigger name as used in the database schema definition.
getEntryPoint	Returns the entry point name as given in the database schema definition. This is decomposed into its three parts in the next three interface functions.
getModuleName	Returns the module (or library name) part of the entry point name.
getRoutineName	Returns the routine name part of the entry point name.
getInfo	Returns the info part of the entry point name. This part is optional and, if present, is passed by Firebird as an opaque string to the UDR. Its purpose is whatever the UDR considers it to be.
getBody	Returns the function, procedure or trigger body as defined in the database schema, Note: this is assumed to be empty for UDRs.
HasInputMetadata	Returns true if an input metadata interface is present.
HasOutputMetadata	Returns true if an output metadata interface is present.
HasTriggerMetadata	Returns true if a triggermetadata interface is present.
getFBInputMetadata	Returns an interface to the input metadata for the UDR function or procedure. (see 4.8.5)
getFBOutputMetadata	Returns an interface to the output metadata for the UDR function or procedure (see 4.8.5).
getFBTriggerMetadata	Returns an interface to the trigger metadata for UDR triggers (see 4.8.5).

getTriggerTable	Returns the name of the table or view to which the trigger applies.
getTriggerType	Returns the trigger type (ttAfter, ttBefore, ttDatabase).

4.8.3 Proc Metadata

The Proc metadata interface is a simplified version of the Routine Metadata interface focusing on the interface specific to UDR functions and procedures.

Note: if necessary, an ***IFBUDRRoutineMetadata*** interface can be obtained from an ***IFBUDRProcMetadata*** interface using the QueryInterface method inherited from ***IUnknown***.

```
 IFBUDRProcMetadata = interface
    ['{d20fc3ae-635e-4841-ad79-b4cd88be75d8}']
    function getPackage: AnsiString;
    function getName: AnsiString;
    function getEntryPoint: AnsiString;
    function getModuleName: AnsiString;
    function getRoutineName: AnsiString;
    function getInfo: AnsiString;
```

4.8.4 Trigger Metadata

The Trigger metadata interface is a simplified version of the Routine Metadata interface focusing on the interface specific to UDR triggers.

Note: if necessary, an ***IFBUDRRoutineMetadata*** interface can be obtained from an ***IFBUDRTriggerMetaData*** interface using the QueryInterface method inherited from ***IUnknown***.

```
IFBUDRTriggerMetaData = interface
    ['{9458bad8-809a-469a-b13f-3a3ab95f8d94}']
    function getName: AnsiString;
    function getModuleName: AnsiString;
    function getRoutineName: AnsiString;
    function getInfo: AnsiString;
    function getTriggerTable: AnsiString;
    function getTriggerType: TFBUDRTriggerType;
  end;
```

4.8.5 Firebird Metadata

This is the Firebird ***IMessageMetadata*** interface presented as a native Pascal interface. The interface functions have the same semantics as the original Firebird interface.

```
IFBUDRMessageMetadata = interface
    ['{da84190f-91a3-40ae-9fab-bbfd98a49dcb}']
    function getCount: Cardinal;
    function getField(index: Cardinal): AnsiString;
    function getRelation(index: Cardinal): AnsiString;
    function getOwner(index: Cardinal): AnsiString;
    function getAlias(index: Cardinal): AnsiString;
    function getType(index: Cardinal): Cardinal;
    function isNullable(index: Cardinal): Boolean;
    function getSubType(index: Cardinal): Integer;
    function getLength(index: Cardinal): Cardinal;
    function getScale(index: Cardinal): Integer;
    function getCharSet(index: Cardinal): Cardinal;
    function getOffset(index: Cardinal): Cardinal;
    function getNullOffset(index: Cardinal): Cardinal;
    function getBuilder: IFBUDRMetadataBuilder;
    function getMessageLength: Cardinal;
```

```
    function getAlignment: Cardinal;
    function getAlignedLength: Cardinal;
  end;
```

4.8.6 Input Params

The input params interface is a specialised version of the *fbintf* ***IResults*** interface. It provides read only access to:

- the input parameters to a UDR function or procedure, or
- the old values for before update and delete triggers, or
- to the new values for after insert and update triggers.

The actual parameter values (and metadata) are accessed either positionally (zero based) or by name as defined in the database schema definition for the UDR function, procedure or trigger. Each parameter is accessed using the *fbintf* ***ISQLData*** interface.

```
 IFBUDRInputParams = interface
    ['{e49d096e-3a9c-4f75-bb39-db32b1897312}']
    function getCount: integer;
    function getSQLData(index: integer): ISQLData;
    function ParamExists(Idx: AnsiString): boolean;
    function ByName(Idx: AnsiString): ISQLData;
    property Data[index: integer]: ISQLData read getSQLData; default;
    property Count: integer read getCount;
  end;
```

4.8.7 Output Data

The output data interface is a specialised version of the *fbintf* ***ISQLParams*** interface. It provides read/write interface to:

- each field in the output data for UDR Procedures
- the new values for before insert or update triggers.

The actual parameter values (and metadata) are accessed or set either positionally (zero based) or by name as defined in the database schema definition for the UDR procedure or trigger. Each parameter is accessed using the *fbintf* ***ISQLParam*** interface.

```
IFBUDROutputData = interface
    ['{8a7d7890-e9a4-430b-8cbc-3874b5f66b31}']
    function getCount: integer;
    function getSQLParam(index: integer): ISQLParam;
    function FieldExists(Idx: AnsiString): boolean;
    function GetModified: Boolean;
    function GetHasCaseSensitiveParams: Boolean;
    function ByName(Idx: AnsiString): ISQLParam ;
    procedure Clear;
    property Modified: Boolean read GetModified;
    property Params[index: integer]: ISQLParam read getSQLParam; default;
    property Count: integer read getCount;
  end;
```

4.8.8 Metadata Builder Interface

This is the Firebird ***IMetadataBuilder*** interface presented as a native Pascal interface. The interface functions have the same semantics as the original Firebird interface.

```
 IFBUDRMetadataBuilder = interface
    ['{a6876fed-fd70-40f0-b965-6c43b8c5c00d}']
    procedure setType(index: Cardinal; type_: Cardinal);
    procedure setSubType(index: Cardinal; subType: Integer);
    procedure setLength(index: Cardinal; length: Cardinal);
    procedure setCharSet(index: Cardinal; charSet: Cardinal);
```

```
  procedure setScale(index: Cardinal; scale: Integer);
  procedure truncate(count: Cardinal);
  procedure moveNameToIndex(name: AnsiString; index: Cardinal);
  procedure remove(index: Cardinal);
  function addField:Cardinal;
  procedure setField(index: Cardinal; field: AnsiString);
  procedure setRelation(index: Cardinal; relation: AnsiString);
  procedure setOwner(index: Cardinal; owner: AnsiString);
  procedure setAlias(index: Cardinal; alias: AnsiString);
end;
```

5

UDRs and Transactions

When a UDR Function, Procedure or Trigger is called, it is called in the context of the current (user) transaction. An interface to this transaction is provided as part of the external context (see 4.8.1). Most calls to the current database will use this transaction.

You may not Commit or Rollback this transaction, or call Commit Retaining or Rollback Retaining.

You can start additional transactions within a UDR in the normal way. These transactions must be completed (Commit or Rollback) before the UDR returns. Note that even after a transaction has been committed, any changes made to the database under that transaction will not be visible to the user transaction unless the user transaction was started with Read_Committed in its attributes.

The UDR Controller will forcibly release any uncompleted transactions when the UDR Function, Procedure or Trigger executes. It will first attempt to complete them using their default completion. However, if that fails, the transaction handle is released and the transaction may hence be left in limbo.

6

Testing Strategies

Testing a UDR library can be more difficult than testing a normal program. An IDE with an appropriate debugger (e.g. GDB) can be used to step through a shared library. However, this can be difficult to get working reliably, and in some cases may not be possible. As an alternative, the following testing strategies may be considered:

- **Exception Handling:** All exceptions raised by UDR functions, procedures or triggers are caught by the UDR controller and returned to the client using the Firebird status vector. They should thus be faithfully rendered to the client. Using exceptions to trap and report unexpected or incorrect actions should be part of the programming strategy for UDR functions, procedures and triggers.

  Note: access violations are not, in general, caught by the UDR Controller. These are often reported as a general protection fault, a segmentation fault or a lost connection, with limited additional information. Exception handling cannot be relied on to debug such events.

- **Using the Log File**. The UDR library log file can provide copious amounts of debugging information. Each step in the creation and use of a UDR function, procedure or trigger can be logged, in sequence, including parameter input and output values and metadata. It is also possible to log each database query made by a UDR function, procedure or trigger. In addition, a UDR library programmer can log internal steps and results made by their UDR function, procedure or trigger.

  Note: logging internal steps may sometimes be the only workable strategy for catching an access violation. However, a bad crash can mean that the log file loses the last one or two entries. To overcome this, set the ForceWriteLogEntries Controller option to true. This forces the log file to be closed and re-opened after each entry is written to the file. Normally, this option should be set to false in order to minimise processing overhead.

- **Using the UDR Engine Testbed:** This is found in the *fbudrtestbed* package and can be used to test UDR library logic in a client side program. Such a program can readily be debugged using an IDE and is useful for locating otherwise hard to detect logic errors.

6.1 The UDR Testbed

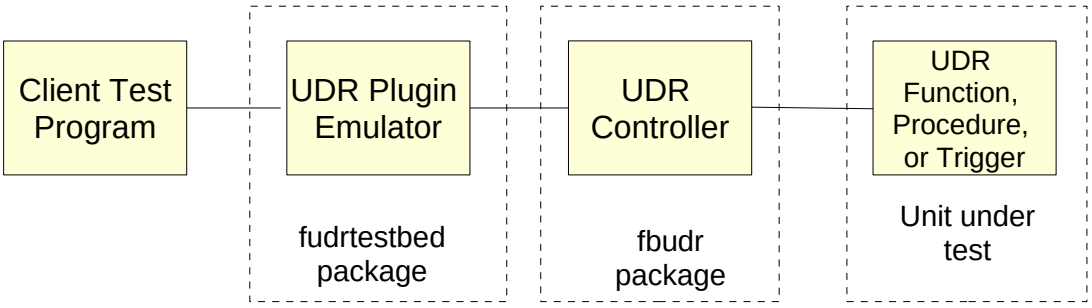

Figure 6: Testbed Functional Components

The figure above illustrates the functional components that make up a UDR Function, Procedure or Trigger Testbed, and which are linked together to create a clientside testbed program. From the right:

- The unit under test contains the UDR Function, Procedure or Trigger under test. Figure 3 on page 9 provides an example of such a unit.
- The UDR Controller is part of the *fbudr* package and is normally the core part of a UDR library built using the *fbudr* package. It is included here as part of the testbed and performs the same role as in UDR library.
- The UDR Plugin emulator is the core component of the *fbudrtestbed* package. Here it replaces the Firebird UDR Engine as far as the UDR Controller is concerned. It provide *makeFunction*, *makeProcedure* and *makeTrigger* functions to the Client Test Program similar to the same services that the Firebird UDR Engine provides to Firebird.
- The Client Test Program is written by the user. It calls the UDR Plugin emulator to create a new instance of the UDR Function, Procedure or Trigger. It may then execute the UDR Function, Procedure or Trigger, after first providing it with input parameters. It can examine the output parameters, if any, on completion.

The testbed is built as a single program that can be run under the Lazarus or Delphi IDE allowing the user to step through the UDR Function, Procedure or Trigger under test in order to locate and fix any logic errors that are proving hard to detect using exception reporting or log file analysis.

Note: the testbed was initially used to debug the UDR Controller itself.

6.2 Example Client Test Program

Assuming that the objective is to debug the simple example UDR Function presented in 3.2, a testbed program may be built as follows.

6.2.1 With Lazarus

Prior to first use, you should first open each of the package files “fbintf.lpk”, “fbudr.lpk” and fbudrtestbed.lpk, using the menu item Packages->Open Package. This is sufficient for Lazarus to recognise each package.

In the Lazarus IDE, select File->New and click on “Simple Program” as the project type. This will create and show a basic program source code file. In the project inspector, select Add->New Requirement and select the *fbudrtestbed* package. This will automatically bring in *fbudr* and *fbintf*.

You should also add the unit under test to the program source code units.

You should finally save the project using some suitable program name.

6.2.2 With Delphi

Prior to first use, you should open, in the Delphi IDE, each of the package files “fbintf.dproj” , “fbudr.dproj” and “fbudrtestbed.dproj”, and compile the packages each in turn.

In the Delphi IDE, select File->New->Other and double click on “Console Application” when the dialog opens showing each of the options. You should now save the project using a suitable program name.

The project should now be linked to the *fbudr* package. This is done by

1. Select Project->Options
2. In the Options dialog, select Packages->Run Time Packages.
3. Click on the ellipses in the right hand window at the end of the current list of runtime packages.
4. In the Run Time Packages dialog, select the *fbudrtestbed* package by clicking on the yellow folder icon and browsing for the fbudrtestbed.dcp file. This is, by default, located in the fbintf\Win32\Debug, or the fbintf\Win64\Debug folders. Select the fbudrtestbed.dcp file, click on the “open” button”, click on the “Add” button and finally the “OK” button to close the dialog.

You should also add the unit under test to the program source code units.

6.2.3 Completing the Testbed Client

Figure 7 shows an example testbed client program to test out the example UDR function presented in Figure 3. Note that this is simple enough to comprise a single source file.

The program:

1. Sets the FBUDRControllerOptions as appropriate.
2. Opens a connection to the database used by the unit under test.
3. Creates an instance of the UDR Plugin emulator and assigns the database connection to it.

   Note that the UDR Plugin emulator should only be created once per testbed program.
4. Calls the plugin to “make” an instance of the 'row_count' UDR function. It provides:
   - The name of the function in the database schema,
   - The package name if appropriate, and
   - The routine name under which it is registered with the UDR Controller.
5. Sets the input parameter.
6. Creates a suitable transaction to execute the UDR.
7. Executes the UDR Function and displays the result.

```
program myudrtestbed;

uses Classes, FBUDRController, FBUdrPlugin, IB, udr_myrowcount;

procedure TestRowCount(UDRPlugin: TFBUdrPluginEmulator);
var MyRowCount: TExternalFunctionWrapper;
    Transaction: ITransaction;
    Rows: integer;
begin
  {Get the emulator wrapper for the row_count function, declared as MyRowCount}
  MyRowCount := UDRPlugin.makeFunction('MYROWCOUNT',  {Name of Function in database
                                                       - case sensitive}
                                       '',            {package name is empty}
                                       'myudrlibrary!row_count' {entry point}
                                       );
  try
    writeln('Row Count for Employee');
    {set the input parameter to the EMPLOYEE table}
    MyRowCount.InputParams[0].AsString := 'EMPLOYEE';

    Transaction := UDRPlugin.Attachment.StartTransaction(
                     [isc_tpb_read,isc_tpb_nowait,isc_tpb_concurrency],taCommit);

    {invoke the function and print the result}
    writeln('Employee Row Count = ',MyRowCount.Execute(Transaction).AsInteger);
    writeln;
  finally
    MyRowCount.Free
  end;
end;

procedure RunTest;
var Attachment: IAttachment;
    DPB: IDPB;
    UDRPlugin: TFBUdrPluginEmulator;
begin
  {Open a connection with the example employee database. Amend database parameters
   as needed.}
  DPB := FirebirdAPI.AllocateDPB;
  DPB.Add(isc_dpb_user_name).setAsString('SYSDBA');
  DPB.Add(isc_dpb_password).setAsString('masterkey');
  DPB.Add(isc_dpb_lc_ctype).setAsString('UTF8');
  DPB.Add(isc_dpb_set_db_SQL_dialect).setAsByte(3);
  Attachment := FirebirdAPI.OpenDatabase('localhost:employee',DPB);
  try
    UDRPlugin := TFBUdrPluginEmulator.Create(FBUDRControllerOptions.ModuleName);
    try
      {initialize the emulator with the database connection}
      UDRPlugin.Attachment := Attachment;
      TestRowCount(UDRPlugin);
    finally
      UDRPlugin.Free;
    end;
  finally
    Attachment.Disconnect(true);
  end;
end;

begin
  with FBUDRControllerOptions do
  begin
    ModuleName := 'myudrlibrary';
    AllowConfigFileOverrides := true;
    LogFileNameTemplate := '$LOGDIR$MODULE.log';
    LogOptions := [loLogFunctions, loLogProcedures, loLogTriggers, loDetails];
  end;
  RunTest;
end.
```

Figure 7: Example Testbed program

Testbeds for Execute and Select procedures and triggers will follow a similar approach.

6.3 Reference

6.3.1 UDR Plugin

A simplified declaration of the TFBUdrPluginEmulator class is:

```
TFBUdrPluginEmulator = class(Firebird.IUdrPluginImpl)
    constructor Create(aModuleName: AnsiString);
    destructor Destroy; override;
    function makeFunction(aFunctionName, aPackageName,
      aEntryPoint: AnsiString): TExternalFunctionWrapper;
    function makeProcedure(aProcName, aPackageName, aEntryPoint: AnsiString):
                                                          TExternalProcedureWrapper;
    function makeTrigger(aName, aEntryPoint, datasetName: AnsiString; aTriggerType: cardinal):
                                                          TExternalTriggerWrapper;
    property Attachment: IAttachment read FAttachment write SetAttachment;
    property ModuleName: AnsiString read FModuleName;
  end;
```

Prior to use, the Attachment property must be set to a valid connection to a database. This can be updated between tests of UDR Functions, Procedures or Triggers.

The functions makeFunction, makeProcedure and makeTrigger are called respectively to obtain and instance of the UDR Function, Procedure or Trigger identified on the function call.

6.3.2 The UDR Function Wrapper

A simplified declaration of this class is:

```
TExternalFunctionWrapper = class(TExternalWrapper)
  public
    constructor Create(aManager: TFBUdrPluginEmulator;aName, aPackageName, aEntryPoint: AnsiString;
       aFunctionFactory: TFBUDRFunctionFactory;
       preparedStmt: IStatement);
    function Execute(aTransaction: ITransaction): ISQLData;
    property InputParams: ISQLParams read FInputParams;
  end;
```

The InputParams property is used to set the input parameter values prior to the function being executed by the Execute function.

The ***ISQLParams*** interface is defined by the *fbintf* package **[1]**.

The function result is returned as an *fbintf* ***ISQLData*** interface from which the returned value may be obtained.

6.3.3 The UDR Procedure Wrapper

This class is used for both Execute and Select Procedures. A simplified declaration of the class is:

```
TExternalProcedureWrapper = class(TExternalWrapper)
  public
    constructor Create(aManager: TFBUdrPluginEmulator; aName, aPackageName, aEntryPoint: AnsiString;
       aProcedureFactory: TFBUDRProcedureFactory;
       preparedStmt: IStatement);
    function Execute(aTransaction: ITransaction): IProcedureResults;
    property InputParams: ISQLParams read FInputParams;
  end;
```

An object of this class is used much in the same way as for a UDR Function. The difference is that when the UDR Procedure is executed an ***IProcedureResults*** interface is returned.

The ***ISQLParams*** interface is defined by the *fbintf* package **[1]**.

6.3.4 The IProcedureResults Interface

This interface is returned from a call to Execute a UDR Procedure - see above. It is declared as:

```
IProcedureResults = interface
  ['{1b851373-a7c2-493e-b457-6a19980e0f5f}']
  function getCount: integer;
  function ByName(Idx: AnsiString): ISQLData;
  function getSQLData(index: integer): ISQLData;
  function FetchNext: boolean; {fetch next record}
  function IsEof: boolean;
  property Data[index: integer]: ISQLData read getSQLData; default;
  property Count: integer read getCount;
end;
```

This is a simplified version of the *fbintf* ***IResultSet*** interface and is used to return one or more rows from the UDR Procedure. FetchNext must be called before accessing any row data.

- For an Execute Procedure, the first call to FetchNext returns true, and the output row may then be accessed. All subsequent calls to FetchNext return false.
- For a Select Procedure, FetchNext is used to return each row in turn. It returns false once all rows have been returned.

6.3.5 The UDR Trigger Wrapper

A simplified declaration of this class is:

```
TExternalTriggerWrapper = class(TExternalWrapper)
public
  constructor Create(aManager: TFBUdrPluginEmulator; aName,  aTableName, aEntryPoint: AnsiString;
     aTriggerType: cardinal;
     aTriggerFactory: TFBUDRTriggerFactory;
     preparedStmt: IStatement);
  destructor Destroy; override;
  procedure Execute(aTransaction: ITransaction; action: cardinal);
  property OldValues: IFBUDROutputData read FOldValues;
  property NewValues: IFBUDROutputData read FNewValues;
end;
```

The ***IFBUDROutputData*** interface is described in 4.8.7.

Prior to executing the trigger:

- For an Update or Delete Trigger the OldValues must be set.
- For an Update or Insert Trigger the NewValues must be set.

After trigger execution, the NewValues may contain trigger outputs for Before Update or Insert Triggers. Otherwise, in order to determine correct execution, it may be necessary for the client testbed to query database tables in order to ensure that the trigger has correctly updated them.

7

Security Considerations

UDRs should be significantly more secure that legacy UDFs, if only because a UDR library has to be specifically written as a Firebird UDR library rather than allowing a UDF to be created out of any shared library entry point.

System and Database Administrators should be aware that a UDR operates in a privileged environment in that it is possible for a UDR to open a new database connection to any local database accessible to the server, and to do so without needing a password. This is because the UDR is running in the same process and with the same privileges as the database engine. It similarly has privileged access to the service manager.

Prior to installation, UDRs should thus be checked to ensure that they do not contain any unauthorised functionality.

A Security Policy for UDR use should include the following rules :

1. Access Rights should be used to control access to UDR Functions and Procedures in order to ensure use by authorised users and groups only.

2. Where appropriate, access rights should be grant to UDR Functions, Procedures and Triggers to allow them to access database resources in their own right rather than relying on inherited rights from the current user.

3. The Firebird udr directory should be protected to ensure that only an authorised user can install or modify a UDR library. Such protection should extend to all directories on the path to the udr directory.

4. Where a UDR has to present security credentials for access to a resource outside of Firebird, any required credentials must not be compiled into the UDR or read from an external file. Instead, they should always be provided by the user.

   Note: the reason for this rule is that any authorised database administrator can add a UDR Function, Procedure or Trigger declaration to their database schema. However, access to such

resources is not necessary intended for any database user and may be restricted by database or to authorised users only on a single database. Forcing the user to provide the credentials ensures that such resources are accessed only on behalf of authorised users.