Writing Firebird UDRs in Pascal

Denis Simonov, Martin Köditz

Version 1.0.0, 22 Sep 2023

Table of Contents

Chapter 1. Introduction

Firebird has had the ability to extend features of the PSQL language by writing external functions - UDF (User Defined Functions). UDF can be written on almost any compilable programming language.

Firebird 3.0 introduced a plugin architecture to extend Firebird features. One such plugin is External Engine (external engines). UDR mechanism (User Defined Routines - defined user subroutine) adds a layer on top of the engine interface of firebird external. UDRs have the following advantages over UDFs:

- you can write not only functions that return a scalar result, but also stored procedures (both executable and selective), as well as triggers;
- improved control of input and output parameters. In a number of cases (passing by descriptor) the types and other properties of the input parameters were not controlled at all, however, you could get these properties inside the UDF. UDRs provide a more uniform way of declaring input and output parameters, as is done with regular PSQL functions and procedures;
- the context of the current connection or transaction is available in the UDR, which allows you to perform some manipulations on the current database in this context;
- external procedures and functions (UDR) can be grouped in PSQL packages;
- UDRs can be written in any programming language (optional compiled into object codes). This requires an appropriate external engine plugin. For example, there are plugins for writing external modules in Java or any of the .NET languages.

In this guide, we will describe how to declare UDRs, their internal mechanisms, capabilities, and give examples of writing UDRs in Pascal. In addition, some aspects of using the new object-oriented API will be touched upon.

Further in the various chapters of this manual, when using the terms external procedure, function or trigger, we will mean exactly UDR, not UDF.

All our examples work on Delphi 2009 and older, as well as on Free Pascal. All examples can be compiled in both Delphi and Free Pascal unless otherwise noted.

Chapter 2. Firebird API

To write external procedures, functions or triggers on compiled programming languages, we need knowledge about the new object oriented Firebird API. This manual does not include a complete Firebird API description. You can find it in the documentation catalog distributed with Firebird (`doc/Using_OO_API.html`). For Russian-speaking users there is a translation of this document available at https://github.com/sim1984/fbooapi.

Included files for various programming languages that contain APIs are not distributed as part of the Firebird distribution for Windows, but you can extract them from compressed tarbar files distributed for Linux (path inside the archive `/opt/firebird/include/firebird/Firebird.pas`).

2.1. CLOOP

CLOOP - Cross Language Object Oriented Programming. This tool is not included with Firebird. It can be found in the source code https://github.com/FirebirdSQL/firebird/tree/B3_0_Release/extern/cloop. After the tool is built, you can generate an API for your programming language (`IdlFbInterfaces.h` or `Firebird.pas`) based on the interface description file `include/firebird/FirebirdInterface.idl`.

For Object Pascal this is done with the following command:

```
cloop FirebirdInterface.idl pascal Firebird.pas Firebird --uses SysUtils \
  --interfaceFile Pascal.interface.pas \
  --implementationFile Pascal.implementation.pas \
  --exceptionClass FbException --prefix I \
  --functionsFile fb_get_master_interface.pas
```

The files `Pascal.interface.pas`, `Pascal.implementation.pas` and `fb_get_master_interface.pas` can be found at https://github.com/FirebirdSQL/firebird/tree/B3_0_Release/src/misc/pascal.

Comment

In this case, for Firebird API interfaces will be added prefix I, as it is accepted in Object Pascal.

2.1.1. Constants

The resulting `Firebird.pas` file is missing `isc_*` constants. These constants for C/C++ languages can be found under https://github.com/FirebirdSQL/firebird/blob/B3_0_Release/src/include/consts_pub.h. To obtain constants for the Pascal language, we use the AWK script for syntax transformations. On Windows you will need to install Gawk for Windows or use the Windows Subsystem for Linux (available at Windows 10). This is done with the following command:

```
awk -f Pascal.Constants.awk consts_pub.h > const.pas
```

TThe contents of the resulting file must be copied into the empty const section of the `Firebird.pas` file immediately after implementation. The file `Pascal.Constants.awk` can be found at https://github.com/FirebirdSQL/firebird/tree/B3_0_Release/src/misc/pascal.

2.2. Life time management

Firebird interfaces are not based on the COM specification, so their lifetime is managed differently.

There are two interfaces in Firebird that deal with lifetime management: `IDisposable` and `IReferenceCounted`. The latter is especially active when creating other interfaces: `IPlugin` counts links, like many other interfaces used by plug-ins. These include interfaces that describe the database connection, transaction management, and SQL statements.

You don't always need the extra overhead of a reference-counted interface. For example, `IMaster`, the main interface that calls functions available to the rest of the API, has an unlimited lifetime by definition. For other APIs, the lifetime is strictly determined by the lifetime of the parent interface; interface `ISatus` is not multithreaded. For interfaces with limited lifetimes, it is useful to have an easy way to destroy them, i.e. the `dispose()` function.

Clue

If you don't know how an object is destroyed, look up its hierarchy if it has the `IReferenceCounted` interface. For reference-counted interfaces, upon completion of work with the object, it is necessary to decrement the reference count by calling the `release()` method.

Example 1. Important

Some methods of interfaces derived from `IReferenceCounted` release the interface after successful completion. There is no need to call `release()` after calling such methods.

This is done for historical reasons, because similar functions from the ISC API freed the corresponding handle.

Here is a list of such methods:

- `IAttachment` interface
  - `detach(status: IStatus)` - disconnect the connection to the database. On success, releases the interface.
  - `dropDatabase(status: IStatus)` - drop database. On success, releases the interface.
- Interface `ITransaction`
  - `commit(status: IStatus)` - transaction confirmation. On success, releases the interface.
  - `rollback(status: IStatus)` - transaction rollback. On success, releases the interface.
- `IStation` interface
  - `free(status: IStatus)` - removes a prepared statement. On success, releases the interface.

- `IResultSet` interface
  - `close(status: IStatus)` closes the cursor. On success, releases the interface.
- `IBlob` interface
  - `cancel(status: IStatus)` - cancels all changes made to the temporary BLOB (if any) and closes the BLOB. On success, releases the interface.
  - `close(status: IStatus)` - saves all changes made to the temporary BLOB (if any) and closes the BLOB. On success, releases the interface.
- Interface `IService`
  - `detach(status: IStatus)` - disconnect the connection with the service manager. On success, releases the interface.
- `IEvents` interface
  - `cancel(status: IStatus)` - cancels event subscription. On success, releases the interface.

Chapter 3. UDR announcements

UDRs can be added to or removed from the database using DDL commands, much like you add or remove normal PSQL procedures, functions, or triggers. In this case, instead of the body of the trigger, its location in the external module is specified using the `EXTERNAL NAME` clause.

Consider the syntax of this sentence, it will be common to external procedures, functions and triggers.

Syntax

```
EXTERNAL NAME '<extname>' ENGINE <engine>
[AS <extbody>]

<extname> ::= '<module name>!<routine name>[!<misc info>]'
```

The argument to this `EXTERNAL NAME` clause is a string indicating the location of the function in the external module. For plug-ins using the UDR engine, this line contains the name of the plug-in, the name of the function inside the plug-in, and user-defined information separated by a delimiter. An exclamation point is used as a separator (!).

The ENGINE clause specifies the name of the engine to handle the connection external modules. In Firebird, to work with external modules written in compiled languages (C, C++, Pascal) use the UDR engine. External functions written in Java require the Java engine.

After the `AS` keyword, a string literal can be specified - the "body" of the external module (procedure, function or trigger), it can be used by the external module for various purposes. For example, an SQL query may be specified to access an external database, or text in some language for interpretation by your function.

3.1. External functions

Syntax

```
{CREATE [OR ALTER] | RECREATE} FUNCTION funcname [(<inparam> [, <inparam> ...])]
RETURNS <type> [COLLATE collation] [DETERMINISTIC]
EXTERNAL NAME <extname> ENGINE <engine>
[AS <extbody>]


<inparam> ::= <param_decl> [{= | DEFAULT} <value>]

<value> ::=  {literal | NULL | context_var}

<param_decl> ::= paramname <type> [NOT NULL] [COLLATE collation]

<extname> ::= '<module name>!<routine name>[!<misc info>]'

<type> ::= <datatype> | [TYPE OF] domain | TYPE OF COLUMN rel.col
```

```
<datatype> ::=
    {SMALLINT | INT[EGER] | BIGINT}
  | BOOLEAN
  | {FLOAT | DOUBLE PRECISION}
  | {DATE | TIME | TIMESTAMP}
  | {DECIMAL | NUMERIC} [(precision [, scale])]
  | {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR} [(size)]
    [CHARACTER SET charset]
  | {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR} [VARYING] [(size)]
  | BLOB [SUB_TYPE {subtype_num | subtype_name}]
    [SEGMENT SIZE seglen] [CHARACTER SET charset]
  | BLOB [(seglen [, subtype_num])]
```

All parameters of an external function can be changed using the `ALTER statement FUNCTION`.

Syntax

```
ALTER FUNCTION funcname [(<inparam> [, <inparam> ...])]
RETURNS <type> [COLLATE collation] [DETERMINISTIC]
EXTERNAL NAME <extname> ENGINE <engine>
[AS <extbody>]

<extname> ::= '<module name>!<routine name>[!<misc info>]'
```

You can remove an external function using the DROP FUNCTION statement.

Syntax

```
DROP FUNCTION funcname
```

Table 1. Some parameters of the external function

Parameter	**Description**
funcname	Name of the stored function. Can contain up to 31 bytes.
inparam	Description of the input parameter.
module name	Name of the external module where the function resides.
routine name	The internal name of the function inside the external module.
misc info	User-defined information to pass to the function external module.
engine	Name of the engine to use external functions. Usually specifies the name of the UDR.
extbody	External function body. A string literal that can be used by UDR for various purposes.

Here we will not describe the syntax of the input parameters and the output result. It fully corresponds to the syntax for regular PSQL functions, which is described in detail in the SQL

Language Manual. Instead, we give examples of declaring external functions with explanations.

```
create function sum_args (
    n1 integer,
    n2 integer,
    n3 integer
)
returns integer
external name 'udrcpp_example!sum_args'
engine udr;
```

The implementation of the function is in the `udrcpp_example` module. Within this module, the function is registered under the name `sum_args`. The UDR engine is used to operate the external function.

```
create or alter function regex_replace (
  regex varchar(60),
  str varchar(60),
  replacement varchar(60)
)
returns varchar(60)
external name 'org.firebirdsql.fbjava.examples.fbjava_example.FbRegex.replace(
      String, String, String)'
engine java;
```

The implementation of the function is in the `udrcpp_example` module. Within this module, the function is registered under the name `sum_args`. The UDR engine is used to operate the external function.

3.2. External Procedures

Syntax

```
{CREATE [OR ALTER] | RECREATE} PROCEDURE procname [(<inparam> [, <inparam> ...])]
RETURNS (<outparam> [, <outparam> ...])
EXTERNAL NAME <extname> ENGINE <engine>
[AS <extbody>]

<inparam> ::= <param_decl> [{= | DEFAULT} <value>]

<outparam>  ::=  <param_decl>

<value> ::=  {literal | NULL | context_var}

<param_decl> ::= paramname <type> [NOT NULL] [COLLATE collation]

<extname> ::= '<module name>!<routine name>[!<misc info>]'
```

```
<type> ::= <datatype> | [TYPE OF] domain | TYPE OF COLUMN rel.col

<datatype> ::=
    {SMALLINT | INT[EGER] | BIGINT}
  | BOOLEAN
  | {FLOAT | DOUBLE PRECISION}
  | {DATE | TIME | TIMESTAMP}
  | {DECIMAL | NUMERIC} [(precision [, scale])]
  | {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR} [(size)]
    [CHARACTER SET charset]
  | {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR} [VARYING] [(size)]
  | BLOB [SUB_TYPE {subtype_num | subtype_name}]
    [SEGMENT SIZE seglen] [CHARACTER SET charset]
  | BLOB [(seglen [, subtype_num])]
```

All parameters of an external procedure can be changed using the `ALTER PROCEDURE` statement.

Syntax

```
ALTER PROCEDURE procname [(<inparam> [, <inparam> ...])]
RETURNS (<outparam> [, <outparam> ...])
EXTERNAL NAME <extname> ENGINE <engine>
[AS <extbody>]
```

You can drop an external procedure using the `DROP PROCEDURE` statement.

Syntax

```
DROP PROCEDURE procname
```

Table 2. Some parameters of the external procedure

Parameter	**Description**
procname	Name of the stored procedure. Can contain up to 31 bytes.
inparam	Description of the input parameter.
outparam	Description of the output parameter.
module name	The name of the external module in which the procedure resides.
routine name	Internal name of the procedure inside the external module.
misc info	User-defined information to pass to external module procedure.
engine	Name of the engine to use external procedures. Usually specifies the name of the UDR.
extbody	The body of the external procedure. A string literal that can be used by UDR for various purposes.

Here we will not describe the syntax of input and output parameters. It is fully consistent with the syntax for regular PSQL procedures, which is described in detail in the SQL Language Manual.

Instead, let's take examples of declaration of external procedures with explanations.

```
create procedure gen_rows_pascal (
    start_n integer not null,
    end_n integer not null
)
returns (
    result integer not null
)
external name 'pascaludr!gen_rows'
engine udr;
```

The implementation of the function is in the `pascaludr` module. Within this module, the procedure is registered under the name `gen_rows`. The UDR engine i is used to run the external procedure.

```
create or alter procedure write_log (
  message varchar(100)
)
external name 'pascaludr!write_log'
engine udr;
```

The implementation of the function is in the `pascaludr` module. Within this module, the procedure is registered under the name `write_log`. The UDR engine is used to run the external procedure.

```
create or alter procedure employee_pgsql (
  -- Firebird 3.0.0 has a bug with external procedures without parameters
  dummy integer = 1
)
returns (
  id type of column employee.id,
  name type of column employee.name
)
external name 'org.firebirdsql.fbjava.examples.fbjava_example.FbJdbc
    .executeQuery()!jdbc:postgresql:employee|postgres|postgres'
engine java
as 'select * from employee';
```

The implementation of the function is in the static function executeQuery of the class `org.firebirdsql.fbjava.examples.fbjava_example.FbJdbc`. After exclamation mark "!" contains information for connecting to an external database via JDBC. The Java engine is used to run the external function. Here, as the "body" of the external procedure, an SQL query is passed to retrieve data.

Comment

This procedure uses a stub that passes unused parameter. This is due to the fact that in Firebird 3.0 there is a bug with the processing of external procedures

without parameters.

3.3. Placing External Procedures and Functions Inside Packages

A group of related procedures and functions is conveniently placed in PSQL packages. The packages can contain both external and conventional psql procedures and functions.

Syntax

```
{CREATE [OR ALTER] | RECREATE} PACKAGE package_name
AS
BEGIN
  [<package_item> ...]
END

{CREATE | RECREATE} PACKAGE BODY package_name
AS
BEGIN
  [<package_item> ...]
  [<package_body_item> ...]
END

<package_item> ::=
    <function_decl>;
  | <procedure_decl>;

<function_decl> ::=
  FUNCTION func_name [(<in_params>)]
  RETURNS <type> [COLLATE collation]
  [DETERMINISTIC]

<procedure_decl> ::=
  PROCEDURE proc_name [(<in_params>)]
  [RETURNS (<out_params>)]

<package_body_item> ::=
    <function_impl>
  | <procedure_impl>

<function_impl> ::=
  FUNCTION func_name [(<in_impl_params>)]
  RETURNS <type> [COLLATE collation]
  [DETERMINISTIC]
  <routine body>

<procedure_impl> ::=
  PROCEDURE proc_name [(<in_impl_params>)]
  [RETURNS (<out_params>)]
```

```
  <routine body>

<routine body> ::= <sql routine body> | <external body reference>

<sql routine body> ::=
  AS
    [<declarations>]
  BEGIN
    [<PSQL_statements>]
  END

<declarations> ::= <declare_item> [<declare_item> ...]

<declare_item> ::=
    <declare_var>;
  | <declare_cursor>;
  | <subroutine declaration>;
  | <subroutine implimentation>

<subroutine declaration> ::= <subfunc_decl> | <subproc_decl>

<subroutine implimentation> ::= <subfunc_impl> | <subproc_impl>

<external body reference> ::=
  EXTERNAL NAME <extname> ENGINE <engine> [AS <extbody>]

<extname> ::= '<module name>!<routine name>[!<misc info>]'
```

For external procedures and functions, the package header specifies the name, input parameters, their types, default values, and output parameters, and in the body of the package everything is the same, except for the default values, as well as the location in the external module (clause `EXTERNAL NAME`), the name of the engine, and possibly the "body" of the procedure/function.

Let's say you wrote a UDR to work with regular expressions, which is located in an external module (dynamic library) PCRE, and you have several other UDRs that perform other tasks. If we did not use PSQL packages, then all our external procedures and would be intermingled both with each other and with regular PSQL procedures and functions. This makes it difficult to find dependencies and make changes to external modules, and also creates confusion, and forces at least the use of prefixes to group procedures and functions. PSQL packages make this task much easier for us.

```
SET TERM ^;

CREATE OR ALTER PACKAGE REGEXP
AS
BEGIN
  PROCEDURE preg_match(
      APattern VARCHAR(8192), ASubject VARCHAR(8192))
    RETURNS (Matches VARCHAR(8192));
```

```
  FUNCTION preg_is_match(
      APattern VARCHAR(8192), ASubject VARCHAR(8192))
    RETURNS BOOLEAN;

  FUNCTION preg_replace(
      APattern VARCHAR(8192),
      AReplacement VARCHAR(8192),
      ASubject VARCHAR(8192))
    RETURNS VARCHAR(8192);

  PROCEDURE preg_split(
      APattern VARCHAR(8192),
      ASubject VARCHAR(8192))
    RETURNS (Lines VARCHAR(8192));

  FUNCTION preg_quote(
      AStr VARCHAR(8192),
      ADelimiter CHAR(10) DEFAULT NULL)
    RETURNS VARCHAR(8192);
END^

RECREATE PACKAGE BODY REGEXP
AS
BEGIN
  PROCEDURE preg_match(
      APattern VARCHAR(8192),
      ASubject VARCHAR(8192))
    RETURNS (Matches VARCHAR(8192))
    EXTERNAL NAME 'PCRE!preg_match' ENGINE UDR;

  FUNCTION preg_is_match(
      APattern VARCHAR(8192),
      ASubject VARCHAR(8192))
    RETURNS BOOLEAN
  AS
  BEGIN
    RETURN EXISTS(
      SELECT * FROM preg_match(:APattern, :ASubject));
  END

  FUNCTION preg_replace(
      APattern VARCHAR(8192),
      AReplacement VARCHAR(8192),
      ASubject VARCHAR(8192))
    RETURNS VARCHAR(8192)
    EXTERNAL NAME 'PCRE!preg_replace' ENGINE UDR;

  PROCEDURE preg_split(
      APattern VARCHAR(8192),
      ASubject VARCHAR(8192))
    RETURNS (Lines VARCHAR(8192))
```

```
    EXTERNAL NAME 'PCRE!preg_split' ENGINE UDR;

  FUNCTION preg_quote(
      AStr VARCHAR(8192),
      ADelimiter CHAR(10))
    RETURNS VARCHAR(8192)
    EXTERNAL NAME 'PCRE!preg_quote' ENGINE UDR;
END^

SET TERM ;^
```

3.4. External triggers

Syntax

```
{CREATE [OR ALTER] | RECREATE} TRIGGER trigname
{
    <relation_trigger_legacy>
  | <relation_trigger_sql2003>
  | <database_trigger>
  | <ddl_trigger>
}
<external-body>

<external-body> ::=
  EXTERNAL NAME <extname> ENGINE <engine>
  [AS <extbody>]

<relation_trigger_legacy> ::=
  FOR {tablename | viewname}
  [ACTIVE | INACTIVE]
  {BEFORE | AFTER} <mutation_list>
  [POSITION number]

<relation_trigger_sql2003> ::=
  [ACTIVE | INACTIVE]
  {BEFORE | AFTER} <mutation_list>
  [POSITION number]
  ON {tablename | viewname}

<database_trigger> ::=
  [ACTIVE | INACTIVE]
  ON db_event
  [POSITION number]

<ddl_trigger> ::=
  [ACTIVE | INACTIVE]
  {BEFORE | AFTER} <ddl_events>
  [POSITION number]
```

```
<mutation_list> ::= <mutation> [OR <mutation> [OR <mutation>]]

<mutation> ::= INSERT | UPDATE | DELETE

<db_event> ::=
    CONNECT
  | DISCONNECT
  | TRANSACTION START
  | TRANSACTION COMMIT
  | TRANSACTION ROLLBACK


<ddl_events> ::=
    ANY DDL STATEMENT
  | <ddl_event_item> [{OR <ddl_event_item>} ...]

<ddl_event_item> ::=
    CREATE TABLE | ALTER TABLE | DROP TABLE
  | CREATE PROCEDURE | ALTER PROCEDURE | DROP PROCEDURE
  | CREATE FUNCTION | ALTER FUNCTION | DROP FUNCTION
  | CREATE TRIGGER | ALTER TRIGGER | DROP TRIGGER
  | CREATE EXCEPTION | ALTER EXCEPTION | DROP EXCEPTION
  | CREATE VIEW | ALTER VIEW | DROP VIEW
  | CREATE DOMAIN | ALTER DOMAIN | DROP DOMAIN
  | CREATE ROLE | ALTER ROLE | DROP ROLE
  | CREATE SEQUENCE | ALTER SEQUENCE | DROP SEQUENCE
  | CREATE USER | ALTER USER | DROP USER
  | CREATE INDEX | ALTER INDEX | DROP INDEX
  | CREATE COLLATION | DROP COLLATION
  | ALTER CHARACTER SET
  | CREATE PACKAGE | ALTER PACKAGE | DROP PACKAGE
  | CREATE PACKAGE BODY | DROP PACKAGE BODY
  | CREATE MAPPING | ALTER MAPPING | DROP MAPPING
```

An external trigger can be changed with the `ALTER TRIGGER` statement.

Syntax

```
ALTER TRIGGER trigname {
[ACTIVE | INACTIVE]
[
    {BEFORE | AFTER} {<mutation_list> | <ddl_events>}
  | ON db_event
]
[POSITION number]
[<external-body>]

<external-body> ::=
  EXTERNAL NAME <extname> ENGINE <engine>
  [AS <extbody>]
```

```
<extname> ::= '<module name>!<routine name>[!<misc info>]'

<mutation_list> ::= <mutation> [OR <mutation> [OR <mutation>]]

<mutation> ::= { INSERT | UPDATE | DELETE }
```

You can remove an external trigger using the `DROP TRIGGER` statement.

Syntax

```
DROP TRIGGER trigname
```

Table 3. Some external trigger parameters

Parameter	Description
trigname	Trigger name. Can contain up to 31 bytes.
relation_trigger_legacy	Table trigger declaration (inherited).
relation_trigger_sql2003	Table trigger declaration according to SQL-2003 standard.
database_trigger	Declaration of a database trigger.
ddl_trigger	DDL trigger declaration.
tablename	Table name.
viewname	The name of the view.
mutation_list	List of table events.
mutation	One of the table events.
db_event	Connection or transaction event.
ddl_events	List of metadata change events.
ddl_event_item	One of the metadata change events.
number	The order in which the trigger fires. From 0 to 32767.
extbody	External trigger body. A string literal that can be used by UDR for various purposes.
module name	Name of the external module where the trigger is located.
routine name	Internal name of the trigger inside the external module.
misc info	User-defined information to pass to the trigger external module.
engine	Name of the engine to use external triggers. Usually specifies the name of the UDR.

Here are examples of declaring external triggers with explanations.

```
create database 'c:\temp\slave.fdb';

create table persons (
```

```
    id integer not null,
    name varchar(60) not null,
    address varchar(60),
    info blob sub_type text
);

commit;

create database 'c:\temp\master.fdb';

create table persons (
    id integer not null,
    name varchar(60) not null,
    address varchar(60),
    info blob sub_type text
);

create table replicate_config (
    name varchar(31) not null,
    data_source varchar(255) not null
);

insert into replicate_config (name, data_source)
   values ('ds1', 'c:\temp\slave.fdb');

create trigger persons_replicate
after insert on persons
external name 'udrcpp_example!replicate!ds1'
engine udr;
```

The trigger implementation is in the `udrcpp_example` module. Within this module, the trigger is registered under the name `replicate`. The UDR engine is used to operate the external trigger.

The link to the external module uses an additional parameter `ds1`, according to which, inside the external trigger, the configuration for connecting to the external database is read from the *replicate_config* table.

Chapter 4. Structure UDR

We will describe the UDR structure in Pascal. To explain the minimum structure for constructing a UDR, we will use the standard examples from `examples/udr/` translated into Pascal.

Create a new dynamic library project, which we will call MyUdr. The result should be a `MyUdr.dpr` file (if you created the project in Delphi) or a `MyUdr.lpr` file (if you created the project in Lazarus). Now let's change the main project file so that it looks like this:

```
library MyUdr;

{$IFDEF FPC}
  {$MODE DELPHI}{$H+}
{$ENDIF}

uses
{$IFDEF unix}
    cthreads,
    // the c memory manager is on some systems much faster for multi-threading
    cmem,
{$ENDIF}
  UdrInit in 'UdrInit.pas',
  SumArgsFunc in 'SumArgsFunc.pas';

exports firebird_udr_plugin;

end.
```

In this case, only one `firebird_udr_plugin` function needs to be exported, which is the entry point for the UDR plug-in plugin. The implementation of this function will be in the `UdrInit` module.

Comment

If you are developing your UDR in Free Pascal, then you will need additional directives. The `{$mode objfpc}` directive is required to enable Object Pascal mode. Instead, you can use the `{$mode delphi}` directive to ensure compatibility with Delphi. Because my examples should compile successfully in both FPC and Delphi, I choose `{$mode delphi}`.

The `{$H+}` directive enables support for long strings. This is necessary if you use the `string`, `ansistring` types, and not just the null-terminated strings `PChar`, `PAnsiChar`, `PWideChar`.

In addition, we will need to include separate modules to support multithreading on Linux and other Unix-like operating systems.

4.1. Registering Functions

Now let's add the `UdrInit` module, it should look like this:

```
unit UdrInit;

{$IFDEF FPC}
  {$MODE DELPHI}{$H+}
{$ENDIF}

interface

uses
  Firebird;

// entry point for the External Engine of the UDR module
function firebird_udr_plugin(AStatus: IStatus; AUnloadFlagLocal: BooleanPtr;
  AUdrPlugin: IUdrPlugin): BooleanPtr; cdecl;

implementation

uses
  SumArgsFunc;

var
  myUnloadFlag: Boolean;
  theirUnloadFlag: BooleanPtr;

function firebird_udr_plugin(AStatus: IStatus; AUnloadFlagLocal: BooleanPtr;
  AUdrPlugin: IUdrPlugin): BooleanPtr; cdecl;
begin
  // register our functions
  AUdrPlugin.registerFunction(AStatus, 'sum_args',
    TSumArgsFunctionFactory.Create());
  // register our procedures
  //AUdrPlugin.registerProcedure(AStatus, 'sum_args_proc',
  //  TSumArgsProcedureFactory.Create());
  //AUdrPlugin.registerProcedure(AStatus, 'gen_rows', TGenRowsFactory.Create());
  // registering our triggers
  //AUdrPlugin.registerTrigger(AStatus, 'test_trigger',
  //  TMyTriggerFactory.Create());

  theirUnloadFlag := AUnloadFlagLocal;
  Result := @myUnloadFlag;
end;

initialization

myUnloadFlag := false;
```

```
finalization

if ((theirUnloadFlag <> nil) and not myUnloadFlag) then
  theirUnloadFlag^ := true;

end.
```

In the `firebird_udr_plugin` function, we need to register the factories of our external procedures, functions, and triggers. For each function, procedure or trigger, you must write your own factory. This is done using the methods of the `IUdrPlugin` interface:

- `registerFunction` - registers an external function;
- `registerProcedure` - registers an external procedure;
- `registerTrigger` - registers an external trigger.

The first argument to these functions is a pointer to a status vector, followed by the internal name of the function (procedure or trigger). The internal name will be used when creating procedure/function/trigger in SQL. The third argument is a factory instance for creating a function (procedure or trigger).

4.2. Function factory

Now we need to write the factory and the function itself. They will be located in the `SumArgsFunc` module. Examples for writing procedures and triggers would be presented later.

```
unit SumArgsFunc;

{$IFDEF FPC}
{$MODE DELPHI}{$H+}
{$ENDIF}

interface

uses
  Firebird;

{ ******************************************************
    create function sum_args (
      n1 integer,
      n2 integer,
      n3 integer
    ) returns integer
    external name 'myudr!sum_args'
    engine udr;
 ****************************************************** }

type
  // the structure to which the input message will be mapped
```

```
TSumArgsInMsg = record
  n1: Integer;
  n1Null: WordBool;
  n2: Integer;
  n2Null: WordBool;
  n3: Integer;
  n3Null: WordBool;
end;
PSumArgsInMsg = ^TSumArgsInMsg;

// the structure to which the output message will be mapped
TSumArgsOutMsg = record
  result: Integer;
  resultNull: WordBool;
end;
PSumArgsOutMsg = ^TSumArgsOutMsg;

// Factory for instantiating the external function TSumArgsFunction
TSumArgsFunctionFactory = class(IUdrFunctionFactoryImpl)
  // Called when the factory is destroyed
  procedure dispose(); override;

  { Executed each time an external function is loaded into the metadata cache.
     Used to change the format of the input and output messages.

    @param(AStatus status vector)
    @param(AContext External function execution context)
    @param(AMetadata External Function Metadata)
    @param(AInBuilder Message builder for input metadata)
    @param(AOutBuilder Message builder for output metadata)
  }
  procedure setup(AStatus: IStatus; AContext: IExternalContext;
    AMetadata: IRoutineMetadata; AInBuilder: IMetadataBuilder;
    AOutBuilder: IMetadataBuilder); override;

  { Creating a new external function instance TSumArgsFunction

    @param(AStatus status vector)
    @param(AContext External function execution context)
    @param(AMetadata External Function Metadata)
    @returns(Экземпляр external function)
  }
  function newItem(AStatus: IStatus; AContext: IExternalContext;
    AMetadata: IRoutineMetadata): IExternalFunction; override;
end;

// External function TSumArgsFunction.
TSumArgsFunction = class(IExternalFunctionImpl)
  // Called when the function instance is destroyed
  procedure dispose(); override;
```

```
    { This method is called just before execute and tells
       kernel our requested character set to exchange data internally
       this method. During this call, the context uses the character set
       obtained from ExternalEngine::getCharSet.

      @param(AStatus Status vector)
      @param(AContext External function execution context)
      @param(AName Character set name)
      @param(AName Character set name length)
    }
    procedure getCharSet(AStatus: IStatus; AContext: IExternalContext;
      AName: PAnsiChar; ANameSize: Cardinal); override;

    { Executing an external function

      @param(AStatus Status vector)
      @param(AContext External function execution context)
      @param(AInMsg Pointer to input message)
      @param(AOutMsg Pointer to output message)
    }
    procedure execute(AStatus: IStatus; AContext: IExternalContext;
      AInMsg: Pointer; AOutMsg: Pointer); override;
  end;

implementation

{ TSumArgsFunctionFactory }

procedure TSumArgsFunctionFactory.dispose;
begin
  Destroy;
end;

function TSumArgsFunctionFactory.newItem(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata): IExternalFunction;
begin
  Result := TSumArgsFunction.Create();
end;

procedure TSumArgsFunctionFactory.setup(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata;
  AInBuilder, AOutBuilder: IMetadataBuilder);
begin

end;

{ TSumArgsFunction }

procedure TSumArgsFunction.dispose;
begin
  Destroy;
```

```
end;

procedure TSumArgsFunction.execute(AStatus: IStatus; AContext: IExternalContext;
  AInMsg, AOutMsg: Pointer);
var
  xInput: PSumArgsInMsg;
  xOutput: PSumArgsOutMsg;
begin
  // convert pointers to input and output to typed ones
  xInput := PSumArgsInMsg(AInMsg);
  xOutput := PSumArgsOutMsg(AOutMsg);
  // by default, the output argument is NULL, so set it to nullFlag
  xOutput^.resultNull := True;
  // if one of the arguments is NULL, then the result is NULL
  // otherwise, we calculate the sum of the arguments
  with xInput^ do
  begin
    if not (n1Null or n2Null or n3Null) then
    begin
      xOutput^.result := n1 + n2 + n3;
      // if there is a result, then reset the NULL flag
      xOutput^.resultNull := False;
    end;
  end;
end;

procedure TSumArgsFunction.getCharSet(AStatus: IStatus;
  AContext: IExternalContext; AName: PAnsiChar; ANameSize: Cardinal);
begin
end;

end.
```

The external function factory must implement the interface `IUdrFunctionFactory`. To simplify, we simply inherit the class `IUdrFunctionFactoryImpl`. Each external function needs its own factory. However, if factories do not have specifics for creating some function, then you can write a generic factory using generics. Later we will give an example of how to do this.

The `dispose` method is called when the factory is destroyed, in which we must release the previously allocated resources. In this case, we simply call the destructor.

The setup method is executed each time an external function is loaded into the metadata cache. In it, you can do various actions that are necessary before creating an instance of a function, for example, change the format for input and output messages. We'll talk about it in more detail later.

The `newItem` method is called to instantiate the external function. This method is passed a pointer to the status vector, the context of the external function, and the metadata of the external function. With `IRoutineMetadata` you can get the format of the input and output message, the body of the external function, and other metadata. In this method, you can create different instances of an external function depending on its declaration in PSQL. Metadata can be passed to the created

external function instance if needed. In our case, we simply create an instance of an external function `TSumArgsFunction`.

4.3. Function instance

An external function must implement the `IExternalFunction` interface. To simplify, we simply inherit the `IExternalFunctionImpl` class.

The `dispose` method is called when the function instance is destroyed, in which we must release the previously allocated resources. In this case, we simply call the destructor.

The `getCharSet` method is used to tell the external function context the character set we want to use when working with the connection from the current context. By default, the connection from the current context works in the encoding of the current connection, which is not always convenient.

The `execute` method handles the function call itself. This method is passed a pointer to the status vector, a pointer to the context of the external function, pointers to the input and output messages.

We may need the context of an external function to get the context of the current connection or transaction. Even if you do not use database queries in the current connection, you may still need these contexts, especially when working with the BLOB type. Examples working with the BLOB type, as well as the use of connection and transaction contexts will be shown later.

The input and output messages have a fixed width, which depends on the data types declared for the input and output variables, respectively. This allows typed pointers to fixed-width structures whose members must match the data types. The example shows that for each variable in the structure, a member of the corresponding type is indicated, after which there is a member that is a sign of a special NULL value (hereinafter referred to as the Null flag). In addition to working with buffers of input and output messages through structures, there is another way using address arithmetic on pointers using offsets, the values of which can be obtained from the `IMessageMetadata` interface. We'll talk more about working with messages later, but now we'll just explain what was done in the execute method.

First of all, we convert untyped pointers to typed ones. For the output value, set the Null flag to `True` (this is necessary for the function to return `NULL` if one of the input arguments is `NULL`). Then we check the Null flags of all input arguments, if none of the input arguments is equal to `NULL`, then the output value will be equal to the sum of the argument values. It is important to remember to reset the Null flag of the output argument to `False`.

4.4. Registration of procedures

It's time to add a stored procedure to our UDR module. As you know, there are two types of stored procedures: executable stored procedures and stored procedures for retrieving data. First, let's add an executable stored procedure, i.e. a stored procedure that can be called with the `EXECUTE PROCEDURE` statement and can return at most one record.

Go back to the `UdrInit` module and change the `firebird_udr_plugin` function to look like this.

```
function firebird_udr_plugin(AStatus: IStatus; AUnloadFlagLocal: BooleanPtr;
  AUdrPlugin: IUdrPlugin): BooleanPtr; cdecl;
begin
  // register our functions
  AUdrPlugin.registerFunction(AStatus, 'sum_args',
    TSumArgsFunctionFactory.Create());
  // register our procedures
  AUdrPlugin.registerProcedure(AStatus, 'sum_args_proc',
    TSumArgsProcedureFactory.Create());
  //AUdrPlugin.registerProcedure(AStatus, 'gen_rows', TGenRowsFactory.Create());
  // register our triggers
  //AUdrPlugin.registerTrigger(AStatus, 'test_trigger',
  // TMyTriggerFactory.Create());

  theirUnloadFlag := AUnloadFlagLocal;
  Result := @myUnloadFlag;
end;
```

Comment

Do not forget to add `uses` module `SumArgsProc` to the list our procedure is located.

4.5. Factory of procedures

The factory of the external procedure should implement the interface `IUdrProcedureFactory`. To simplify, we just inherit the class `IUdrProcedureFactoryImpl`. Each external procedure needs its own factory. However, if factories have no specifics to create some procedures, you can write a generalized factory using generics. Later we will give an example of how to do this.

The `dispose` method is called when the factory is destroyed, in it we must free previously allocated resources. In this case, we simply call Destructor.

The `setup` method is performed each time when loading the external procedure in cache metadata. In it you can make various actions that are necessary Before creating a copy of the procedure, for example, a change in format for input and output messages. Let's talk about him in more detail later.

The `Newitem` method is caused to create a copy of the external procedure. IN This method is transmitted to the indicator to the status of the vector, the context of the external Procedures and metadata external procedure. Using `IRoutineMetadata` you can get the input and output format, the body of the external functions and other metadata. In this method you can create various copies of the external function depending on its ad in PSQL. Metadata can be transferred to the created copy of the external procedure if it's necessary. In our case, we simply create a copy of the external `TSumArgsProcedure` procedures.

The factory of the procedure, as well as the very procedure in the module `SumArgsProc`.

```
unit SumArgsProc;
```

```
{$IFDEF FPC}
{$MODE DELPHI}{$H+}
{$ENDIF}

interface

uses
  Firebird;

  { ************************************************************

    create procedure sp_sum_args (
      n1 integer,
      n2 integer,
      n3 integer
    ) returns (result integer)
    external name 'myudr!sum_args_proc'
    engine udr;

    ************************************************************ }
type
  // The structure of which the input message will be displayed
  TSumArgsInMsg = record
    n1: Integer;
    n1Null: WordBool;
    n2: Integer;
    n2Null: WordBool;
    n3: Integer;
    n3Null: WordBool;
  end;
  PSumArgsInMsg = ^TSumArgsInMsg;

  // The structure for which the output will be displayed
  TSumArgsOutMsg = record
    result: Integer;
    resultNull: WordBool;
  end;
  PSumArgsOutMsg = ^TSumArgsOutMsg;

  // Factory to create a copy of the external TSUMARGSPROCEDURE procedure
  TSumArgsProcedureFactory = class(IUdrProcedureFactoryImpl)
    // Called when the factory is destroyed
    procedure dispose(); override;

    { It is performed each time when loading the external procedure in the cache of
metadata
       Used to change the input and output format.

      @param(AStatus Status vector)
      @param(AContext The context of the external procedure)
```

```
      @param(AMetadata Metadata of the external procedure)
      @param(AInBuilder Message builder for input metadata)
      @param(AOutBuilder Message builder for weekend metadata)
    }
    procedure setup(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata; AInBuilder: IMetadataBuilder;
      AOutBuilder: IMetadataBuilder); override;

    { Creating a new copy of the external procedure TSumArgsProcedure

      @param(AStatus Status vector)
      @param(AContext The context of the external procedure)
      @param(AMetadata Metadata of the external procedure)
      @returns(Экземпляр external procedure)
    }
    function newItem(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata): IExternalProcedure; override;
  end;

  TSumArgsProcedure = class(IExternalProcedureImpl)
  public
    // Called when destroying a copy of the procedure
    procedure dispose(); override;

    { This method is called just before open and tells the kernel
      our requested character set to communicate within this
      method. During this call, the context uses the character set
      obtained from ExternalEngine::getCharSet.

      @param(AStatus Status vector)
      @param(AContext The context of external function)
      @param(AName The name of the set of characters)
      @param(AName The length of the name of the set of characters)
    }
    procedure getCharSet(AStatus: IStatus; AContext: IExternalContext;
      AName: PAnsiChar; ANameSize: Cardinal); override;

    { External procedure

      @param(AStatus Status vector)
      @param(AContext The context of external function)
      @param(AInMsg Input message pointer)
      @param(AOutMsg Output indicator)
      @returns(Data set for a selective procedure or
                Nil for the procedures)
    }
    function open(AStatus: IStatus; AContext: IExternalContext; AInMsg: Pointer;
      AOutMsg: Pointer): IExternalResultSet; override;
  end;

implementation
```

```
{ TSumArgsProcedureFactory }

procedure TSumArgsProcedureFactory.dispose;
begin
  Destroy;
end;

function TSumArgsProcedureFactory.newItem(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata): IExternalProcedure;
begin
  Result := TSumArgsProcedure.create;
end;

procedure TSumArgsProcedureFactory.setup(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata; AInBuilder,
  AOutBuilder: IMetadataBuilder);
begin

end;

{ TSumArgsProcedure }

procedure TSumArgsProcedure.dispose;
begin
  Destroy;
end;

procedure TSumArgsProcedure.getCharSet(AStatus: IStatus;
  AContext: IExternalContext; AName: PAnsiChar; ANameSize: Cardinal);
begin

end;

function TSumArgsProcedure.open(AStatus: IStatus; AContext: IExternalContext;
  AInMsg, AOutMsg: Pointer): IExternalResultSet;
var
  xInput: PSumArgsInMsg;
  xOutput: PSumArgsOutMsg;
begin
  // The set of data for the procedures performed is not necessary
  Result := nil;
  // We convert the signs to the input and access to the typized
  xInput := PSumArgsInMsg(AInMsg);
  xOutput := PSumArgsOutMsg(AOutMsg);
  // By default, the output argument = NULL, and therefore we expose him nullflag
  xOutput^.resultNull := True;
  // If one of the arguments NULL means the result NULL
  // Otherwise, we consider the amount of arguments
  with xInput^ do
  begin
```

```
    if not (n1Null or n2Null or n3Null) then
    begin
      xOutput^.result := n1 + n2 + n3;
      // since there is a result, then drop the NULL flag
      xOutput^.resultNull := False;
    end;
  end;
end;

end.
```

4.6. Procedure instance

An external procedure must implement the `IExternalProcedure` interface. To simplify, we simply inherit the `IExternalProcedureImpl` class.

The `dispose` method is called when the procedure instance is destroyed, in which we must release the previously allocated resources. In this case, we simply call the destructor.

The `getCharSet` method is used to tell the outer procedure context the character set we want to use when working with the connection from the current context. By default, the connection from the current context works in the encoding of the current connection, which is not always convenient.

The `open` method directly handles the procedure call itself. This method is passed a pointer to the status vector, a pointer to the context of the external procedure, pointers to the input and output messages. If you have an executable procedure, then the method must return `nil`, otherwise it must return an instance of the output set for the procedure. In this case, we don't need to instantiate the dataset. We just transfer the logic from the `TSumArgsFunction.execute` method.

4.7. Store a choice procedure

Now let's add a simple selection procedure to our UDR module. To do this, we will change the registration function `firebird_udr_plugin`.

```
function firebird_udr_plugin(AStatus: IStatus; AUnloadFlagLocal: BooleanPtr;
  AUdrPlugin: IUdrPlugin): BooleanPtr; cdecl;
begin
  // We register our functions
  AUdrPlugin.registerFunction(AStatus, 'sum_args',
    TSumArgsFunctionFactory.Create());
  // We register our procedures
  AUdrPlugin.registerProcedure(AStatus, 'sum_args_proc',
    TSumArgsProcedureFactory.Create());
  AUdrPlugin.registerProcedure(AStatus, 'gen_rows', TGenRowsFactory.Create());
  // We register our triggers
  //AUdrPlugin.registerTrigger(AStatus, 'test_trigger',
  //  TMyTriggerFactory.Create());
```

```
  theirUnloadFlag := AUnloadFlagLocal;
  Result := @myUnloadFlag;
end;
```

Comment

Don't forget to add the `GenRowsProc` module to the `uses` list, which will contain our procedure is located.

The procedure factory is completely identical as for the case with an executable stored procedure. The procedure instance methods are also identical, with the exception of the `open` method, which we will analyze in a little more detail.

```
unit GenRowsProc;

{$IFDEF FPC}
{$MODE DELPHI}{$H+}
{$ENDIF}

interface

uses
  Firebird, SysUtils;

type
  { *********************************************************

    create procedure gen_rows (
      start  integer,
      finish integer
    ) returns (n integer)
    external name 'myudr!gen_rows'
    engine udr;

    ********************************************************* }

  TInput = record
    start: Integer;
    startNull: WordBool;
    finish: Integer;
    finishNull: WordBool;
  end;
  PInput = ^TInput;

  TOutput = record
    n: Integer;
    nNull: WordBool;
  end;
  POutput = ^TOutput;
```

```
// Factory for creating an instance of the external procedure TGenRowsProcedure
 TGenRowsFactory = class(IUdrProcedureFactoryImpl)
   // Called when the factory is destroyed
   procedure dispose(); override;

   { Executed each time an external function is loaded into the metadata cache.
     Used to change the format of the input and output messages.

     @param(AStatus Status vector)
     @param(AContext External function execution context)
     @param(AMetadata External function metadata)
     @param(AInBuilder Message builder for input metadata)
     @param(AOutBuilder Message builder for output metadata)
   }
   procedure setup(AStatus: IStatus; AContext: IExternalContext;
     AMetadata: IRoutineMetadata; AInBuilder: IMetadataBuilder;
     AOutBuilder: IMetadataBuilder); override;

   { Create a new instance of the external procedure TGenRowsProcedure

     @param(AStatus Status vector)
     @param(AContext External function execution context)
     @param(AMetadata External function metadata)
     @returns(External function instance)
   }
   function newItem(AStatus: IStatus; AContext: IExternalContext;
     AMetadata: IRoutineMetadata): IExternalProcedure; override;
 end;

 // External procedure TGenRowsProcedure.
 TGenRowsProcedure = class(IExternalProcedureImpl)
 public
   // Called when the procedure instance is destroyed
   procedure dispose(); override;

   { This method is called just before open and tells
     to the kernel our requested set of characters to exchange data within this
     method. During this call, the context uses the character set obtained from
     ExternalEngine::getCharSet.

     @param(AStatus Status vector)
     @param(AContext External function execution context)
     @param(AName Character set name)
     @param(AName Character set name length)
   }
   procedure getCharSet(AStatus: IStatus; AContext: IExternalContext;
     AName: PAnsiChar; ANameSize: cardinal); override;

   { Execution of external procedure

     @param(AStatus Status vector)
```

```
        @param(AContext External function execution context)
        @param(AInMsg Pointer to input message)
        @param(AOutMsg Pointer to output message)
        @returns(Data set for selective procedure or
                 nil for run procedures)
      }
      function open(AStatus: IStatus; AContext: IExternalContext; AInMsg: Pointer;
        AOutMsg: Pointer): IExternalResultSet; override;
    end;

    // Output data set for the TGenRowsProcedure procedure
    TGenRowsResultSet = class(IExternalResultSetImpl)
      Input: PInput;
      Output: POutput;

      // Called when the dataset instance is destroyed
      procedure dispose(); override;

      { Retrieve the next record from the dataset. Somewhat analogous to
        SUSPEND. In this method, the next record from the data set should
        be prepared.

        @param(AStatus Status vector)
        @returns(True if the dataset has an entry to retrieve,
                 False if there are no more entries)
      }
      function fetch(AStatus: IStatus): Boolean; override;
    end;

implementation

{ TGenRowsFactory }

procedure TGenRowsFactory.dispose;
begin
   Destroy;
end;

function TGenRowsFactory.newItem(AStatus: IStatus; AContext: IExternalContext;
   AMetadata: IRoutineMetadata): IExternalProcedure;
begin
   Result := TGenRowsProcedure.create;
end;

procedure TGenRowsFactory.setup(AStatus: IStatus; AContext: IExternalContext;
   AMetadata: IRoutineMetadata; AInBuilder, AOutBuilder: IMetadataBuilder);
begin

end;

{ TGenRowsProcedure }
```

```
procedure TGenRowsProcedure.dispose;
begin
   Destroy;
end;

procedure TGenRowsProcedure.getCharSet(AStatus: IStatus;
   AContext: IExternalContext; AName: PAnsiChar; ANameSize: cardinal);
begin

end;

function TGenRowsProcedure.open(AStatus: IStatus; AContext: IExternalContext;
   AInMsg, AOutMsg: Pointer): IExternalResultSet;
begin
   Result := TGenRowsResultSet.create;
   with TGenRowsResultSet(Result) do
   begin
     Input := AInMsg;
     Output := AOutMsg;
   end;

   // if one of the input arguments is NULL, return nothing
   if PInput(AInMsg).startNull or PInput(AInMsg).finishNull then
   begin
     POutput(AOutMsg).nNull := True;
// intentionally set the output so that
// TGenRowsResultSet.fetch method returned false
     Output.n := Input.finish;
     exit;
   end;
   // checks
   if PInput(AInMsg).start > PInput(AInMsg).finish then
     raise Exception.Create('First parameter greater then second parameter.');

   with TGenRowsResultSet(Result) do
   begin
     // initial value
     Output.nNull := False;
     Output.n := Input.start - 1;
   end;
end;

{ TGenRowsResultSet }

procedure TGenRowsResultSet.dispose;
begin
   Destroy;
end;

// If it returns True, then the next record from the data set is retrieved.
```

```
// If it returns False, then the records in the data set are over
// new values in the output vector are calculated each time
// when calling this method
function TGenRowsResultSet.fetch(AStatus: IStatus): Boolean;
begin
  Inc(Output.n);
  Result := (Output.n <= Input.finish);
end;

end.
```

In the `open` method of the `TGenRowsProcedure` procedure instance, we check the first and second input arguments for the value `NULL`, if one of the arguments is `NULL`, then the output argument is `NULL`, in addition, the procedure should not return any row when fetching via the `SELECT` statement, so we assign `Output.n` such a value that the TGenRowsResultSet.fetch` method returns `False`.

In addition, we check that the first argument does not exceed the value of the second, otherwise we throw an exception. Don't worry, this exception will be caught in the UDR subsystem and converted to a Firebird exception. This is one of the advantages of the new UDRs over Legacy UDFs.

Since we are creating a selection procedure, the open method must return a dataset instance that implements the `IExternalResultSet` interface. To simplify, let's inherit our data set from the `IExternalResultSetImpl` class.

The `dispose` method is designed to release allocated resources. In it, we simply call the destructor.

The `fetch` method is called when the next record is retrieved by the `SELECT` statement. This method is essentially analogous to the `SUSPEND` statement used in regular PSQL stored procedures. Each time it is called, it prepares new values for the output message. The method returns `true` if the record should be returned to the caller, and `false` if there is no more data to retrieve. In our case, we simply increment the current value of the output variable until it is greater than the maximum limit.

Comment

Delphi does not support the `yield` operator, so you will not be able to write code like

```
while(...) do {
  ...
  yield result;
}
```

You can use any collection class, populate it in the `open` method of the stored procedure, and then return the values from that collection element-by-element to `fetch`. However, in this case, you lose the opportunity to prematurely abort the execution of the procedure (incomplete fetch in `SELECT` or FIRST / ROWS / FETCH delimiters in the `SELECT` statement.)

4.8. Registering triggers

Now let's add an external trigger to our UDR module.

Comment

In the original C++ examples, the trigger copies the record to another external database. I think that such an example is too complicated for the first acquaintance with external triggers. Working with connections to external databases will be discussed later.

Go back to the `UdrInit` module and change the `firebird_udr_plugin` function so that it looks like this.

```
function firebird_udr_plugin(AStatus: IStatus; AUnloadFlagLocal: BooleanPtr;
   AUdrPlugin: IUdrPlugin): BooleanPtr; cdecl;
begin
   // register our functions
   AUdrPlugin.registerFunction(AStatus, 'sum_args',
     TSumArgsFunctionFactory.Create());
   // register our procedures
   AUdrPlugin.registerProcedure(AStatus, 'sum_args_proc',
     TSumArgsProcedureFactory.Create());
   AUdrPlugin.registerProcedure(AStatus, 'gen_rows', TGenRowsFactory.Create());
   // register our triggers
   AUdrPlugin.registerTrigger(AStatus, 'test_trigger',
     TMyTriggerFactory.Create());

   theirUnloadFlag := AUnloadFlagLocal;
   Result := @myUnloadFlag;
end;
```

Comment

Don't forget to add the `TestTrigger` module to the `uses` list, where our trigger will be located.

4.9. Trigger Factory

An external trigger factory must implement the `IUdrTriggerFactory` interface. To simplify things, we simply inherit the `IUdrTriggerFactoryImpl` class. Each external trigger needs its own factory.

The `dispose` method is called when the factory is destroyed, in which we must release previously allocated resources. In this case, we simply call the destructor.

The `setup` method is executed every time an external trigger is loaded into the metadata cache. In it, you can do various actions that are necessary before creating a trigger instance, for example, to change the format of messages for table fields. We'll talk about it in more detail later.

The `newItem` method is called to instantiate an external trigger. This method is passed a pointer to

the status vector, the context of the external trigger, and the metadata of the external trigger. With `IRoutineMetadata` you can get the message format for new and old field values, the body of the external trigger, and other metadata. In this method, you can create different instances of the external trigger depending on its declaration in PSQL. Metadata can be passed to the created external trigger instance if necessary. In our case, we simply instantiate the external trigger `TMyTrigger`.

We will place the trigger factory, as well as the trigger itself, in the `TestTrigger` module.

```
unit TestTrigger;

{$IFDEF FPC}
{$MODE DELPHI}{$H+}
{$ENDIF}

interface

uses
  Firebird, SysUtils;

type
  { *************************************************************
    create table test (
      id int generated by default as identity,
      a int,
      b int,
      name varchar(100),
      constraint pk_test primary key(id)
    );

    create or alter trigger tr_test_biu for test
    active before insert or update position 0
    external name 'myudr!test_trigger'
    engine udr;
  }

  // structure for displaying NEW.* and OLD.* messages
  // must match the field set of the test table
  TFieldsMessage = record
    Id: Integer;
    IdNull: WordBool;
    A: Integer;
    ANull: WordBool;
    B: Integer;
    BNull: WordBool;
    Name: record
      Length: Word;
      Value: array [0 .. 399] of AnsiChar;
    end;
    NameNull: WordBool;
```

```
end;

PFieldsMessage = ^TFieldsMessage;

// Factory for instantiating external trigger TMyTrigger
TMyTriggerFactory = class(IUdrTriggerFactoryImpl)
  // Called when the factory is destroyed
  procedure dispose(); override;

  { Executed each time an external trigger is loaded into the metadata cache.
    Used to change the message format for fields.

    @param(AStatus Status vector)
    @param(AContext External trigger execution context)
    @param(AMetadata External trigger metadata)
    @param(AFieldsBuilder Message builder for table fields)
  }
  procedure setup(AStatus: IStatus; AContext: IExternalContext;
    AMetadata: IRoutineMetadata; AFieldsBuilder: IMetadataBuilder); override;

  { Creating a new instance of the external trigger TMyTrigger

    @param(AStatus Status vector)
    @param(AContext External trigger execution context)
    @param(AMetadata External trigger metadata)
    @returns(Instance of external trigger)
  }
  function newItem(AStatus: IStatus; AContext: IExternalContext;
    AMetadata: IRoutineMetadata): IExternalTrigger; override;
end;

TMyTrigger = class(IExternalTriggerImpl)
  // Called when the trigger is destroyed
  procedure dispose(); override;

  { This method is called just before execute and tells
    kernel our requested character set to exchange data internally
    this method. During this call, the context uses the character set
    obtained from ExternalEngine::getCharSet.

    @param(AStatus Status vector)
    @param(AContext External trigger execution context)
    @param(AName Character set name)
    @param(AName Character set name length)
  }
  procedure getCharSet(AStatus: IStatus; AContext: IExternalContext;

    AName: PAnsiChar; ANameSize: Cardinal); override;

  { trigger execution TMyTrigger
```

```
      @param(AStatus Status vector)
      @param(AContext External trigger execution context)
      @param(AAction Action (current event) trigger)
      @param(AOldMsg Message for old field values :OLD.*)
      @param(ANewMsg Message for new field values :NEW.*)
    }
    procedure execute(AStatus: IStatus; AContext: IExternalContext;
      AAction: Cardinal; AOldMsg: Pointer; ANewMsg: Pointer); override;
  end;

implementation

{ TMyTriggerFactory }

procedure TMyTriggerFactory.dispose;
begin
  Destroy;
end;

function TMyTriggerFactory.newItem(AStatus: IStatus; AContext: IExternalContext;
  AMetadata: IRoutineMetadata): IExternalTrigger;
begin
  Result := TMyTrigger.create;
end;

procedure TMyTriggerFactory.setup(AStatus: IStatus; AContext: IExternalContext;
  AMetadata: IRoutineMetadata; AFieldsBuilder: IMetadataBuilder);
begin

end;

{ TMyTrigger }

procedure TMyTrigger.dispose;
begin
  Destroy;
end;

procedure TMyTrigger.execute(AStatus: IStatus; AContext: IExternalContext;
  AAction: Cardinal; AOldMsg, ANewMsg: Pointer);
var
  xOld, xNew: PFieldsMessage;
begin
  // xOld := PFieldsMessage(AOldMsg);
  xNew := PFieldsMessage(ANewMsg);
  case AAction of
    IExternalTrigger.ACTION_INSERT:
      begin
        if xNew.BNull and not xNew.ANull then
        begin
          xNew.B := xNew.A + 1;
```

```
          xNew.BNull := False;
        end;
      end;

    IExternalTrigger.ACTION_UPDATE:
      begin
        if xNew.BNull and not xNew.ANull then
        begin
          xNew.B := xNew.A + 1;
          xNew.BNull := False;
        end;
      end;

    IExternalTrigger.ACTION_DELETE:
      begin

      end;
  end;
end;

procedure TMyTrigger.getCharSet(AStatus: IStatus; AContext: IExternalContext;
  AName: PAnsiChar; ANameSize: Cardinal);
begin

end;

end.
```

4.10. trigger instance

An external trigger must implement the `IExternalTrigger` interface. To simplify, we simply inherit the `IExternalTriggerImpl` class.

The `dispose` method is called when the trigger instance is destroyed, in which we must release the previously allocated resources. In this case, we simply call the destructor.

The `getCharSet` method is used to tell the external trigger context the character set we want to use when working with the connection from the current context. By default, the connection from the current context works in the encoding of the current connection, which is not always convenient.

The `execute` method is called when a trigger is executed on one of the events for which the trigger was created. This method is passed a pointer to the status vector, a pointer to the context of the external trigger, the action (event) that caused the trigger to fire, and pointers to messages for the old and new field values. Possible trigger actions (events) are listed by constants in the `IExternalTrigger` interface. Such constants start with the `ACTION_` prefix. Knowing about the current action is necessary because Firebird has triggers created for several events at once. Messages are needed only for triggers on table actions, for DDL triggers, as well as for triggers for database connection and disconnection events and triggers for transaction start, end and rollback events, pointers to messages will be initialized to `nil`. Unlike procedures and functions, trigger messages

are built for the fields of the table on the events of which the trigger was created. Static structures for such messages are built according to the same principles as message structures for input and output parameters of a procedure, but table fields are taken instead of variables.

Comment

Please note that if you are using message-to-struct mapping, then your triggers may break after changing the composition of table fields and their types. To prevent this from happening, use the work with the message through offsets obtained from `IMessageMetadata`. This is not so true for procedures and functions, since the input and output parameters do not change very often. Or at least you do it explicitly, which may lead you to think that you need to redo the outer procedure/function as well.

In our simplest trigger, we define the event type, and in the body of the trigger we execute the following PSQL analogue

```
...
  if (:new.B IS NULL) THEN
    :new.B = :new.A + 1;
...
```

Chapter 5. Messages

A message in UDR is a fixed-size memory area for passing input arguments to a procedure or function, or returning output arguments. For external event triggers, the message table entries are used to receive and return data in NEW and OLD.

To access individual variables or fields of a table, you need to know at least the type of that variable, and the offset from the beginning of the message buffer. As mentioned earlier, there are two ways to do this:

- conversion of a pointer to a message buffer to a pointer to a static structure (in Delphi this is a record, i.e. `record`);
- getting offsets using an instance of the class that implements the `IMessageMetadata` interface, and reading / writing from the data buffer, the size corresponding to the type of the variable or field.

The first method is the fastest, the second is more flexible, since in some cases it allows you to change the types and sizes for input and output variables or table fields without recompiling the dynamic library containing the UDR.

5.1. Working with the message buffer using a structure

As mentioned above, we can work with the message buffer through a pointer to a structure. This structure looks like this:

Syntax

```
TMyStruct = record
  <var_1>: <type_1>;
  <nullIndicator_1>: WordBool;
  <var_2>: <type_1>;
  <nullIndicator_2>: WordBool;
  ...
  <var_N>: <type_1>;
  <nullIndicator_N>: WordBool;
end;
PMyStruct = ^TMyStruct;
```

The types of data members must match the types of input/output variables or fields (for triggers). There must be a null indicator after each variable/field, even if they have a `NOT NULL` constraint. Null indicator takes 2 bytes. The value -1 means that the variable/field has the value `NULL`. Since at the moment only the `NULL` attribute is written to the NULL-indicator, it is convenient to reflect it on a 2-byte logical type. SQL data types appear in the structure as follows:

Table 4. Mapping SQL types to Delphi types

Sql type	Delphi type	Remark
BOOLEAN	Boolean, ByteBool	
SMALLINT	Smallint	

Sql type	Delphi type	Remark
INTEGER	Integer	
BIGINT	Int64	
INT128	FB_I128	Available since Firebird 4.0.
FLOAT	Single	
DOUBLE PRECISION	Double	
DECFLOAT(16)	FB_DEC16	Available since Firebird 4.0.
DECFLOAT(34)	FB_DEC34	Available since Firebird 4.0.
NUMERIC(N, M)	The data type depends on the precision and dialect: • 1-4 — Smallint; • 5-9 — Integer; • 10-18 (3 dialect) — Int64; • 10-15 (1 dialect) — Double; • 19-38 - FB_I128 (since Firebird 4.0).	As a value, the number multiplied by 10^{M}.
DECIMAL(N, M)	The data type depends on the precision and dialect: • 1-4 — Integer; • 5-9 — Integer; • 10-18 (3 dialect) — Int64; • 10-15 (1 dialect) — Double; • 19-38 - FB_I128 (since Firebird 4.0).	As a value, the number multiplied by 10^{M}.
CHAR(N)	array[0 .. M] of AnsiChar	M is calculated by the formula M = N * BytesPerChar - 1, where BytesPerChar - number of bytes per character, depends on encoding variable/field. For example, for UTF-8 it is 4 bytes/character, for WIN1251 it is 1 byte/char.
VARCHAR(N)	record Length: Smallint; Data: array[0 .. M] of AnsiChar; end	M is calculated by the formula M = N * BytesPerChar - 1, where BytesPerChar - number of bytes per character, depends on encoding variable/field. For example, for UTF-8 it is 4 bytes/character, for WIN1251 it is 1 byte/char. Length is the actual length of the string in characters.
DATE	ISC_DATE	
TIME	ISC_TIME	

Sql type	Delphi type	Remark
TIME WITH TIME ZONE	ISC_TIME_TZ	Available since Firebird 4.0.
TIMESTAMP	ISC_TIMESTAMP	
TIMESTAMP WITH TIME ZONE	ISC_TIMESTAMP_TZ	Available since Firebird 4.0.
BLOB	ISC_QUAD	The contents of the BLOB are never passed directly; the BlobId is passed instead. How to work with the BLOB type will be described in the chapter Working with the BLOB type.

Now let's look at a few examples of how to build message structures from procedure, function, or trigger declarations.

Suppose we have an external function declared like this:

```
function SUM_ARGS(A SMALLINT, B INTEGER) RETURNS BIGINT
....
```

In this case, the structures for input and output messages will look like So:

```
TInput = record
  A: Smallint;
  ANull: WordBool;
  B: Integer;
  BNull: WordBool;
end;
PInput = ^TInput;

TOutput = record
  Value: Int64;
  Null: WordBool;
end;
POutput = ^TOutput;
```

If the same function is defined with other types (in dialect 3):

```
function SUM_ARGS(A NUMERIC(4, 2), B NUMERIC(9, 3)) RETURNS NUMERIC(18, 6)
....
```

In this case, the structures for input and output messages will look like this:

```
TInput = record
  A: Smallint;
```

```
  ANull: WordBool;
  B: Integer;
  BNull: WordBool;
end;
PInput = ^TInput;

TOutput = record
  Value: Int64;
  Null: WordBool;
end;
POutput = ^TOutput;
```

Suppose we have an external procedure declared as follows:

```
procedure SOME_PROC(A CHAR(3) CHARACTER SET WIN1251, B VARCHAR(10) CHARACTER SET UTF8)
....
```

In this case, the structure for the input message will look like this:

```
TInput = record
  A: array[0..2] of AnsiChar;
  ANull: WordBool;
  B: record
    Length: Smallint;
    Value: array[0..39] of AnsiChar;
  end;
  BNull: WordBool;
end;
PInput = ^TInput;
```

5.2. Working with the message buffer using IMessageMetadata

As described above, you can work with the message buffer using an instance of an object that implements the `IMessageMetadata` interface. This interface allows you to learn the following information about a variable/field:

- variable/field name;
- data type;
- character set for string data;
- subtype for BLOB data type;
- buffer size in bytes for variable/field;
- whether a variable/field can take on a NULL value;
- offset in the message buffer for data;

- offset in message buffer for NULL indicator.

5.2.1. Methods of the IMessageMetadata interface

1. getCount

   ```
   unsigned getCount(StatusType* status)
   ```

   returns the number of fields/parameters in the message. In all calls containing an index parameter, this value should be: `0 <= index < getCount()`.

2. getField

   ```
   const char* getField(StatusType* status, unsigned index)
   ```

   returns the name of the field.

3. getRelation

   ```
   const char* getRelation(StatusType* status, unsigned index)
   ```

   returns the name of the relation (from which the given field is selected).

4. getOwner

   ```
   const char* getOwner(StatusType* status, unsigned index)
   ```

   returns the name of the relationship owner.

5. getAlias

   ```
   const char* getAlias(StatusType* status, unsigned index)
   ```

   returns the field alias.

6. getType

   ```
   unsigned getType(StatusType* status, unsigned index)
   ```

   returns the SQL type of the field.

7. isNullable

```
FB_BOOLEAN isNullable(StatusType* status, unsigned index)
```

returns true if the field can be null.

8. getSubType

```
int getSubType(StatusType* status, unsigned index)
```

returns the subtype of the BLOB field (0 - binary, 1 - text, etc.).

9. getLength

```
unsigned getLength(StatusType* status, unsigned index)
```

returns the maximum length of the field in bytes.

10. getScale

```
int getScale(StatusType* status, unsigned index)
```

returns the scale for a numeric field.

11. getCharSet

```
unsigned getCharSet(StatusType* status, unsigned index)
```

returns the character set for character fields and text BLOB.

12. getOffset

```
unsigned getOffset(StatusType* status, unsigned index)
```

returns the field data offset in the message buffer (use it to accessing data in the message buffer).

13. getNullOffset

```
unsigned getNullOffset(StatusType* status, unsigned index)
```

returns the NULL offset of the indicator for the field in the message buffer.

14. getBuilder

```
IMetadataBuilder* getBuilder(StatusType* status)
```

returns the `IMetadataBuilder` interface initialized with metadata this message.

15. getMessageLength

```
unsigned getMessageLength(StatusType* status)
```

returns the length of the message buffer (use it to allocate memory under the buffer).

5.2.2. Getting and Using IMessageMetadata

Instances of objects that implement the `IMessageMetadata` interface for input and output variables can be obtained from the `IRoutineMetadata` interface. It is not passed directly to an instance of a procedure, function, or trigger. This must be done explicitly in the factory of the appropriate type. For example:

```
// Factory for instantiating the external function TSumArgsFunction
TSumArgsFunctionFactory = class(IUdrFunctionFactoryImpl)
  // Called when the factory is destroyed
  procedure dispose(); override;

  { Executed each time an external function is loaded into the metadata cache

    @param(AStatus Status vector)
    @param(AContext External function execution context)
    @param(AMetadata External function metadata)
    @param(AInBuilder Message builder for input metadata)
    @param(AOutBuilder Message builder for output metadata)
  }
  procedure setup(AStatus: IStatus; AContext: IExternalContext;
    AMetadata: IRoutineMetadata; AInBuilder: IMetadataBuilder;
    AOutBuilder: IMetadataBuilder); override;

  { Creating a new instance of the external function TSumArgsFunction

    @param(AStatus Status vector)
    @param(AContext External function execution context)
    @param(AMetadata External function metadata)
    @returns(External function instance)
  }
  function newItem(AStatus: IStatus; AContext: IExternalContext;
    AMetadata: IRoutineMetadata): IExternalFunction; override;
end;

// External function TSumArgsFunction.
TSumArgsFunction = class(IExternalFunctionImpl)
private
```

```
    FMetadata: IRoutineMetadata;
  public
    property Metadata: IRoutineMetadata read FMetadata write FMetadata;
  public
    // Called when the function instance is destroyed
    procedure dispose(); override;

    { This method is called just before execute and tells the kernel
      our requested character set to communicate within this method.
      During this call, the context uses the character set obtained
      from ExternalEngine::getCharSet.

      @param(AStatus Status vector)
      @param(AContext External function execution context)
      @param(AName Character set name)
      @param(AName Character set name length)
    }
    procedure getCharSet(AStatus: IStatus; AContext: IExternalContext;
      AName: PAnsiChar; ANameSize: Cardinal); override;

    { Executing an external function

      @param(AStatus Status vector)
      @param(AContext External function execution context)
      @param(AInMsg Pointer to input message)
      @param(AOutMsg Pointer to output message)
    }
    procedure execute(AStatus: IStatus; AContext: IExternalContext;
      AInMsg: Pointer; AOutMsg: Pointer); override;
  end;
.......................

{ TSumArgsFunctionFactory }

procedure TSumArgsFunctionFactory.dispose;
begin
  Destroy;
end;

function TSumArgsFunctionFactory.newItem(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata): IExternalFunction;
begin
  Result := TSumArgsFunction.Create();
  with Result as TSumArgsFunction do
  begin
    Metadata := AMetadata;
  end;
end;

procedure TSumArgsFunctionFactory.setup(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata;
```

```
  AInBuilder, AOutBuilder: IMetadataBuilder);
begin

end;
```

Instances of `IMessageMetadata` for input and output variables can be obtained using the `getInputMetadata` and `getOutputMetadata` methods from `IRoutineMetadata`. Metadata for the fields of the table on which the trigger is written can be obtained using the `getTriggerMetadata` method.

Important

Please note that the lifecycle of `IMessageMetadata` interface objects is controlled using reference counting. It inherits the `IReferenceCounted` interface. The `getInputMetadata` and `getOutputMetadata` methods increase the reference count by 1 for the returned objects, so after finishing using these objects you need to decrease the reference count for the `xInputMetadata` and `xOutputMetadata` variables by calling the `release` method.

To obtain the value of the corresponding input argument, we need to use address arithmetic. To do this, we get the offset from `IMessageMetadata` using the `getOffset` method and add it to the buffer address for the input message. Then we reduce the resulting result to the corresponding typed pointer. Approximately the same scheme of work for obtaining null indicators of arguments, only the `getNullOffset` method is used to obtain offsets.

```
.......................

procedure TSumArgsFunction.execute(AStatus: IStatus; AContext: IExternalContext;
  AInMsg, AOutMsg: Pointer);
var
  n1, n2, n3: Integer;
  n1Null, n2Null, n3Null: WordBool;
  Result: Integer;
  resultNull: WordBool;
  xInputMetadata, xOutputMetadata: IMessageMetadata;
begin
  xInputMetadata := FMetadata.getInputMetadata(AStatus);
  xOutputMetadata := FMetadata.getOutputMetadata(AStatus);
  try
    // get the values of the input arguments by their offsets
    n1 := PInteger(PByte(AInMsg) + xInputMetadata.getOffset(AStatus, 0))^;
    n2 := PInteger(PByte(AInMsg) + xInputMetadata.getOffset(AStatus, 1))^;
    n3 := PInteger(PByte(AInMsg) + xInputMetadata.getOffset(AStatus, 2))^;
    // get values of null indicators of input arguments by their offsets
    n1Null := PWordBool(PByte(AInMsg) +
      xInputMetadata.getNullOffset(AStatus, 0))^;
    n2Null := PWordBool(PByte(AInMsg) +
      xInputMetadata.getNullOffset(AStatus, 1))^;
    n3Null := PWordBool(PByte(AInMsg) +
      xInputMetadata.getNullOffset(AStatus, 2))^;
```

```
    //by default, the output argument is NULL, so we set it to nullFlag
    resultNull := True;
    Result := 0;
    // if one of the arguments is NULL, then the result is NULL
    // otherwise, we calculate the sum of the arguments
    if not(n1Null or n2Null or n3Null) then
    begin
      Result := n1 + n2 + n3;
      // once there is a result, then reset the NULL flag
      resultNull := False;
    end;
    PWordBool(PByte(AInMsg) + xOutputMetadata.getNullOffset(AStatus, 0))^ :=
      resultNull;
    PInteger(PByte(AInMsg) + xOutputMetadata.getOffset(AStatus, 0))^ := Result;
  finally
    xInputMetadata.release;
    xOutputMetadata.release;
  end;
end;
```

Comment

In the Connection and Transaction Context chapter, great example to work with various SQL types using interface `IMessageMetadata`.

Chapter 6. Factories

You have already encountered factories before. It's time to consider them in detail.

Factories are designed to create instances of procedures, functions, or triggers. The factory class must inherit from one of the `IUdrProcedureFactory`, `IUdrFunctionFactory` or `IUdrTriggerFactory` interfaces depending on the UDR type. Instances of these must be registered as UDR entry points in the `firebird_udr_plugin` function.

```
function firebird_udr_plugin(AStatus: IStatus; AUnloadFlagLocal: BooleanPtr;
  AUdrPlugin: IUdrPlugin): BooleanPtr; cdecl;
begin
  // register our function
  AUdrPlugin.registerFunction(AStatus, 'sum_args',
    TSumArgsFunctionFactory.Create());
  // register our procedure
  AUdrPlugin.registerProcedure(AStatus, 'gen_rows', TGenRowsFactory.Create());
  // register our trigger
  AUdrPlugin.registerTrigger(AStatus, 'test_trigger',
    TMyTriggerFactory.Create());

  theirUnloadFlag := AUnloadFlagLocal;
  Result := @myUnloadFlag;
end;
```

In this example, the `TSumArgsFunctionFactory` class inherits the `IUdrFunctionFactory` interface, `TGenRowsFactory` inherits the `IUdrProcedureFactory` interface, and `TMyTriggerFactory` inherits the `IUdrTriggerFactory` interface.

Factory instances are created and bound to entry points the first time an external procedure, function, or trigger is loaded. This happens once per Firebird process creation. Thus, for the SuperServer architecture, for all connections there will be exactly one factory instance associated with each entry point; for Classic, this number of instances will be multiplied by the number of connections.

When writing factory classes, you need to implement the `setup` and `newItem` methods from the `IUdrProcedureFactory`, `IUdrFunctionFactory` or `IUdrTriggerFactory` interfaces.

```
  IUdrFunctionFactory = class(IDisposable)
    const VERSION = 3;

    procedure setup(status: IStatus; context: IExternalContext;
      metadata: IRoutineMetadata; inBuilder: IMetadataBuilder;
        outBuilder: IMetadataBuilder);

    function newItem(status: IStatus; context: IExternalContext;
      metadata: IRoutineMetadata): IExternalFunction;
  end;
```

```
  IUdrProcedureFactory = class(IDisposable)
    const VERSION = 3;

    procedure setup(status: IStatus; context: IExternalContext;
      metadata: IRoutineMetadata; inBuilder: IMetadataBuilder;
        outBuilder: IMetadataBuilder);

    function newItem(status: IStatus; context: IExternalContext;
      metadata: IRoutineMetadata): IExternalProcedure;
  end;

  IUdrTriggerFactory = class(IDisposable)
    const VERSION = 3;

    procedure setup(status: IStatus; context: IExternalContext;
      metadata: IRoutineMetadata; fieldsBuilder: IMetadataBuilder);

    function newItem(status: IStatus; context: IExternalContext;
      metadata: IRoutineMetadata): IExternalTrigger;
  end;
```

Also, since these interfaces inherit the `IDisposable` interface, you must also implement the `dispose` method. This means that Firebird will unload the factory itself when needed. In the `dispose` method, you need to place code that releases resources when the factory instance is destroyed. To simplify the implementation of interface methods, it is convenient to use the classes `IUdrProcedureFactoryImpl`, `IUdrFunctionFactoryImpl`, `IUdrTriggerFactoryImpl`. Let's consider each of the methods in more detail.

6.1. Method newItem

The `newItem` method is called to instantiate an external procedure, function, or trigger. A UDR is instantiated when it is loaded into the metadata cache, i.e. the first time a procedure, function, or trigger is called. Currently, the metadata cache is per-connection per-connection cache for all server architectures.

The procedure and function metadata cache is associated with their names in the database. For example, two external functions with different names but the same entry points will be different instances of `IUdrFunctionFactory`. The entry point consists of the name of the external module and the name under which the factory is registered. How this can be used will be shown later.

The `newItem` method is passed a pointer to the status vector, the UDR execution context, and UDR metadata.

In the simplest case, the implementation of this method is trivial

```
function TSumArgsFunctionFactory.newItem(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata): IExternalFunction;
begin
```

```
  // create an instance of an external function
  Result := TSumArgsFunction.Create();
end;
```

With `IRoutineMetadata` you can get the input and output message format, UDR body and other metadata. Metadata can be passed to the created UDR instance. In this case, you need to add a field for storing metadata to an instance of the class that implements your UDR.

```
  // External function TSumArgsFunction.
  TSumArgsFunction = class(IExternalFunctionImpl)
  private
    FMetadata: IRoutineMetadata;
  public
    property Metadata: IRoutineMetadata read FMetadata write FMetadata;
  public
  ...
  end;
```

In this case, the implementation of the newItem method looks like this:

```
function TSumArgsFunctionFactory.newItem(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata): IExternalFunction;
begin
  Result := TSumArgsFunction.Create();
  with Result as TSumArgsFunction do
  begin
    Metadata := AMetadata;
  end;
end;
```

6.2. Creating instances of UDRs depending on their declaration

In the `newItem` method, you can create different instances of an external procedure or function, depending on its declaration in PSQL. To do this, you can use the information obtained from `IMessageMetadata`.

Suppose we want to implement a PSQL package with the same set of external functions for squaring a number for various data types and a single entry point.

```
SET TERM ^ ;

CREATE OR ALTER PACKAGE MYUDR2
AS
begin
  function SqrSmallint(AInput SMALLINT) RETURNS INTEGER;
```

```
  function SqrInteger(AInput INTEGER) RETURNS BIGINT;
  function SqrBigint(AInput BIGINT) RETURNS BIGINT;
  function SqrFloat(AInput FLOAT) RETURNS DOUBLE PRECISION;
  function SqrDouble(AInput DOUBLE PRECISION) RETURNS DOUBLE PRECISION;
end^

RECREATE PACKAGE BODY MYUDR2
AS
begin
  function SqrSmallint(AInput SMALLINT) RETURNS INTEGER
  external name 'myudr2!sqrt_func'
  engine udr;

  function SqrInteger(AInput INTEGER) RETURNS BIGINT
  external name 'myudr2!sqrt_func'
  engine udr;

  function SqrBigint(AInput BIGINT) RETURNS BIGINT
  external name 'myudr2!sqrt_func'
  engine udr;

  function SqrFloat(AInput FLOAT) RETURNS DOUBLE PRECISION
  external name 'myudr2!sqrt_func'
  engine udr;

  function SqrDouble(AInput DOUBLE PRECISION) RETURNS DOUBLE PRECISION
  external name 'myudr2!sqrt_func'
  engine udr;

end
^

SET TERM ; ^
```

To test the functions, we will use the following query

```
select
  myudr2.SqrSmallint(1) as n1,
  myudr2.SqrInteger(2) as n2,
  myudr2.SqrBigint(3) as n3,
  myudr2.SqrFloat(3.1) as n4,
  myudr2.SqrDouble(3.2) as n5
from rdb$database
```

To make it easier to work with `IMessageMetadata` and buffers, you can write a convenient wrapper or try to use `IMessageMetadata` and structures to display messages together. Here we will show the use of the second method.

The implementation of this idea is quite simple: in the function factory, we will create different

function instances depending on the type of the input argument. In modern versions of Delphi, you can use generics to generalize code.

```
.....................
type
  // the structure to which the input message will be mapped
  TSqrInMsg<T> = record
    n1: T;
    n1Null: WordBool;
  end;

  // the structure to which the output message will be mapped
  TSqrOutMsg<T> = record
    result: T;
    resultNull: WordBool;
  end;

  // Factory for instantiating external function TSqrFunction
  TSqrFunctionFactory = class(IUdrFunctionFactoryImpl)
    // Called when the factory is destroyed
    procedure dispose(); override;

    { Executed each time an external function is loaded into the metadata cache.
      Used to change the format of the input and output messages.

      @param(AStatus Status vector)
      @param(AContext External function execution context)
      @param(AMetadata External function metadata)
      @param(AInBuilder Message builder for input metadata)
      @param(AOutBuilder Message builder for output metadata)
    }
    procedure setup(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata; AInBuilder: IMetadataBuilder;
      AOutBuilder: IMetadataBuilder); override;

    { Creating a new instance of an external TSqrFunction

      @param(AStatus Status vector)
      @param(AContext External function execution context)
      @param(AMetadata External function metadata)
      @returns(External function instance)
    }
    function newItem(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata): IExternalFunction; override;
  end;


  // External function TSqrFunction.
  TSqrFunction<TIn, TOut> = class(IExternalFunctionImpl)
  private
```

```
    function sqrExec(AIn: TIn): TOut; virtual; abstract;
  public
    type
      TInput = TSqrInMsg<TIn>;
      TOutput = TSqrOutMsg<TOut>;
      PInput = ^TInput;
      POutput = ^TOutput;
    // Called when the function instance is destroyed
    procedure dispose(); override;

    { This method is called just before execute and
      tells the kernel our requested character set to communicate within this
      method. During this call, the context uses the character set obtained from
      ExternalEngine::getCharSet.

      @param(AStatus Status vector)
      @param(AContext External function execution context)
      @param(AName Character set name)
      @param(AName Character set name length)
    }
    procedure getCharSet(AStatus: IStatus; AContext: IExternalContext;
      AName: PAnsiChar; ANameSize: Cardinal); override;

    { Executing an external function

      @param(AStatus Status vector)
      @param(AContext External function execution context)
      @param(AInMsg Pointer to input message)
      @param(AOutMsg Pointer to output message)
    }
    procedure execute(AStatus: IStatus; AContext: IExternalContext;
      AInMsg: Pointer; AOutMsg: Pointer); override;
  end;

  TSqrExecSmallint = class(TSqrFunction<Smallint, Integer>)
  public
    function sqrExec(AIn: Smallint): Integer; override;
  end;

  TSqrExecInteger = class(TSqrFunction<Integer, Int64>)
  public
    function sqrExec(AIn: Integer): Int64; override;
  end;

  TSqrExecInt64 = class(TSqrFunction<Int64, Int64>)
  public
    function sqrExec(AIn: Int64): Int64; override;
  end;

  TSqrExecFloat = class(TSqrFunction<Single, Double>)
  public
```

```
    function sqrExec(AIn: Single): Double; override;
  end;

  TSqrExecDouble = class(TSqrFunction<Double, Double>)
  public
    function sqrExec(AIn: Double): Double; override;
  end;

implementation

uses
  SysUtils, FbTypes, System.TypInfo;

{ TSqrFunctionFactory }

procedure TSqrFunctionFactory.dispose;
begin
  Destroy;
end;

function TSqrFunctionFactory.newItem(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata): IExternalFunction;
var
  xInputMetadata: IMessageMetadata;
  xInputType: TFBType;
begin
  // get the type of the input argument
  xInputMetadata := AMetadata.getInputMetadata(AStatus);
  xInputType := TFBType(xInputMetadata.getType(AStatus, 0));
  xInputMetadata.release;
  // create an instance of a function depending on the type
  case xInputType of
    SQL_SHORT:
      result := TSqrExecSmallint.Create();

    SQL_LONG:
      result := TSqrExecInteger.Create();
    SQL_INT64:
      result := TSqrExecInt64.Create();

    SQL_FLOAT:
      result := TSqrExecFloat.Create();
    SQL_DOUBLE, SQL_D_FLOAT:
      result := TSqrExecDouble.Create();
  else
    result := TSqrExecInt64.Create();
  end;

end;

procedure TSqrFunctionFactory.setup(AStatus: IStatus;
```

```
  AContext: IExternalContext; AMetadata: IRoutineMetadata;
  AInBuilder, AOutBuilder: IMetadataBuilder);
begin

end;

{ TSqrFunction }

procedure TSqrFunction<TIn, TOut>.dispose;
begin
  Destroy;
end;

procedure TSqrFunction<TIn, TOut>.execute(AStatus: IStatus;
  AContext: IExternalContext; AInMsg, AOutMsg: Pointer);
var
  xInput: PInput;
  xOutput: POutput;
begin
  xInput := PInput(AInMsg);
  xOutput := POutput(AOutMsg);
  xOutput.resultNull := True;
  if (not xInput.n1Null) then
  begin
    xOutput.resultNull := False;
    xOutput.result := Self.sqrExec(xInput.n1);
  end;
end;

procedure TSqrFunction<TIn, TOut>.getCharSet(AStatus: IStatus;
  AContext: IExternalContext; AName: PAnsiChar; ANameSize: Cardinal);
begin
end;


{ TSqrtExecSmallint }

function TSqrExecSmallint.sqrExec(AIn: Smallint): Integer;
begin
  Result := AIn * AIn;
end;

{ TSqrExecInteger }

function TSqrExecInteger.sqrExec(AIn: Integer): Int64;
begin
  Result := AIn * AIn;
end;

{ TSqrExecInt64 }
```

```
function TSqrExecInt64.sqrExec(AIn: Int64): Int64;
begin
  Result := AIn * AIn;
end;

{ TSqrExecFloat }

function TSqrExecFloat.sqrExec(AIn: Single): Double;
begin
  Result := AIn * AIn;
end;

{ TSqrExecDouble }

function TSqrExecDouble.sqrExec(AIn: Double): Double;
begin
  Result := AIn * AIn;
end;

.................
```

6.3. setup method

The `setup` method allows you to change the types of input parameters and output variables for external procedures and functions or fields for triggers. For this, the `iMetadatabuilder` interface is used, which allows you to build input and output messages with specified types, dimension and a set of characters. Entrance messages will be rebuilt into the format set in the `setup` method, and the weekend is rebuilt from the format set in the` setup` format to the format of the message format in the DLL procedure, function or trigger. Types of fields or parameters should be compatible for transformation.

This method allows you to simplify the creation of generalized for different types of parameters and functions by bringing them to the most general type. A more complicated and useful example will be considered later, but for now, we will slightly change the existing example of the external function of `sumargs`.

Our function will work with messages described by the following structure

```
type
  // the structure to which the input message will be mapped
  TSumArgsInMsg = record
    n1: Integer;
    n1Null: WordBool;
    n2: Integer;
    n2Null: WordBool;
    n3: Integer;
    n3Null: WordBool;
  end;
```

```
  PSumArgsInMsg = ^TSumArgsInMsg;

  // the structure to which the output message will be mapped
  TSumArgsOutMsg = record
    result: Integer;
    resultNull: WordBool;
  end;

  PSumArgsOutMsg = ^TSumArgsOutMsg;
```

Now let's create a function factory, in the `setup` method we set the format messages that match the above structures.

```
{ TSumArgsFunctionFactory }

procedure TSumArgsFunctionFactory.dispose;
begin
  Destroy;
end;

function TSumArgsFunctionFactory.newItem(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata): IExternalFunction;
begin
  Result := TSumArgsFunction.Create();
end;

procedure TSumArgsFunctionFactory.setup(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata;
  AInBuilder, AOutBuilder: IMetadataBuilder);
begin
  // building a message for the input parameters
  AInBuilder.setType(AStatus, 0, Cardinal(SQL_LONG));
  AInBuilder.setLength(AStatus, 0, sizeof(Int32));
  AInBuilder.setType(AStatus, 1, Cardinal(SQL_LONG));
  AInBuilder.setLength(AStatus, 1, sizeof(Int32));
  AInBuilder.setType(AStatus, 2, Cardinal(SQL_LONG));
  AInBuilder.setLength(AStatus, 2, sizeof(Int32));
  // building a message for output parameters
  AOutBuilder.setType(AStatus, 0, Cardinal(SQL_LONG));
  AOutBuilder.setLength(AStatus, 0, sizeof(Int32));
end;
```

Implementation functions trivial

```
procedure TSumArgsFunction.execute(AStatus: IStatus; AContext: IExternalContext;
  AInMsg, AOutMsg: Pointer);
var
  xInput: PSumArgsInMsg;
  xOutput: PSumArgsOutMsg;
```

```
begin
  // convert pointers to input and output to typed
  xInput := PSumArgsInMsg(AInMsg);
  xOutput := PSumArgsOutMsg(AOutMsg);
  // by default, the output argument is NULL, so we set it to nullFlag
  xOutput^.resultNull := True;
  // if one of the arguments is NULL, then the result is NULL
   // otherwise, we calculate the sum of the arguments
  with xInput^ do
  begin
    if not(n1Null or n2Null or n3Null) then
    begin
      xOutput^.result := n1 + n2 + n3;
      // once there is a result, then reset the NULL flag
      xOutput^.resultNull := False;
    end;
  end;
end;
```

Now, even if we declare the functions as follows, it still will remain operational, since the input and output messages will be converted to the format we set in the `setup` method.

```
CREATE OR ALTER FUNCTION FN_SUM_ARGS (
    N1 VARCHAR(15),
    N2 VARCHAR(15),
    N3 VARCHAR(15))
RETURNS VARCHAR(15)
EXTERNAL NAME 'MyUdrSetup!sum_args'
ENGINE UDR;
```

You can check the above statement by running the following request

```
select FN_SUM_ARGS('15', '21', '35') from rdb$database
```

6.4. Generic factories

In the process of developing UDR, it is necessary for each external procedure, function or trigger to write your factory creating an instance is UDR. This task can be simplified by writing generalized factories using the so -called generics. They are available starting with Delphi 2009, in Free Pascal starting with the FPC 2.2 version.

Comment

In Free Pascal, the syntax for creating generic types is different from Delphi. Since version FPC 2.6.0 the syntax compatible with Delphi is declared.

Consider the two main cases for which generalized factories will be written:

- copies of external procedures, functions and triggers do not require any information about metadata, do not need special actions in the logic of creating UDR copies, fixed structures are used to work with messages;
- Corps of external procedures, functions and triggers require information about metadata, special actions are not needed in the logic of creating UDR copies, and instances of messages `IMessagemetadata` are used to work with messages.

In the first case, it is enough to simply create the desired copy of the class in the `Newitem` method without additional actions. To do this, we will use the restriction of the designer in the classrooms of the classes `IUdrFunctionFactoryImpl`, `IUdrProcedureFactoryImpl`, `IUdrTriggerFactoryImpl`. The ads of such factories are as follows:

```
unit UdrFactories;

{$IFDEF FPC}
{$MODE DELPHI}{$H+}
{$ENDIF}

interface

uses SysUtils, Firebird;

type

  // A simple external function factory
  TFunctionSimpleFactory<T: IExternalFunctionImpl, constructor> = class
    (IUdrFunctionFactoryImpl)
    procedure dispose(); override;

    procedure setup(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata; AInBuilder: IMetadataBuilder;
      AOutBuilder: IMetadataBuilder); override;

    function newItem(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata): IExternalFunction; override;
  end;

  // A simple external procedure factory
  TProcedureSimpleFactory<T: IExternalProcedureImpl, constructor> = class
    (IUdrProcedureFactoryImpl)
    procedure dispose(); override;

    procedure setup(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata; AInBuilder: IMetadataBuilder;
      AOutBuilder: IMetadataBuilder); override;

    function newItem(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata): IExternalProcedure; override;
  end;
```

```
  // A simple external trigger factory
  TTriggerSimpleFactory<T: IExternalTriggerImpl, constructor> = class
    (IUdrTriggerFactoryImpl)
    procedure dispose(); override;

    procedure setup(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata; AFieldsBuilder: IMetadataBuilder); override;

    function newItem(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata): IExternalTrigger; override;
  end;
```

In the implementation section, the body of the `setup` method can be left empty, nothing is done in them, in the body of the` dispose 'method, just call the destructor. And in the body of the `Newitem` method, you just need to call the default designer for the substitution type` t`.

```
implementation

{ TProcedureSimpleFactory<T> }

procedure TProcedureSimpleFactory<T>.dispose;
begin
  Destroy;
end;

function TProcedureSimpleFactory<T>.newItem(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata): IExternalProcedure;
begin
  Result := T.Create;
end;

procedure TProcedureSimpleFactory<T>.setup(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata;
  AInBuilder, AOutBuilder: IMetadataBuilder);
begin

end;


{ TFunctionFactory<T> }

procedure TFunctionSimpleFactory<T>.dispose;
begin
  Destroy;
end;

function TFunctionSimpleFactory<T>.newItem(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata): IExternalFunction;
begin
  Result := T.Create;
```

```
end;

procedure TFunctionSimpleFactory<T>.setup(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata;
  AInBuilder, AOutBuilder: IMetadataBuilder);
begin

end;

{ TTriggerSimpleFactory<T> }

procedure TTriggerSimpleFactory<T>.dispose;
begin
  Destroy;
end;

function TTriggerSimpleFactory<T>.newItem(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata): IExternalTrigger;
begin
  Result := T.Create;
end;

procedure TTriggerSimpleFactory<T>.setup(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata;
  AFieldsBuilder: IMetadataBuilder);
begin

end;
```

Now for case 1, you can not write factories for each procedure, function or trigger. Instead, register them with generic factories as follows:

```
function firebird_udr_plugin(AStatus: IStatus; AUnloadFlagLocal: BooleanPtr;
  AUdrPlugin: IUdrPlugin): BooleanPtr; cdecl;
begin
  // register our function
  AUdrPlugin.registerFunction(AStatus, 'sum_args',
    TFunctionSimpleFactory<TSumArgsFunction>.Create());
  // register our procedure
  AUdrPlugin.registerProcedure(AStatus, 'gen_rows',
    TProcedureSimpleFactory<TGenRowsProcedure>.Create());
  // register our trigger
  AUdrPlugin.registerTrigger(AStatus, 'test_trigger',
    TTriggerSimpleFactory<TMyTrigger>.Create());

  theirUnloadFlag := AUnloadFlagLocal;
  Result := @myUnloadFlag;
end;
```

The second case is more complicated. By default, metadata information is not transmitted into copies of procedures, functions and triggers. However, metadata is transmitted as a parameter in the method of `newitem` factories. UDR metadata has the type of `IRoutineMetadata`, the life cycle of which is controlled by the Firebird engine itself, so it can be safely transferred to UDR copies. From it you can get copies of interfaces for the input and output message, metadata and trigger type, UDR name, package, entrance points and UDR body. The classes themselves for the implementation of external procedures, functions and triggers do not have fields for storing metadata, so we will have to make their heirs.

```
unit UdrFactories;

{$IFDEF FPC}
{$MODE DELPHI}{$H+}
{$ENDIF}

interface

uses SysUtils, Firebird;

type
...

  // External function with metadata
  TExternalFunction = class(IExternalFunctionImpl)
    Metadata: IRoutineMetadata;
  end;

  // External procedure with metadata
  TExternalProcedure = class(IExternalProcedureImpl)
    Metadata: IRoutineMetadata;
  end;

  // External trigger with metadata
  TExternalTrigger = class(IExternalTriggerImpl)
    Metadata: IRoutineMetadata;
  end;
```

In this case, your own stored procedures, functions, and triggers should be inherited from new classes with metadata.

Now let’s declare the factories that will create the UDR and initialize metadata.

```
unit UdrFactories;

{$IFDEF FPC}
{$MODE DELPHI}{$H+}
{$ENDIF}

interface
```

```
uses SysUtils, Firebird;

type
...

  // Factory of external functions with metadata
  TFunctionFactory<T: TExternalFunction, constructor> = class
    (IUdrFunctionFactoryImpl)
    procedure dispose(); override;

    procedure setup(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata; AInBuilder: IMetadataBuilder;
      AOutBuilder: IMetadataBuilder); override;

    function newItem(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata): IExternalFunction; override;
  end;

  // Factory of external procedures with metadata
  TProcedureFactory<T: TExternalProcedure, constructor> = class
    (IUdrProcedureFactoryImpl)
    procedure dispose(); override;

    procedure setup(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata; AInBuilder: IMetadataBuilder;
      AOutBuilder: IMetadataBuilder); override;

    function newItem(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata): IExternalProcedure; override;
  end;

  // Factory of external triggers with metadata
  TTriggerFactory<T: TExternalTrigger, constructor> = class
    (IUdrTriggerFactoryImpl)
    procedure dispose(); override;

    procedure setup(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata; AFieldsBuilder: IMetadataBuilder); override;

    function newItem(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata): IExternalTrigger; override;
  end;
```

The implementation of the method `newitem` is trivial and is similar to the first case, except that it is necessary to initialize the field with metadan.

```
implementation
...
```

```
{ TFunctionFactory<T> }

procedure TFunctionFactory<T>.dispose;
begin
  Destroy;
end;

function TFunctionFactory<T>.newItem(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata): IExternalFunction;
begin
  Result := T.Create;
  (Result as T).Metadata := AMetadata;
end;

procedure TFunctionFactory<T>.setup(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata;
  AInBuilder, AOutBuilder: IMetadataBuilder);
begin

end;

{ TProcedureFactory<T> }

procedure TProcedureFactory<T>.dispose;
begin
  Destroy;
end;

function TProcedureFactory<T>.newItem(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata): IExternalProcedure;
begin
  Result := T.Create;
  (Result as T).Metadata := AMetadata;
end;

procedure TProcedureFactory<T>.setup(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata;
  AInBuilder, AOutBuilder: IMetadataBuilder);
begin

end;

{ TTriggerFactory<T> }

procedure TTriggerFactory<T>.dispose;
begin
  Destroy;
end;

function TTriggerFactory<T>.newItem(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata): IExternalTrigger;
```

```
begin
  Result := T.Create;
  (Result as T).Metadata := AMetadata;
end;

procedure TTriggerFactory<T>.setup(AStatus: IStatus; AContext: IExternalContext;
  AMetadata: IRoutineMetadata; AFieldsBuilder: IMetadataBuilder);
begin

end;
```

A ready-made module with generic factories can be downloaded at https://github.com/sim1984/udr-book/blob/master/examples/Common/UdrFactories.pas.

Chapter 7. Working with the BLOB type

Unlike other BLOB data types are transmitted by the link (BLOB identifier), and not by value. This is logical, Blob can be enormous, and therefore it is impossible to place them in a fixed width buffer. Instead, the so called BLOB identifier is placed in the message buffer, and working with data of the BLOB type is carried out through the `IBlob` interface.

Another important feature of the BLOB type is that Blob is an unchanged type, you cannot change the contents of the BLOB with a given identifier, instead you need to create BLOB with new contents and the identifier.

Since the size of the BLOB type can be very large, the BLOB data is read and written in portions (segments), the maximum segment size is 64 KB. The segment is read by the `getSegment` interface` Iblob `. The segment is recorded by the `putSegment` interface` Iblob `.

7.1. Reading data from BLOB

As an example of reading a BLOB, consider a procedure that splits string by delimiter (reverse procedure for the built-in aggregate LIST functions). It is declared like this

```
create procedure split (
    txt blob sub_type text character set utf8,
    delimiter char(1) character set utf8 = ','
)
returns (
    id integer
)
external name 'myudr!split'
engine udr;
```

Let's register our procedure factory:

```
function firebird_udr_plugin(AStatus: IStatus; AUnloadFlagLocal: BooleanPtr;
  AUdrPlugin: IUdrPlugin): BooleanPtr; cdecl;
begin
  // register our procedure
  AUdrPlugin.registerProcedure(AStatus, 'split',
TProcedureSimpleFactory<TSplitProcedure>.Create());

  theirUnloadFlag := AUnloadFlagLocal;
  Result := @myUnloadFlag;
end;
```

Here I used a generalized factory for simple cases when the factory simply creates a copy of the procedure without the use of metadata. Such a factory is declared as follows:

```
...
interface

uses SysUtils, Firebird;

type

  TProcedureSimpleFactory<T: IExternalProcedureImpl, constructor> =
  class(IUdrProcedureFactoryImpl)
    procedure dispose(); override;

    procedure setup(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata; AInBuilder: IMetadataBuilder;
      AOutBuilder: IMetadataBuilder); override;

    function newItem(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata): IExternalProcedure; override;
  end;

...

implementation

{ TProcedureSimpleFactory<T> }

procedure TProcedureSimpleFactory<T>.dispose;
begin
  Destroy;
end;

function TProcedureSimpleFactory<T>.newItem(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata): IExternalProcedure;
begin
  Result := T.Create;
end;

procedure TProcedureSimpleFactory<T>.setup(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata; AInBuilder,
  AOutBuilder: IMetadataBuilder);
begin

...
```

Now let's move on to the implementation of the procedure. Let's first declare structures for input and output messages.

```
  TInput = record
    txt: ISC_QUAD;
    txtNull: WordBool;
```

```
    delimiter: array [0 .. 3] of AnsiChar;
    delimiterNull: WordBool;
  end;

  TInputPtr = ^TInput;

  TOutput = record
    Id: Integer;
    Null: WordBool;
  end;

  TOutputPtr = ^TOutput;
```

As you can see, instead of the BLOB value, the Blob identifier is transmitted, which is described by the `ISC_QUAD` structure.

Now let's describe the procedure class and the returned data set:

```
  TSplitProcedure = class(IExternalProcedureImpl)
  private
    procedure SaveBlobToStream(AStatus: IStatus; AContext: IExternalContext;
      ABlobId: ISC_QUADPtr; AStream: TStream);
    function readBlob(AStatus: IStatus; AContext: IExternalContext;
      ABlobId: ISC_QUADPtr): string;
  public
    // Called when destroying a copy of the procedure
    procedure dispose(); override;

    procedure getCharSet(AStatus: IStatus; AContext: IExternalContext;
      AName: PAnsiChar; ANameSize: Cardinal); override;

    function open(AStatus: IStatus; AContext: IExternalContext; AInMsg: Pointer;
      AOutMsg: Pointer): IExternalResultSet; override;
  end;

  TSplitResultSet = class(IExternalResultSetImpl)
{$IFDEF FPC}
    OutputArray: TStringArray;
{$ELSE}
    OutputArray: TArray<string>;
{$ENDIF}
    Counter: Integer;
    Output: TOutputPtr;

    procedure dispose(); override;
    function fetch(AStatus: IStatus): Boolean; override;
  end;
```

Additional `SaveBlobToStream` and` readBlob` are designed to read Blob. The first reads Blob in a

stream, the second is based on the first and performs a convert for the read flow into a Delphi line. The data set of the lines of the `OutputArray` and the counter of the returned records `Counter` are transmitted.

```
In the `open` method, Blob is read and converted into a line.
The resulting line is divided into a separator using the built
-in `split` method from a Hellper for lines. The resulting array
of lines is transmitted to the resulting data set.
```

```
function TSplitProcedure.open(AStatus: IStatus; AContext: IExternalContext;
  AInMsg, AOutMsg: Pointer): IExternalResultSet;
var
  xInput: TInputPtr;
  xText: string;
  xDelimiter: string;
begin
  xInput := AInMsg;

  Result := TSplitResultSet.Create;
  TSplitResultSet(Result).Output := AOutMsg;

  if xInput.txtNull or xInput.delimiterNull then
  begin
    with TSplitResultSet(Result) do
    begin
      // We create an empty array
      OutputArray := [];
      Counter := 1;
    end;
    Exit;
  end;

  xText := readBlob(AStatus, AContext, @xInput.txt);
  xDelimiter := TFBCharSet.CS_UTF8.GetString(TBytes(@xInput.delimiter), 0, 4);
  // automatically is not correctly determined because the lines
  // not completed by zero
  // Place the backing of byte/4
  SetLength(xDelimiter, 1);

  with TSplitResultSet(Result) do
  begin
    OutputArray := xText.Split([xDelimiter], TStringSplitOptions.ExcludeEmpty);
    Counter := 0;
  end;
end;
```

Comment

Type of `TFBCharSet` is not included in `Firebird.pas`. It was written by me to relieve

> work with Firebird encodings. In this case, we believe that all our lines come in the UTF-8 encoding.

Now we will describe the data reading procedure from BLOB to the stream. In order to read data from BLOB, it must be opened. This can be done by calling the `openBlob` method` IAttachment`. Since we read Blob from our database, we will open it in the context of the current connection. The context of the current connection and the context of the current transaction can be obtained from the context of the external procedure, function or trigger (the ` `IEXTERNALCONTEXT`).

Blob is read in portions (segments), the maximum size of the segment is 64 KB. The segment is read by the `getSegment` interface` IBlob`.

```
procedure TSplitProcedure.SaveBlobToStream(AStatus: IStatus;
  AContext: IExternalContext; ABlobId: ISC_QUADPtr; AStream: TStream);
var
  att: IAttachment;
  trx: ITransaction;
  blob: IBlob;
  buffer: array [0 .. 32767] of AnsiChar;
  l: Integer;
begin
  try
    att := AContext.getAttachment(AStatus);
    trx := AContext.getTransaction(AStatus);
    blob := att.openBlob(AStatus, trx, ABlobId, 0, nil);
    while True do
    begin
      case blob.getSegment(AStatus, SizeOf(buffer), @buffer, @l) of
        IStatus.RESULT_OK:
          AStream.WriteBuffer(buffer, l);
        IStatus.RESULT_SEGMENT:
          AStream.WriteBuffer(buffer, l);
      else
        break;
      end;
    end;
    AStream.Position := 0;
    // CLOSE method in case of success combines the IBLOB interface
    // Therefore, the subsequent call is not needed
    blob.close(AStatus);
    blob := nil;
  finally
    if Assigned(blob) then
      blob.release;
    if Assigned(trx) then
      trx.release;
    if Assigned(att) then
      att.release;
  end;
```

```
end;
```

Comment

Please note that the interfaces `IAttachment`, `ITransaction` and `IBlob` inherit the` IReferenCecauted` interface, which means these are objects with the calculation of links. Methods of the returning objects of these interfaces set the link meter in 1. Upon completion of work with such objects, you need to reduce the link counter using the `release` method.

Important

The `close` method of the` IBlob` interface in case of successful execution frees the interface, so there is no need to call the `release` method.

In the example of the variable `blob` assigned the value of `nil`. Further in the `finally` section, whether the pointer is initialized to the `IBlob` interface, and only if the execution was completed earlier than the call `blob.close (AStatus)` or if this challenge ended with an error, `Iblob.release` is called.

On the basis of the `SaveBlobToStream` method, the Blob reading procedure in the line is written:

```
function TSplitProcedure.readBlob(AStatus: IStatus; AContext: IExternalContext;
  ABlobId: ISC_QUADPtr): string;
var
{$IFDEF FPC}
  xStream: TBytesStream;
{$ELSE}
  xStream: TStringStream;
{$ENDIF}
begin
{$IFDEF FPC}
  xStream := TBytesStream.Create(nil);
{$ELSE}
  xStream := TStringStream.Create('', 65001);
{$ENDIF}
  try
    SaveBlobToStream(AStatus, AContext, ABlobId, xStream);
{$IFDEF FPC}
    Result := TEncoding.UTF8.GetString(xStream.Bytes, 0, xStream.Size);
{$ELSE}
    Result := xStream.DataString;
{$ENDIF}
  finally
    xStream.Free;
  end;
end;
```

Comment

> Unfortunately, Free Pascal does not provide full reverse compatibility with Delphi for the `TStringStream` class. In the version for FPC, you cannot specify the encoding with which the stream will work, and therefore it is necessary to process the transformation for it into the line in a special way.

The `fetch` method of the output set of data extracts an element with the `Counter` index from the line and increases it until the last element of the array is extracted. Each extracted line is converted to the whole. If this is impossible to do, then an exception will be excited with the `isc_convert_error` code.

```
procedure TSplitResultSet.dispose;
begin
  SetLength(OutputArray, 0);
  Destroy;
end;

function TSplitResultSet.fetch(AStatus: IStatus): Boolean;
var
  statusVector: array [0 .. 4] of NativeIntPtr;
begin
  if Counter <= High(OutputArray) then
  begin
    Output.Null := False;
    // Exceptions will be intercepted in any case with the ISC_Random code
    // Here we will throw out the standard for Firebird
    // error ISC_CONVERT_ERROR
    try
      Output.Id := OutputArray[Counter].ToInteger();
    except
      on e: EConvertError do
      begin

        statusVector[0] := NativeIntPtr(isc_arg_gds);
        statusVector[1] := NativeIntPtr(isc_convert_error);
        statusVector[2] := NativeIntPtr(isc_arg_string);
        statusVector[3] := NativeIntPtr(PAnsiChar('Cannot convert string to
integer'));
        statusVector[4] := NativeIntPtr(isc_arg_end);

        AStatus.setErrors(@statusVector);
      end;
    end;
    inc(Counter);
    Result := True;
  end
  else
    Result := False;
end;
```

Comment

In fact, the processing of any errors except `isc_random` is not very convenient, you can write your wrapper to simplify.

The performance of the procedure can be checked as follows:

```
SELECT ids.ID
FROM SPLIT((SELECT LIST(ID) FROM MYTABLE), ',') ids
```

Comment

The main drawback of this implementation is that Blob will always be read entirely, even if you want to interrupt the extraction of records from the procedure ahead of schedule. If desired, you can change the procedure code so that smashing into tunes is carried out in smaller portions. To do this, the reading of these portions must be carried out in the `Fetch` method as the result is extracted.

7.2. Data recording in Blob

As an example of Blob recording, consider the function of the reader contents of the Blob from the file.

Comment

This example is an adapted version of UDF functions for reading and recording BLOB from/to a file. Original UDF is available at http://www.ibase.ru/files/download/blobsaveload.zip#blobsaveload.zip]

Blob read and record utilities from/to the file are issued in the form of a package

```
CREATE PACKAGE BlobFileUtils
AS
BEGIN
  PROCEDURE SaveBlobToFile(ABlob BLOB, AFileName VARCHAR(255) CHARACTER SET UTF8);

  FUNCTION LoadBlobFromFile(AFileName VARCHAR(255) CHARACTER SET UTF8) RETURNS BLOB;
END^

CREATE PACKAGE BODY BlobFileUtils
AS
BEGIN
  PROCEDURE SaveBlobToFile(ABlob BLOB, AFileName VARCHAR(255) CHARACTER SET UTF8)
  EXTERNAL NAME 'BlobFileUtils!SaveBlobToFile'
  ENGINE UDR;

  FUNCTION LoadBlobFromFile(AFileName VARCHAR(255) CHARACTER SET UTF8) RETURNS BLOB
  EXTERNAL NAME 'BlobFileUtils!LoadBlobFromFile'
  ENGINE UDR;
```

```
END^
```

Let's register the factories of our procedures and functions:

```
function firebird_udr_plugin(AStatus: IStatus; AUnloadFlagLocal: BooleanPtr;
  AUdrPlugin: IUdrPlugin): BooleanPtr; cdecl;
begin
  // registerable
  AUdrPlugin.registerProcedure(AStatus, 'SaveBlobToFile',
TSaveBlobToFileProcFactory.Create());
  AUdrPlugin.registerFunction(AStatus, 'LoadBlobFromFile',
TLoadBlobFromFileFuncFactory.Create());

  theirUnloadFlag := AUnloadFlagLocal;
  Result := @myUnloadFlag;
end;
```

In this case, we give an example only for the feature reading BLOB from the file, the full example of udr you can download at https://github.com/sim1984/uDr-book/Master/EXAMPles/06.%20BLOBLOBOLOODALOADX . The interface part of the module with a description of the `Loadblobfromfile` function is as follows:

```
interface

uses
  Firebird, Classes, SysUtils;

type

  // Input messages of the function
  TInput = record
    filename: record
      len: Smallint;
      str: array [0 .. 1019] of AnsiChar;
    end;
    filenameNull: WordBool;
  end;
  TInputPtr = ^TInput;

  // Output function
  TOutput = record
    blobData: ISC_QUAD;
    blobDataNull: WordBool;
  end;
  TOutputPtr = ^TOutput;

  // realization features Loadblobfromfile
  TLoadBlobFromFileFunc = class(IExternalFunctionImpl)
```

```
  public
    procedure dispose(); override;

    procedure getCharSet(AStatus: IStatus; AContext: IExternalContext;
      AName: PAnsiChar; ANameSize: Cardinal); override;

    procedure execute(AStatus: IStatus; AContext: IExternalContext;
      AInMsg: Pointer; AOutMsg: Pointer); override;
  end;

  // Factory for creating a copy of the external function Loadblobfromfile
  TLoadBlobFromFileFuncFactory = class(IUdrFunctionFactoryImpl)
    procedure dispose(); override;

    procedure setup(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata; AInBuilder: IMetadataBuilder;
      AOutBuilder: IMetadataBuilder); override;

    function newItem(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata): IExternalFunction; override;
  end;
```

Let us only give the implementation of the basic Execute class `tloadblobfromfile`, the rest of the classes of classes are elementary.

```
procedure TLoadBlobFromFileFunc.execute(AStatus: IStatus;
  AContext: IExternalContext; AInMsg: Pointer; AOutMsg: Pointer);
const
  MaxBufSize = 16384;
var
  xInput: TInputPtr;
  xOutput: TOutputPtr;
  xFileName: string;
  xStream: TFileStream;
  att: IAttachment;
  trx: ITransaction;
  blob: IBlob;
  buffer: array [0 .. 32767] of Byte;
  xStreamSize: Integer;
  xBufferSize: Integer;
  xReadLength: Integer;
begin
  xInput := AInMsg;
  xOutput := AOutMsg;
  if xInput.filenameNull then
  begin
    xOutput.blobDataNull := True;
    Exit;
  end;
  xOutput.blobDataNull := False;
```

```
  // We get the file name
  xFileName := TEncoding.UTF8.GetString(TBytes(@xInput.filename.str), 0,
    xInput.filename.len * 4);
  SetLength(xFileName, xInput.filename.len);
  // We read the file in the stream
  xStream := TFileStream.Create(xFileName, fmOpenRead or fmShareDenyNone);
  att := AContext.getAttachment(AStatus);
  trx := AContext.getTransaction(AStatus);
  blob := nil;
  try
    xStreamSize := xStream.Size;
    // Determine the maximum size of the buffer (segment)
    if xStreamSize > MaxBufSize then
      xBufferSize := MaxBufSize
    else
      xBufferSize := xStreamSize;
    // We create a new Blob
    blob := att.createBlob(AStatus, trx, @xOutput.blobData, 0, nil);
    // We read the contents of the stream and write it in Blob
    while xStreamSize <> 0 do
    begin
      if xStreamSize > xBufferSize then
        xReadLength := xBufferSize
      else
        xReadLength := xStreamSize;
      xStream.ReadBuffer(buffer, xReadLength);

      blob.putSegment(AStatus, xReadLength, @buffer[0]);

      Dec(xStreamSize, xReadLength);
    end;
    // Close Blob
    // CLOSE method in case of success combines the IBLOB interface
    // Therefore, the subsequent call is not needed
    blob.close(AStatus);
    blob := nil;
  finally
    if Assigned(blob) then
      blob.release;
    trx.release;
    att.release;
    xStream.Free;
  end;
end;
```

First of all, it is necessary to create a new Blob and tie it in the Blobid output using the `createBlob` method `IAttachment`. Since we write a temporary Blob for our database, we will create it in the context of the current connection. The context of the current connection and the context of the current transaction can be obtained from the context of the external procedure, function or trigger (the `IExternalContext`).

As in the case of reading data from Blob, the record is carried out by segmented using the `putSegment` method `IBlob` until the data in the file flow is completed. Upon completion of the data recording in Blob, it is necessary to close it using the `close` method.

Important

The `close` method of the` IBlob` interface in case of successful execution frees the interface. Therefore, there is no need to call the `Release 'method.

7.3. Helper for working with the Blob type

In the described examples, we used the preservation of BLOB contents into the flow, as well as the loading of the contents of BLOB into the stream. This is a rather frequent operation when working with the BLOB type, so it would be good to write a special set of utilities for reuse of code.

Modern versions of Delphi and Free Pascal allow you to expand existing classes and types without inheritance using the so called Halper. Add the methods to the IBlob interface to save and load the contents of the flow from/to Blob.

Create a special module `FbBlob`, where our Halper will be placed.

```
unit FbBlob;

interface

uses Classes, SysUtils, Firebird;

const
  MAX_SEGMENT_SIZE = $7FFF;

type
  TFbBlobHelper = class helper for IBlob
    { Loads in Blob the contents of the stream

      @param(AStatus Статус вектор)
      @param(AStream Поток)
    }
    procedure LoadFromStream(AStatus: IStatus; AStream: TStream);
    { Loads BLOB contents into the stream

      @param(AStatus Статус вектор)
      @param(AStream Поток)
    }
    procedure SaveToStream(AStatus: IStatus; AStream: TStream);
  end;

implementation

uses Math;
```

```
procedure TFbBlobHelper.LoadFromStream(AStatus: IStatus; AStream: TStream);
var
  xStreamSize: Integer;
  xReadLength: Integer;
  xBuffer: array [0 .. MAX_SEGMENT_SIZE] of Byte;
begin
  xStreamSize := AStream.Size;
  AStream.Position := 0;
  while xStreamSize <> 0 do
  begin
    xReadLength := Min(xStreamSize, MAX_SEGMENT_SIZE);
    AStream.ReadBuffer(xBuffer, xReadLength);
    Self.putSegment(AStatus, xReadLength, @xBuffer[0]);
    Dec(xStreamSize, xReadLength);
  end;
end;

procedure TFbBlobHelper.SaveToStream(AStatus: IStatus; AStream: TStream);
var
  xInfo: TFbBlobInfo;
  Buffer: array [0 .. MAX_SEGMENT_SIZE] of Byte;
  xBytesRead: Cardinal;
  xBufferSize: Cardinal;
begin
  AStream.Position := 0;
  xBufferSize := Min(SizeOf(Buffer), MAX_SEGMENT_SIZE);
  while True do
  begin
    case Self.getSegment(AStatus, xBufferSize, @Buffer[0], @xBytesRead) of
      IStatus.RESULT_OK:
        AStream.WriteBuffer(Buffer, xBytesRead);
      IStatus.RESULT_SEGMENT:
        AStream.WriteBuffer(Buffer, xBytesRead);
    else
      break;
    end;
  end;
end;

end.
```

Now you can greatly simplify operations with the BLOB type, for example, the above function of saving Blob to the file can be rewritten as follows:

```
procedure TLoadBlobFromFileFunc.execute(AStatus: IStatus;
  AContext: IExternalContext; AInMsg: Pointer; AOutMsg: Pointer);
var
  xInput: TInputPtr;
  xOutput: TOutputPtr;
  xFileName: string;
```

```
  xStream: TFileStream;
  att: IAttachment;
  trx: ITransaction;
  blob: IBlob;
begin
  xInput := AInMsg;
  xOutput := AOutMsg;
  if xInput.filenameNull then
  begin
    xOutput.blobDataNull := True;
    Exit;
  end;
  xOutput.blobDataNull := False;
  // We get the file name
  xFileName := TEncoding.UTF8.GetString(TBytes(@xInput.filename.str), 0,
    xInput.filename.len * 4);
  SetLength(xFileName, xInput.filename.len);
  // We read the file in the stream
  xStream := TFileStream.Create(xFileName, fmOpenRead or fmShareDenyNone);
  att := AContext.getAttachment(AStatus);
  trx := AContext.getTransaction(AStatus);
  blob := nil;
  try
    // We create a new Blob
    blob := att.createBlob(AStatus, trx, @xOutput.blobData, 0, nil);
    // We load the contents of the flow into Blob
    blob.LoadFromStream(AStatus, xStream);
    // Close Blob
    // CLOSE method in case of success combines the IBLOB interface
    // Therefore, the subsequent call is not needed
    blob.close(AStatus);
    blob := nil;
  finally
    if Assigned(blob) then
      blob.release;
    att.release;
    trx.release;
    xStream.Free;
  end;
end;
```

Chapter 8. Connection and transaction context

If your external procedure, function or trigger should receive data from your own database not through input arguments, but for example through a request, then you will need to receive the context of the current connection and/or transactions. In addition, the context of the connection and transaction is necessary if you work with the BLOB type.

The context of the current procedure, function or trigger is transmitted as a parameter with the type of `IExternalContext` into the `execute` trigger method or function, or in the open procedure method. The `IExternalContext` interface allows you to get the current connection using the `getAttachment` method, and the current transaction using the `getTransaction` method. This gives greater flexibility to your UDR, for example, you can fulfill the current database requests while maintaining the current session environment, in the same transaction or in a new transaction created using the StartTransaction `IExternalContext` interface method. In the latter case, the request will be made as if it is executed in an autonomous transaction. In addition, you can comply with the external database using the transaction attached to the current transaction, i.e. Transactions with two phase confirmation (2PC).

As an example of working with the context of the function of the function, we will write a function that will serialize the result of the execution of `SELECT` request in JSON format. It is declared as follows:

```
create function GetJson (
    sql_text blob sub_type text character set utf8,
    sql_dialect smallint not null default 3
)
returns returns blob sub_type text character set utf8
external name 'JsonUtils!getJson'
engine udr;
```

Since we allow us to execute an arbitrary SQL request, we do not know in advance the format of the output fields, and we will not be able to use a structure with fixed fields. In this case, we will have to work with the `IMessageMetadata` interface. We have already encountered it earlier, but this time we will have to work with it more thoroughly, since we must process all the existing Firebird types.

Comment

In JSON, you can encode almost any type of data except binary. For coding the types of `char`, `varchar` with `octets none` and `blob sub_type binary` we will encode binary contents using base64 coding, which can already be placed in JSON.

We will register the factory of our function:

```
function firebird_udr_plugin(AStatus: IStatus; AUnloadFlagLocal: BooleanPtr;
  AUdrPlugin: IUdrPlugin): BooleanPtr; cdecl;
```

```
begin
  // We register a function
  AUdrPlugin.registerFunction(AStatus, 'getJson',
TFunctionSimpleFactory<TJsonFunction>.Create());

  theirUnloadFlag := AUnloadFlagLocal;
  Result := @myUnloadFlag;
end;
```

Now we will declare structures for the input and output message, as well as the interface part of our function:

```
unit JsonFunc;

{$IFDEF FPC}
{$MODE objfpc}{$H+}
{$DEFINE DEBUGFPC}
{$ENDIF}

interface

uses
  Firebird,
  UdrFactories,
  FbTypes,
  FbCharsets,
  SysUtils,
  System.NetEncoding,
  System.Json;

{ ************************************************************
 create function GetJson (
   sql_text blob sub_type text,
   sql_dialect smallint not null default 3
 ) returns blob sub_type text character set utf8
 external name 'JsonUtils!getJson'
 engine udr;
 ************************************************************ }

type

  TInput = record
    SqlText: ISC_QUAD;
    SqlNull: WordBool;
    SqlDialect: Smallint;
    SqlDialectNull: WordBool;
  end;

  InputPtr = ^TInput;
```

```
TOutput = record
  Json: ISC_QUAD;
  NullFlag: WordBool;
end;

OutputPtr = ^TOutput;

// External Tsumargsfunction function.
TJsonFunction = class(IExternalFunctionImpl)
public
  procedure dispose(); override;

  procedure getCharSet(AStatus: IStatus; AContext: IExternalContext;
    AName: PAnsiChar; ANameSize: Cardinal); override;

  { Converts the whole into a line in accordance with the scale

    @param(AValue Meaning)
    @param(Scale Scale)
    @returns(Strokal representation of a scaled whole)
  }
  function MakeScaleInteger(AValue: Int64; Scale: Smallint): string;

  { Adds an encoded entry to an array of JSON objects

    @param(AStatus Vecto statusp)
    @param(AContext The context of external function)
    @param(AJson An array of JSON objects)
    @param(ABuffer Buffer records)
    @param(AMeta Metadata cursor)
    @param(AFormatSetting Setting date and time)
  }
  procedure writeJson(AStatus: IStatus; AContext: IExternalContext;
    AJson: TJsonArray; ABuffer: PByte; AMeta: IMessageMetadata;
    AFormatSettings: TFormatSettings);

  { External function

    @param (AStatus status vector)
    @PARAM (ACONTEXT Context of external function)
    @param (AINMSG input message)
    @PARAM (AUTMSG Office for output)
  }
  procedure execute(AStatus: IStatus; AContext: IExternalContext;
    AInMsg: Pointer; AOutMsg: Pointer); override;
end;
```

The additional method of `MakeScaleInteger` is designed to convert scalable numbers into a line, the ` Writejson method encods the next recording of the object selected from the cursor to Json and adds it to the massif of such objects.

In this example, we will need to implement the `getCharSet` method to indicate the desired encoding of the request for the request of the current connection within the external function. By default, this internal request will be carried out in the encoding of the current connection. However, this is not entirely convenient. We do not know in advance what encoding the client will work, so we will have to determine the encoding of each returned string field and transcode into UTF8. To simplify the task, we will immediately indicate to the context that we are going to work inside the procedure in UTF8 encoding.

```
procedure TJsonFunction.getCharSet(AStatus: IStatus; AContext: IExternalContext;
  AName: PAnsiChar; ANameSize: Cardinal);
begin
  // grind the previous encoding
  Fillchar (aname, anamesize, #0);
  // put the desired encoding
  Strcopy (aname, 'UTF8');
end;
```

We will describe these methods later, but for now we will give the main method of `execute` to perform an external function.

```
procedure TJsonFunction.execute(AStatus: IStatus; AContext: IExternalContext;
  AInMsg, AOutMsg: Pointer);
var
  xFormatSettings: TFormatSettings;
  xInput: InputPtr;
  xOutput: OutputPtr;
  att: IAttachment;
  tra: ITransaction;
  stmt: IStatement;
  inBlob, outBlob: IBlob;
  inStream: TBytesStream;
  outStream: TStringStream;
  cursorMetaData: IMessageMetadata;
  rs: IResultSet;
  msgLen: Cardinal;
  msg: Pointer;
  jsonArray: TJsonArray;
begin
  xInput := AInMsg;
  xOutput := AOutMsg;
  // If one of the input arguments is null, then the result is null
  if xInput.SqlNull or xInput.SqlDialectNull then
  begin
    xOutput.NullFlag := True;
    Exit;
  end;
  xOutput.NullFlag := False;
  // setting date and time formatting
{$IFNDEF FPC}
```

```
  xFormatSettings := TFormatSettings.Create;
{$ELSE}
  xFormatSettings := DefaultFormatSettings;
{$ENDIF}
  xFormatSettings.DateSeparator := '-';
  xFormatSettings.TimeSeparator := ':';
  // We create a byte stream for Blob reading
  inStream := TBytesStream.Create(nil);
{$IFNDEF FPC}
  outStream := TStringStream.Create('', 65001);
{$ELSE}
  outStream := TStringStream.Create('');
{$ENDIF}
  jsonArray := TJsonArray.Create;
  // obtaining the current connection and transaction
  att := AContext.getAttachment(AStatus);
  tra := AContext.getTransaction(AStatus);
  stmt := nil;
  rs := nil;
  inBlob := nil;
  outBlob := nil;
  try
    // We read Blob in a stream
    inBlob := att.openBlob(AStatus, tra, @xInput.SqlText, 0, nil);
    inBlob.SaveToStream(AStatus, inStream);
    // The Close method, if successful, combines the IBLOB interface
    // Therefore, the subsequent call is not needed
    inBlob.close(AStatus);
    inBlob := nil;
    // Prepare the operator
    stmt := att.prepare(AStatus, tra, inStream.Size, @inStream.Bytes[0],
      xInput.SqlDialect, IStatement.PREPARE_PREFETCH_METADATA);
    // We get a weekend of metadata cursor
    cursorMetaData := stmt.getOutputMetadata(AStatus);
    // We're getting off the cursor
    rs := stmt.openCursor(AStatus, tra, nil, nil, nil, 0);
    // We highlight the buffer of the desired size
    msgLen := cursorMetaData.getMessageLength(AStatus);
    msg := AllocMem(msgLen);
    try
      // We read each cursor record
      while rs.fetchNext(AStatus, msg) = IStatus.RESULT_OK do
      begin
        // and write it in json
        writeJson(AStatus, AContext, jsonArray, msg, cursorMetaData,
          xFormatSettings);
      end;
    finally
      // We release the buffer
      FreeMem(msg);
    end;
```

```
    // Close the cursor
    // CLOSE method in case of success combines the IRESULTSET interface
    // Therefore, the subsequent call is not needed
    rs.close(AStatus);
    rs := nil;
    // We release the prepared request
    // Free method, in case of success, combines the ISTATEMENT interface
    // Therefore, the subsequent call is not needed
    stmt.free(AStatus);
    stmt := nil;
    // We write json in stream
{$IFNDEF FPC}
    outStream.WriteString(jsonArray.ToJSON);
{$ELSE}
    outStream.WriteString(jsonArray.AsJSON);
{$ENDIF}
    // We write json on the blany Blob
    outBlob := att.createBlob(AStatus, tra, @xOutput.Json, 0, nil);
    outBlob.LoadFromStream(AStatus, outStream);
    // CLOSE method in case of success combines the IBLOB interface
    // Therefore, the subsequent call is not needed
    outBlob.close(AStatus);
    outBlob := nil;
  finally
    if Assigned(inBlob) then
      inBlob.release;
    if Assigned(stmt) then
      stmt.release;
    if Assigned(rs) then
      rs.release;
    if Assigned(outBlob) then
      outBlob.release;
    tra.release;
    att.release;
    jsonArray.Free;
    inStream.Free;
    outStream.Free;
  end;
end;
```

First of all, we get a current connection from the context of the function and the current transaction using the `getAttachment` and `getTransaction` methods of interface `IExternalContext`. Then we read the contents of the BLOB for obtaining the text of the SQL request. The request is prepared using the Prepare method of the `IAttachment` interface. The fifth parameter is transmitted by SQL dialect obtained from the input parameter of our function. The sixth parameter we convey the flag `IStatement.PREPARE_PREFETCH_METADATA`, which means that we want to get a metadata cursor along with the result of the preparation of the request. We get the weekend of the metadata cursor using the `getOutputMetadata` interface` IStatement`.

Comment

> In fact, the getoutPutmetadata method will return the weekend metadata in any case. The flag IStatement.PREPARE_PREFETCH_METADATA will force metadata along with the result of preparing a request for one network package. Since we comply with a request within the current connection of any network exchange, and this is not fundamentally.

Next, open the cursor using the openCursor method as part of the current transaction (parameter 2). We get the size of the output buffer to the result of the cursor using the getMessageLength interface` IMessageMetadata`. This allows you to highlight the memory under the buffer, which we will free immediately after the latching of the last recording of the cursor.

The cursor records are read using the fetchNext method from IResultSet. This method fills the msg buffer with the values of the cursor fields and returns IStatus.RESULT_OK until the cursor records are over. Each record read is transmitted to the Writejson method, which adds an object like TJsonObject with a serialized cursor recording in the TJsonArray array.

After completing the work with the cursor, we close it by the close method, convert an array of json objects into a line, write it to the output stream, which we write down in the Blob output.

Now let's analyze the writeJson method. The IUtil object will need us in order to receive functions to decode the date and time. This method actively involves working with metadata output fields of the cursor using the IMessageMetadata interface. First of all, we create an object type TJsonObject into which we will record the values of the fields of the current record. As the names of the keys, we will use the alias of fields from the cursor. If Nullflag is installed, then we write the value of NULL for the key and go to the next field, otherwise we analyze the field type and write its value in JSON.

```
function TJsonFunction.MakeScaleInteger(AValue: Int64; Scale: Smallint): string;
var
  L: Integer;
begin
  Result := AValue.ToString;
  L := Result.Length;
  if (-Scale >= L) then
    Result := '0.' + Result.PadLeft(-Scale, '0')
  else
    Result := Result.Insert(Scale + L, '.');
end;


procedure TJsonFunction.writeJson(AStatus: IStatus; AContext: IExternalContext;
  AJson: TJsonArray; ABuffer: PByte; AMeta: IMessageMetadata;
  AFormatSettings: TFormatSettings);
var
  jsonObject: TJsonObject;
  i: Integer;
  FieldName: string;
  NullFlag: WordBool;
  fieldType: Cardinal;
```

```
  pData: PByte;
  util: IUtil;
  metaLength: Integer;
  // types
  CharBuffer: TBytes;
  charLength: Smallint;
  charset: TFBCharSet;
  StringValue: string;
  SmallintValue: Smallint;
  IntegerValue: Integer;
  BigintValue: Int64;
  Scale: Smallint;
  SingleValue: Single;
  DoubleValue: Double;
  Dec16Value: FB_DEC16Ptr;
  xDec16Buf: array[0..IDecFloat16.STRING_SIZE-1] of AnsiChar;
  xDecFloat16: IDecFloat16;
  Dec34Value: FB_DEC34Ptr;
  xDec34Buf: array[0..IDecFloat34.STRING_SIZE-1] of AnsiChar;
  xDecFloat34: IDecFloat34;
  BooleanValue: Boolean;
  DateValue: ISC_DATE;
  TimeValue: ISC_TIME;
  TimeValueTz: ISC_TIME_TZPtr;
  TimestampValue: ISC_TIMESTAMP;
  TimestampValueTz: ISC_TIMESTAMP_TZPtr;
  tzBuffer: array[0..63] of AnsiChar;
  DateTimeValue: TDateTime;
  year, month, day: Cardinal;
  hours, minutes, seconds, fractions: Cardinal;
  blobId: ISC_QUADPtr;
  BlobSubtype: Smallint;
  att: IAttachment;
  tra: ITransaction;
  blob: IBlob;
  textStream: TStringStream;
  binaryStream: TBytesStream;
{$IFDEF FPC}
  base64Stream: TBase64EncodingStream;
  xFloatJson: TJSONFloatNumber;
{$ENDIF}
  xInt128: IInt128;
  Int128Value: FB_I128Ptr;
  xInt128Buf: array[0..IInt128.STRING_SIZE-1] of AnsiChar;
begin
  // We get ITIL
  util := AContext.getMaster().getUtilInterface();
  // We create an object of Tjsonobject in which we will
  // Write the value of the recording fields
  jsonObject := TJsonObject.Create;
  for i := 0 to AMeta.getCount(AStatus) - 1 do
```

```
  begin
    // We get Alias Fields in the request
    FieldName := AMeta.getAlias(AStatus, i);
    NullFlag := PWordBool(ABuffer + AMeta.getNullOffset(AStatus, i))^;
    if NullFlag then
    begin
      // If Null we write it in json and move on to the next field
{$IFNDEF FPC}
      jsonObject.AddPair(FieldName, TJsonNull.Create);
{$ELSE}
      jsonObject.Add(FieldName, TJsonNull.Create);
{$ENDIF}
      continue;
    end;
    // We get a pointer to these fields
    pData := ABuffer + AMeta.getOffset(AStatus, i);
    // аналог AMeta->getType(AStatus, i) & ~1
    fieldType := AMeta.getType(AStatus, i) and not 1;
    case fieldType of
      // VARCHAR
      SQL_VARYING:
        begin
          // Boofer size for Varchar
          metaLength := AMeta.getLength(AStatus, i);
          SetLength(CharBuffer, metaLength);
          charset := TFBCharSet(AMeta.getCharSet(AStatus, i));
          charLength := PSmallint(pData)^;
          // Binary data is encoded in Base64
          if charset = CS_BINARY then
          begin
{$IFNDEF FPC}
            StringValue := TNetEncoding.base64.EncodeBytesToString((pData + 2),
              charLength);
{$ELSE}
            // For Varchar first 2 bytes - length in bytes
            // therefore copy to the buffer starting with 3 bytes
            Move((pData + 2)^, CharBuffer[0], metaLength);
            StringValue := charset.GetString(CharBuffer, 0, charLength);
            StringValue := EncodeStringBase64(StringValue);
{$ENDIF}
          end
          else
          begin
            // For Varchar first 2 bytes - length in bytes
            // therefore copy to the buffer starting with 3 bytes
            Move((pData + 2)^, CharBuffer[0], metaLength);
            StringValue := charset.GetString(CharBuffer, 0, charLength);
          end;
{$IFNDEF FPC}
          jsonObject.AddPair(FieldName, StringValue);
{$ELSE}
```

```
          jsonObject.Add(FieldName, StringValue);
{$ENDIF}
        end;
      // CHAR
      SQL_TEXT:
        begin
          // Boofer size for Char
          metaLength := AMeta.getLength(AStatus, i);
          SetLength(CharBuffer, metaLength);
          charset := TFBCharSet(AMeta.getCharSet(AStatus, i));
          Move(pData^, CharBuffer[0], metaLength);
          // Binary data encoded in Base64
          if charset = CS_BINARY then
          begin
{$IFNDEF FPC}
            StringValue := TNetEncoding.base64.EncodeBytesToString(pData,
              metaLength);
{$ELSE}
            StringValue := charset.GetString(CharBuffer, 0, metaLength);
            StringValue := EncodeStringBase64(StringValue);
{$ENDIF}
          end
          else
          begin
            StringValue := charset.GetString(CharBuffer, 0, metaLength);
            charLength := metaLength div charset.GetCharWidth;
            SetLength(StringValue, charLength);
          end;
{$IFNDEF FPC}
          jsonObject.AddPair(FieldName, StringValue);
{$ELSE}
          jsonObject.Add(FieldName, StringValue);
{$ENDIF}
        end;
      // FLOAT
      SQL_FLOAT:
        begin
          SingleValue := PSingle(pData)^;
{$IFNDEF FPC}
          jsonObject.AddPair(FieldName, TJSONNumber.Create(SingleValue));
{$ELSE}
          jsonObject.Add(FieldName, TJSONFloatNumber.Create(SingleValue));
{$ENDIF}
        end;
      // DOUBLE PRECISION
      // DECIMAL(p, s), where p = 10..15 in 1 dialect
      SQL_DOUBLE, SQL_D_FLOAT:
        begin
          DoubleValue := PDouble(pData)^;
{$IFNDEF FPC}
          jsonObject.AddPair(FieldName, TJSONNumber.Create(DoubleValue));
```

```
{$ELSE}
          jsonObject.Add(FieldName, TJSONFloatNumber.Create(DoubleValue));
{$ENDIF}
        end;
      // DECFLOAT(16)
      SQL_DEC16:
        begin
          Dec16Value := FB_Dec16Ptr(pData);
          xDecFloat16 := util.getDecFloat16(AStatus);
          xDecFloat16.toString(AStatus, Dec16Value, IDecFloat16.STRING_SIZE,
@xDec16Buf[0]);
          StringValue := AnsiString(@xDec16Buf[0]);
          StringValue := Trim(StringValue);
{$IFNDEF FPC}
          jsonObject.AddPair(FieldName, StringValue);
{$ELSE}
          jsonObject.Add(FieldName, StringValue);
{$ENDIF}
        end;
      // DECFLOAT(34)
      SQL_DEC34:
        begin
          Dec34Value := FB_Dec34Ptr(pData);
          xDecFloat34 := util.getDecFloat34(AStatus);
          xDecFloat34.toString(AStatus, Dec34Value, IDecFloat34.STRING_SIZE,
@xDec34Buf[0]);
          StringValue := AnsiString(@xDec34Buf[0]);
          StringValue := Trim(StringValue);
{$IFNDEF FPC}
          jsonObject.AddPair(FieldName, StringValue);
{$ELSE}
          jsonObject.Add(FieldName, StringValue);
{$ENDIF}
        end;
      // INTEGER
      // NUMERIC(p, s), где p = 1..4
      SQL_SHORT:
        begin
          Scale := AMeta.getScale(AStatus, i);
          SmallintValue := PSmallint(pData)^;
          if (Scale = 0) then
          begin
{$IFNDEF FPC}
            jsonObject.AddPair(FieldName, TJSONNumber.Create(SmallintValue));
{$ELSE}
            jsonObject.Add(FieldName, SmallintValue);
{$ENDIF}
          end
          else
          begin
            StringValue := MakeScaleInteger(SmallintValue, Scale);
```

```
{$IFNDEF FPC}
            jsonObject.AddPair(FieldName, TJSONNumber.Create(StringValue));
{$ELSE}
            xFloatJson := TJSONFloatNumber.Create(0);
            xFloatJson.AsString := StringValue;
            jsonObject.Add(FieldName, xFloatJson);
{$ENDIF}
          end;
        end;
      // INTEGER
      // NUMERIC(p, s), где p = 5..9
      // DECIMAL(p, s), где p = 1..9
      SQL_LONG:
        begin
          Scale := AMeta.getScale(AStatus, i);
          IntegerValue := PInteger(pData)^;
          if (Scale = 0) then
          begin
{$IFNDEF FPC}
            jsonObject.AddPair(FieldName, TJSONNumber.Create(IntegerValue));
{$ELSE}
            jsonObject.Add(FieldName, IntegerValue);
{$ENDIF}
          end
          else
          begin
            StringValue := MakeScaleInteger(IntegerValue, Scale);
{$IFNDEF FPC}
            jsonObject.AddPair(FieldName, TJSONNumber.Create(StringValue));
{$ELSE}
            xFloatJson := TJSONFloatNumber.Create(0);
            xFloatJson.AsString := StringValue;
            jsonObject.Add(FieldName, xFloatJson);
{$ENDIF}
          end;
        end;
      // BIGINT
      // NUMERIC(p, s), where p = 10..18 in dialect 3
      // DECIMAL(p, s), where p = 10..18 in dialect 3
      SQL_INT64:
        begin
          Scale := AMeta.getScale(AStatus, i);
          BigintValue := Pint64(pData)^;
          if (Scale = 0) then
          begin
{$IFNDEF FPC}
            jsonObject.AddPair(FieldName, TJSONNumber.Create(BigintValue));
{$ELSE}
            jsonObject.Add(FieldName, BigintValue);
{$ENDIF}
          end
```

```
          else
          begin
            StringValue := MakeScaleInteger(BigintValue, Scale);
{$IFNDEF FPC}
            jsonObject.AddPair(FieldName, TJSONNumber.Create(StringValue));
{$ELSE}
            xFloatJson := TJSONFloatNumber.Create(0);
            xFloatJson.AsString := StringValue;
            jsonObject.Add(FieldName, xFloatJson);
{$ENDIF}
          end;
        end;
      SQL_INT128:
        begin
          Scale := AMeta.getScale(AStatus, i);
          Int128Value := FB_I128Ptr(pData);
          xInt128 := util.getInt128(AStatus);
          xInt128.toString(AStatus, Int128Value, Scale, IInt128.STRING_SIZE,
@xInt128Buf[0]);
          StringValue := AnsiString(@xInt128Buf[0]);
          StringValue := Trim(StringValue);
{$IFNDEF FPC}
          jsonObject.AddPair(FieldName, StringValue);
{$ELSE}
          jsonObject.Add(FieldName, StringValue);
{$ENDIF}
        end;
      // TIMESTAMP
      SQL_TIMESTAMP:
        begin
          TimestampValue := PISC_TIMESTAMP(pData)^;
          // we get the components of the date-time
          util.decodeDate(TimestampValue.timestamp_date, @year, @month, @day);
          util.decodeTime(TimestampValue.timestamp_time, @hours, @minutes, @seconds,
            @fractions);
          // We get a date-time in our delphi type
          DateTimeValue := EncodeDate(year, month, day) +
            EncodeTime(hours, minutes, seconds, fractions div 10);
          // We format a date-time according to a given format
          StringValue := FormatDateTime('yyyy/mm/dd hh:nn:ss', DateTimeValue,
            AFormatSettings);
{$IFNDEF FPC}
          jsonObject.AddPair(FieldName, StringValue);
{$ELSE}
          jsonObject.Add(FieldName, StringValue);
{$ENDIF}
        end;
      // TIMESTAMP WITH TIME_ZONE
      SQL_TIMESTAMP_TZ:
        begin
          TimestampValueTz := ISC_TIMESTAMP_TZPtr(pData);
```

```
          // We get the components of the date-time and the time zone
          util.decodeTimeStampTz(AStatus, TimestampValueTz, @year, @month, @day,
@hours, @minutes, @seconds,
            @fractions, 64, @tzBuffer[0]);

          // We get a date-time in our delphi type
          DateTimeValue := EncodeDate(year, month, day) +
            EncodeTime(hours, minutes, seconds, fractions div 10);
          // Format the date-time according to the given format + time zone
          StringValue := FormatDateTime('yyyy/mm/dd hh:nn:ss', DateTimeValue,
            AFormatSettings) + ' ' + AnsiString(@tzBuffer[0]);
{$IFNDEF FPC}
          jsonObject.AddPair(FieldName, StringValue);
{$ELSE}
          jsonObject.Add(FieldName, StringValue);
{$ENDIF}
        end;
      // DATE
      SQL_DATE:
        begin
          DateValue := PISC_DATE(pData)^;
          // We get the components of the date
          util.decodeDate(DateValue, @year, @month, @day);
          // We get a date in the native type Delphi
          DateTimeValue := EncodeDate(year, month, day);
          // We format the date according to the given format
          StringValue := FormatDateTime('yyyy/mm/dd', DateTimeValue,
            AFormatSettings);
{$IFNDEF FPC}
          jsonObject.AddPair(FieldName, StringValue);
{$ELSE}
          jsonObject.Add(FieldName, StringValue);
{$ENDIF}
        end;
      // TIME
      SQL_TIME:
        begin
          TimeValue := PISC_TIME(pData)^;
          // We get the components of the time
          util.decodeTime(TimeValue, @hours, @minutes, @seconds, @fractions);
          // We get time in the native type Delphi
          DateTimeValue := EncodeTime(hours, minutes, seconds,
            fractions div 10);
          // We format the time according to a given format
          StringValue := FormatDateTime('hh:nn:ss', DateTimeValue,
            AFormatSettings);
{$IFNDEF FPC}
          jsonObject.AddPair(FieldName, StringValue);
{$ELSE}
          jsonObject.Add(FieldName, StringValue);
{$ENDIF}
```

```
        end;
      // TIME WITH TIME ZONE
      SQL_TIME_TZ:
        begin
          TimeValueTz := ISC_TIME_TZPtr(pData);
          // We get the components of the time and the time zone
          util.decodeTimeTz(AStatus, TimeValueTz, @hours, @minutes, @seconds,
            @fractions, 64, @tzBuffer[0]);
          // We get time in the native type Delphi
          DateTimeValue := EncodeTime(hours, minutes, seconds,
            fractions div 10);
          // We format the time according to a given format + time zone
          StringValue := FormatDateTime('hh:nn:ss', DateTimeValue,
            AFormatSettings) + ' ' + AnsiString(@tzBuffer[0]);
{$IFNDEF FPC}
          jsonObject.AddPair(FieldName, StringValue);
{$ELSE}
          jsonObject.Add(FieldName, StringValue);
{$ENDIF}
        end;
      // BOOLEAN
      SQL_BOOLEAN:
        begin
          BooleanValue := PBoolean(pData)^;
{$IFNDEF FPC}
          jsonObject.AddPair(FieldName, TJsonBool.Create(BooleanValue));
{$ELSE}
          jsonObject.Add(FieldName, BooleanValue);
{$ENDIF}
        end;
      // BLOB
      SQL_BLOB, SQL_QUAD:
        begin
          BlobSubtype := AMeta.getSubType(AStatus, i);
          blobId := ISC_QUADPtr(pData);
          att := AContext.getAttachment(AStatus);
          tra := AContext.getTransaction(AStatus);
          blob := att.openBlob(AStatus, tra, blobId, 0, nil);
          try
            if BlobSubtype = 1 then
            begin
              // lyrics
              charset := TFBCharSet(AMeta.getCharSet(AStatus, i));
              // Create a stream with a given encoding
{$IFNDEF FPC}
              textStream := TStringStream.Create('', charset.GetCodePage);
              try
                blob.SaveToStream(AStatus, textStream);
                blob.close(AStatus);
                blob := nil;
                StringValue := textStream.DataString;
```

```
              finally
                textStream.Free;
              end;
{$ELSE}
              binaryStream := TBytesStream.Create(nil);
              try
                blob.SaveToStream(AStatus, binaryStream);
                blob.close(AStatus);
                blob := nil;
                StringValue := TEncoding.UTF8.GetString(binaryStream.Bytes, 0,
                  binaryStream.Size);
              finally
                binaryStream.Free;
              end;
{$ENDIF}
            end
            else
            begin
{$IFNDEF FPC}
              // all other subtypes are considered binary
              binaryStream := TBytesStream.Create;
              try
                blob.SaveToStream(AStatus, binaryStream);
                blob.close(AStatus);
                blob := nil;
                // encode the string in base64
                StringValue := TNetEncoding.base64.EncodeBytesToString
                  (binaryStream.Memory, binaryStream.Size);
              finally
                binaryStream.Free;
              end
{$ELSE}
              textStream := TStringStream.Create('');
              base64Stream := TBase64EncodingStream.Create(textStream);
              try
                blob.SaveToStream(AStatus, base64Stream);
                blob.close(AStatus);
                blob := nil;
                StringValue := textStream.DataString;
              finally
                base64Stream.Free;
                textStream.Free;
              end;
{$ENDIF}
            end;
          finally
            if Assigned(blob) then blob.release;
            if Assigned(tra) then tra.release;
            if Assigned(att) then att.release;
          end;
{$IFNDEF FPC}
```

```
          jsonObject.AddPair(FieldName, StringValue);
{$ELSE}
          jsonObject.Add(FieldName, StringValue);
{$ENDIF}
        end;
    end;
  end;
  // Adding an entry in json format to array
{$IFNDEF FPC}
  AJson.AddElement(jsonObject);
{$ELSE}
  AJson.Add(jsonObject);
{$ENDIF}
end;
```

Comment

Listing the type `TFbType` is absent in the standard module `Firebird.pas`. However, it is not convenient to use numerical values, so I wrote a special module FbTypes in which I placed some additional types for convenience.

The enumeration of `TFBCharSet` is also absent in the `Firebird.pas` module. I wrote a separate module FbCharsets in which this transfer is posted. In addition, for this type, a special helper is written, which contains functions for obtaining the name of the set of characters, the code page, the size of the symbol in bytes, obtaining the `TEncoding` class in the necessary encoding, as well as the function for converting the byte massif into the Delphi unicode line.

For lines of the type `CHAR` and `VARCHAR`, check the encoding, if its encoding is `OCTETS`, then we encode the line with the base64 algorithm, otherwise we convert data from the buffer to the Delphi line. Please note that for the type of VARCHAR the first 2 bytes contain the length of the line in the characters.

Types of `SMALLINT`, `INTEGER`, `BIGINT` can be as ordinary integers, so scalable. The scale of the number can be obtained by the `getScale` interface `IMessageMetadata`. If the scale is not equal to 0, then a special processing of the number is required, which is carried out by the `MakeScaleInteger`.

Types `DATE`,` TIME` and `TIMESTAMP` are decoded on the components of the date and time using the methods `decodeDate` and `decodeTime` of interface `IUtil`. We use parts of the date and time to receive the date-time in the standard Delphi type TDateTime.

With the BLOB type, we work through Delphi flows. If Blob is binary, then we create a stream like `TBytesStream`. The resulting an array of byte is encoded using the base64 algorithm. If BLOB is textual, then we use a specialized stream `TStringStream` for lines, which allows you to take into account the code page. We get the code page from the BLOB field encoding.

To work with the data of `INT128` there is a special interface `IInt128`. It can be obtained by calling the `getInt128` of interface `IUtil` interface. This type appeared in Firebird 4.0 and is designed to accurately represent very large numbers. There is no direct type of data in Delphi, which could work with this type, so we simply display its string performance.

To work with the types of `DECFLOAT(16)` and `DECFLOAT(34)` there are special interfaces `IDecFloat16` and` IDecFloat34`. They can be obtained by calling `getDecFloat16` or `getDecFloat34` of interface `IUtil`. These types are available from Firebird 4.0. There are no direct types of data in Delphi that could work with these types. These types can be displayed in BCD or presented in the form of a string.

Types of `TIME WITH TIME ZONE` and `TIMESTAMP WITH TIME ZONE` are decoded on the components of the date and time, as well as the name of the time zone, using the `decodeTimeStampTz` and `decodeTimeTz` methods. We use parts of the date and time to receive the date-time in the standard Delphi type TDateTime. Next, we convert the value of this type into the line and add the name of the time zone to it.

Appendices

Appendix A: License notice

The original documentation is called *Writing UDR Firebird in Pascal*.

The original authors of the original documentation are: Denis Simonov. The authors of the text in Russian are Denis Simonov.

Author(s): Denis Simonov.

(Author contacts: sim-mail at list dot ru).

Contributor(s): Martin Köditz.

(Author contacts: martin koeditz at it syn dot de).

Appendix B: Document history

The exact file history is recorded in the firebird-documentation git repository; see https://github.com/FirebirdSQL/firebird-documentation

Revision History			
1.0.0	22 Sep 2023	MK	English translation of the Russian document by Martin Köditz.
1.0.0-ru	21 Sep 2023	DS	Document's first version. The original was contributed by Denis Simonov in Russian language.