Firebird and IBExpert White Paper

Firebird external engine and UDR's written in Pascal

An example of UDR functions, triggers and procedures

Fikret Hasovic, January 2022

Introduction

In Firebird 3, the remodelling of the architecture was completed with the implementation of full SMP support for the Superserver model. The remodelled architecture integrates the core engine for Classic/SuperClassic, Superserver and embedded models in a common binary.

From version 3 onward, Firebird's architecture supports plug-ins. For a number of predefined points in the Firebird code, a developer can now write his own fragment of code for execution when needed.

A plug-in is not necessarily written by a third party: Firebird has a number of intrinsic plug-ins. Even some core parts of Firebird are implemented as plug-ins.

External engine

One of the plug-in types implemented in Firebird 3 is *ExternalEngine.*

The UDR (User Defined Routines) engine adds a layer on top of the *FirebirdExternal* interface with these objectives:

- Establish a way to place external modules into server and make them available for usage;
- Create an API so that external modules can register their available routines;
- Make routines instances per-attachment, instead of per-database like the *FirebirdExternal* does in Superserver mode.

External names of the UDR engine are defined as follows:

```
'<module name>!<routine name>!<misc info>'
```

The `<module name>` is used to locate the library, `<routine name>` is used to locate the routine registered by the given module, and `<misc info>` is a user-defined string passed to the routine and can be read by the user. `"!<misc info>"` may be omitted.

Modules available to the UDR engine should be stored in a directory listed through the path attribute of the corresponding plugin config tag. By default, UDR modules should be on `<fbroot>/plugins/udr` accordingly, to its path attribute in `<fbroot>/plugins/udr_engine.conf`.

The user library should include `FirebirdUdr.h` (or `FirebirdUdrCpp.h`, or `Firebird.pas`) and link with the `udr_engine` library. Routines are easily defined and registered using some macros, but nothing prevents you from doing things manually. An example routine library is implemented in

`examples/plugins`, showing you how to write functions, selectable procedures and triggers. Also, it shows you how to interact with the current database through the ISC API.

The UDR routines state (i.e. member variables) are shared between multiple invocations of the same routine until it's unloaded from the metadata cache. But note that it isolates the instances per session, different to the raw interface which shares instances by multiple sessions in Superserver.

By default, UDR routines use the same character set specified by the client. They can modify it by overriding the `getCharSet` method. The chosen character set is valid for communication with the ISC library as well as the communications done through the *FirebirdExternal* API.

Enabling an external routine in the database involves a DDL command to "create" it. Of course, it should already have been created externally and well tested.

Syntax Pattern

```
{ CREATE [ OR ALTER ] | RECREATE | ALTER } PROCEDURE <name>
    [ ( <parameter list> ) ]
    [ RETURNS ( <parameter list> ) ]
    EXTERNAL NAME '<external name>' ENGINE <engine>

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

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

Lazarus and FPC

To create a UDR using Lazarus and have your function written in Pascal, you will need to start with the library, in our case:

```
library PascalUDR;
uses
  Udr_Init,
  UdrGenRows in 'UdrGenRows.pas',
  UdrInc in 'UdrInc.pas',
  TestTrigger in 'TestTrigger.pas';
exports firebird_udr_plugin;
begin
  IsMultiThread := true;
end.
```

`Udr_Init` is the location where you register your function, trigger and/or stored procedure, and it can look like the following:

```
unit Udr_Init;
```

```
interface

uses Firebird;

function firebird_udr_plugin(status: iStatus; theirUnloadFlagLocal: BooleanPtr;
udrPlugin: iUdrPlugin): BooleanPtr; cdecl;

implementation
uses UdrGenRows, UdrInc, TestTrigger;
var
      myUnloadFlag   : Boolean;
      theirUnloadFlag: BooleanPtr;


function firebird_udr_plugin(status: iStatus; theirUnloadFlagLocal: BooleanPtr;
udrPlugin: iUdrPlugin): BooleanPtr; cdecl;
begin
      udrPlugin.registerProcedure(status, 'gen_rows', GenRowsFactory.create());
      udrPlugin.registerFunction(status, 'pas_inc', IncFactory.create());
      udrPlugin.registerTrigger(status, 'test_trigger',
TMyTriggerFactory.Create());
      theirUnloadFlag := theirUnloadFlagLocal;
      Result := @myUnloadFlag;
end;

initialization
      myUnloadFlag := false;
finalization
      if ((theirUnloadFlag <> nil) and not myUnloadFlag) then
            theirUnloadFlag^ := true;
end.
```

Note the usage of `Firebird.pas` here. You can find that file with your Firebird distribution, usually in the `<fbroot>/include/firebird` directory. I needed to delete `Classes` from the users list, to be able to use it here.

Unit `UdrGenRows` has a Pascal implementation of a stored procedure for generating a number of rows. You can find the same example written in C++ in your Firebird install directory.

The interface section should, in this case, look like:

```
interface

uses Firebird;

type
  GenRowsInMessage = record
    start: integer;
    startNull: wordbool;
    end_: integer;
    endNull: wordbool;
  end;
```

```
GenRowsInMessagePtr = ^GenRowsInMessage;

GenRowsOutMessage = record
  Result: integer;
  resultNull: wordbool;
end;

GenRowsOutMessagePtr = ^GenRowsOutMessage;

GenRowsResultSet = class(iExternalResultSetImpl)
  procedure dispose(); override;
  function fetch(status: iStatus): boolean; override;


public
  inMessage: GenRowsInMessagePtr;
  outMessage: GenRowsOutMessagePtr;
end;

GenRowsProcedure = class(iExternalProcedureImpl)
  procedure dispose(); override;

  procedure getCharSet(status: iStatus; context: iExternalContext;
    Name: pansichar; nameSize: cardinal); override;

  function Open(status: iStatus; context: iExternalContext; inMsg: Pointer;
    outMsg: Pointer): iExternalResultSet; override;
end;

GenRowsFactory = class(iUdrProcedureFactoryImpl)
  procedure dispose(); override;

  procedure setup(status: iStatus; context: iExternalContext;
    metadata: iRoutineMetadata; inBuilder: iMetadataBuilder;
    outBuilder: iMetadataBuilder); override;

  function newItem(status: iStatus; context: iExternalContext;
    metadata: iRoutineMetadata): iExternalProcedure; override;
end;
```

The `GenRowsInMessage` record in this example has 2 parameters, in fact 4, to store the starting value, as well the ending value. When you examine the code here, you will notice interface classes defined in the `Firebird.pas` unit mentioned above.

The implementation block is not hard to understand:

```
implementation

procedure GenRowsResultSet.dispose();
begin
  Destroy;
end;

function GenRowsResultSet.fetch(status: iStatus): boolean;
begin
  if (outMessage.Result >= inMessage.end_) then
    Result := False
  else
  begin
    outMessage.Result := outMessage.Result + 1;
    Result := True;
  end;
end;


procedure GenRowsProcedure.dispose();
begin
  Destroy;
end;

procedure GenRowsProcedure.getCharSet(status: iStatus; context:
iExternalContext;
  Name: pansichar; nameSize: cardinal);
begin
end;

function GenRowsProcedure.Open(status: iStatus; context: iExternalContext;
  inMsg: Pointer; outMsg: Pointer): iExternalResultSet;
var
  Ret: GenRowsResultSet;
begin
  Ret := GenRowsResultSet.Create();
  Ret.inMessage := inMsg;
  Ret.outMessage := outMsg;

  Ret.outMessage.resultNull := False;
  Ret.outMessage.Result := Ret.inMessage.start - 1;

  Result := Ret;
end;


procedure GenRowsFactory.dispose();
begin
  Destroy;
end;
```

```
procedure GenRowsFactory.setup(status: iStatus; context: iExternalContext;
  metadata: iRoutineMetadata; inBuilder: iMetadataBuilder; outBuilder:
iMetadataBuilder);
begin
end;

function GenRowsFactory.newItem(status: iStatus; context: iExternalContext;
  metadata: iRoutineMetadata): iExternalProcedure;
begin
  Result := GenRowsProcedure.Create;
end;
```

Unit UdrInc has a Pascal implementation of the function, to return an incremented value, as a simple example.

The interface section is:

```
interface

uses Firebird;

type
  IncInMessage = record
    val: integer;
    valNull: wordbool;
  end;

  IncInMessagePtr = ^IncInMessage;

  IncOutMessage = record
    Result: integer;
    resultNull: wordbool;
  end;

  IncOutMessagePtr = ^IncOutMessage;

  IncFunction = class(IExternalFunctionImpl)
    procedure dispose(); override;

    procedure getCharSet(status: iStatus; context: iExternalContext;
      Name: pansichar; nameSize: cardinal); override;

    procedure Execute(status: iStatus; context: iExternalContext;
      inMsg: Pointer; outMsg: Pointer); override;
  end;

  IncFactory = class(IUdrFunctionFactoryImpl)
    procedure dispose(); override;

    procedure setup(status: iStatus; context: iExternalContext;
      metadata: iRoutineMetadata; inBuilder: iMetadataBuilder;
      outBuilder: iMetadataBuilder); override;
```

```
    function newItem(status: iStatus; context: iExternalContext;
      metadata: iRoutineMetadata): IExternalFunction; override;
  end;
```

The `IncInMessage` record in this example has 1 member, in fact 2, to store returned data. When you examine the code here, you will notice different interface classes to the ones mentioned above, because we are working on a UDR function, and we must be careful with that.

The implementation block is also not difficult to understand:

```
implementation

procedure IncFunction.dispose();
begin
  Destroy;
end;

procedure IncFunction.getCharSet(status: iStatus; context: iExternalContext;
  Name: pansichar; nameSize: cardinal);
begin
end;

procedure IncFunction.Execute(status: iStatus; context: iExternalContext;
  inMsg: Pointer; outMsg: Pointer);
var
  xInput: IncInMessagePtr;
  xOutput: IncOutMessagePtr;
begin
  xInput := IncInMessagePtr(inMsg);
  xOutput := IncOutMessagePtr(outMsg);

  xOutput^.resultNull := xInput^.valNull;
  xOutput^.Result := xInput^.val + 1;
end;


procedure IncFactory.dispose();
begin
  Destroy;
end;

procedure IncFactory.setup(status: iStatus; context: iExternalContext;
  metadata: iRoutineMetadata; inBuilder: iMetadataBuilder; outBuilder:
iMetadataBuilder);
begin
end;

function IncFactory.newItem(status: iStatus; context: iExternalContext;
  metadata: iRoutineMetadata): IExternalFunction;
begin
  Result := IncFunction.Create;
end;
```

The third unit, `TestTrigger` contains the implementation of a UDR-based trigger, the interface section is:

```
interface

uses
  Firebird;

type

  // structure for mapping messages for NEW. * and OLD. *
  // must match the set of fields in 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 creating an instance of the external trigger TMyTrigger
  TMyTriggerFactory = class(IUdrTriggerFactoryImpl)
    // Called when the factory is destroyed
    procedure dispose(); override;

    {Executed every time an external trigger is loaded into the metadata cache

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

    {Create a new instance of the external trigger TMyTrigger

      @param (AStatus Status vector)
      @param (AContext External trigger execution context)
      @param (AMetadata External trigger metadata)
      @returns (External Trigger Instance)
    }
    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 immediately before execute and reports
      the kernel is our requested character set for exchanging 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;

    {execution of trigger 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;
```

The implementation block is:

```
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
  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;
```

Examples of using a UDR-based stored procedure, function and trigger

To be able to use the UDR library we just created, we need to register those in our Firebird database:

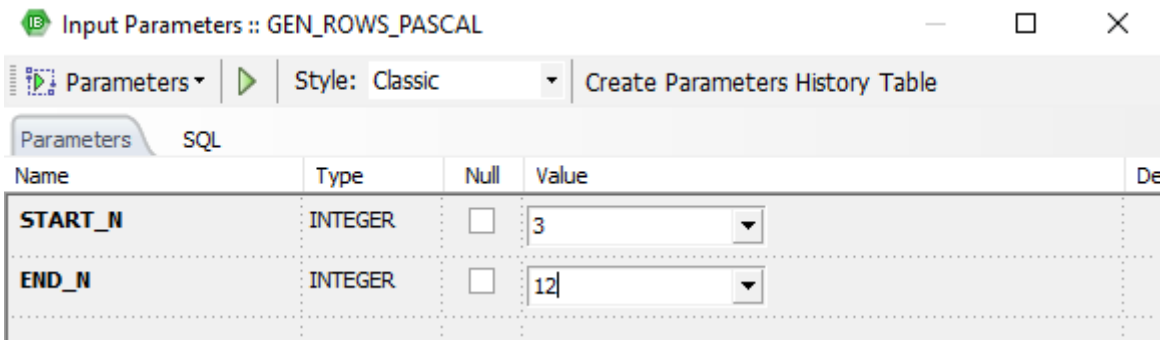
```
create procedure gen_rows_pascal (
    start_n integer not null,
    end_n integer not null
) returns (
    result integer not null
```

```
)
    external name 'pascaludr!gen_rows'
    engine udr;

create function Inc (
    v integer
) returns integer
    external name 'pascaludr!pas_inc'
    engine udr;

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 'pascaludr!test_trigger'
    engine udr;
```

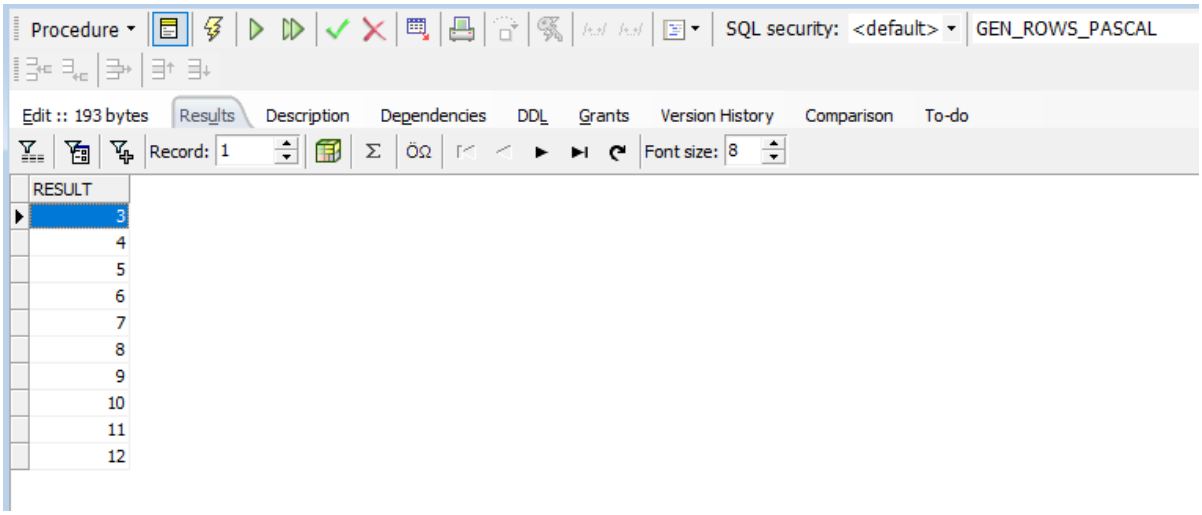
Note that I have included a `create table` statement here, because I have a trigger written as UDR, so I need a sample table. The trigger's job in this simple example is to set the value of field B as the value of A incremented by one.

Results

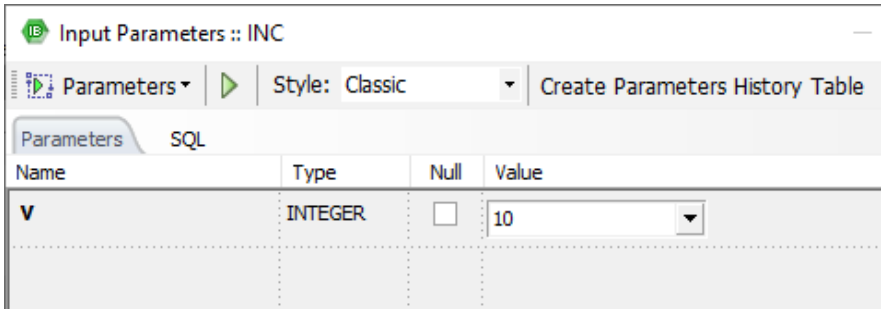
When you execute the procedure `gen_rows_pascal`, by using IBExpert (or writing the select statement), for example, such as:



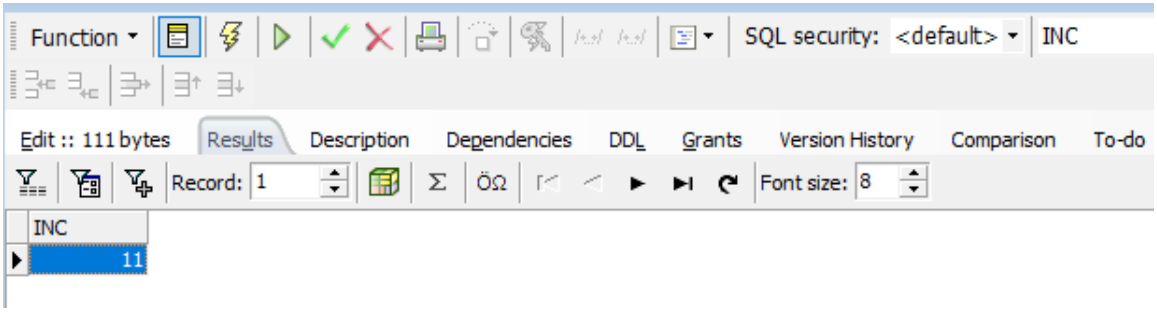
You will get results the following results:



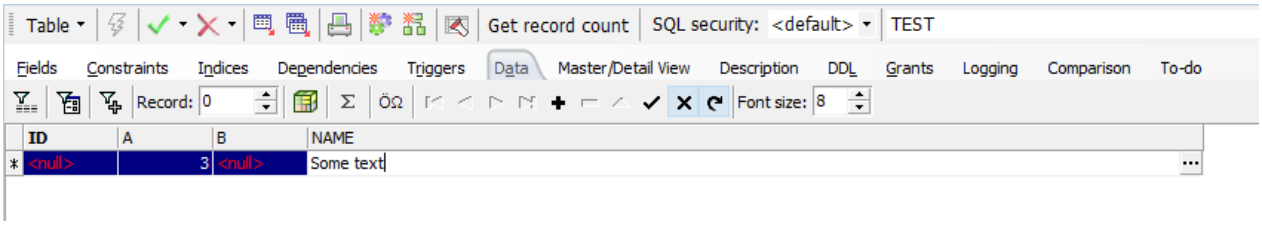
The second example, using the `inc` function, with the parameter 10:



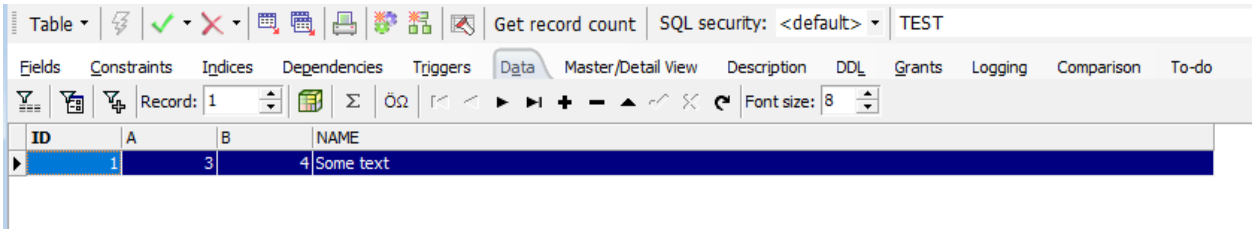
will result in the following:



The third example, a simple trigger on the table, on insert new value in table, and skipping value of field B:



After committing the transaction, you will get following in your table:



How everything works

External names are recognized by specific external engines. External engines are declared in the config files (possibly in the same file as a plugin, like in the config example below):

```
<external_engine UDR>
    plugin_module UDR_engine
</external_engine>

<plugin_module UDR_engine>
    filename $(this)/udr_engine
    plugin_config UDR_config
</plugin_module>

<plugin_config UDR_config>
    path $(this)/udr
</plugin_config>
```

When Firebird wants to load an external routine (function, procedure or trigger) into its metadata cache, it gets (if not already done for the database*) the external engine through the plugin external engine factory and asks it for the routine. The plugin used is the one referenced by the attribute plugin module of the external engine.

* This is in Superserver. In [Super-]Classic, different attachments to a single database create multiple metadata caches and hence multiple external engine instances.