FB/Java External Engine Plugin

Documentation for FB/Java External Engine Plugin

Adriano dos Santos Fernandes

28 May 2016 – Document version 1.0.1

Table of Contents

What is the FB/Java External Engine Plugin?

FB/Java is an External Engine plugin for Firebird that makes Firebird capable of run functions, procedures and triggers made in the Java platform.

It complements Jaybird making it interface with Firebird engine and handling the infrastructure necessary to support user routines. It also has a client utility with functions to install and uninstall the plugin in a database and that users may use to deploy and undeploy Java classes and resources stored in JAR files to a database.

Installing FB/Java in Firebird

Server installation of FB/Java is a simple task involving two passes. The first necessary pass is to extract the zip/tarball package in a place subsequently named here as <fbjava-root> (this is the parent directory of the packaged bin and others directories). The second pass is to include a line in the end of <firebird-root>/plugins.conf file pointing to the plugin:

```
include <fbjava-root>/conf/fbjava.conf
```

FB/Java can also be installed in the client making possible to use fbjava-deployer utility remotely. In this case it's just necessary to extract the zip/tarball in some directory and optionally add its bin directory to the PATH environment variable.

To develop Java code the only necessary file in the compiling classpath is <fbjava-root>/jar/fbjava-<version>.jar

For Maven users, there is a provisory repository which more info can be found in https://github.com/asfernandes/fbjava-maven.

Features

FB/Java features a number of important things:

Database and Java routines mappings

There are basically two ways to map database functions and procedures to Java methods. By fixed or generic signatures. Triggers can be mapped only with generic signatures.

> **Note**
>
> Generics here does not refer to Java 5 generics in any way.

- Fixed signatures

  Fixed signatures means that for each database routine parameter there should be a Java parameter in the method.

- Generic signatures

  Generic signatures doesn't have parameters. The Java code can obtain from the call Context all parameters or fields values passed by the database routine.

  > **Warning**
  >
  > It's expected that in a subsequent test release fixed signatures accepts an optional context parameter and that generic signatures requires a single context parameter.

Security

One of the more important features of the Java platform is it security system, the so called sandbox. FB/Java integrates the J2SE/JAAS security mechanism with Firebird so that permissions may be assigned to database users running the Java code.

Users permissions works at server level. They are stored in the <fbjava-root>/conf/java-security.fdb database. That database contains the tables PERMISSION_GROUP (with columns ID, NAME), PERMISSION (with columns PERMISSION_GROUP, CLASS_NAME, ARG1, ARG2) and PERMISSION_GROUP_GRANT (with colums PERMISSION_GROUP, DATABASE_PATTERN, GRANTEE_TYPE, GRANTEE_PATTERN).

PERMISSION_GROUP names a set of PERMISSION associated by PERMISSION.PERMISSION_GROUP column.

In PERMISSION table, there is CLASS_NAME column which stores the Java permission class name and ARG1/ARG2 which stores the arguments passed to the permission class constructor.

The PERMISSION_GROUP_GRANT table associates PERMISSION_GROUP with Firebird users and roles. This association is done by a DATABASE_PATTERN and a GRANTEE_TYPE/GRANTEE_PATTERN. Patterns are SIMILAR TO patterns escaped by the '|' symbol. Pay attention when using special SIMILAR TO characters like underline, if they refer to actual database, user or role name parts, they need to be escaped. GRANTEE_TYPE defines if GRANTEE_PATTERN refers to a ROLE or USER.

The plugin ships with a number of permissions grouped as COMMON and granted to all (% pattern) users of all databases. They are:

Table 1. Default permissions granted

CLASS_NAME	ARG1	ARG2
java.util.PropertyPermission	file.separator	read
java.util.PropertyPermission	java.version	read
java.util.PropertyPermission	java.vendor	read
java.util.PropertyPermission	java.vendor.url	read
java.util.PropertyPermission	line.separator	read
java.util.PropertyPermission	os.*	read
java.util.PropertyPermission	path.separator	read

Warning

Permissions configured in java-security.fdb are valid only for classes stored inside the database. Classes at file system are granted java.security.AllPermission.

The java.security.AllPermission is effectively valid in this context if the code marks itself as privileged, like with java.security.AccessController.doPrivileged method.

ClassLoaders

FB/Java looks for classes in two different places: the file system and the current database, in this order. It is general recommendation that users store they classes in the database.

Classes in the file system are shared between all databases handled by a Firebird process. For example, static variables have per-process values. In an analogy with an application server, they are the system classes. The internal classes necessary for FB/Java are in <fbjava-root>/jar/*.jar and is not recommended to put more jar files there.

Classes in the database are isolated per-database (and process) and unloaded when the last user disconnects from the database/process. So static variables are shared between attachments to the same database, but are reinitialized when a closed database is opened. In an analogy with an application server, they are the application classes, although an application server does not reload application classes when the application is idle.

Note

Classes does not share static variables when used by different Classic or embedded process.

Note

Classes are unloaded by closing the database classloader. It is subject to garbage collection really unload them.

Classes can be stored in the database by two different methods: the fbjava-deployer (.bat in Windows and .sh in Linux) utility or the SQLJ package.

The deployer utility

FB/Java allows execution of Java routines stored in the file system without any per-database installation. But to have per-database classes, the plugin need to be installed on the database.

fbjava-deployer is the utility to install and uninstall the plugin in databases and to install, remove and replace JAR files in databases. Its command line options are:

- **`--database <connection string>`**

  Jaybird connection string, without jdbc: prefix.

- **`--user <user name>`**

  Database user name.

- **`--password <password>`**

  User password.

- **`--install-plugin`**

  Installs the plugin in the database. The installation process consists of the creation of some database objects, prefixed with FB$JAVA$ and the SQLJ package.

  > **Note**
  >
  > Details of the install process can be seen in the install.sql file in the scripts directory of the plugin.

- **`--uninstall-plugin`**

  Uninstalls the plugin from the database. The uninstall process consists of dropping all the objects created by the installation process.

  > **Note**
  >
  > Details of the uninstall process can be seen in the uninstall.sql file in the scripts directory of the plugin.

  > **Warning**
  >
  > All stored JARs are deleted when the plugin is uninstalled.

- **`--install-jar <URL or filename> <name>`**

  Installs a JAR in the database. **`<name>`** is an unique identifier to refer to the JAR in subsequent calls, like **`--replace-jar`**, **`--update-jar`** or **`--remove-jar`**.

- **`--update-jar <URL or filename> <name>`**

  Updates an existing JAR in the database. It is an error to try to update a JAR with a **`<name>`** that is not installed.

- **`--replace-jar <URL or filename> <name>`**

  Replaces a JAR in the database. The replacement is done deleting the current JAR if it exists and installing the new one.

- **`--remove-jar <name>`**

  Removes a JAR from the database.

SQLJ package

JAR installation, replacement, updation and remotion can also be done with the help of the SQLJ package. The SQLJ package just runs the same class used in fbjava-deployer, but in the server, as Java stored procedures. With that package, paths and URLs are server-based. Its procedures are:

- **SQLJ.INSTALL_JAR(<URL or filename>, <name>)**
- **SQLJ.UPDATE_JAR(<URL or filename>, <name>)**
- **SQLJ.REPLACE_JAR(<URL or filename>, <name>)**
- **SQLJ.REMOVE_JAR(<name>)**

Mapping database routines to Java methods

Database routines are mapped to Java methods by a database declaration with an external call specification and usage of ENGINE JAVA clause. The call specification consists of the method signature, in this format:

```
<call specification> ::=
  <fully qualified class name>.<static method name>(
      [<type> [{, <type>}...]])
      [!<name info>]

<type> ::=
  <primitive type> |
  <fully qualified class name or unqualified class name from java.lang package>
```

> **Note**
>
> The java.lang package prefix may be avoided in parameters types but not in the class name containing the static method.

> **Note**
>
> <name info> prefixed by an exclamation point is an optional info you can pass to the Java method that it can obtain with the method getNameInfo from the Context interface.

Table 2. Supported Java types

Java type	Compatible Firebird type	Notes
byte[]	BLOB, CHAR, VARCHAR	
boolean	any	[1]
short	any	[1]
int	any	[1]
long	any	[1]
float	any	[1]
double	any	[1]
java.lang.Boolean	any	
java.lang.Short	any	
java.lang.Integer	any	
java.lang.Long	any	
java.lang.Float	any	
java.lang.Double	any	
java.lang.Object	any	[2]
java.lang.String	any	
java.math.BigDecimal	any	
java.sql.Blob	BLOB	
java.sql.Date	any	
java.sql.Time	any	
java.sql.Timestamp	any	
java.util.Date	any	

Note

[1] A database NULL is converted to 0 (zero) when passed to a primitive numeric type. and false to boolean.

[2] Parameters and trigger values are converted accordingly to default mapping rules.

Note

Any compatible type means the plugin doesn't care about the type, it just tries to get the value as a Firebird type compatible with the Java type. Basically, this means that a CAST will be done from the Firebird value to the default mapping type of the Java type, or vice-versa.

Table 3. Default mappings

Firebird type	Java type
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
SMALLINT	java.math.BigDecimal
INTEGER	java.math.BigDecimal
BIGINT	java.math.BigDecimal
FLOAT	java.lang.Float
DOUBLE PRECISION	java.lang.Double
BOOLEAN	java.lang.Boolean
CHAR	java.lang.String
VARCHAR	java.lang.String
BLOB	java.sql.Blob
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

Functions mapping

Functions are the only routine type that always requires a non-void return type in the Java method.

Examples:

```
create or alter function get_system_property (
  name varchar(80)
) returns varchar(80)
  external name 'java.lang.System.getProperty(String)'
  engine java;
```

```
-- A single method being mapped to two different Firebird functions.

create function funcSum2 (n1 integer, n2 integer)
    returns integer
  external name 'org.firebirdsql.fbjava.FuncTest.sum()'
  engine java;

create function funcSum4 (n1 integer, n2 integer, n3 integer, n4 integer)
    returns integer
  external name 'org.firebirdsql.fbjava.FuncTest.sum()'
  engine java;
```

Procedures mapping

Procedures can have a return type (of org.firebirdsql.fbjava.ExternalResultSet type or a class implementing that interface) or void, depending of it being a selectable procedure or not. Output parameters should appear on the call specification as arrays. FB/Java pass each output parameter as an array of length 1, and routines can change their [0] element.

Examples:

```
-- Executable procedure.
create procedure procInsert (n integer, s varchar(10))
  external name 'org.firebirdsql.fbjava.ProcTest.insert(int, String)'
  engine java;
```

```
-- Selectable procedure.
create procedure procGenRows (numRows integer) returns (n integer)
  external name 'org.firebirdsql.fbjava.ProcTest.genRows(int, int[])'
  engine java;
```

Triggers mapping

Call specification of triggers has always one zero parameter and the Java method should return void. Details of the call and the OLD and NEW values can be read and write from the call context.

Examples:

```
create or alter trigger employee_log_bdiu
    before delete or insert or update on employee
  external name 'org.firebirdsql.example.fbjava.FbLogger.info()'
  engine java;
```

Context

Java code can obtain a Context object which has information about the Firebird routine that mapped to the Java code as well inspect and manipulate metadata and data about the call. The Context interface has the following hierarchy:

```
        CallableRoutineContext extends Context
        FunctionContext extends CallableRoutineContext
        ProcedureContext extends CallableRoutineContext
        TriggerContext extends Context
```

All these interfaces have a static get() method to obtain the object instance. More details of what can be done from them can be consulted in the Java Docs present in <fbjava-root>/docs/apidocs

Known issues

In this test version it's not possible to use Jaybird's non-pure-Java connection string protocols like embedded, native and local.