Table of Contents

FTS Database Code Guidelines

This describes the guidelines for writing and deploying database (PL/SQL) modules for use by the FTS schema.

Code types

The FTS database schema consists of two parts:

- **The core schema**. This is loaded (manuallly, as instructed by YAIM) upon installation of the FTS. Core schema upgrade scripts are integrated into YAIM and must be run before YAIM will configure a new version of FTS. It consists of core FTS tables, indicies and triggers for FTS's core functionality. The core schema is stable and the change control is managed by the FTS development team.
- **Modular schemas**. These are loaded to provide specific functions to the FTS service administrator, or to provide specific monitoring packages to check different aspects of the FTS's behaviour. Examples are the "FTS history package" (cleaning out the core tables), the "FTS admin pack" (providing channel rename functions) and the newer "FTS monitoring modules" (looking at different aspects of the FTS's behaviour). These are loaded from the RPM concerned and should have limited interdependencies. They can consist of any reasonable schema objects (database or code) as long as they do not chnage or impact on the core schema. They are not configured by YAIM.

This page describes how to register FTS modules

Migration from module schema to core schema

Stable modules might eventually get absorbed into the core FTS schema if they are useful and stable enough. The decision to do this is with the FTS development team.

Package contents

A module should be packaged in a single RPM.

Code should be delivered as PL/SQL packages.

The RPM should contain maximum one PL/SQL package and associated schema objects.

It should contain:

- A `README` to explain what it does.
- A SQL file containing the package code header, if any.
- A SQL file containing the package code body, if any.
- A SQL file containing the associated schema, if any.

Package registration

There is a special package called `FTS_REGISTER` with associated schema `FTS_PLUGIN` and `FTS_PLUGIN_SCHEMA`.

Every package must register itself using the `FTS_REGISTER` package.

For each registered package there will be:

- One row in `FTS_PLUGIN` plugin describing the package and its schema dependencies.
- One row in `FTS_PLUGIN_SCHEMA` describing the associated schema version for the package.

The registration provides:

- Name - this must match the PL/SQL package name, capitalised (without the schema owner prefix).
- Description - a short package description
- Author - the email of the maintainer
- Package version - the package version. This must be the same as the RPM that delivered it.
- Core schema requirements:
  - Minimum version of the core schema required for the package to run, or `NULL` if no restriction
  - Maximum version of the core schema required for the package to run, or `NULL` if no restriction
- Associated schema requirements:
  - Minimum version of the associated schema required for the package to run, or `NULL` if no restriction
  - Maximum version of the associated schema required for the package to run, or `NULL` if no restriction
- Maximum DB state for this package ('==TEST' or '=PRODUCTION='). For core schema versions >= 3.2.0. This allows you to specify that this version of the package will not run on a '=PRODUCTION=' FTS, but is only suitable for a 'TEST' FTS.

Version number format

It should be three `major.minor.patch` format, numbers only.

e.g. `1.0.5` is valid. `2.0.3-4` and `2.1` are not valid.

Registration and checking

A packages is required to implement some code to make use of the `FTS_REGISTER` functionality.

In particular:

It should implement the register 'interface':

- The package must implement a `registerMe` method that will register the package in the `FTS_PLUGIN` table using the `fts_register.registerPackage( .. )` method.
- The associated schema loading script should call the `fts_register.registerSchema( .. )` method to register the schema version in the `FTS_PLUGIN_SCHEMA` table.

Every public method of the package should first call:

```
fts_register.checkPackage('MODULENAME')
```

This will check that the package has been registered and that the schema versions are compatible, and will not allow the package to run if this is not the case.

Suggested code

Package header

In the package header, you should define constants to give all the relevant version information and dependencies:

```
g_packageName varchar2(100) := 'FTS_SERVICESTATE';
g_description varchar2(1024) := 'FTS service state information collection';
g_author varchar2(100) := 'gavin.mccance@cern.ch';
g_packageVersion varchar2(10) := '1.0.0';
g_requirePackageSchemaMin varchar2(10) := '1.0.0';
g_requirePackageSchemaMax varchar2(10) := NULL;
g_requireCoreSchemaMin varchar2(10) := '3.1.0';
g_requireCoreSchemaMax varchar2(10) := NULL;
g_requireMaxDbState varchar2(15) := 'TEST';
```

and add the registerMe method:

```
procedure registerMe;
```

Package body

In the package body you should implement the registerMe method. It is suggested to use this code:

```
   procedure registerMe
   as
   begin
      fts_register.registerPackage(
                     g_packageName,
                       g_description,
                     g_author,
                       g_packageVersion,
                       g_requirePackageSchemaMin,
                       g_requirePackageSchemaMax,
                       g_requireCoreSchemaMin,
                       g_requireCoreSchemaMax,
                       g_requireMaxDbState );
      commit;
   END registerMe;
```

You should also add, at the top of every public method a call to check the package validity, e.g:

```
procedure countStates
as
begin
  fts_register.checkPackage(g_packageName);
  execute immediate('
      :
      :
end countStates;
```

Schema loading script

At the bottom, include a call to register the package in `FTS_PLUGIN_SCHEMA`.

```
CREATE TABLE m_agent_avail (
-- id
     :
     :

-- register schema
exec fts_register.registerSchema('FTS_SERVICESTATE', '1.0.0');
```

The name should match the package name, upper case.

What if I have no associated schema?

You still need a row in the `FTS_PLUGIN_SCHEMA` table, with `NULL` schema version.

Add this into your registerMe function, in this case:

```
   procedure registerMe
   as
   begin
      fts_register.registerPackage(
                     g_packageName,
                       g_description,
                     g_author,
                       g_packageVersion,
                       g_requirePackageSchemaMin,
                       g_requirePackageSchemaMax,
                       g_requireCoreSchemaMin,
                       g_requireCoreSchemaMax,
                       g_requireMaxDbState );
      fts_register.registerSchema(g_packageName, NULL);
      commit;
   END registerMe;
```

DBMS jobs

For regular cron-like running of tasks, your code should implement the job 'interface':

- The package must implement a `submit_job` method which will submit a DBMS job with a given periodicity in minutes (which should have a reasonable default).
- The package must implement a =stop_job+ method which will remove the DBMS job.
- The package must implement a `runjob` method which is called by DBMS every time the job is run.

Suggested code

Package header

In the package header, you should define the three methods and the default periodicity of the job:

```
-- The job 'interface'
procedure runjob;
procedure submit_job (v_minutes binary_integer DEFAULT 1);
procedure stop_job;
```

Package body

In the package body you should implement the three methods. The suggested implementation is here (TODO: make this a bit better!):

```
-- submit the DBMS job
procedure submit_job
   (v_minutes binary_integer)
as
njob USER_JOBS.JOB%TYPE;
begin
   fts_register.checkPackage(g_packageName);
   dbms_job.submit(njob, 'begin fts_servicestate.runjob; end;', sysdate, 'SYSDATE + '||to_char(v_
end submit_job;

-- delete the DBMS job
procedure stop_job
```

```
as
v_jobid USER_JOBS.JOB%TYPE;
begin
   fts_register.checkPackage(g_packageName);
   select JOB into v_jobid from user_jobs where what like '%fts_servicestate.runjob%';
   dbms_job.remove(v_jobid);
end stop_job;

-- run the DBMS job
procedure runjob
as
begin
   -- do whatever is needed, for example:
   agentStates;
   countStates;
end runjob;
```

Error handling

There are two type of procedure and package: those designed to be run by hand from the administrator (to accomplish some discrete task) and those designed to be run from DBMS regularly.

Error handling for hand run scripts

It is recommended top handle all reasonable exceptions properly - i.e. try to fix the problem if possible, otherwise, log using a `dbms_output.put_line` and exit the script, rolling back any changes as necessary.

It is recommended to catch all other unexpected exceptions and log them using a `dbms_output.put_line` and then rethrow them, either as they are or wrapped in a application exception type (set the exception propagation to `TRUE`). This way the command line gets a printed log plus a detailed exception of explain where the unexpected problem was.

Example code to catch unexpected exceptions is here:

```
E_FATAL EXCEPTION;
PRAGMA EXCEPTION_INIT(E_FATAL, -20100);
   :
   :
procedure testRandom
as
begin
   dothestuff;
exception
   when OTHERS then
      rollback;
      dbms_output.put_line('Error: an unexpected error occurred.');
      raise_application_error (-20100, 'Error: an unexpected error occurred', TRUE);
end testRandom;
```

Error handling for DBMS tools

In this case, the error handling is harder since there is no command line to report on.

You should define an error table as part of your associated schema and log there. For example:

```
-- Error table
CREATE TABLE m_servicestate_err (
    LOGTIME     TIMESTAMP(6) WITH TIME ZONE,
    ROUTINE     VARCHAR(20),
    ERR_CODE    NUMBER,
```

```
    ERR_MSG     VARCHAR2(2048)
);
```

with associated function in the package:

```
-- Log any errors
procedure log_error
   ( routine varchar, errcode number, errmessage varchar)
as
PRAGMA AUTONOMOUS_TRANSACTION;
begin
   execute immediate('insert into M_SERVICESTATE_ERR (LOGTIME, ROUTINE, ERR_CODE, ERR_MSG) VALUES
      using systimestamp, routine, errcode, errmessage;
end log_error;
```

The error handling should roll back any transactions, log the error and then exit. Example code is:

```
procedure agentStates
as
begin
  fts_register.checkPackage(g_packageName);
  dothestuff;
  commit;
exception
   when OTHERS then
       rollback;
       log_error('agentStates', SQLCODE, SQLERRM);
end agentStates;
```

This way the job will continue to run but will log any errors in the timestamped error table. Your application or procedures should check this table regularly.

Coding conventions

There are a few conventions which are strongly encouraged.

Variable scope naming

Global public variables, those declared in the package header should be prefixed with `g_`.

Global private variables, those declared in the package body should be prefixed with `gp_`.

Parameter variables (i.e. passed into a procedure or function) should be prefixed with `p_`.

Procedure local variables should be prefixed with `l_`.

Record types should be prefixed with `_rec`.

Typing

The `%TYPE` directive should be used when defining variables for the destination of a `select .. into`. This makes the code more resilient to schema changes. e.g:

procedure stop_job as l_jobid USER_JOBS.JOB%TYPE; begin fts_register.checkPackage(g_packageName); select JOB into l_jobid from user_jobs where what like '%fts_servicestate.runjob%'; dbms_job.remove(l_jobid); end stop_job;

Record types

In general, if more than a third of the variables from a cursor select are used, then you should use the full record type (using =%ROWTYPE) and load the entire row. As in:

```
rec_plugin FTS_PLUGIN%ROWTYPE;
      :
      :
select    F.packageName, F.description, F.author,F.dateAdded, F.packageVersion,
      F.requirePackageSchemaMin, F.requirePackageSchemaMax, F.requireCoreSchemaMin,
      F.requireCoreSchemaMax, F.requireMaxDbState
      INTO rec_plugin   FROM FTS_PLUGIN F
      WHERE F.packageName = l_packageName;
```

If fetching more than one row

Do use a cursor, either implicit or explicit.

EXECUTE IMMEDIATE

Should generally be avoided except in the case of logging or unless there is no other reasonable choice.

DDL

Should not be used, except for:

- Administrator 'specials' where you have to disable a constraint to do the work. The constraint should always be re-enabled regardless of the outcome of the work (i.e. even if the function fails).
- Discrete administrator functionality where the purpose is clearly to alter the data dictionary - e.g. schema upgrade.
- Automatic function or trigger definition on package bootstrap.

DCL

Should not be used except for explicit packages such as `FTS_WRITER_ACCOUNT` where the purpose is clearly to grant access to another schema.

Changes and updates

New versions of code and schema should be released in a new RPM with an increased version number.

- If the schema has changed, the schema version should be updated using the `registerSchema( .. )` method call. This merges, so you will not create duplicate entries.
- If the package code has changed, the package version (and possibly its schema dependencies) shoulf be re-registered using the `registerPackage( .. )` method call. This merges, so you will not create duplicate entries.

Utility packages

There are some 'standardised utility packages available for FTS code to use.

FTS_REGISTER

Is mostly described above. It will provide extra functions to suggest which packages need attention upon a schema upgrade.

FTS_TIMING

Provide timestamp casting functions. The most useful are:

```
select fts_timing.t_ms( timestamp ) from dual
```

which returns the number of millseconds since the UTC epoch (midnight Jan 1st 1970).

and:

```
select fts_timing.t_s( timestamp ) from dual
```

which returns the rounded number of seconds.

These functions can be used for timing purposes to measure how long a given query or function is taking to run.

Example code

The core modules delivering the above functionality are attached here as reference, together with the `service state` example.

- package_lcg_fts_prod.fts_register.sql: The register package header
- packagebody_lcg_fts_prod.fts_register.sql: The register package body
- package_lcg_fts_prod.fts_timing.sql: Utility timing package
- package_lcg_fts_prod.fts_servicestate.sql: Example package (the service state toolset)
- packagebody_lcg_fts_prod.fts_servicestate.sql: Example package body (the service state toolset)

This topic: LCG > FtsDbCodeGuide
Topic revision: r4 - 2009-02-05 - RosaGarciaRioja