Contents

The documents in this directory contain some of the earliest information on the design and implementation of PL/Java. They are not current information (see the project web site for that), but they may offer interesting insights into why certain aspects of PL/Java are the way they are.

Note that the documentation available on the project web site and on the wiki does include much of the information you can find here, but more actively kept up to date.

The files in this directory (excepting the one you are reading now) have not been modified since 2007, and so predate the release of PL/Java 1.4.0. PL/Java had recently been updated to support PostgreSQL 8.1 and Java 1.4!

An early introduction and description of the project and its capabilities. In March of 2006, an acknowledgment of sponsorship by EnterpriseDB was added.

A README from a PL/Java 1.2.x distribution, with builds for Windows and for i386-based Linux, and instructions for how it was built and installed at that time.

User Guide for PL/Java 1.2. Described installation using the Deployer or install.sql and making entries in postgresql.conf. People really lived like that.

Explains the original reasoning for PL/Java's choice to load a Java virtual machine in each PostgreSQL session backend using it, rather than to share a VM in another process, the approach a contemporaneous project was exploring.

Describes several challenges that were encountered in PL/Java's development, and the techniques adopted to meet them. This document is good background on: the locking strategy used to serialize access by Java threads into PostgreSQL, the integration of PostgreSQL exceptions and savepoints with Java exceptions, and the management of pointer lifetimes by Invocation, CallLocal, and JavaWrapper.

Bringing the power of Java™ to PostgreSQL™ Functions and Triggers.

Java™ is a registered trademark of Sun Microsystems, Inc. in the United States and other countries. PostgreSQL™ is a trademark of PostgreSQL Inc and Regents of the University of California.

PL/Java is an add on module to the PostgreSQL backend. It falls into the same category as PL/SQL, PL/TCL, PL/Perl, PL/Python, and PL/R. When installed, functions and triggers can be written in Java using your favorite Java IDE and installed into the database.

The PL/Java 1.2.0 release of PL/Java provides the following features.

Ability to write both functions and triggers using Java 1.4 or higher.

Standardized utilities (modeled after the SQL 2003 proposal) to install and maintain Java code in the database.

Standardized mappings of parameters and result. Complex types as well as sets are supported.

An embedded, high performance, JDBC driver utilizing the internal PostgreSQL SPI routines.

Metadata support for the JDBC driver. Both DatabaseMetaData and ResultSetMetaData is included.

The ability to return a ResultSet that origins from a query as an alternative to build a ResultSet row by row

Full support for PostgreSQL 8.0 savepoints and exception handling.

Ability to use IN, INOUT, and OUT parameters when used with PostgreSQL 8.1

Two language handlers, one TRUSTED (the default) and one that is not TRUSTED (language tag is javaU to conform with the defacto standard)

Transaction and Savepoint listeners enabling code execution when a transaction or savepoint is commited or rolled back.

Integration with GNU GCJ on selected platforms.

PL/Java in brief

A function or trigger in SQL will appoint a static method in a Java class. In order for the function to execute, the appointed class must be installed in the database. PL/Java adds a set of functions that helps installing and maintaining the java classes. Classes are stored in normal Java archives (AKA jars). A Jar may optionally contain a deployment descriptor that in turn contains SQL commands to be executed when the jar is deployed/undeployed. The functions are modeled after the standards proposed for SQL 2003.

PL/Java implements a standardized way of passing parameters and return values. Complex types and sets are passed using the standard JDBC ResultSet

class. Great care has been taken not to introduce any proprietary interfaces unless absolutely necessary so that Java code written using PL/Java becomes as database agnostic as possible.

A JDBC driver is included in PL/Java. This driver is written directly on top of the PostgreSQL internal SPI routines. This driver is essential since it's very common for functions and triggers to reuse the database. When they do, they must use the same transactional boundaries that where used by the caller.

PL/Java is optimized for performance. The Java virtual machine executes within the same process as the backend itself. This vouches for a very low call overhead. PL/Java is designed with the objective to enable the power of Java to the database itself so that database intensive business logic can execute as close to the actual data as possible.

The standard Java Native Interface (JNI) is used when bridging calls from the backend into the Java VM and vice versa. Please read the rationale behind the choice of technology and a more in-depth discussion about some implementation details. PL/Java is primarily targeted to the new 8.1 version but will run with PostgreSQL 8.0 versions too albeit with some limitations.

For info on how to get started, please read the release notes. A user guide contains more information on how to create and manage our Java functions and triggers.

Source and selected binaries are available for download. See the download page for more details.

Rationale behind using JNI as opposed to threads in a remote JVM process

Rationale behind using JNI as opposed to threads in a remote JVM process.

Java™ is a registered trademark of Sun Microsystems, Inc. in the United States and other countries.

Reasons to use a high level language like Java™ in the backend

A large part of the reason why JNI was chosen in favor of an RPC based, single JVM solution was due to the expected use-cases. Enterprise systems today are almost always 3-tier or n-tier. Database functions, triggers, and stored procedures are mechanisms that extend the functionality of the backend tier. They typically rely on a tight integration with the database due to a very high rate of interactions and execute inside of the database largely to limit the number of interactions between the middle tier and the backend tier. Some typical use-cases:

Referential integrity enforcement. Using Java, referential integrity that goes beyond what can be provided using the standard SQL semantics can be provided. It might involve checking XML documents, enforcing some meta-driven rule system, or other complex tasks that put high demands on the implementation language.

Advanced pattern recognition. Soundex, image comparison, etc.

XML support functions. Java comes with a lot of XML support. Parsers etc. are readily available.

Support functions for O/R mappers. A variety of support can be implemented depending on design. One example is an O/R mapper that allows methods on persistent objects. A lot can be gained if such methods are pushed down and executed within the database. Consider the following (OQL):

SELECT AVG(x.salary - x.computeTax()) FROM Employee x WHERE x.salary > 120000;

Pushing the computeTax logic down to the database instead of computing it in the middle tier (where much or the O/R logic resides) is a huge gain from a performance standpoint. The statement could be transformed into SQL as:

SELECT AVG(x.salary - computeTax(x.salary)) FROM Employee x WHERE x.salary > 120000;

As a result, very few interactions (typically only one) need to be made between the middle and the backend tier.

Views and indexes making use of computed values. In the above example and index could be created on computeTax(x.salary) and a view could express that as net_income.

Message queue management. Delivering or fetching things using message queues or other delivery mechanisms. As with most interactions with other processes, this requires transaction coordination of some kind.

One might argue that since a JVM often is present running an app-server in the middle tier, would it not be more efficient if that JVM also executed the database functions and triggers? In my opinion, this would be very bad. One major reason for moving execution down to the database is performance (by minimizing the number of roundtrips between the app-server and the database) another is separation of concern. Referential data integrity and other ways to extend the functionality of the database should not be the app-servers concern, it belongs in the backend tier. Other aspects like database versus app-server administration, replication of code and permission changes for functions, and running different tiers on different servers, makes it even worse.

Resource consumption

Having one JVM per connection instead of one thread per connection running in the same JVM will undoubtedly consume more resources. There are however a couple of facts that must be remembered:

The overhead of multiple processes is already present due to the fact that each connection is a process in a PostgreSQL system.

In order to keep connections separated in case they run in the same JVM, some kind of “compartments” must be created. Either you create them using parallel

class loader chains (similar to how EAR files are managed in an EJB server) or you use a less protective model similar to a servlet engine. In order to get a separation that comparable to what you get using separate JVM's, you have to go for the former. That consumes some resources.

The JVM has undergone a series of improvements in order to reduce footprint and startup time. Some significant improvements where made in Java 1.4 and Java 1.5 introduces Java Heap Self Tuning, Class Data Sharing, and Garbage Collector Ergonomics (read more here), technologies that will minimize the startup time and make the JVM adopt its resource consumption in a much improved way.

PL/Java can make use of the GCJ. Using this technology, all core classes will be compiled into binaries and optionally pre-loaded by the postmaster. It also means that all modules that are loaded using the install_jar/replace_jar can be compiled into real shared objects. Finally, it means that the footprint for each "JVM" will be significantly decreased.

Connection pooling

In the Java community you are very likely to use a connection pool. The pool will ensure that the number of connections stays as low as possible and that connections are reused (instead of closed and reestablished). New JVMs are started rarely.

Connection isolation

Separate JVMs gives you a much higher degree of isolation. This brings a number of advantages:

There's no problem attaching a debugger to one connection (one JVM) while the others run unaffected.

There's no chance that one connection manages to accidentally (or maliciously) exchange dirty data with another connection.

A process that performs tasks that consume a lot of CPU under a long period of time can be scheduled with a lower priority using a simple OS command.

The JVMs can be brought down and restarted individually.

Security policies are much easier to enforce.

Transaction visibility

In order to maintain the correct visibility, the transaction must somehow be propagated to the Java layer. I can see two solutions for this using RPC. Either an XA aware JDBC-driver is used (requires XA support from PostgreSQL) or a JDBC driver is written so that it calls back to the SPI functions in the invoking process. Both choices results in an increased number of RPC calls and a negative performance impact.

The PL/Java approach is to use the underlying SPI interfaces directly through JNI by providing a "pseudo connection" that implements the JDBC interfaces. The mapping is thus very direct. Data need never be serialized nor duplicated.

RPC performance

Remote procedure calls are extremely expensive compared to in-process calls. Relying on an RPC mechanism for Java calls will cripple the usefulness of such an implementation a great deal. Here are two examples:

In order for an update trigger to function using RPC, you can choose one of two approaches. Either you limit the number of RPC calls and send two full Tuples (old and new) and a Tuple Descriptor to the remote JVM, and then pass a third Tuple (the modified new) back to the original, or you pass those structures by reference (as CORBA remote objects) and perform one RPC call each time you access them. You have a tradeoff between on one hand, limited functionality and poor performance, and on the other, good functionality and really bad performance.

When one or several Java functions are used in the projection or filter of a SELECT statement on a query processing several thousand rows, each row will cause at least one call to Java. In case of RPC, this implies that the OS needs to do at least two context switches (back and forth) for each row in the query.

Using JNI to directly access structures like TriggerData, Relation, TupleDesc, and HeapTuple minimizes the amount of data that needs to be copied. Parameters and return values that are primitives need not even become Java objects. A 32-bit int4 Datum can be directly passed as a Java int (jint in JNI).

Simplicity

I've have some experience of work involving CORBA and other RPCs. They add a fair amount of complexity to the process. JNI however, is invisible to the user.

PL/Java readme

PL/Java, 1.2.x

Java™ is a registered trademark of Sun Microsystems, Inc. in the United States and other countries.

A source tarball and four of binary builds can be found in the download area. Two for Windows and two for i386 based Linux. Although all pre-compiled binaries must run using a standard JVM, the PL/Java can also be compiled and linked using GNU GCJ. PL/Java has no pre-compiled binaries for GCJ but the make system contain what's needed to make the build easy.

Please note that all use of GCJ should be regarded as experimental. Due to limitations in the implementation of java.security in some versions of GCJ the PL/Java trusted language implementation is, in fact, not trusted. At present

this applies regardless of what GCJ version since the code is conditionally compiled.

Prerequisites

PostgreSQL >= 8.0.3

PostgreSQL JDBC drivers (needed by the client Deployer program).

A Java runtime >= Java 1.4 or GCJ >= 4.0.x (Linux only).

Get the binary distribution of PL/Java for your platform. Unzip it into a directory of your own choice.

Postmaster configuration

Get the PostgreSQL environment up and running. You will need to modify the postgresql.conf file. In order to find the PL/Java shared object, you can do one of two things. Either you install the shared object in a directory already searched by the postmaster (such as the data directory) or you tell the postmaster where to find it using the dynamic_library_path. I.e. you have a setting similar to this:

Note that on the win32 platform you need to use a semicolon as a path separator and double backslashes (since backslash is the escape character in the postgresql.conf file) as directory separators.

In order to see the logging from the tests add the following:

Add the following entry:

System classpath

Normally, all Java code is loaded into the database using the install_jar/replace_jar SQL functions. Most of PL/Java (those functions included) is however implemented in Java. Unless you use GCJ, where this Java code is compiled and linked with the pljava shared object module, this hen and egg problem needs to be resolved using the system classpath. Add the following entry to the postgresql.conf file:

Shared object issues

Unless you use GCJ, the postmaster must be made aware of the location of the shared objects used by the Java Runtime Environment (JRE). Please note that this applies to the postmaster, i.e. the backend process, and not to the client. The client will not need these settings.

Linux/Unix

Setting the LD_LIBRARY_PATH environment will work on most Linux/Unix platforms:

Apparently, on some Linux platforms you will also need to include $JAVA_HOME/jre/lib/i386/native_threads.

An alternative to use LD_LIBRARY_PATH is to edit the /etc/ld.so.conf file and then use /sbin/ldconfig on some Linux/Unix systems.

zlib conflict

On some platforms there will be a conflict between the libzip.so included in the JRE and the libz.so used by PostgreSQL (the JRE libzip.so includes a libz.so). The symptom is an InternalError in the java.util.zip.Inflater.init when an attempt is made to load the first class. You can verify the version of libzip.so using the following command:

The problem can be resolved in one of the following ways depending on your needs and ability to recompile:

Check if you can use a newer version of your JRE where the versions match. If so, that's probably the best solution.

Set the environment LD_PRELOAD in effect for the postmaster process to point to the libzip.so present in the JRE. That will force the JRE version to be used. This might break postgres own use of compression for the affected processes.

Reconfigure PostgreSQL with –without-zlib, recompile and reinstall. This will effetively disable all compression support in PostgreSQL.

Obtain a zlib version from somewhere that corresponds to the version used by the JRE and relink your PostgreSQL executables with that version.

Windows

A standard install on a Windows box would be to add the following entries to you PATH environment:

You are now ready to start the postmaster.

Deploying the PL/Java

PL/Java adds a schema named SQLJ to the database (the naming is from the proposed SQL standard for Java backend mapping) and adds a couple of tables and functions to that schema. The deployment can be done in one of two ways. The simplest way is probably to just execute the file install.sql as a super user (the uninstall.sql will remove the PL/Java installation). PL/Java also comes with deploy program that lets you install, reinstall, or uninstall PL/Java. This program will assert that you indeed are a super user and then execute the correct commands using jdbc. In order to run this program, you must see to that the PostgreSQL jdbc driver package postgresql.jar and the deploy.jar file is in your CLASSPATH, then run:

This will result in a list of options. Typically you would use something like:

That's all there's to it. You are now ready to start using the PL/Java system.

Run the example tests

The tests are divided into two jar files. One is the client part found in the test.jar. It contains some methods that executes SQL statements and prints the output (all contained there can of course also be executed from psql or any other client). The other is the examples.jar which contains the sample code that runs in the backend. The latter must be installed in the database in order to function. An easy way to do this is to use psql and issue the command:

Please note that the deployment descriptor stored in examples.jar will attempt to create the schema javatest so the user that executes the sqlj.install_jar must have permission to do that. If this command succeeds, everything is working correctly. You may get a couple of errors here though.

A complaint that the class org.postgresql.pljava.<something> cannot be found. The probable cause of this is that the CLASSPATH seen by the postmaster is incorrect so that the pljava.jar is not found.

A complaint that the libpljava.so or pljava.dll cannot be found. Probable cause is that the dynamic_library_path in the postgresql.conf file is incorrect.

Once loaded, you must also set the classpath used by the PL/Java runtime. This classpath is set per schema (namespace). A schema that lacks a classpath will default to the classpath that has been set for the public schema. The tests will use the schema javatest. To define the classpath for this schema, simply use psql and issue the command:

The first argument is the name of the schema, the second is a colon separated list of jar names. The names must reflect jars that are installed in the system.

NOTE: If you don't use schemas, you must still issue the set_classpath command to assign a correct classpath to the 'public' schema. This can only be done by a super user.

Now, you should be able to run the tests:

Building

Building should be very stright forward:

No PosgreSQL source is needed. Your path must be set to find the pg_config binary. The 'pgxs' concept in PostgreSQL will take care of the rest.

If you are using GCJ, you must supply USE_GCJ=1 to the make command.

If you are using a normal Java VM, be sure to set the JAVA_HOME environment variable.

Some problems and their solution

Some problems and their solution.

When writing the PL/Java, mapping the JVM into the same process-space as the PostgreSQL backend code, some concerns have been raised regarding multiple threads, exception handling, and memory management. Here is a brief text explaining how these issues where resolved.

Multi threading.

Problem

Java is inherently multi threaded. The PostgreSQL backend is not. There's nothing stopping a developer from utilizing multiple Threads class in the Java code. Finalizers that call out to the backend might have been spawned from a background Garbage Collection thread. Several third party Java-packages that are likely to be used make use of multiple threads. How can this model coexist with the PostgreSQL backend in the same process without creating havoc?

Solution

The solution is simple. PL/Java defines a special object called the Backend.THREADLOCK. When PL/Java is initialized, the backend will immediately grab this objects monitor (i.e. it will synchronize on this object). When the backend calls a Java function, the monitor is released and then immediately regained when the call returns. All calls from Java out to backend code are synchronized on the same lock. This ensures that only one thread at a time can call the backend from Java, and only at a time when the backend is awaiting the return of a Java function call.

Exception handling

Problem

Java makes frequent use of try/catch/finally blocks. PostgreSQL sometimes use an exception mechanism that calls longjmp to transfer control to a known state. Such a jump would normally effectively bypass the JVM. Prior to PostgreSQL version 8.0, the error was propagated before the actual jump and then discarded, thus there was no way to catch and handle the error.

Solution

The backend now allows errors to be caught using the macros PG_TRY/PG_CATCH/PG_END_TRY and in the catch block, the error can be examined using the ErrorData structure. PL/Java implements a java.sql.SQLException subclass called org.postgresql.pljava.ServerException. The ErrorData can be retrieved and examined from that exception. A catch handler is allowed to issue a rollback to a savepoint. After a successful rollback, execution can continue.

Java Garbage Collector versus palloc() and stack allocation.

Problem

Primitive types will be passed by value always. This includes the String type (this is a must since Java uses double byte characters). Complex types are

however often wrapped in Java objects and passed by reference. I.e, a Java object will contain a pointer to a palloc'ed or stack allocated memory and use native JNI calls to extract and manipulate data. Such data will become "stale" once a call has ended. Further attempts to access such data will at best give very unpredictable results but more likely cause a memory fault and a crash.

Solution

The PL/Java contains code that ensures that stale pointers are cleared when the MemoryContext or stack where they where allocated goes out of scope. The Java wrapper objects might live on but any attempt to use them will result in a "stale native handle" exception.

PL/Java 1.2 User Guide

PL/Java 1.2 User Guide

Table of contents

Utilities

Deployer

When running the deployer, you must use a classpath that can see the deploy.jar found in the PL/Java distribution and the postgresql.jar from the PostgreSQL distribution. The former contains the code for the deployer command and the second includes the PostgreSQL JDBC driver. You then run the deployer with the command:

It's recommended that create a shell script or a .bat script that does this for you so that you don't have to do this over and over again.

Deployer options

-install

Installs the Java™ language along with the sqlj procedures. The deployer will fail if the language is installed already.

-reinstall

Reinstalls the Java™ language and the sqlj procedures. This will effectively drop all jar files that have been loaded.

-remove

Drops the Java™ language and the sqlj procedures and loaded jars.

-user <user name>

Name of user that connects to the database. Default is the current user.

-password <password>

Password of user that connects to the database. Default is no password.

-database <database>

The name of the database to connect to. Default is to use the user name.

-host <host name>

Name of the host. Default is “localhost”.

-port <port number>

Port number. Default is blank.

-cygwin

Use this option if the host runs on a Cygwin based windows platform. Affects the name used for the PL/Java dynamic library.

NOTE This option should not be used when running native the Win32 port.

Deploying using SQL

An alternative to using the deployer is to run the install.sql and uninstall.sql scripts that are included in the distribution.

SQLJ functions

Deployment descriptor

The install_jar, replace_jar, and remove_jar can act on a deployment descriptor allowing SQL commands to be executed after the jar has been installed or prior to removal. The format of the deployment descriptor is stipulated by ISO/IEC 9075-13:2003.

The descriptor is added as a normal text file to your jar file. In the Manifest of the jar there must be an entry that appoints the file as the SQLJ deployment descriptor.

The deployment descriptor must have the following form:

If implementor blocks are used, PL/Java will consider only those with implementor name PostgreSQL (case insensitive). Here is a small sample of a deployment descriptor:

install_jar

The install_jar command loads a jarfile from a location appointed by an URL into the SQLJ jar repository. It is an error if a jar with the given name already exists in the repository.

Usage

Parameters

jar_url

The URL that denotes the location of the jar that should be loaded.

jar_name

This is the name by which this jar can be referenced once it has been loaded.

deploy

True if the jar should be deployed according to a deployment descriptor, false otherwise.

replace_jar

The replace_jar will replace a loaded jar with another jar. Use this command to update already loaded files. It's an error if the jar is not found.

Usage

Parameters

jar_url

The URL that denotes the location of the jar that should be loaded.

jar_name

The name of the jar to be replaced.

redeploy

True if the jar should be undeployed according to the deployment descriptor of the old jar and deployed according to the deployment descriptor of the new jar, false otherwise.

remove_jar

The remove_jar will drop the jar from the jar repository. Any classpath that references this jar will be updated accordingly. It's an error if the jar is not found.

Usage

Parameters

jar_name

The name of the jar to be removed.

undeploy

True if the jar should be undeployed according to a deployment descriptor, false otherwise.

get_classpath

The get_classpath will return the classpath that has been defined for the given schema or NULL if the schema has no classpath. It's an error if the given schema does not exist.

Usage

Parameters

schema

The name of the schema.

set_classpath

The set_classpath will define a classpath for the given schema. A classpath consists of a colon separated list of jar names. It's an error if the given schema does not exist or if one or more jar names references non existent jars.

Usage

Parameters

schema

The name of the schema.

classpath

The colon separated list of jar names.

Writing Java functions

SQL declaration

A Java function is declared with the name of a class and a static method on that class. The class will be resolved using the classpath that has been defined for the schema where the function is declared. If no classpath has been defined for that schema, the "public" schema is used. If no classpath is found there either, the class is resolved using the system classloader.

The following function can be declared to access the static method getProperty on java.lang.System class:

Type mapping

Scalar types are mapped in a straight forward way. Here's a table of the current mappings (will be updated as more mappings are implemented).

PostgreSQL

Java

bool

boolean

‘char’

byte

int2

short

int4

int

int8

long

float4

float

float8

double

varchar

java.lang.String

text

java.lang.String

bytea

byte[]

date

java.sql.Date

time

java.sql.Time (stored value treated as local time)

timetz

java.sql.Time

timestamp

java.sql.Timestamp (stored value treated as local time)

timestamptz

java.sql.Timestamp

complex

java.sql.ResultSet

setof complex

java.sql.ResultSet

All other types are currently mapped to java.lang.String and will utilize the standard textin/textout routines registered for respective type.

NULL handling

The scalar types that map to Java primitives can not be passed as null values. To enable this, those types can have an alternative mapping. You enable this mapping by explicitly denoting it in the method reference.

In java, you would have something like:

The following two statements should both yield true:

In order to return null values from a Java method, you simply use the object type that corresponds to the primitive (i.e. you return java.lang.Integer instead of int). The PL/Java resolve mechanism will find the method regardless. Since Java cannot have different return types for methods with the same name, this does not introduce any ambiguity.

Complex types

A complex type will always be passed as a read-only java.sql.ResultSet with exaclty one row. The ResultSet will be positioned on its row so no call to next should be made. The values of the complex type are retrieved using the standard getter methods of the ResultSet.

Example:

In class Fum we add the static following static method:

Returning complex types

Java does not stipulate any way to create a ResultSet from scratch. Hence, returning a ResultSet is not an option. The SQL-2003 draft suggest that a complex return value instead is handled as an IN/OUT parameter and PL/Java implements it that way. If you declare a function that returns a complex type, you will need to use a Java method with boolean return type with a last parameter of type java.sql.ResultSet. The parameter will be initialized to an empty updateable ResultSet that contains exactly one row.

Assume that we still have the complexTest type created above.

The PL/Java method resolve will now find the following method in the Fum class:

The return value denotes if the receiver should be considered as a valid tuple (true) or NULL (false).

Functions returning sets

Returning sets is tricky. You don't want to first build a set and then return it since large sets would eat too much resources. Its far better to produce one row at a time. Incidentally, that's exactly what the PostgreSQL backend expects a function with SETOF return to do. You can return a SETOF a scalar type such as an int, float or varchar, or you can return a SETOF a complex type.

Returning a SETOF <scalar type>

In order to return a set of a scalar type, you need create a Java method that returns something that implements the java.util.Iterator interface. Here's an example of a method that returns a SETOF varchar:

The very rudimentary java method that returns an interator:

Returning a SETOF <complex type>

A method returning a SETOF <complex type> must use either the interface org.postgresql.pljava.ResultSetProvider or org.postgresql.pljava.ResultSetHandle. The reason for having two interfaces is that they cater for optimal handling of two distinct use cases. The former is great when you want to dynamically create each row that is to be returned from the SETOF function. The latter makes sense when you want to return the result of an executed query.

Using the ResultSetProvider interface

This interface has two methods. The boolean assignRowValues(java.sql.ResultSet tupleBuilder, int rowNumber) and the void close() method. The PostgreSQL query evaluator will call the assignRowValues repeatedly until it returns false or until the evaluator decides that it does not need any more rows. It will then call close.

You can use this interface the following way:

The function maps to a static java method that returns an instance that implements the ResultSetProvider interface.

The listComplextTests method is called once. It may return null if no results are available or an instance of the ResultSetProvider. Here the Fum implements this interface so it returns an instance of itself. The method assignRowValues will then be called repeatedly until it returns false. At that time, close will be called

Using the ResultSetHandle interface

This interface is similar to the ResultSetProvider interface in that it has a close() method that will be called at the end. But instead of having the evaluator call a method that builds one row at a time, this method has a method that returns a ResultSet. The query evaluator will iterate over this set and deliver it's contents, one tuple at a time, to the caller until a call to next() returns false or the evaluator decides that no more rows are needed.

Here is an example that executes a query using a statement that it obtained using the default connection. The SQL suitable for the deployment descriptor looks like this:

And in the Java package org.postgresql.pljava.example a class Users is added:

Triggers

The method signature of a trigger is predefined. A trigger method must always return void and have a org.postgresql.pljava.TriggerData parameter. No more, no less. The TriggerData interface provides access to two ResultSet instances; one representing the old row and one representing the new. The old row is read-only, the new row is updateable.

The sets are only available for triggers that are fired ON EACH ROW. Delete triggers have no new row, and insert triggers have no old row. Only update triggers have both.

In addition to the sets, several boolean methods exists to gain more information about the trigger. Here's an example trigger:

The Java method in class org.postgresql.pljava.example.Triggers looks like this:

Using JDBC

PL/Java contains a JDBC driver that maps to the PostgreSQL SPI functions. A connection that maps to the current transaction can be obtained using the following statement:

From there on, you can prepare and execute statements, just like with any other JDBC connection. There are a couple of limitations though:

The transaction cannot be managed in any way. Thus, you cannot use methods on the connection like:

commit()

rollback()

setAutoCommit()

setTransactionIsolation()

Savepoints are available but only if you use PostgreSQL 8.0 or later and with some restrictions. A savepoint cannot outlive the function in which it was set and it must also be rolled back or released by that same function.

ResultSet's returned from executeQuery() are always FETCH_FORWARD and CONCUR_READ_ONLY.

Meta-data is only available in PL/Java 1.1 or higher.

CallableStatement (for stored procedures) is not yet implemented.

Clob/Blob types need more work. byte[] and String works fine for bytea/text respectively. A more efficient mapping is planned where the actual array is not copied.

Exception handling

You can catch and handle an exception in the Postgres backend just like any other exceptoin. The backend ErrorData structure is exposed as a property in a class called org.postgresql.pljava.ServerException (derived from java.sql.SQLException) and the Java try/catch mechanism is synchronized with the backend mechanism.

Important Note:

You will not be able to continue executing backend functions until your function has returned and the error has been propagated when the backend has generated an exception unless you have used a savepoint. When a savepoint is rolled back, the exceptional condition is reset and you can continue your execution.

Savepoints

PostgreSQL savepoints are exposed using the java.sql.Connection interface. Two restrictions apply.

A savepoint must be rolled back or released in the function where it was set.

A savepoint must not outlive the function where it was set

Logging

PL/Java uses the standard Java 1.4 Logger. Hence, you can write things like:

At present, the logger is hardwired to a handler that maps the current state of the PostgreSQL configuration setting log_min_messages to a valid Logger level and that outputs all messages using the backend function elog(). The following mapping apply between the Logger levels and the PostgreSQL backend levels.

java.util.logging.Level

PostgreSQL level

SEVERE

ERROR

WARNING

WARNING

INFO

INFO

FINE

DEBUG1

FINER

DEBUG2

FINEST

DEBUG3

Security

Installation

Only a PostgreSQL super user can install PL/Java. The PL/Java utility functions are installed using SECURITY DEFINER so that they execute with the access permissions that where granted to the creator of the functions.

Trusted language

PL/Java is now a TRUSTED language. PostgreSQL stipulates that a language marked as trusted has no access to the filesystem and PL/Java enforces this. Any user can create and access functions or triggers in a trusted language. PL/Java also installs a language handler for the language "javaU". This version is not trusted and only a superuser can create new functions that use it. Any user can still call the functions.

Execution of the deployment descriptor

The install_jar, replace_jar, and remove_jar, optionally executes commands found in a SQL deployment descriptor. Such commands are executed with the permissions of the caller. In other words, although the utility function is declared with SECURITY DEFINER, it switches back to the session user during execution of the deployment descriptor commands.

Classpath manipulation

The function set_classpath requires the caller of the function has been granted CREATE permission on the affected schema.

Module Configuration.

PL/Java makes use of PostgreSQL custom variable classes in the postgresql.conf configuration file to add some configuration parameters. PL/Java introduces a custom variable class named "pljava". Here's a sample postgresql.conf entry using all (3) of the variables currently introduced: