Using the Python Database API with Red Hat Database

by Patrick Macdonald (patrickm@redhat.com) of Red Hat, Inc.

The Python Database API (DB-API) is a product-neutral interface to access and manipulate data stored in database management systems that allows developers to write database-independent Python applications. This document provides an overview of how to use the Python Database API with Red Hat Database (RHDB), a powerful and robust open-source database solution.

Introduction

Python is a portable, platform-independent, open-source programming language that has gained favor with developers for its versatility and ease of use. Python is an interpreted, scripting language that allows developers to do everything from creating web sites, to developing applications and scripts, to accessing data in databases.

The Python Database API (DB-API) Specification was produced by the Python Database Special Interest Group (SIG) to provide a product-neutral interface for accessing and manipulating data stored in database management systems. The DB-API allows developers to write applications that are transportable between database products. Red Hat Database (RHDB) ships with a Python Database API module, `pgdb.py`, which implements the Python Database API Specification Version 2.0.

This document provides an overview of how to use the Python Database API with Red Hat Database, a powerful and robust open-source database solution. We first present how to obtain and install the appropriate Python and database environment, then we use a simple example application to illustrate some basic aspects of the Python Database API.

Getting Ready to Use the Python Database API with Red Hat Database

To use the Python Database API, you need the Python interpreter as well as the Red Hat Database implementation of the Python DB-API, `pgdb.py`.

Getting and Installing Python and the Database API Code

Python is freely available at `http://www.python.org`. You may download whichever version of Python you like in whatever format you prefer. Red Hat Linux 7.1 ships with Python version 1.5.2.

Red Hat Database ships with the RHDB implementation of the Python Database API. The module is contained in the `rh-postgresql-python` package. Once installed, you can find the Red Hat Database Python Database API module in `/usr/lib/python1.5/site-packages`. This module, `pgdb.py`, is compliant with the Python Database API Specification Version 2.0. Refer to the *Red Hat Database Getting Started / Installation Guide* for details on installing this package.

The Python Database API Specification states that the preferred object types for the date/time objects are those defined in the `mxDateTime` package. Red Hat Database ships with the `mx-2.0.1-1.i386.rpm`, which is the 2.0.1 version of the eGenix.com `mx` Extensions for Python. Included in this package is the `mxDateTime` module. This RPM is installed when the `rh-postgresql-python` package is installed.

Setting Up Red Hat Database

As the RHDB Python Database API implementation uses TCP/IP to communicate with the database server, the `postmaster` must be started with TCP/IP enabled. To enable TCP/IP, you can modify the `tcpip_socket` setting in the `postgresql.conf` file and restart the `postmaster`. Alternatively, you can start that `postmaster` with the `-i` option. Refer to the *Red Hat Database Administrator and User's Guide* for further information on enabling TCP/IP connections.

Python Database API Version 2.0 Concepts

As mentioned, the purpose of the Python Database API is to provide a database neutral interface from Python to access data stored in database management systems.

The Python Database API Specification consists of the following sections:

- Module interface
- Connection objects
- Cursor objects
- Type objects and constructors

Module Interface

Access to RHDB databases is made available through connection objects, and the Python module provides a constructor for the connection that returns a Connection object proper. The interface also defines a set of global variables that describe the API. See the section “Code: example.py” on page 13 for the `about()` method, which displays some of this information for the RHDB Python Database API implementation.

Connection Objects

Once a connection has been established, you can use the Connection Object for transaction control (commits and rollbacks) and cursor object instantiation.

Cursor Objects

These objects represent database cursors. Cursor Object methods allow the user to query and manipulate database data and also describe the contents of the cursor itself.

Type Objects and Constructors

Type Objects allow database-independent data types to be used when programming with the Python Database API. These types and constructors are mapped in the DB-API implementation to the underlying RHDB data types.

Conceptual Level Example

Given the concepts from the Python Database API Specification, and taking an object-level view, let’s walk through the process of connecting to a database, inserting

a row, and then disconnecting:

- Connect to the database by creating a Connection Object
  - Use the Connection Object to create a Cursor Object
    - Insert a row using the Cursor Object
    - Close the Cursor Object
  - Commit the transaction using the Connection Object
- Disconnect from the database using the Connection Object.

Sample Application

In this section we examine various aspects of the Python Database API using a sample program. The topics that will be covered are:

- Database schema
- Loading the RHDB Python Database API module and connecting to a database
- Performing queries and updates
- Retrieving results
- Transactions
- Closing a cursor and connection.

The examples will focus on using the Python Database API interactively with the Python interpreter. Python, and the Python Database API, can also be used to construct web pages that access Red Hat Database clusters. This feature is beyond the scope of this document but information is available in “References and Further Reading” on page 8.

Database Schema

For all of the sample code, we assume that a database called `basketball` has been created. The `basketball` database contains one table called `players`, which contains a player's name and team.

The table definition is as follows:

```
CREATE TABLE players (
   name varchar(25),
   team varchar(20)
)
```

and is populated with the following data:

```
INSERT INTO players VALUES ('Michael Jordan', 'Washington Wizards')
INSERT INTO players VALUES ('Tim Duncan', 'San Antonio Spurs')
INSERT INTO players VALUES ('Vince Carter', 'Toronto Raptors')
```

A function in the `example.py` module, `cleantable()`, creates the `players` table as defined above. See the section “Code: example.py” on page 13.

Loading the RHDB Python Database API Module and Connecting to a Database

In order to connect to an RHDB database using the Python Database API, you must load the RHDB Python Database API module, `pgdb.py`.

Start the Python interpreter:

```
$ python
```

and then load the RHDB Python Database API module:

```
>>> import pgdb
```

This allows access to the objects and methods defined in the API.

To connect to a database, call the `connect()` method of the Connection Object. The parameters of the `connect()` method are as follows:

```
dsn      = data source name (host:database)
user     = user name / identifier (optional)
password = password for given user (optional)
host     = host name (optional)
database = database name (optional)
```

The Python Database API states as a guideline that the connection parameters should be implemented as keyword parameters and follow the above ordering.

To connect to a database named `basketball` on my local host, using user `patrickm` and password `fakepswd`:

```
>>> dbConnect = pgdb.connect (dsn='localhost:basketball',
      user='patrickm', password='fakepswd')
```

The above statement creates a local Connection Object, `dbConnect`.

Performing Queries and Updates

To perform queries and to update data (that is, to execute INSERT, UPDATE, or DELETE commands), you need to allocate a Cursor Object by invoking the `cursor()` method of your newly allocated Connection Object `dbConnect`.

```
>>> cursor = dbConnect.cursor ()
```

The `execute()` method of the Cursor Object prepares and executes a database operation. A reference to the operation performed will be retained by the cursor to allow result set iteration.

To perform a query on the players table:

```
>>> cursor.execute (“select * from players”)
```

To perform a delete on the players table:

```
>>> cursor.execute(“delete from players where name = 'Michael Jordan'”)
```

Retrieving Results

The Python Database API supports the concept of retrieving one or more rows from a result set using the Cursor Object's `fetchone()`, `fetchmany()`, and `fetchall()` methods. Given the following query:

```
>>> cursor.execute (“select * from players”)
```

To retrieve one row from the result set, call the `fetchone()` method of the Cursor Object:

```
>>> row = cursor.fetchone ()
```

If there are no more rows in the result set, `fetchone()` will return `None`; otherwise it will return the row/tuple.

Let's look at an example of iterating through the result set one row at a time and displaying the result:

```
>>> cursor.execute (“select * from players”)
>>> while (1):
...   row = cursor.fetchone()
...   if row == None:
...     break
...   print row
```

As mentioned, there are other methods of the Cursor Object that return rows from result sets. Call the `fetchall()` method to display all remaining rows of the result set:

```
>>> cursor.fetchall ()
```

The `fetchmany()` method returns the next set of rows from the result set, based on the method call parameter, as a list of rows. To return the next two rows of a result set, call the `fetchmany()` method with a size of `2`:

```
>>> cursor.fetchmany (2)
```

Transactions

A transaction is a grouping of SQL statements that are perceived by the database as one atomic action. They are guaranteed to either all succeed or all fail. Transactions are implicitly started when a Cursor Object is created and are explicitly ended using the transaction-ending method calls, `commit()` or `rollback()`, of the Connection Object. These transaction ending method calls implicitly start a new transaction after ending the current one. Note that since Red Hat Database supports an auto-commit feature, it is initially turned off as per the Python Database API Specification.

To commit a transaction and make the results visible to the rest of the system, call the `commit()` method of the Connection Object:

```
>>> dbConnect.commit ()
```

To rollback a transaction, call the `rollback()` method of the Connection Object:

```
>>> dbConnect.rollback ()
```

Closing a Cursor and Connection

To close a cursor, call the `close()` method of the Cursor Object:

```
>>> cursor.close ()
```

The cursor will be unusable after the `close()` method call. Any operation attempted with the cursor after it is closed will raise an exception.

As with closing a cursor, closing a connection is very simple. All you have to do is call the `close()` method of the Connection Object.

```
>>> dbConnect.close ()
```

The connection will be unusable after the `close()` method call. Any operation attempted with the connection after it is closed will raise an exception.

References and Further Reading

- *Red Hat Database Administrator and User's Guide*
- *Red Hat Database Getting Started / Installation Guide*
- Red Hat Database: `http://www.redhat.com/software/database/`
- Red Hat Database Community Project: `http://sources.redhat.com/rhdb`
- Python Official Site: `http://www.python.org`
- Python Database Special Interest Group (SIG) `http://www.python.org/sigs/db-sig`
- Python Database API Specification 2.0 `http://www.python.org/topics/database/DatabaseAPI-2.0.html`
- PostgreSQL Global Development Group: `http://www.postgresql.org`

Code Samples

This section lists the Python functions resident in the `example.py` module, which you can see at “Code: example.py” on page 13. All functions in this module interact with RHDB using the Python DB-API.

- `about()`: Display information about the RHDB Python DB-API
- `initialize()`: Connect to an RHDB database
- `disconnect()`: Disconnect from an RHDB database
- `cleantable()`: Create a pristine version of the data for use by other functions
- `viewtable()`: Display the contents of a table
- `transaction()`: Commit or rollback a transaction which modifies table data
- `update()`: Update table data

Running the Sample Module (example.py)

1. If they are not already installed, get and install the `rh-postgresql-python` and `mx-2.0.1-1` packages. You can find these on your Red Hat Database CD; updates are available from Red Hat Network.
2. To accept TCP/IP connections, edit the configuration file to allow TCP/IP connections. For a standard installation of Red Hat Database, the configuration file is located at `/var/lib/pgsql/data/postgresql.conf.` Modify and uncomment the value of the `tcpip_socket` from `false` to `true` so that TCP/IP connections are allowed by default every time the `postmaster` starts up. Alternatively, you could (re)start the `postmaster` with the `-i` option.
3. Start the `postmaster`.
4. Create a database called `basketball` using your userid. If your userid is not allowed to create databases, talk to your database administrator or refer to the *Red Hat Database Administrator and User's Guide* for instructions on how to grant “create database” permission to a user.
5. Create the sample python module, `example.py.` (See “Code: example.py” on page 13).
6. Make sure `example.py` is accessible to the Python interpreter. You may need to update the `PYTHONPATH` environment variable on your system to include the directory in which `example.py` resides.
7. Invoke the Python interpreter on the command line:

```
$ python
```

This creates an interactive session for you. You should see a "`>>>`" prompt.

8. Import the example module to allow access to the defined functions. The Red Hat Database Python DB-API module, `pgdb.py`, is imported by the `example.py` module:

```
>>> import example
```

9. Display information about the RHDB Python Database API. A function, `about()`, has been provided to display this information:

```
>>> example.about ()
```

Displays:

```
*****************************************
 About the Red Hat Database Python DB-API
   DB-API level: 2.0
   DB-API Thread Safety level: 1
   DB-API Parameter Formatting: pyformat
*****************************************
```

10. Create a table called `players` within the `basketball` database and seed the table with information as listed under "Database Schema" on page 4. A function, `cleantable()` will drop the `players` table (if it exists), create the table, seed it, and display the table data.

```
>>> example.cleantable ()
```

Displays:

```
Dropping Players table...
   Exception encountered: Table does not exist. Continuing...
   Creating and seeding Players table...
      Players table successfully created.

Content of Players table:
*************************
 Michael Jordan -- Washington Wizards
 Tim Duncan -- San Antonio Spurs
 Vince Carter -- Toronto Raptors

Number of players: 3
```

11. The `transaction()` function is an example that contains a transaction with the following SQL statements:

```
insert into players values ('Tracy McGrady','Orlando Magic')
insert into players values ('Kobe Bryant','NY Knicks')
delete from players where name = 'Michael Jordan'
```

To execute the SQL statements in a transaction and rollback at the end of the transaction:

```
>>> example.transaction (“rollback”)
```

Displays:

```
Before:
Content of Players table:
*************************
 Michael Jordan -- Washington Wizards
 Tim Duncan -- San Antonio Spurs
 Vince Carter -- Toronto Raptors

Number of players: 3

About to issue the following commands:
  insert into players values ('Tracy McGrady','Orlando Magic')
  insert into players values ('Kobe Bryant','NY Knicks')
  delete from players where name = 'Michael Jordan'

About to rollback the transaction...

After:
Content of Players table:
*************************
 Michael Jordan -- Washington Wizards
 Tim Duncan -- San Antonio Spurs
 Vince Carter -- Toronto Raptors

Number of players: 3
```

To execute the SQL statements in a transaction and commit at the end of the transaction:

```
>>> example.transaction (“commit”)
```

Displays:

```
Before:
Content of Players table:
*************************
 Michael Jordan -- Washington Wizards
 Tim Duncan -- San Antonio Spurs
 Vince Carter -- Toronto Raptors
```

```
Number of players: 3

About to issue the following commands:
  insert into players values ('Tracy McGrady','Orlando Magic')
  insert into players values ('Kobe Bryant','NY Knicks')
  delete from players where name = 'Michael Jordan'

About to commit the transaction...

After:
Content of Players table:
*************************
 Kobe Bryant -- NY Knicks
 Tim Duncan -- San Antonio Spurs
 Tracy McGrady -- Orlando Magic
 Vince Carter -- Toronto Raptors

Number of players: 4
```

12. You may notice that Kobe Bryant's team is listed incorrectly. Perform an update on the `players` table to change Kobe's team to the LA Lakers. In the example function, `update()`, the current data for Kobe is returned, an SQL update is performed and Kobe's data is requeried. The query result sets are not formatted in this example.

```
>>> example.update ()
```

Displays:

```
Before: [['Kobe Bryant', 'NY Knicks']]

Updating...

After: [['Kobe Bryant', 'LA Lakers']]

Content of Players table:
*************************
 Kobe Bryant -- LA Lakers
 Tim Duncan -- San Antonio Spurs
 Tracy McGrady -- Orlando Magic
 Vince Carter -- Toronto Raptors

Number of players: 4
```

13. Press Ctrl-D to exit from the Python interpreter.

Code: example.py

```
#! /usr/bin/env python
#
# example.py: Using the Python Database API (DB-API) with Red Hat
#             Database
#
# This example module utilizes the Python DB-API to create and
# modify data within an RHDB database.  These examples have been
# created to display the concepts of the Red Hat Database Python
# DB-API.

# Allow the Python DBI (pgdb.py) to be accessible
import pgdb
import string

# Dump information about the Red Hat Database Python DB-API
def about ():
  print "\n*****************************************"
  print " About the Red Hat Database Python DB-API"
  print "   DB-API level: %s" % pgdb.apilevel
  print "   DB-API Thread Safety level: %d" % pgdb.threadsafety
  print "   DB-API Parameter Formatting: %s" % pgdb.paramstyle
  print "*****************************************\n"


# Return a connection to the database
def initialize ():

  # Connect to the basketball database. Use the dsn as the connection
  # parameter.  The Python DB-API details the other valid connection
  # parameters.  Notify the user of any raised exceptions.
  try:
    db = pgdb.connect (dsn = 'localhost:basketball')
  except:
    print
    print "Exception encountered connecting to database basketball."
    print
    # Force execution to stop by raising the handled exception again
    raise

  return db


# Close a database connection
def disconnect (db):
  db.close ()
```

```
# Create and populate the table which the examples feed off of
def cleantable ():

  # Connect to the database and open a cursor
  db = initialize ()
  cursor = db.cursor ()

  print "\nDropping Players table..."

  # Drop the table if it exists.  This is done in a try/except block
  # as we may be attempting to drop a table that does not exist.
  # Handle exceptions as you see fit.
  try:
    cursor.execute ("drop table players")
  except pgdb.DatabaseError:
    print "  Exception encountered: Table does not exist. Continuing."
    db.rollback ()

  print "  Creating and seeding Players table..."

  # Create and populate the Players table.  Any raised exceptions will
  # cause the program to stop as no handler is declared.
  cursor.execute ("create table players (name varchar(20),   \
                                         team varchar(50))")
  cursor.execute ("insert into players values              \
                   ('Michael Jordan','Washington Wizards')")
  cursor.execute ("insert into players values               \
                   ('Tim Duncan','San Antonio Spurs')")
  cursor.execute ("insert into players values               \
                   ('Vince Carter','Toronto Raptors')")

  db.commit ()
  cursor.close ()
  print "    Players table successfully created.\n"

  # Display the contents of the Players table and leave
  viewtable (db)
  disconnect (db)


# Display the contents of the Players table
def viewtable (db):

  # Create understandable index variables
  playerName, playerTeam = 0, 1

  # Issue a full select from the Players table
  cursor = db.cursor ()
```

```
  cursor.execute ("select * from players order by name asc")

  # Fetch the first result from the cursor
  row = cursor.fetchone ()

  # Display a message on an empty result set
  if row == None:
    print "Empty result set returned on select * from players"
  else:
    # Iterate through the result set displaying the player's name
    # and team
    print "Content of Players table:"
    print "*************************"
    while row != None:
      print " %s -- %s" % (row[playerName], row[playerTeam])
      row = cursor.fetchone ()
    print "\nNumber of players: %d\n" % cursor.rowcount

  cursor.close ()

# Update a member of the Players table
def update ():

  # Connect to the database
  db = initialize ()
  cursor = db.cursor ()

  selectStmt = "select * from players where name = 'Kobe Bryant'"
  updateStmt = "update players set team='LA Lakers' \
                where name = 'Kobe Bryant'"

  # We'll show a raw before/after image of the result sets sent
  # back for Kobe Bryant
  cursor.execute (selectStmt)
  print "\nBefore:", cursor.fetchall ()

  # Update Kobe's real team
  print "\nUpdating..."
  cursor.execute (updateStmt)

  # Display the updated row(s) and then the rest of the table
  cursor.execute (selectStmt)
  print "\nAfter: ", cursor.fetchall ()
  print
  viewtable (db)

  # Cleanup
  db.commit ()
```

```
  cursor.close ()
  db.close ()


# Modify table data and commit or rollback the transaction based
# on user input
def transaction (action):

  # Use the older versions of string manipulation so that
  # the example works with all versions of Python
  command  = string.lower (action)

  # Check parameters
  if (command != "commit") and (command != "rollback"):
    print "Usage: transaction (action) where"
    print "       action=\"commit\" or action=\"rollback\""
    return

  # Connect to the database
  db = initialize ()

  # Display the contents of the Players table before we modify
  # the data
  print "\nBefore:"
  viewtable (db)

  # Create SQL statements to be consumed by RHDB
  insertStmt1 = "insert into players values ('Tracy McGrady','Orlando Magic')"
  insertStmt2 = "insert into players values ('Kobe Bryant','NY Knicks')"
  deleteStmt = "delete from players where name = 'Michael Jordan'"

  print "About to issue the following commands: \n  %s\n  %s\n  %s" \
         % (insertStmt1, insertStmt2, deleteStmt)

  # Modify the table data.  If specific exceptions are returned,
  # rollback the transaction and leave... any other exceptions
  # will cause execution to halt.
  cursor = db.cursor ()
  try:
    cursor.execute (insertStmt1)
    cursor.execute (insertStmt2)
    cursor.execute (deleteStmt)
  except (pgdb.DatabaseError, pgdb.OperationalError,
          pgdb.pgOperationalError):
    print "  Exception encountered while modifying table data."
    db.rollback ()
    return
  print "\nAbout to", command, "the transaction..."
```

```
# Commit or rollback the transaction as requested
if command  == "commit":
  db.commit ()
else:
  db.rollback ()

cursor.close ()

print "\nAfter:"

# Display the contents of the Players table after the (potential)
# data modification
viewtable (db)
disconnect (db)
```