MANUAL PYEXIST

Manual para trabajar con Python en una BD eXist

Josu Boldoba Trapote
Barcelona, 3 de Junio de 2011

Índice

Introducción

Objetivo del manual

El objetivo de este manual es guiar al lector en el desarrollo de un sistema Python que controle remotamente las operaciones propias de una base de datos jerárquica como eXist.

¿Qué es eXist?

EXist es un sistema gestor de base de datos que almacena los datos mediante estructuras jerarquizadas, para ello utiliza el formato XML.

Para realizar operaciones sobre sus datos utiliza el sistema XQuery + XPath, basado en la indexación, y complementado con diferentes módulos como por ejemplo XProc o XSLFO.

Debido a que no es el objetivo del manual, se recomienda la lectura del Anexo I de este manual para la mejor comprensión de términos y conceptos usados más adelante.

¿Qué es pyexist?

Pyexist es un módulo de Python creado por usuarios de la red de programadores GitHub. Este módulo permite la comunicación de un código Python con una base de datos eXist a través de la API REST de eXist.

Pyexist está creado pensando en la distribución del sistema por lo que permite trabajar con bases de datos que no están instaladas en la misma máquina donde se usa.

Instalación y configuración

Base de datos: eXist

EXist es una base de datos de código libre, bajo licencia GNU-LGPL, que nos podemos descargar a través del siguiente enlace: http://www.exist-db.org/download.html .

Para instalar y comenzar a usar eXist, tenemos el enlace: http://www.exist-db.org/quickstart.html .

Por defecto, eXist se instala en el puerto 8080.

Requisitos

Java5 a partir de la release 1.4.x

Controlador: Pyexist

Como ya hemos dicho, se trata de un módulo no oficial de python, desarrollado por knipknap, larsmans y justinvw Debido a las políticas de GitHub, el código fuente se puede obtener y modificar, pero no distribuir, por lo que, en caso de necesidad, podemos solucionar o modificar a nuestro gusto el módulo.

El código se puede descargar en el siguiente enlace: https://github.com/knipknap/pyexist .

Requisitos

Python 2.5 o superior

eXist-db (No necesariamente en la misma máquina)

lxml u otro módulo DOM para tratar los resultados.

UTILIZANDO PYEXIST

En este capítulo se introducen las operaciones y usos ofrecidos por pyexist, así como los diferentes arreglos o mejoras que se le puede aplicar. Todos los ejemplos se harán sobre una supuesta collection “mycoll” definida en el Anexo II.

UBICACIÓN DEL CÓDIGO FUENTE

Para trabajar más cómodos, sin necesidad de hacer importaciones con rutas largas, lo primero de todo que haremos será copiar el directorio “src/pyexist”, obtenido al descomprimir el código obtenido anteriormente, a nuestro directorio de desarrollo.

FUNCIONES DEL MÓDULO

ExistDB

Las funciones de este tipo de objetos nos permitirá trabajar con el SGBD, pero nunca con los datos directamente.

ExistDB(string,string)

Función creadora del controlador. El primer parámetro es la URI de la base de datos y el segundo el path relativo a la zona de la base de datos donde trabajaremos, ésta puede ser una collection o una resource.. A continuación se muestran dos ejemplos:

- Conexión a eXist instalado en la misma máquina. Enlaza a la resource “users” de la collection “mycoll”:

  ```
  db = ExistDB('localhost:8080/exist/rest/db','mycol/users.xml')
  ```

- Conexión a eXist instalado en otra máquina. Enlaza a la collection “mycoll”:

  ```
  db = ExistDB('192.168.42.1:8080/exist/rest/db','mycol')
  ```

Tal y como se irá viendo, cuando hagamos una consulta o actualización simple, es decir, sobre una resource concreta, usaremos la primera forma, mientras que en los demás casos la segunda es más apropiada.

store(string,xml)

Importa un xml dado a un documento de la base de datos . El primer parámetro es la ruta a la resource y el segundo el objeto xml a importar.

Ejemplo: `db.store('users.xml',myXML)`

store_file(self,string,string)

Realiza la misma función que store(string,xml) pero lee el xml desde un documento dado. EL primer parámetro es la resource a modificar y el segundo el documento donde se encuentra el xml.

delete(self,string)

Borra la resource dada. El parámetro es el path a la resource a partir del punto donde hayamos conectado.

Ejemplo: `db.delete('users.xml')`

query(self,string)

Realiza una acción directa sobre los datos de la collection, es decir, consulta o actualiza los datos. Para ello necesita un parámetro que contenga una query válida en formato XQuery.

Debido a que la evaluación de una query es lento, especialmente al trabajar en una estructura jerarquizada, pyexist

trabaja de forma asíncrona, es decir, no envía la query hasta que no se consulta su resultado. Esto nos dará problemas con las consultas de actualización ya que éstas no retornan ningún tipo de dato. Para solucionar este problema usaremos una función definida más adelante.

Ejemplo de consulta de todos los usuarios:

```
. . .
query = ''' let $u := /users/user
            return $u '''
res = db.query(query)
. . .
```

query_from_file(filename)

Realiza el mismo trabajo que la función query pero lee la consulta desde un fichero .xq

Ejemplo,

Tenemos el fichero users.xq en un directorio de nuestro proyecto con el texto:

```
let $u := /users/user return $u
```

Ahora en nuestro código usaremos el siguiente fragmento para consultar los usuarios del sistema:

```
. . .
res = db.query_from_file('path/to/getUsers.xq')
. . .
```

XQuery

Estos objetos contendrán el resultado de una consulta realizada con las funciones anteriores. El módulo da mecanismos para la creación directa de estos objetos, pero lo mejor es crearlos a través de las funciones query o query_from_file del módulo ExistDB. Tiene dependencia de lxml para realizar las lecturas de datos.

En caso de preferir minidom para parsear XML, pyexist utilizara objetos XqueryMinidom aunque a nosotros no nos afectará.

count()

Consulta el número de elementos que tiene el resultado de una consulta.

```
. . .
query = ''' let $u := /users/user
            return $u '''
res = db.query(query)
if res.count() == 0:
      print "No hay usuarios registrados"
else:
      print "Hay %d usuarios registrados"%res.count()
. . .
```

Problemas Conocidos

Debido a mejoras de eficiencia que consideró en su día knipknap, podemos sentir que pasan cosas extrañas como que una query no se ha ejecutado bien o que perdemos datos al pasar de un objeto a otro. Aunque a primera vista parezca que pyexist está funcionando mal, realmente somos nosotros los que no estamos teniendo en cuenta ciertas peculiaridades que tiene por decisión de eficiencia.

A continuación se explican estas características, sus posibles errores fantasma y la forma de solucionarlos. Pero antes esta bien saber un poco sobre el objeto XQuery. Un XQuery es un pseudoXML con el formato

```
<XQery ns="XXXXXX">
      RESULTADO
</XQuery>
```

Decimos que es pseudoXML dado que no se puede parsear directamente con funciones DOM aunque tenga la estructura típica de un XML.

RESULTADO es un array de XML, aunque se codifica con la estructura Element Tree en Python. Por este motivo podemos acceder secuencialmente a los diferentes objetos retornados e incluso acceder aleaoriamente a uno en concreto. Esto se muestra en el siguiente fragmento:

```
. . .
# Conectamos a nuestra BD
db = ExistDB('localhost:8080/exist/rest/db','mycoll/users.xml')
# Creamos y ejecutamos una query guardando el resultado en res.
query = ''' let $u := /users/user
            return $u '''
res = db.query(query)
# Acceso secuencial a todos los usuarios retornados, mediante lxml.etree
for elem in res:
      print etree.tounicode(elem)
# Calculamos el numero de resultados obtenidos, lo usaremos para accesos
# aleatorios protegidos
cont = res.cont()
# Acceso aleatorio a un resultado N si existe
if N < cont:
      print etree.tounicode(res[N])
# Acceso secuencial de todos los resultados usando accesos aleatorios
for i in xrange(0,cont):
      print etree.tounicode(res[i])
. . .
NOTA: Este código puede dar problemas cuya solución se explican en este mismo
capítulo, pero por visualidad de los ejemplos se ha hecho así.
```

Ejecución asíncrona de querys

Tal y como se ha comentado antes, pyexist no envía la query a la base de datos hasta que no se trata el resultado. Cuando la query es de una consulta no nos dará ningún tipo de problema pero en el momento de utilizar querys de actualización (update insert / update replace / update delete) veremos que éstas no se ejecutan y por lo tanto no se producen cambios en el sistema.

Para solucionar esto nos vale con hacer una lectura mínima del resultado. Podemos hacer una lectura directa o utilizar la función count de XQuery.py . Por motivos de coherencia, una query de tipo update no retorna nada, usaremos el primer método. Podemos crear la siguiente función en nuestro código:

```
def updateQuery(db,query):
res = db.query(query)
res[0:1]
```

db es un objeto de tipo ExistDB y query es un string válido para la función query explicada anteriormente. Con la operación `res[0:1]` leemos el primer elemento del "resultado", haciendo así que se ejecute la actualización.

XQuery con elementos volátiles

NOTA: En este apartado se explica la no correctitud del código de ejemplo explicado anteriormente.

Los elementos del resultado de una query no se almacenan en memoria. Por este motivo una vez que son tratados no se pueden volver a consultar salvo que volvamos a ejecutar de nuevo la query que los producía.

En este caso no existe ninguna función mágica que nos haga un parche usable sobre el módulo y lo único que podemos hacer es guardar los resultados en la estructura de datos que nos resulte más adecuada. Hay que tener en cuenta el problema descrito lo dan los objetos de RESULTADO y no la XQuery, por lo que la función cont() no dará problemas ni el retorno de un objeto XQuery. Además, el return de un elemento de RESULTADO se puede hacer sin perder los datos.

Un ejemplo de cómo hacer esto se explica a continuación utilizando una lista para almacenar los datos. En este caso almacenaremos los mails de los usuarios con una edad entre 20 y 30 años.

```
. . .
# Conectamos a la collection
db = ExistDB('localhost:8080/exist/rest/db','mycoll/users.xml')
# Ejecutamos la query
query = ''' for $u in /users/user[age >= 20 and age <= 30]
            return <mail>{$u/age/text()}</mail>'''
res = db.query(query)
# Recorremos la respuesta de la BD y almacenamos el mail, sólo el string ya que
# no queremos nada más que los correos
mails = []
for elem in res:
      for m in elem.getiterator('mail'):
            mails.append(m.text)
. . .
```

En el código de ejemplo del principio de este capítulo, una vez visto esto, nos trataría de leer elementos nulos después de haber hecho el acceso secuencial.

ANEXO I: DEFINICIONES

COLLECTION

Conjunto de resources, interrelacionadas o no, pertenecientes todas a un mismo servidor eXist. Su relación conceptual con las BBDD relacionales sería el concepto de base de datos.

RESOURCE

Archivo en formato XML que almacena una serie de datos, en principio homogéneos, de una forma jerarquizada. Su relación en la con las BBDD relacionales, en el caso de que todos los datos sean homogéneos, sería una tabla. En el caso de que sean heterogéneos, podemos visualizarlo como una unión de tablas.

QUERY

Consulta que se realiza sobre una base de datos para obtener resultados (datos o información) o actualizarlos.

ELEMENT TREE

Estructura de datos que representa un XML. El tag root se define como el root del árbol y a partir de ahí cada nodo tiene tantos hijos como subtags tenga el tag que representan.

ANEXO II: COLLECTION EJEMPLOS

Debido a que el objetivo de este manual no es la comprensión o utilización de eXist, si no la obtención de los conocimientos mínimos para comunicarse desde Python con una BD eXist a través de Pyexist, la collection que usaremos es muy reducida.

MYCOLL

MYCOLL es el nombre que usaremos para la collection de ejemplo. Su path relativo en el servidor donde esté la BD será: ~/mycoll

RESOURCE USERS

USERS es la única resource que nos interesa de esta collection. Almacena datos sobre los usuarios del sistema, para representar un posible caso real vamos a suponer que se trata de los clientes de una empresa, por lo tanto necesitará sus datos personales y algún tipo de información sobre sus costumbres de compra. Su path relativo en el servidor donde esté la BD será: ~/mycoll/users.xml

La estructura de la resource es:

```
<users>
      <user>
            <id></id>
            <name></name>
            <address>
                  <st></st>
                  <city></city>
                  <CP></CP>
            </address>
            <age></age>
            <mail></mail>
            <category>
                  <class></class>
                  .
                  .
                  .
                  <class></class>
            <category>
      </user>
      .
      .
      .
      <user>
      . . .
      </user>
</users>
```