Databases and Jython: Object Relational Mapping and Using JDBC

In this chapter, we will look at zxJDBC package, which is a standard part of Jython since version 2.1 and complies with the Python 2.0 DBI standard. zxJDBC can be an appropriate choice for simple one-off scripts where database portability is not a concern. In addition, it's (generally) necessary to use zxJDBC when writing a new dialect for SQLAlchemy or Django. (But that's not strictly true: you can use pg8000, a pure Python DBI driver, and of course write your own DBI drivers. But please don't do that.) So knowing how zxJDBC works can be useful when working with these packages. However, it's too low level for us to recommend for more general usage. Use SQLAlchemy or Django if at all possible. Finally, JDBC itself is also directly accessible, like any other Java package from Jython. Simply use the java.sql package. In practice this should be rarely necessary.

The second portion of this chapter will focus on using object relational mapping with Jython. The release of Jython 2.5 has presented many new options for object relational mapping. In this chapter we'll focus on using SQLAlchemy with Jython, as well as using Java technologies such as Hibernate. In the end you should have a couple of different choices for using object relational mapping in your Jython applications.

ZxJDBC—Using Python's DB API via JDBC

The zxJDBC package provides an easy-to-use Python wrapper around JDBC. zxJDBC bridges two standards:

JDBC is the standard platform for database access in Java.

DBI is the standard database API for Python apps.

ZxJDBC, part of Jython, provides a DBI 2.0 standard compliant interface to JDBC. Over 200 drivers are available for JDBC (http://developers.sun.com/product/jdbc/drivers), and they all work with zxJDBC. High performance drivers are available for all major relational databases, including DB2, Derby, MySQL, Oracle, PostgreSQL, SQLite, SQL Server, and Sybase. And drivers are also available for non-relational and specialized databases, too.

However, unlike JDBC, zxJDBC when used in the simplest way possible, blocks SQL injection attacks, minimizes overhead, and avoids resource exhaustion. In addition, zxJDBC defaults to using a transactional model (when available), instead of autocommit.

First we will look at connections and cursors, which are the key resources in working with zxJDBC, just like any other DBI package. Then we will look at what you can do them with them, in terms of typical queries and data manipulating transactions.

Getting Started

The first step in developing an application that utilizes a database back-end is to determine what database or databases the application will use. In the case of using zxJDBC or another JDBC implementation, the determination of what database the application will make use of is critical to the overall development process. Many application developers will choose to use an object relational mapper for this very reason. When an application is coded with a JDBC implementation, whereas SQL code is hand-coded, the specified database of choice will cause different dialects of SQL to be used. One of the benefits of object relation mapping (ORM) technology is that the SQL is transparent to the developer. The ORM technology takes care of the different dialects behind the scenes. This is one of the reasons why ORM technology may be slower at implementing support for many different databases. Take SQLAlchemy or Django for instance: each of these technologies must have a different dialect coded for each database. Using an ORM can make an application more portable over many different databases. However, as stated in the preface using zxJDBC would be a fine choice if your application is only going to target one or two databases.

While using JDBC for Java, one has to deal with the task of finding and registering a driver for the database. Most of the major databases make their JDBC drivers readily available for use. Others may make you register prior to downloading the driver, or in some cases purchase it. Because zxJDBC is an alternative implementation of JDBC, one must use a JDBC driver in order to use the API. Most JDBC drivers come in the format of a JAR file that can be installed to an application server container, and IDE. In order to make use of a particular database driver, it must reside within the CLASSPATH. As mentioned previously, to find a given JDBC driver for a particular database, take a look at the Sun Microsystems JDBC Driver search page (http://developers.sun.com/product/jdbc/drivers) as it contains a listing of different JDBC drivers for *most* of the databases available today.

Note

Examples in this section are for Jython 2.5.1 and later. Jython 2.5.1 introduced some simplifications for working with connections and cursors. In addition, we assume PostgreSQL for most examples, using the world sample database (also available for MySQL). In order to follow along with the examples in the following sections, you should have a PostgreSQL database available with the *world* database example. Please go to the PostgreSQL homepage at http://www.postgresql.org to download the database. The world database sample is available with the source for this book. It can be installed into a PostgreSQL database by opening psql and initiating the following command:

```
postgres=# \\i <path to world sql>/world.sql
```

As stated previously, once a driver has been obtained it must be placed into the classpath. What follows are a few examples for adding JDBC drivers to the CLASSPATH for a couple of the most popular databases.

Listing 12-1. Adding JDBC drivers for popular databases to the CLASSPATH

```
# Oracle
# Windows
set CLASSPATH=<PATH TO JDBC>\\ojdbc14.jar;%CLASSPATH%
# OS X
export CLASSPATH=<PATH TO JDBC>/ojdbc14.jar:$CLASSPATH
# PostgreSQL
# Windows
set CLASSPATH=<PATH TO JDBC>\\postgresql-x.x.jdbc4.jar;%CLASSPATH%
# OS X
export CLASSPATH=<PATH TO
JDBC>/postgresql-x.x.jdbc4.jar:$CLASSPATH
```

After the appropriate JAR file for the target database has been added to the CLASSPATH, development can commence. It is important to note that zxJDBC (and all other JDBC implementations) use a similar procedure for working with the database. One must perform the following tasks to use a JDBC implementation:

- Create a connection.
- Create a query or statement.
- Obtain results of query or statement.
- If using a query, obtain results in a cursor and iterate over data

to perform tasks. - Close cursor. - Close connection (If not using the with_statement syntax in versions of Jython prior to 2.5.1).

Over the next few sections, we'll take a look at each of these steps and how zxJDBC can make them easier than using JDBC directly.

Connections

A database connection is simply a resource object that manages access to the database system. Because database resources are generally expensive objects to allocate, and can be readily exhausted, it is important to close them as soon as you're finished using them. There are two ways to create database connections:

- *Direct creation.* Standalone code, such as a script, will directly

create a connection.

- *JNDI.* Code managed by a container should use JNDI for connection

creation. Such containers include GlassFish, JBoss, Tomcat, WebLogic, and WebSphere. Normally connections are pooled when run in this context and are also associated with a given security context.

The following is an example of the best way to create a database connection outside of a managed container using Jython 2.5.1. It is important to note that prior to 2.5.1, the *with_statement* syntax was not available. This is due to the underlying implementation of PyConnection in versions of Jython prior to 2.5.1. As a rule, any object that can be used via the *with_statement* must implement certain functionality, including the *__exit__* method. Please see the note that follows to find out how to implement this functionality in versions prior to 2.5.1. Another thing to notice is that in order to connect, we must use a JDBC url which conforms to the standards of a given database in this case, PostgreSQL.

Listing 12-2

```
from __future__ import with_statement
from com.ziclix.python.sql import zxJDBC

# for example
jdbc_url = "jdbc:postgresql:test"
username = "postgres"
password = "jython25"
driver = "org.postgresql.Driver"

# obtain a connection using the with-statment
with zxJDBC.connect(jdbc_url, username, password, driver) as conn:
    with conn:
        with conn.cursor() as c:
            c.execute("select name from country")
            c.fetchone()
```

Walking through the steps, you can see that the *with_statement* and zxJDBC are imported as we will use them to obtain our connection. The next step is to define a series of string values that will be used for the connection activity. Note that these only need to be defined once if set up as globals. Lastly, the connection is obtained and some work is done. Now let's take a look at this same procedure coded in Java for comparison.

Listing 12-3.

```
import java.sql.*;
import org.postgresql.Driver;
...
// In some method
Connection conn = null;
String jdbc_url = "jdbc:postgresql:test";
String username = "postgres";
String password = "jython25";
String driver = "org.postgresql.Driver";
try {
  DriverManager.registerDriver(new org.postgresql.Driver());
  conn = DriverManager.getConnection(jdbc_url,
               username, password);
  // do something using statement and resultset
  conn.close();
} catch(Exception e) {
  logWriter.error("getBeanConnection ERROR: ",e);
}
```

Note

In versions of Jython prior to 2.5.1, the *with_statement* syntax is not available. For this reason, we must work directly with the connection (i.e. close it when finished). Take a look at the following code for an example of using zxJDBC connections without the with_statement functionality.

```
from __future__ import with_statement from
com.ziclix.python.sql import zxJDBC

# for example jdbc_url = "jdbc:postgresql:test" username =
"postgres" password = "jython25" driver = "org.postgresql.Driver"

conn = zxJDBC.connect(jdbc_url, username, password, driver)
do_something(conn) # Be sure to clean up by closing the connection
(and cursor) conn.close()
```

The *with* statement ensures that the connection is immediately closed following the work. The alternative is to use finally to perform the close. Using the latter technique allows for more tightly controlled exception handling technique, but also adds a considerable amount of code. As noted previously, the with statement is not available in versions of Jython prior to 2.5.1, so this is the recommended approach when using those versions:

Listing 12-4.

```
try:
    conn = zxJDBC.connect(jdbc_url, username, password, driver)
    do_something(conn)
finally:
    conn.close()
```

The connection (PyConnection) object in zxJDBC has a number of methods and attributes that can be used to perform various functions and obtain metadata information. For instance, the *close* method can be used to close the connection. Tables 12-1 and 12-2 are listings of all available methods and attributes for a connection and what they do.

Table 12-1: Connection Methods

Method	**Functionality**
close	Close the connection now (rather than whenever __del__ is called).
commit	Commits all work that has been performed against a connection
cursor	Returns a new cursor object from the connection
rollback	In case a database does provide transactions this method causes the database to roll back to the start of any pending transaction.

Method	Functionality
nativesql	Converts the given SQL statement into the system's native SQL grammar

Table 12-2: Connection Attributes

Attribute	Functionality
autocommit	Enable or disable autocommit on a connection. Default is disabled.
dbname	Returns the name of the database
dbversion	Returns the version of databae
drivername	Returns the database driver name
driverversion	Returns the database driver version
url	Returns the database URL in use
__connection__	Returns the type of connection in use
__cursors__	Returns a listing of all open cursors on the connection
__statements__	Returns a listing of all open statements on the connection
closed	Returns a boolean stating whether connection is closed

Of course, we can always use the connection to obtain a listing of all methods and attributes using the syntax shown in Listing 12-5.

Listing 12-5.

```
>>> conn.__methods__
['close', 'commit', 'cursor', 'rollback', 'nativesql']
>>> conn.__members__
['autocommit', 'dbname', 'dbversion', 'drivername',
'driverversion', 'url', '__connection__', '__cursors__',
'__statements__', 'closed']
```

Note

Connection pools help ensure for more robust operation, by providing for reuse of connections while ensuring the connections are in fact valid. Often naive code will hold a connection for a very long time, to avoid the overhead of creating a connection, and then go to the trouble of managing reconnecting in the event of a network or server failure. It's better to let that be managed by the connection pool infrastructure instead of reinventing it.

All transactions, if supported, are done within the context of a connection. We will be discussing transactions further in the subsection on data modification, but Listing 12-6 is the basic recipe.

Listing 12-6. Transaction Recipe

```
try:
    # Obtain a connection that is not using auto-commit (default for
    zxJDBC)
    conn = zxJDBC.connect(jdbc_url, username, password, driver)
    # Perform all work on connection
    do_something(conn)
    # After all work is complete, commit
    conn.commit()
except:
    # If a failure occurs along the way, rollback all previous work
    conn.rollback()
```

ZxJDBC.lookup

In a managed container, you would use zxJDBC.lookup instead of zxJDBC.connect. If you have code that needs to run both inside and outside containers, we recommend you use a factory to abstract this. Inside a container, like an app server, you should use JDNI to allocate the resource. Generally the connection will be managed by a connection pool (see Listing 12-7).

Listing 12-7.

```
factory = "com.sun.jndi.fscontext.RefFSContextFactory"
db = zxJDBC.lookup('jdbc/postgresDS',
    INITIAL_CONTEXT_FACTORY=factory)
```

This example assumes that the datasource defined in the container is named “jdbc/postgresDS,” and it uses the Sun FileSystem JNDI reference implementation. This lookup process does not require knowing the JDBC URL or the driver factory class. These aspects, as well as possibly the user name and password, are configured by the administrator of the container using tools specific to that container. Most often by convention you will find that JNDI names typically resemble a *jdbc/NAME* format.

Cursors

Once you have a connection, you probably want to do something with it. Because you can do multiple things within a transaction, such as query one table, update another, you need one more resource, which is a cursor. A cursor in zxJDBC is a wrapper around the JDBC statement and resultSet objects that provides a very *Pythonic* syntax for working with the database. The result is an easy to use and extremely flexible API. Cursors are used to hold data that has been obtained via the database, and they can be used in a variety of fashions which we will discuss. There are two types of cursors available for use, static and dynamic. A static cursor is the default type, and it basically performs an iteration on an entire resultSet at once. The latter dynamic cursor is known as a lazy cursor and it only iterates through the resultSet on an as-needed basis. The following listings are examples of creating each type of cursor.

Listing 12-8. Creating all possible cursor types

```
# Assume that necessary imports have been performed
# and that a connection has been obtained and assigned
# to a variable 'conn'
cursor = conn.cursor() # static cursor creation
cursor = conn.cursor(True) # dynamic cursor creation with the Boolean
argument
```

Dynamic cursors tend to perform better due to memory constraints; however, in some cases they are not as convenient as working with a static cursor. For example, if you'd like to query the database to find a row count it is very easy with a static cursor because all rows are obtained at once. This is not possible with a dynamic cursor and one must perform two queries in order to achieve the same result.

Listing 12-9.

```
# Using a static cursor to obtain rowcount
>>> cursor = conn.cursor()
>>> cursor.execute("select * from country")
>>> cursor.rowcount
239
# Using a dynamic cursor to obtain rowcount
>>> cursor = conn.cursor(1)
>>> cursor.execute("select * from country")
>>> cursor.rowcount
0
# Since rowcount does not work with dynamic, we must
# perform a separate count query to obtain information
>>> cursor.execute("select count(*) from country")
>>> cursor.fetchone()
(239L,)
```

Cursors are used to execute queries, inserts, updates, deletes, and/or issue database commands. Like connections, cursors have a number of methods and attributes that can be used to perform actions or obtain metadata information. See Tables 12-3 and 12-4.

Table 12-3: Cursor Methods

Method	**Functionality**
tables	Retrieves a list of tables. (catalog, schema-pattern, table-pattern, types)
columns	Retrieves a list of columns. (catalog, schema-pattern, table-name-pattern, column-name-pattern)
primarykeys	Retrieves a list of primary keys. (catalog, schema, table)
foreignkeys	Retrieves a list of foreign keys. (primary-catalog, primary-schema, primary-table, foreign-catalog, foreign-schema, foreign-table)
procedures	Retrieves a list of procedures. (catalog, schema, tables)
procedurecolumns	Retrieves a list of procedure columns. (catalog, schema-pattern, procedure-pattern, column-pattern)
statistics	Obtains statistics on the query. (catalog, schema, table, unique, approximation)

Method	**Functionality**
bestrow	Optimal set of columns that uniquely identifies a row
versioncolumns	Columns that are automatically updated when any value in a row is updated
close	Closes the cursor
execute	Executes code contained within the cursor
executemany	Used to execute prepared statements or sql with a parameter list
fetchone	Fetch the next row of a query result set, returning a single sequence, or None if no more data exists
fetchall	Fetch all (remaining) rows of a query result, returning them as a sequence of sequnces
fetchmany	Fetch the next set of rows of a query result, returning a sequence of seqences
callproc	Executes a stored procedure
next	Moves to the next row in the cursor
write	Execute the sql written to this file-like object

Table 12-4: Cursor Attributes

Attribute	**Functionality**
arraysize	Number of rows *fetchmany()* should return without any arguments
rowcount	Returns the number of resulting rows
rownumber	Returns the current row number
description	Returns information regarding each column in the query
datahandler	Returns the specified datahandler
warnings	Returns all wornings on the cursor
lastrowid	Returns the rowid of the last row fetched
updatecount	Returns the number of updates that the current cursor has performed
closed	Returns a boolean representing whether the cursor has been closed
connection	Returns the connection object that contains the cursor

A number of the methods and attributes above cannot be used until a cursor has been executed with a query or statement of some kind. Most of the time, the particular method or attribute name will provide a good enough description of its functionality.

Creating and Executing Queries

As you've seen previously, it is quite easy to initiate a query against a given cursor. Simply provide a *sel**ect* statement in string format as a parameter to the cursor *execute()* or *executemany()* methods and then use one of the *fetch* methods to iterate over the returned results. In the following examples we query the world data and display some cursor data via the associated attributes and methods.

Listing 12-10.

```
>>> cursor = conn.cursor()
>>> cursor.execute("select country, region from country")
# Fetch next record
>>> cursor.fetchone()
((AFG,Afghanistan,Asia,"Southern and Central
Asia",652090,1919,22720000,45.9,5976.00,,Afganistan/Afqanestan,"Islamic
Emirate","Mohammad Omar",1,AF), u'Southern and Central Asia')
# Calling fetchmany() without any parameters returns next record
>>> cursor.fetchmany()
[((NLD,Netherlands,Europe,"Western
Europe",41526,1581,15864000,78.3,371362.00,360478.00,Nederland,"Constitutiona
l
Monarchy",Beatrix,5,NL), u'Western Europe')]
# Fetch the next two records
>>> cursor.fetchmany(2)
[((ANT,"Netherlands Antilles","North
America",Caribbean,800,,217000,74.7,1941.00,,"Nederlandse
Antillen","Nonmetropolitan Territory of The Netherlands",Beatrix,33,AN),
u'Caribbean'),
((ALB,Albania,Europe,"Southern
Europe",28748,1912,3401200,71.6,3205.00,2500.00,Shqip?ria,Republic,"Rexhep
Mejdani",34,AL), u'Southern Europe')]
# Calling fetchall() would retrieve the rest of the records
>>> cursor.fetchall()
...
# Using description provides data regarding the query in the cursor
>>> cursor.description
[('country', 1111, 2147483647, None, None, None, 2), ('region', 12,
2147483647, None, None, None, 0)]
```

Creating a cursor using the with_statement syntax is easy, please take a look at the following example for use with Jython 2.5.1 and beyond.

Listing 12-11.

```
with conn.cursor() as c:
    do_some_work(c)
```

Like connections, you need to ensure the resource is appropriately closed. So you can just do this to follow the shorter examples we will look at:

Listing 12-12.

```
>>> c = conn.cursor()
>>> # work with cursor
```

As you can see, queries are easy to work with using cursors. In the previous example, we used the *f**etchall()* method to retrieve all of the results of the query. However, there are other options available for cases where all results are not desired including the *fetchone()* and *fetchmany()* options. Sometimes it is best to iterate over results of a query in order to work with each record separately. Listing 12-13 iterates over the countries contained within the country table.

Listing 12-13.

```
>>> from com.ziclix.python.sql import zxJDBC
>>> conn =
zxJDBC.connect("jdbc:postgresql:test","postgres","jython25","org.postgresql.D
river")
>>> cursor = conn.cursor()
>>> cursor.execute("select name from country")
>>> while cursor.next():
...     print cursor.fetchone()
...
(u'Netherlands Antilles',)
(u'Algeria',)
(u'Andorra',)
...
```

Often, queries are not hard-coded, and we need the ability to substitute values in the query to select the data that our application requires. Developers also need a way to create dynamic SQL statements at times. Of course, there are multiple ways to perform these feats. The easiest way to substitute variables or create a dynamic query is to simply use string concatenation. After all, the *execute()* method takes a string-based query. Listing 12-14 shows how to use string concatenation for dynamically forming a query and also substituting variables.

Listing 12-14. String Concatenation for Dynamic Query Formation

```
...
# Assume that the user selected a pull-down menu choice determining
# what results to retrieve from the database, either continent or country
name.
# The selected choice is stored in the selectedChoice variable. Let's also
assume
# that we are interested in all continents or countries beginning with the
letter "A"
>>> qry = "select " + selectedChoice + " from country where " +
selectedChoice + " like 'A%'"
>>> cursor.execute(qry)
>>> while cursor.next():
... print cursor.fetchone()
...
(u'Albania',)
(u'American Samoa',)
...
```

This technique works very well for creating dynamic queries, but it also has its share of issues. For instance, reading through concatenated strings of code can become troublesome on the eyes. Maintaining such code is a tedious task. Above that, string concatenation is not the safest way to construct a query as it opens an application up for a SQL injection attack. SQL injection is a technique that is used to pass undesirable SQL code into an application in such a way that it alters a query to perform unwanted tasks. If the user has the ability to type free text into a textfield and have that text passed into a string concatenated query, it is best to perform some other means of filtering to ensure certain keywords or commenting symbols are not contained in the value. A better way of getting around these issues is to make use of prepared statements.

Note

Ideally, never construct a query statement directly from user data. SQL injection attacks employ such construction as their attack vector. Even when not malicious, user data will often contain characters, such as quotation marks, that can cause the query to fail if not properly escaped. In all cases, it’s important to scrub and then escape the user data before it’s used in the query.

One other consideration is that such queries will generally consume more resources unless the database statement cache is able to match it (if at all).

But there are two important exceptions to our recommendation:

SQL statement requirements: Bind variables cannot be used everywhere. However, specifics will depend on the database.

Ad hoc or unrepresentative queries: In databases like Oracle, the statement cache will cache the execution plan, without taking in account lopsided distributions of values that are indexed, but are known to the database if presented literally. In those cases, a more efficient execution plan will result if the value is put in the statement directly.

However, even in these exceptional cases, it’s imperative that any user data is fully scrubbed. A good solution is to use some sort of mapping table, either an internal dictionary or a mapping table driven from the database itself. In certain cases, a carefully constructed regular expression may also work. Be careful.

Prepared Statements

To get around using the string concatenation technique for substituting variables, we can use a technique known as *prepared statements*. Prepared statements allow one to use bind variables for data substitution, and they are generally safer to use because most security considerations are taken care of without developer interaction. However, it is always a good idea to filter input to help reduce the risk. Prepared statements in zxJDBC work the same as they do in JDBC, just a simpler syntax. In Listing 12-15, we will perform a query on the country table using a prepared statement. Note that the question marks are used as place holders for the substituted variables. It is also important to note that the *executemany()* method is invoked when using a prepared statement. Any substitution variables being passed into the prepared statement must be in the form of a tuple or list.

Listing 12-15. Using Prepared Statements

```
...
# Passing a string value into the query
qry = "select continent from country where name = ?"
>>> cursor.executemany(qry,['Austria'])
>>> cursor.fetchall()
[(u'Europe',)]

# Passing some variables into the query
>>> continent1 = 'Asia'
>>> continent2 = 'Africa'
>>> qry = "select name from country where continent in (?,?)"
>>> cursor.executemany(qry, [continent1, continent2])
>>> cursor.fetchall()
[(u'Afghanistan',), (u'Algeria',), (u'Angola',), (u'United Arab Emirates',),
(u'Armenia',), (u'Azerbaijan',),
...
```

Resource Management

You should always close connections and cursors. This is not only good practice but absolutely essential in a managed container so as to avoid exhausting the corresponding connection pool, which needs the connections returned as soon as they are no longer in use. The *with* statement makes it easy. See Listing 12-16.

Listing 12-16. Managing Connections Using With Statements

```
from __future__ import with_statement
from itertools import islice
from com.ziclix.python.sql import zxJDBC
# externalize
jdbc_url = "jdbc:oracle:thin:@host:port:sid"
username = "world"
password = "world"
driver = "oracle.jdbc.driver.OracleDriver"
with zxJDBC.connect(jdbc_url, username, password, driver) as conn:
    with conn:
        with conn.cursor() as c:
            c.execute("select * from emp")
            for row in islice(c, 20):
                print row # let's redo this w/ namedtuple momentarily...
```

The older alternative is available. It’s more verbose, and similar to the Java code that would normally have to be written to ensure that the resource is closed. See Listing 12-17.

Listing 12-17. Managing Connections Avoiding the With Statement

```
try:
    conn = zxJDBC.connect(jdbc_url, username, password, driver)
    cursor = conn.cursor()
    # do something with the cursor
    # Be sure to clean up by closing the connection (and cursor)
finally:
    if cursor:
        cursor.close()
    if conn:
        conn.close()
```

Metadata

As mentioned previously in this chapter, it is possible to obtain metadata information via the use of certain attributes that are available to both connection and cursor objects. zxJDBC matches these attributes to the properties that are found in the JDBC *java.sql.DatabaseMetaData* object. Therefore, when one of these attributes is called, the JDBC *DatabaseMetaData* object is actually obtaining the information.

Listing 12-18 shows how to retrieve metadata about a connection, cursor, or even a specific query. Note that whenever obtaining metadata about a cursor, you must fetch the data after setting up the attributes.

Listing 12-18. Retrieving Metadata About a Connection, Cursor or Specific Query

```
# Obtain information about the connection using connection attributes
>>> conn.dbname
'PostgreSQL'
>>> conn.dbversion
'8.4.0'
>>> conn.drivername
'PostgreSQL Native Driver'
# Check for existing cursors
>>> conn.__cursors__
[<PyExtendedCursor object instance at 1>]

# Obtain information about the cursor and the query
>>> cursor = conn.cursor()
# List all tables
>>> cursor.tables(None, None, '%', ('TABLE',))
>>> cursor.fetchall()
[(None, u'public', u'city', u'TABLE', None), (None, u'public', u'country',
u'TABLE', None), (None, u'public', u'countrylanguage',
u'TABLE', None), (None, u'public', u'test', u'TABLE', None)]
```

Data Manipulation Language and Data Definition Language

Any application that will manipulate data contained in a RDBMS must be able to issue Data Manipulation Language (DML). Of course, DML consists of issuing statements such as INSERT, UPDATE, and DELETE. . .the basics of CRUD programming. zxJDBC makes it rather easy to use DML in a standard cursor object. When doing so, the cursor will return a value to provide information about the result. A standard DML transaction in JDBC uses a prepared statement with the cursor object, and assigns the result to a variable that can be read afterwards to determine whether the statement succeeded.

ZxJDBC also uses cursors to define new constructs in the database using Data Definition Language (DDL). Examples of doing such are creating tables, altering tables, creating indexes, and the like. Similarly to performing DML with zxJDBC, a resulting DDL statement returns a value to assist in determining whether the statement succeeded or not.

In the next couple of examples, we'll create a table, insert some values, delete values, and finally delete the table.

Listing 12-19. Using DML

```
>>>
# Create a table named PYTHON_IMPLEMENTATIONS
>>> stmt = "create table python_implementations (id integer,
python_implementation varchar, current_version varchar)"
>>> result = cursor.execute(stmt)
>>> print result
None
>>> cursor.tables(None, None, '%', ('TABLE',))
# Ensure table was created
>>> cursor.fetchall()
[(None, u'public', u'city', u'TABLE', None), (None, u'public', u'country',
u'TABLE', None), (None, u'public', u'countrylanguage',
u'TABLE', None), (None, u'public', u'python_implementations', u'TABLE',
None), (None, u'public', u'test', u'TABLE', None)]
# Insert some values into the table
>>> stmt = "insert into PYTHON_IMPLEMENTATIONS values (?, ?, ?)"
>>> result = cursor.executemany(stmt, [1,'Jython','2.5.1'])
>>> result = cursor.executemany(stmt, [2,'CPython','3.1.1'])
>>> result = cursor.executemany(stmt, [3,'IronPython','2.0.2'])
>>> result = cursor.executemany(stmt, [4,'PyPy','1.1'])
>>> conn.commit()
# Query the database
>>> cursor.execute("select python_implementation, current_version from
python_implementations")
>>> cursor.rowcount
4
>>> cursor.fetchall()
[(u'Jython', u'2.5.1'), (u'CPython', u'3.1.1'), (u'IronPython', u'2.0.2'),
(u'PyPy', u'1.1')]
# Update values and re-query
>>> stmt = "update python_implementations set
python_implementation = 'CPython -Standard Implementation' where id = 2"
>>> result = cursor.execute(stmt)
>>> print result
None
>>> conn.commit()
>>> cursor.execute("select python_implementation, current_version from
python_implementations")
>>> cursor.fetchall()
```

```
[(u'Jython', u'2.5.1'), (u'IronPython', u'2.0.2'), (u'PyPy', u'1.1'), 
(u'CPython -Standard Implementation', u'3.1.1')]
```

It is a good practice to make use of bulk inserts and updates. Each time a commit is issued it incurs a performance penalty. If DML statements are grouped together and then followed by a commit, the resulting transaction will perform much better. Another good reason to use bulk DML statements is to ensure transactional safety. It is likely that if one statement in a transaction fails, all others should be rolled back. As mentioned previously in the chapter, using a try/except clause will maintain transactional dependencies. If one statement fails then all others will be rolled back. Likewise, if they all succeed then they will be committed to the database with one final commit.

Calling Procedures

Database applications often make use of procedures and functions that live inside the database. Most often these procedures are written in a SQL procedural language such as Oracle's PL/SQL or PostgreSQL's PL/pgSQL. Writing database procedures and using them with external applications such written in Python, Java, or the like makes lots of sense, because procedures are often the easiest way to work with data. Not only are they running close to the metal since they are in the database, but they also perform much faster than say a Jython application that needs to connect and close connections on the database. Since a procedure lives within the database, there is no performance penalty due to connections being made.

ZxJDBC can easily invoke a database procedure just as JDBC can do. This helps developers to create applications that have some of the more database-centric code residing within the database as procedures, and other application-specific code running on the application server and interacting seamlessly with the database. In order to make a call to a database procedure, zxJDBC offers the *callproc()* method which takes the name of the procedure to be invoked. In Listing 12-20, we create a relatively useless procedure and then call it using Jython (Listing 12-21).

Listing 12-20. PostgreSQL Procedure

```
CREATE OR REPLACE FUNCTION proc_test(
    OUT out_parameter CHAR VARYING(25) )
AS $$
DECLARE
BEGIN
    SELECT python_implementation
    INTO out_parameter
    FROM python_implementations
    WHERE id = 1;
    RETURN;
END;
$$ LANGUAGE plpgsql;
```

Listing 12-21. Jython Calling Code

```
>>> result = cursor.callproc('proc_test')
>>> cursor.fetchall()
[(u'Jython',)]
```

Although this example was relatively trivial, it is easily to see how the use of database procedures from zxJDBC could easily become important. Combining database procedures and functions with application code is a powerful technique, but it does tie an application to a specific database so it should be used wisely.

Customizing zxJDBC Calls

At times, it is convenient to have the ability to alter or manipulate a SQL statement automatically. This can be done before the statement is sent to the database, after it is sent to the database, or even just to obtain information about the statement that has been sent. To manipulate or customize data calls, it is possible to make use of the *DataHandler* interface that is available via zxJDBC. There are basically three different methods for handling type mappings when using DataHandler. They are called at different times in the process, one when fetching and the other when binding objects for use in a prepared statement. These datatype mapping callbacks are categorized into four different groups: life cycle, developer support, binding prepared statements, and building results.

At first mention, customizing and manipulating statements can seem overwhelming and perhaps even a bit daunting. However, the zxJDBC DataHandler makes this task fairly trivial. Simply create a handler class and implement the functionality that is required by overriding a given handler method. What follows is a listing of the various methods that can be overridden, and we'll look at a simple example afterward.

Life Cycle

*public void preExecute(Statement stmt)**throws SQLException;*

A callback prior to each execution of the statement. If the statement is a PreparedStatement (created when parameters are sent to the execute method), all the parameters will have been set.

*public void postExecute**(Statement stmt) throws SQLException;*

A callback after successfully executing the statement. This is particularly useful for cases such as auto-incrementing columns where the statement knows the inserted value.

Developer Support

*public String getMetaDataName**(String name);*

A callback for determining the proper case of a name used in a DatabaseMetaData method, such as getTables(). This is particularly useful for Oracle which expects all names to be upper case.

*public PyObject getRowId**(Statement stmt) throws SQLException;*

A callback for returning the row id of the last insert statement.

Binding Prepared Statements

*public Object getJDBCObject**(PyObject object, int type);*

This method is called when a PreparedStatement is created through use of the execute method. When the parameters are being bound to the statement, the DataHandler gets a callback to map the type. *This is only called if type bindings are present.*

*public Object getJDBCObject**(PyObject object);*

This method is called when no type bindings are present during the execution of a PreparedStatement.

Building Results

*public PyObject ge**tPyObject(**ResultSet set, int col, int type);*

This method is called upon fetching data from the database. Given the JDBC type, return the appropriate PyObject subclass from the Java object at column col in the ResultSet set.

Now we'll examine a simple example of utilizing this technique. The recipe basically follows these steps:

1. Create a handler class to implement a particular functionality (must implement the DataHandler interface).
2. Assign the created handler class to a given cursor object.
3. Use the cursor object to make database calls.

In Listing 12-22, we override the *preExecute* method to print a message stating that the functionality has been altered. As you can see, it is quite easy to do and opens up numerous possibilities.

Listing 12-22. PyHandler.py

```
from com.ziclix.python.sql import DataHandler

class PyHandler(DataHandler):
    def __init__(self, handler):
        self.handler = handler
        print 'Inside DataHandler'
    def getPyObject(self, set, col, datatype):
        return self.handler.getPyObject(set, col, datatype)
    def getJDBCObject(self, object, datatype):
        print "handling prepared statement"
        return self.handler.getJDBCObject(object, datatype)
    def preExecute(self, stmt):
        print "calling pre-execute to alter behavior"
        return self.handler.preExecute(stmt)
```

Jython Interpreter Code

```
>>> cursor.datahandler = PyHandler(cursor.datahandler)
Inside DataHandler
>>> cursor.execute("insert into test values (?,?)", [1,2])
calling pre-execute
```

History

zxJDBC was contributed by Brian Zimmer, one-time lead committer for Jython. This API was written to enable Jython developers to have the capability of working with databases using techniques that more closely resembled the Python DB API. The package eventually became part of the Jython distribution and today it is one of the most important underlying APIs for working with higher level frameworks such as Django. The zxJDBC API is evolving at the time of this publication, and it is likely to become more useful in future releases.

Object Relational Mapping

Although zxJDBC certainly offers a viable option for database access via Jython, there are many other solutions available. Many developers today are choosing to use ORM (Object Relational Mapping) solutions to work with the database. This section is not an introduction to ORM, we assume that you are at least a bit familiar with the topic. Furthermore, the ORM solutions that are about to be discussed have an enormous amount of very good documentation already available either on the web or in book format. Therefore, this section will give insight on how to use these technologies with Jython, but it will not go into great detail on how each ORM solution works. With that said, there is no doubt in stating that these solutions are all very powerful and capable for standalone and enterprise applications alike.

In the next couple of sections, we'll cover how to use some of the most popular ORM solutions available today with Jython. You'll learn how to set up your environment and how to code Jython to work with each ORM. By the end of this chapter, you should have enough knowledge to begin working with these ORMs using Jython, and even start building Jython ORM applications.

SqlAlchemy

No doubt about it, SqlAlchemy is one of the most widely known and used ORM solutions for the Python programming language. It has been around long enough that its maturity and stability make it a great contender for use in your applications. It is simple to setup, and easy-to-use for both new databases and legacy databases alike. You can download and install SqlAlchemy and begin using it in a very short amount of time. The syntax for using this solution is very straight forward, and as with other ORM technologies, working with database entities occurs via the use of a mapper that links a special Jython class to a particular table in the database. The overall result is that the application persists through the use of entity classes as opposed to database SQL transactions.

In this section we will cover the installation and configuration of SqlAlchemy with Jython. The section will then show you how to get started using it through a few short examples; we will not get into great detail as there are plenty of excellent references on SqlAlchemy already. However, this section should fill in the gaps for making use of this great solution on Jython.

Installation

We'll begin by downloading SqlAlchemy from the web site (www.sqlalchemy.org), at the time of this writing the version that should be used is 0.6. This version has been installed and tested with the Jython 2.5.0 release. Once you've downloaded the package, unzip it to a directory on your workstation and then traverse to that directory in your terminal or command prompt. Once you are inside of your SqlAlchemy directory, issue the following command to install:

```
jython setup.py install
```

Once you've completed this process, SqlAlchemy should be successfully installed into your jython Libsite-packages directory. You can now access the SqlAlchemy modules from Jython, and you can open up your terminal and check to ensure that the install was a success by importing sqlalchemy and checking the version. See Listing 12-23.

Listing 12-23.

```
>>> import sqlalchemy
>>> sqlalchemy.__version__
'0.6beta1'
>>>
```

After we've ensured that the installation was a success, it is time to begin working with SqlAlchemy via the terminal. However, we have one step left before we can begin. Jython uses zxJDBC to implement the Python database API in Java. The end result is that most of the dialects that are available for use with SqlAlchemy will not work with Jython out of the box. This is because the dialects need to be rewritten to implement zxJDBC. At the time of this writing, we could only find one completed dialect, zxoracle, that was rewritten to use zxJDBC, and we'll be showing you some examples based upon zxoracle in the next sections. However, other dialects are in the works including SQL Server and MySQL. The bad news is that SqlAlchemy will not yet work with every database available, on the other hand, Oracle is a very good start and implementing a new dialect is not very difficult. You can find the zxoracle.py dialect included in the source for this book. Browse through it and you will find that it may not be too difficult to implement a similar dialect for the database of your choice. You can either place zxoracle somewhere on your Jython path, or place it into the Lib directory in your Jython installation.

Lastly, we will need to ensure that our database JDBC driver is somewhere on our path so that Jython can access it. Once you've performed the procedures included in this section, start up Jython and practice some basic SqlAlchemy using the information from the next couple of sections.

Using SqlAlchemy

We can work directly with SqlAlchemy via the terminal or command line. There is a relatively basic set of steps you'll need to follow in order to work with it. First, import the necessary modules for the tasks you plan to perform. Second, create an engine to use while accessing your database. Third, create your database tables if you have not yet done so, and map them to Python classes using a SqlAlchemy mapper. Lastly, begin to work with the database.

Now there are a couple of different ways to do things in this technology, just like any other. For instance, you can either follow a very granular process for table creation, class creation, and mapping that involves separate steps for each, or you can use what is known as a declarative procedure and perform all of these tasks at the same time. We will show you how to do each of these in this chapter, along with performing basic database activities using SqlAlchemy. If you are new to SqlAlchemy, we suggest reading through this section and then going to sqlalchemy.org and reading through some of the large library of documentation available there. However, if you're already familiar with SqlAlchemy, you can move on if you wish because the rest of this section is a basic tutorial of the ORM solution itself.

Our first step is to create an engine that can be used with our database. Once we've got an engine created then we can begin to perform database tasks making use of it. Type the following lines of code (Listing 12-24) in your terminal, replacing database specific information with the details of your development database.

Listing 12-24. Creating a Database Engine and Performing Database Tasks

```
>>> import zxoracle
>>> from sqlalchemy import create_engine
>>> db = create_engine('zxoracle://schema:password@hostname:port/database)
```

Next, we'll create the metadata that is necessary to create our database table using SqlAlchemy (Listing 12-25). You can create one or more tables via metadata, and they are not actually created until after the metadata is applied to your database engine using a create_all() call on the metadata. In this example, we are going to walk you through the creation of a table named Player that will be used in an application example in the next section.

Listing 12-25. Creating a Database Table

```
>>>player = Table('player', metadata,
... Column('id', Integer, primary_key=True),
... Column('first', String(50)),
... Column('last', String(50)),
... Column('position', String(30)))
>>> metadata.create_all(engine)
```

Our table should now exist in the database and the next step is to create a Python class to use for accessing this table. See Listing 12-26.

Listing 12-26. Creating a Python Class to Access a Database Table

```
class Player(object):
    def __init__(self, first, last, position):
        self.first = first
        self.last = last
        self.position = position
    def __repr__(self):
        return "<Player('%s', '%s', '%s')>" %(self.first, self.last,
self.position)
```

The next step is to create a mapper to correlate the Player python object and the player database table. To do this, we use the mapper() function to create a new Mapper object binding the class and table together (Listing 12-27). The mapper function then stores the object away for future reference.

Listing 12-27. Create a Mapper to Correlate the Python Object and the Database Table

```
>>> from sqlalchemy.orm import mapper
>>> mapper(Player, player)
<Mapper at 0x4; Player>
```

Creating the mapper is the last step in the process of setting up the environment to work with our table. Now, let's go back and take a quick look at performing all of these steps in an easier way. If we want to create a table, class, and mapper all at once, then we can do this declaratively. Please note that with the Oracle dialect, we need to use a sequence to generate the auto-incremented id column for the table. To do so, import the sqlalchemy.schema.Sequence object and pass it to the id column when creating. You must ensure that you've manually created this sequence in your Oracle database or this will not work. See Listing 12-28.

Listing 12-28. Creating a Table, Class and Mapper at Once

```
SQL> create sequence id_seq
2 start with 1
3 increment by 1;
```

```
Sequence created.

# Delarative creation of the table, class, and mapper
>>> from sqlalchemy.ext.declarative import declarative_base
>>> from sqlalchemy.schema import Sequence
>>> Base = declarative_base()
>>> class Player(object):
...     __tablename__ = 'player'
...     id = Column(Integer, Sequence(‘id_seq’), primary_key=True)
...     first = Column(String(50))
...     last = Column(String(50))
...     position = Column(String(30))
...     def __init__(self, first, last, position):
...         self.first = first
...         self.last = last
...         self.position = position
...     def __repr__(self):
...         return "<Player('%s','%s','%s')>" % (self.first, self.last,
self.position)
...
```

It is time to create a session and begin working with our database. We must create a session class and bind it to our database engine that was defined with create_engine earlier. Once created, the Session class will create new session object for our database. The Session class can also do other things that are out of scope for this section, but you can read more about them at sqlalchemy.org or other great references available on the web. See Listing 12-29.

Listing 12-29. Creating a Session Class

```
>>> from sqlalchemy.orm import sessionmaker
>>> Session = sessionmaker(bind=db)
```

We can start to create Player objects now and save them to our session. The objects will persist in the database once they are needed; this is also known as a flush(). If we create the object in the session and then query for it, SqlAlchemy will first persist the object to the database and then perform the query. See Listing 12-30.

Listing 12-30. Creating and Querying the Player Object

```
#Import sqlalchemy module and zxoracle
>>> import zxoracle
>>> from sqlalchemy import create_engine
>>> from sqlalchemy import Table, Column, String, Integer, MetaData,
ForeignKey
>>> from sqlalchemy.schema import Sequence

# Create engine
>>> db = create_engine('zxoracle://schema:password@hostname:port/database’)
# Create metadata and table
>>> metadata = MetaData()
>>> player = Table('player', metadata,
... Column('id', Integer, Sequence('id_seq'), primary_key=True),
... Column('first', String(50)),
... Column('last', String(50)),
... Column('position', String(30)))
>>> metadata.create_all(db)

# Create class to hold table object
>>> class Player(object):
... def __init__(self, first, last, position):
... self.first = first
... self.last = last
... self.position = position
... def __repr__(self):
... return "<Player('%s','%s','%s')>" % (self.first, self.last,
self.position)

# Create mapper to map the table to the class
>>> from sqlalchemy.orm import mapper
>>> mapper(Player, player)
<Mapper at 0x4; Player>

# Create Session class and bind it to the database
>>> from sqlalchemy.orm import sessionmaker
>>> Session = sessionmaker(bind=db)
>>> session = Session()

# Create player objects, add them to the session
>>> player1 = Player('Josh', 'Juneau', 'forward')
>>> player2 = Player('Jim', 'Baker', 'forward')
>>> player3 = Player('Frank', 'Wierzbicki', 'defense')
>>> player4 = Player('Leo', 'Soto', 'defense')
>>> player5 = Player('Vic', 'Ng', 'center')
>>> session.add(player1)
>>> session.add(player2)
>>> session.add(player3)
>>> session.add(player4)
>>> session.add(player5)

# Query the objects
>>> forwards = session.query(Player).filter_by(position='forward').all()
>>> forwards
[<Player('Josh','Juneau','forward')>, <Player('Jim','Baker','forward')>]
>>> defensemen = session.query(Player).filter_by(position='defense').all()
>>> defensemen
[<Player('Frank','Wierzbicki','defense')>, <Player('Leo','Soto','defense')>]
>>> center = session.query(Player).filter_by(position='center').all()
>>> center
[<Player('Vic','Ng','center')>]
```

Well, hopefully from this example you can see the benefits of using SqlAlchemy. Of course, you can perform all of the necessary SQL actions such as insert, update, select, and delete against the objects. However, as said before, there are many very good tutorials where you can learn how to do these things. We've barely scratched the surface of what you can do with SqlAlchemy, it is a very powerful tool to add to any Jython or Python developer's arsenal.

Hibernate

Hibernate is a very popular object relational mapping solution used in the Java world. As a matter of fact, it is so popular that many other ORM solutions are either making use of Hibernate or extending it in various ways. As Jython developers, we can make use of Hibernate to create powerful hybrid applications. Because Hibernate works by mapping POJO (plain old Java object) classes to database tables, we cannot map our Jython objects to it directly. While we could always try to make use of an object factory to coerce our Jython objects into a format that Hibernate could use, this approach leaves a bit to be desired. Therefore, if you wish to create an application coded entirely using Jython, this would probably not be the best ORM solution. However, most Jython developers are used to doing a bit of work in Java and as such, they can harness the maturity and power of the Hibernate API to create first-class hybrid applications. This section will show you how to create database persistence objects using Hibernate and Java, and then use them directly from a Jython application. The end result, code the entity POJOs in Java, place them into a JAR file along with Hibernate and all required mapping documents, and then import the JAR into your Jython application and use.

We have found that the easiest way to create such an application is to make use of an IDE such as Eclipse or Netbeans. Then create two separate projects, one of the projects would be a pure Java application that will include the entity beans. The other project would be a pure Jython application that would include everything else. In this situation, you could simply add resulting JAR from your Java project into the sys.path of your Jython project and you'll be ready to go. However, this works just as well if you do not wish to use an IDE.

It is important to note that this section will provide you with one use case for using Jython, Java, and Hibernate together. There may be many other scenarios in which this combination of technologies would work out just as well, if not better. It is also good to note that this section will not cover Hibernate in any great depth; we'll just scratch the surface of what it is capable of doing. There are a plethora of great Hibernate tutorials available on the web if you find this solution to be useful.

Entity Classes and Hibernate Configuration

Because our Hibernate entity beans must be coded in Java, most of the Hibernate configuration will reside in your Java project. Hibernate works in a straightforward manner. You basically map a table to a POJO and use a configuration file to map the two together. It is also possible to use annotations as opposed to XML configuration files, but for the purposes of this use case we will show you how to use the configuration files.

The first configuration file we need to assemble is the hibernate.cfg.xml, which you can find in the root of your Java project directory tree. The purpose of this file is to define your database connection information as well as declare which entity configuration files will be used in your project. For the purposes of this example, we will be using the PostgreSql database, and we'll be using the classic examples of the hockey roster application. This makes for a very simple use-case as we only deal with one table here, the Player table. Hibernate makes it very possible to work with multiple tables and even associate them in various ways.

Listing 12-31.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate
Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
    <!-- Database connection settings -->
    <property
    name="connection.driver_class">org.postgresql.Driver</property>
    <property
    name="connection.url">jdbc:postgresql://localhost/database-
name</property>
    <property name="connection.username">username</property>
    <property name="connection.password">password</property>
    <!-- JDBC connection pool (use the built-in) -->
    <property name="connection.pool_size">1</property>
    <!-- SQL dialect -->
    <property
    name="dialect">org.hibernate.dialect.PostgreSQLDialect</property>
    <mapping resource="org/jythonbook/entity/Player.hbm.xml"/>
</session-factory>
</hibernate-configuration>
```

Our next step is to code the plain old Java object for our database table. In this case, we’ll code an object named Player that contains only four database columns: id, first, last, and position. As you’ll see, we use standard public accessor methods with private variables in this class.

Listing 12-32.

```
package org.jythonbook.entity;

public class Player {

    public Player(){}

    private long id;
    private String first;
    private String last;
    private String position;

    public long getId(){
        return this.id;
    }
    private void setId(long id){
        this.id = id;
    }
    public String getFirst(){
        return this.first;
    }
    public void setFirst(String first){
        this.first = first;
    }
    public String getLast(){
        return this.last;
    }
    public void setLast(String last){
        this.last = last;
    }
    public String getPosition(){
        return this.position;
    }
    public void setPosition(String position){
        this.position = position;
    }
}
```

Lastly, we will create a configuration file that will be used by Hibernate to map our POJO to the database table itself. We'll ensure that the primary key value is always populated by using a generator class type of increment. Hibernate also allows for the use of other generators, including sequences if desired. The player.hbm.xml file should go into the same package as our POJO, in this case, the org.jythonbook.entity package.

Listing 12-33. Creating a Hibernate Configuration File

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="org.jythonbook.entity">
    <class name="Player" table="player" lazy="true">
        <comment>Player for Hockey Team</comment>
        <id name="id" column="id">
            <generator class="increment"/>
        </id>
        <property name="first" column="first"/>
        <property name="last" column="last"/>
        <property name="position" column="position"/>
    </class>
</hibernate-mapping>
```

That is all we have to do inside of the Java project for our simple example. Of course, you can add as many entity classes as you'd like to your own project. The main point to remember is that all of the entity classes are coded in Java, and we will code the rest of the application in Jython.

Jython Implementation Using the Java Entity Classes

The remainder of our use-case will be coded in Jython. Although all of the Hibernate configuration files and entity classes are coded and place within the Java project, we'll need to import that project into the Jython project, and also import the Hibernate JAR file so that we can make use of its database session and transactional utilities to work with the entities. In the case of Netbeans, you'd create a Python application then set the Python platform to Jython 2.5.0. After that, you should add all of the required Hibernate JAR files as well as the Java project JAR file to the Python path from within the project properties. Once you've set up the project and taken care of the dependencies, you're ready to code the implementation.

As said previously, for this example we are coding a hockey roster implementation. The application runs on the command line and basically allows one to add players to a roster, remove players, and check the current roster. All of the database transactions will make use of the Player entity we coded in our Java application, and we'll make use of Hibernate's transaction management from within our Jython code.

Listing 12-34. Hockey Roster Application Code

```
from org.hibernate.cfg import Environment
from org.hibernate.cfg import Configuration
from org.hibernate import Query
from org.hibernate import Session
from org.hibernate import SessionFactory
from org.hibernate import Transaction
from org.jythonbook.entity import Player


class HockeyRoster:

    def __init__(self):
        self.cfg = Configuration().configure()
        self.factory = self.cfg.buildSessionFactory()

    def make_selection(self):
        '''
        Creates a selector for our application. The function prints output to
the
        command line. It then takes a parameter as keyboard input at the
command
        line in order to choose our application option.
        '''
        options_dict = {1:self.add_player,
                        2:self.print_roster,
                        3:self.search_roster,
                        4:self.remove_player}
        print "Please chose an option\\n"

        selection = raw_input('''Press 1 to add a player, 2 to print the
roster,
                3 to search for a player on the team,
                4 to remove player, 5 to quit: ''')
        if int(selection) not in options_dict.keys():
            if int(selection) == 5:
                print "Thanks for using the HockeyRoster application."
            else:
                print "Not a valid option, please try again\\n"
                self.make_selection()
        else:
            func = options_dict[int(selection)]
            if func:
                func()
            else:
                print "Thanks for using the HockeyRoster application."

    def add_player(self):
        '''
        Accepts keyboard input to add a player object to the roster list.
        This function creates a new player object each time it is invoked
        and inserts a record into the corresponding database table.
        '''
        addNew = 'Y'
        print "Add a player to the roster by providing the following
information\\n"
        while addNew.upper() == 'Y':
            first = raw_input("First Name: ")
            last = raw_input("Last Name: ")
            position = raw_input("Position: ")
            id = len(self.return_player_list())
            session = self.factory.openSession()
            try:
                tx = session.beginTransaction()
                player = Player()
                player.first = first
                player.last = last
                player.position = position
                session.save(player)
                tx.commit()
            except Exception,e:
                if tx!=None:
                    tx.rollback()
                    print e
            finally:
                session.close()

            print "Player successfully added to the roster\\n"
            addNew = raw_input("Add another? (Y or N)")
        self.make_selection()

    def print_roster(self):
        '''
        Prints the contents of the Player database table
        '''
        print "====================\\n"
        print "Complete Team Roster\\n"
        print "======================\\n\\n"
        playerList = self.return_player_list()
        for player in playerList:
            print "%s %s - %s" % (player.first, player.last, player.position)
        print "\\n"
        print "=== End of Roster ===\\n"
        self.make_selection()

    def search_roster(self):
        '''
        Takes input from the command line for a player's name to search
within the
        database. If the player is found in the list then an affirmative
message
        is printed. If not found, then a negative message is printed.
        '''
        index = 0
        found = False
        print "Enter a player name below to search the team\\n"
        first = raw_input("First Name: ")
        last = raw_input("Last Name: ")
        position = None
        playerList = self.return_player_list()
        while index < len(playerList):
            player = playerList[index]
            if player.first.upper() == first.upper():
                if player.last.upper() == last.upper():
                    found = True
                    position = player.position
            index = index + 1
        if found:
            print '%s %s is in the roster as %s' % (first, last, position)
        else:
            print '%s %s is not in the roster.' % (first, last)
        self.make_selection()

    def remove_player(self):
        '''
        Removes a designated player from the database
        '''
        index = 0
        found = False
        print "Enter a player name below to remove them from the team
roster\\n"
        first = raw_input("First Name: ")
        last = raw_input("Last Name: ")
        position = None
        playerList = self.return_player_list()
        found_player = Player()
        while index < len(playerList):
            player = playerList[index]
            if player.first.upper() == first.upper():
                if player.last.upper() == last.upper():
                    found = True
                    found_player = player
            index = index + 1

        if found:
            print '''%s %s is in the roster as %s,
                are you sure you wish to remove?''' % (found_player.first,
                                                    found_player.last,
                                                                 found_player.position)
            yesno = raw_input("Y or N")
            if yesno.upper() == 'Y':
                session = self.factory.openSession()
                tx = None
                try:
                    delQuery = "delete from Player player where id = %s" %
(found_player.id)

                    tx = session.beginTransaction()
                    q = session.createQuery(delQuery)
                    q.executeUpdate()
                    tx.commit()
                    print 'The player has been removed from the roster',
found_player.id
                except Exception,e:
                    if tx!=None:
                        tx.rollback()
                        print e
                finally:
                    s   ession.close
            else:
                print 'The player will not be removed'
        else:
            print '%s %s is not in the roster.' % (first, last)
        self.make_selection()

    def return_player_list(self):
        '''
        Connects to database and retrieves the contents of the player table
        '''
        session = self.factory.openSession()
        try:
            tx = session.beginTransaction()
            playerList = session.createQuery("from Player").list()
            tx.commit()
        except Exception,e:
            if tx!=None:
                tx.rollback()
            print e
        finally:
            session.close
        return playerList

    # main
    #
    # This is the application entry point. It simply prints the application
title
    # to the command line and then invokes the makeSelection() function.
    if __name__ == "__main__":
        print "Hockey Roster Application\\n\\n"
        hockey = HockeyRoster()
        hockey.make_selection()
```

We begin our implementation in the main block, where the HockeyRoster class is instantiated. As you can see, the hibernate configuration is initialized and the session factory is built within the class initializer. Next, the make_selection() method is invoked which begins the actual execution of the program. The entire Hibernate configuration resides within the Java project, so we are not working with XML here, just making use of it. The code then begins to branch so that various tasks can be performed. In the case of adding a player to the roster, a user could enter the number 1 at the command prompt. You can see that the addPlayer() function simply creates a new Player object, populates it, and saves it into the database. Likewise, the searchRoster() function calls another function named returnPlayerList() which queries the player table using Hibernate query language and returns a list of Player objects.

In the end, we have a completely scalable solution. We can code our entities using a mature and widely used Java ORM solution, and then implement the rest of the application in Jython. This allows us to make use of the best features of the Python language, but at the same time, persist our data using Java.

Summary

You would be hard-pressed to find too many enterprise-level applications today that do not make use of a relational database in one form or another. The majority of applications in use today use databases to store information as they help to provide robust solutions. That being said, the topics covered in this chapter are very important to any developer. In this chapter, we learned that there are many different ways to implement database applications in Jython, specifically through the Java database connectivity API or an object relational mapping solution.

Source: http://www.jython.org/jythonbook/en/1.0/DatabasesAndJython.html