Technical Explorations

by Keith R. Bennett

Using Oracle in JRuby with Rails and Sequel

Jan 9, 2013 • keithrbennett

The Ruby culture prefers open source technologies, and when it comes to relational data bases, MySQL and Postgres are commonly used. However, there are times when the Rubyist will not be in a position to choose technologies and must inherit legacy decisions. For example, a common issue in the enterprise is the need to integrate with Oracle. In this article, I'll talk about integrating Oracle and JRuby (1), using both Active Record (Ruby on Rails) and the Sequel gem.

JDBC

The JVM typically communicates with data bases using JDBC (Java Data Base Connectivity). The JDBC layer provides an abstraction with which application code can access the data base without (for the most part) needing to know anything about its underlying implementation.

JDBC Jar Files

A JDBC jar (jar = **j**ava **ar**chive) file is provided for each data base product, typically by the vendor, and contains the code needed to implement the JDBC contract for that target data base. Since MySQL and Postgres are open source, their jar files can be freely copied around, and as a result, are included in the respective JDBC gems. This greatly simplifies configuration, since the gem takes care of storing and providing the location of the JDBC jar file.

The activerecord-jdbc-adapter gem includes several JDBC jar files. The complete list can be found by looking at the directory names beginning with "jdbc-" on the activerecord-jdbc-adapter GitHub page here, and, at the time of this writing, consists of Derby, H2, Hypersonic, jTDS (SQL Server and Sybase), MySQL, Postgres, and SQLite.

Dealing with the Oracle JDBC Jar File

With Oracle, it's a different story. Oracle does not permit freely coping their JDBC jar file, and in order to download it, you'll probably need to go to the Oracle web site and log in first. It would not be legal to write an Oracle JDBC gem that packaged this jar file, so, unfortunately, extra work is required.

The solution I chose was to:

1) download it to a directory outside of my project (/opt/oracle-jdbc/ojdbc6.jar)

2) have an environment variable point to it – I added this to my ~/.zshrc file (you might use ~/.bashrc instead):

```
>export ORACLE_JDBC_JAR=/opt/oracle-jdbc/ojdbc6.jar
```

3) use that variable at runtime to locate it (separate solutions for Rails and Sequel below).

Oracle and Rails - environment.rb and database.yml

For Rails, you'll need this in your config/initializers/environment.rb so that the JDBC jar file can be found at runtime:

```
>$CLASSPATH << ENV['ORACLE_JDBC_JAR']
```

Now you'll need to provide the appropriate values in the database.yml file (test and production groups are omitted for brevity):

```
><%
  require 'socket';
  host_name = ENV['OJTEST_DB_HOSTNAME']
  host_ip   = IPSocket.getaddress(host_name)
  db_name   = ENV['OJTEST_DB_DBNAME']
  userid    = ENV['OJTEST_DB_USERID']
  password  = ENV['OJTEST_DB_PASSWORD']
  dev_url = "jdbc:oracle:thin://@#{host_ip}:1521:#{db_name}"
%>

development:
  adapter: jdbc
  username: <%= userid %>
  password: <%= password %>
  driver: oracle.jdbc.OracleDriver
  url:  <%= dev_url %>
```

As you can see, I used environment variables beginning with "OJTEST_DB_" to provide the required values, although that is not important and you can use any approach that works for you.

More importantly, note that I am translating the Oracle host's name to its IP address. This was necessary due to an apparent bug in Oracle's driver.

Oracle and the Sequel Gem

There is also the excellent Sequel gem that can be used for general data base access, even (perhaps especially) in a minimal script. Here's a sample script that worked succesfully for me:

```
>#!/usr/bin/env ruby

require 'sequel'
require 'socket'

$CLASSPATH << ENV['ORACLE_JDBC_JAR']

def init_connection

  host_name = ENV['OJTEST_DB_HOSTNAME']
  host_ip   = IPSocket.getaddress(host_name)
  db_name   = ENV['OJTEST_DB_DBNAME']
  userid    = ENV['OJTEST_DB_USERID']
  password  = ENV['OJTEST_DB_PASSWORD']
  dev_url   = "jdbc:oracle:thin://@#{host_ip}:1521:#{db_name}"
  url       = "jdbc:oracle:thin:#{userid}/#{password}@#{host_name}:1521:#{db_name}"

  puts "Connecting to #{url}..."
  db = Sequel.connect(url)
  puts 'Initialized connection.'
  db
end

def create_table(db)
  puts 'Creating table'
  # db.run "CREATE TABLE artists (id NUMBER, name VARCHAR(255) NOT NULL)"
  DB.create_table(:artists) do
    primary_key :id
    String :name
  end
end
```

```
def add_records(db)
  puts 'Adding records'
  dataset = db[:artists]
  dataset.insert(name: 'Vincent Van Gogh')
  dataset.insert(name: 'Lino of Cebu')
  dataset
end

def show_records(dataset)
  puts 'Showing records'
  dataset.map { |row| puts row[:name] }
end

# Call this once to set up the table with some records:
def setup(db)
  create_table(db)
  add_records(db)
end

DB = init_connection
# DB.run 'drop table artists'
setup(DB)
show_records(DB[:artists])
```

Conclusion

This setup took me some time to figure out, but after I did, things went smoothly. I'd like to hear if the approaches described here worked for you or if you have any problems with them.

Footnotes:

(1) an implementation of Ruby that runs on the Java Virtual Machine (aka *JVM*)

Published with GitHub Pages