

Practically Groovy: JDBC programming with Groovy

Build your next reporting application using GroovySql

Skill Level: Intermediate

[Andrew Glover \(andrew@thirstyhead.com\)](mailto:andrew@thirstyhead.com)

Co-Founder

ThirstyHead.com

[Scott Davis \(scott@thirstyhead.com\)](mailto:scott@thirstyhead.com)

Founder

ThirstyHead.com

11 Jan 2005

Take your practical knowledge of Groovy one step further this month, as Andrew Glover shows you how to use GroovySql to build a simple data-reporting application. GroovySql combines closures and iterators to ease Java Database Connectivity (JDBC) programming by shifting the burden of resource management from you to the Groovy framework itself.

In the previous installments of the *Practically Groovy* series, you discovered some pretty nifty features of Groovy. In the [first article](#), you learned how to apply Groovy for simpler, speedier unit testing of normal Java™ code. In the [second installment](#), you saw the expressiveness that Groovy can bring to Ant builds. This time you find out another practical use of Groovy; that is, how you can use it to quickly build SQL-based reporting applications.

Scripting languages are typically excellent tools for quickly building reporting applications, but building such applications is distinctively breezy with Groovy. Groovy's lightweight syntax can alleviate some of the verbosity of JDBC in the Java language, but its real punch comes from closures, which elegantly shift the responsibility of resource handling from the client to the framework itself, where the weight is easier to handle.

In this month's article, I'll start with a quick overview of the features of GroovySql and show you how to put them to work by building a simple data-reporting application. To get the most out of the discussion, you should be familiar with JDBC programming on the Java platform. You may also want to review [last month's introduction to closures](#) in Groovy, because they play an important role here. The most important concept to focus on this month, however, is iteration, because iterators play an important role in Groovy's enhancement of JDBC. So I'll start you out with an overview of iterator methods in Groovy.

Enter the iterator

Iteration is one of the most common and useful tactics in all kinds of programming situations. An *iterator* is a kind of code helper that lets you quickly access data in any collection or container, one at a time. Groovy improves the Java language's concept of iterators by making them implicit and more simple to use. In Listing 1, you can see the effort it takes to print out each element of a `String` collection using the Java language.

Listing 1. Iterators in normal Java code

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
public class JavaIteratorExample {
    public static void main(String[] args) {
        Collection coll = new ArrayList();
        coll.add("JMS");
        coll.add("EJB");
        coll.add("JMX");
        for(Iterator iter = coll.iterator(); iter.hasNext();){
            System.out.println(iter.next());
        }
    }
}
```

In Listing 2, you can see how Groovy simplifies my efforts. Here, I get to bypass the `Iterator` interface and use iterator-like methods on collections themselves. What's more, Groovy's iterator methods accept closures, which are evoked for each iteration cycle. Listing 2 shows the preceding Java language-based example transformed by Groovy.

Listing 2. Iterators in Groovy

```
class IteratorExample1{
    static void main(String[] args) {
        def coll = ["JMS", "EJB", "JMX"]
        coll.each{ item ->
            println item
        }
    }
}
```

```
}
```

As you can see, unlike typical Java code, Groovy controls my iteration-specific code while allowing me to pass in the behavior I need. With this control, Groovy neatly shifts the responsibility of resource handling from me to itself. Putting Groovy in charge of resource handling is extremely powerful. It also makes the job of programming much easier and consequently, quicker.

About this series

The key to incorporating any tool into your development practice is knowing when to use it and when to leave it in the box. Scripting languages can be an extremely powerful addition to your toolkit, but only when applied properly to appropriate scenarios. To that end, [Practically Groovy](#) is a series of articles that explores the practical uses of Groovy, helping you learn when and how to apply them successfully.

Introducing GroovySql

Groovy's SQL magic is found in an elegant API called GroovySql. Using closures and iterators, GroovySql neatly shifts JDBC resource management from you, the developer, to the Groovy framework. In so doing, it removes the drudgery from JDBC programming so that you can focus on queries and their results.

Just in case you've forgotten what a hassle normal Java JDBC programming can be, I'm all too happy to remind you! In Listing 3, you can see a simple JDBC programming example in the Java language.

Listing 3. JDBC programming in normal Java

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
public class JDBCExample1 {
    public static void main(String[] args) {
        Connection con = null;
        Statement stmt = null;
        ResultSet rs = null;
        try{
            Class.forName("com.mysql.jdbc.Driver");
            con = DriverManager.getConnection("jdbc:mysql://localhost:3306/words",
                "words", "words");
            stmt = con.createStatement();
            rs = stmt.executeQuery("select * from word");
            while (rs.next()) {
                System.out.println("word id: " + rs.getLong(1) +
                    " spelling: " + rs.getString(2) +
                    " part of speech: " + rs.getString(3));
            }
        }
    }
}
```

```
    }catch(SQLException e){
        e.printStackTrace();
    }catch(ClassNotFoundException e){
        e.printStackTrace();
    }finally{
        try{rs.close();}catch(Exception e){}
        try{stmt.close();}catch(Exception e){}
        try{con.close();}catch(Exception e){}
    }
}
```

Wow. Listing 3 contains roughly 40 lines of code just to view the contents of a table! How many do you think it takes using GroovySql? If you guess more than 10 lines, you would be wrong. Watch how Groovy elegantly lets me focus on the task at hand -- performing a simple query-- and handles the underlying resource handling for me in Listing 4.

Listing 4. Welcome to GroovySql!

```
import groovy.sql.Sql
class GroovySqlExample1{
    static void main(String[] args) {
        def sql = Sql.newInstance("jdbc:mysql://localhost:3306/words", "words",
            "words", "com.mysql.jdbc.Driver")
        sql.eachRow("select * from word"){ row ->
            println row.word_id + " " + row.spelling + " " + row.part_of_speech
        }
    }
}
```

Not bad. Using just a few lines, I've just coded the same behavior from Listing 3, without relying on `Connection` closing, `ResultSet` closing, or any of the other familiar heavy lifters found in JDBC programming. Pretty exciting stuff, if you ask me -- and so easy, too. Now let me show you exactly how I did it.

Performing a simple query

In the first line of [Listing 4](#), I created an instance of Groovy's `Sql` class, which is used to connect to a desired database. In this case, I created an `Sql` instance pointing to a MySQL database running on my machine. So far, pretty basic, right? The real knockout is the next part, where the one-two punches of iterators and closures shows their power.

Think of the `eachRow` method as an iterator on the result of the passed-in query. Underneath, you can visualize a JDBC `ResultSet` object being returned and its contents being passed into a `for` loop. Consequently, the closure I passed in is executed for each iteration. If the `word` table found in the database only had three rows, the closure is executed three times -- printing out the `word_id`, `spelling`, and `part_of_speech` values.

The code is simplified even further by dropping my named variable, `row`, from the equation and using one of Groovy's implicit variables: `it`, which happens to be the instance of the iterator. If I did this, the preceding code would be written as shown in Listing 5.

Listing 5. Groovy's `it` variable in GroovySql

```
import groovy.sql.Sql
class GroovySqlExample2{
    static void main(String[] args) {
        def sql = Sql.newInstance("jdbc:mysql://localhost:3306/words", "words",
            "words", "com.mysql.jdbc.Driver")
        sql.eachRow("select * from word"){
            println it.spelling + " ${it.part_of_speech}"
        }
    }
}
```

In this code I was able to drop the `row` variable and use `it` instead. Additionally, I can reference the `it` variable in `String` statements, as I did with `${it.part_of_speech}`.

Doing more complex queries

The previous examples are fairly simple, but GroovySql is just as solid when it comes to more complex data manipulation queries such as `insert`, `update`, and `delete` queries. For these, you wouldn't necessarily want to use iterators, so Groovy's `Sql` object provides the `execute` and `executeUpdate` methods instead. These methods are reminiscent of the normal JDBC `statement` class, which has an `execute` and an `executeUpdate` method as well.

In Listing 6, you see a simple `insert` that uses variable substitution again with the `${}` syntax. This code simply inserts a new row into the `word` table.

Listing 6. Inserts with GroovySql

```
def wid = 5
def spelling = "Nefarious"
def pospeech = "Adjective"
sql.execute("insert into word (word_id, spelling, part_of_speech)
    values (${wid}, ${spelling}, ${pospeech})")
```

Groovy also provides an overridden version of the `execute` method, which takes a list of values that correspond to any `?` elements found in the query. In Listing 7, I've simply queried for a particular row in the `word` table. Underneath the hood, GroovySql creates an instance of the normal Java language `java.sql.PreparedStatement`.

Listing 7. PreparedStatements with GroovySql

```
def val = sql.execute("select * from word where word_id = ?", [5])
```

Updates are much the same in that they utilize the `executeUpdate` method. Notice, too, that in Listing 8 the `executeUpdate` method takes a list of values that will be matched to the corresponding `?` elements in the query.

Listing 8. Updates with GroovySql

```
def nid = 5
def newSpelling = "Dastardly"
sql.executeUpdate("update word set spelling = ? where word_id = ?", [newSpelling, nid])
```

Deletes are essentially the same as inserts, except, of course, that the query's syntax is different, as shown in Listing 9.

Listing 9. Deletes with GroovySql

```
sql.execute("delete from word where word_id = ?" , [5])
```

Simplifying data manipulation

Any API or utility that intends to simplify JDBC programming had better have some rock-solid data manipulation features and in this section, I'll show you three more.

DataSets

Building on its foundation of simplicity, GroovySql supports the notion of `DataSet` types, which are basically object representations of database tables. With a `DataSet`, you can iterate over rows and add new rows. Indeed, using datasets is a convenient way of representing a collection of data common to a table.

In Listing 10, I've created a `DataSet` from the `word` table.

Listing 10. Datasets with GroovySql

```
import groovy.sql.Sql
class GroovyDatasetsExample1{
    static void main(String[] args) {
        def sql = Sql.newInstance("jdbc:mysql://localhost:3306/words", "words",
            "words", "com.mysql.jdbc.Driver")
        def words = sql.dataSet("word")
        words.add(word_id:"9999", spelling:"clerisy", part_of_speech:"Noun")
        words.each{ word ->
```

```
        println word.word_id + " " + word.spelling
    }
}
```

As you can see, GroovySql's `DataSet` type makes it easy to iterate over the contents of a table with the `each` method and add a new rows with the `add` method, which takes a `map` representing the desired data.

Using stored procedures and negative indexing

Stored procedure calling and negative indexing can be essential aspects of data manipulation. GroovySql makes stored procedure calling as simple as using the `call` method on the `Sql` class. For negative indexing, GroovySql provides its enhanced `ResultSet` type, which works much like *collections* in Groovy. For example, if you wanted to grab the last column in a result set, you could do as shown in Listing 11.

Listing 11. Negative indexing with GroovySql

```
sql.eachRow("select * from word"){ row ->
    (0..2).each{ i ->
        print "Field ${i}: "
        println row.getAt(i)
    }
    println "Last field using -1 index = " + row.getAt(-1)
}
```

As you can see in Listing 11, grabbing the last column in a result set is as easy as indexing with a `-1`. If wanted to, I could also access the same column using the `2` index.

Again, these examples are pretty basic, but they should give you a good sense of GroovySql's powers. I'll close this month's lesson with a real-world example demonstrating all the features discussed so far.

Writing a simple reporting application

Reporting applications usually pull information from a database. In a typical business environment, you might be asked to write a reporting application to inform the sales team about current Web sales or to let the development team do daily checkups on the performance of some aspect of the system, such as its database.

For the sake of this simple example, let's assume you've just deployed an enterprisewide Web application. It's running flawlessly, of course, because you wrote a wealth of unit tests (in Groovy) as you went along; but you still need to generate a report about the state of the database for tuning purposes. You want to know how

the application is being used by your customers so you can anticipate performance issues and address them.

Usually time constraints limit the number of bells and whistles you can apply to such an application. But your newly acquired knowledge of GroovySql will let you knock out this application in a snap, leaving you with time to add in extra features if you so desire.

The details

Your target database is in this case MySQL, which just so happens to support the notion of discovering status information with a query. The status information you're interested in is as follows:

- Uptime
- Total number of overall queries processed
- Proportions of specific queries, such as `insert`, `update`, and `select`

Getting this information out of a MySQL database is almost too easy using GroovySql. Since you're building it for the development team, you'll probably just start out with a simple command-line report, but you could easily Web-enable the report in a later iteration. The use case for the reporting example might look something like this:

1.	Connect to our application's live database
2.	Issue <code>show status</code> queries and capture:
	a. <code>uptime</code>
	b. <code>total queries</code>
	c. <code>total inserts</code>
	d. <code>total updates</code>
	e. <code>total selects</code>
3.	With those data points, calculate:
	a. <code>queries per minute</code>
	b. <code>percentage of total insert queries</code>
	c. <code>percentage of total update queries</code>
	d. <code>percentage of total select queries</code>

In Listing 12, you can see the final result: an application that reports the desired database statistics. The initial lines of code obtain a connection to the production database, followed by a series of `show status` queries that let you calculate queries per minute and then break them down by type. Notice how variables like

uptime pop into existence as they're defined.

Listing 12. Database status reporting with GroovySql

```
import groovy.sql.Sql
class DBStatusReport{
    static void main(String[] args) {
        def sql = Sql.newInstance("jdbc:mysql://yourserver.anywhere/tiger", "scott",
            "tiger", "com.mysql.jdbc.Driver")

        def uptime = ""
        def questions = ""
        sql.eachRow("show status"){ status ->
            if(status.variable_name == "Uptime"){
                uptime = status[1]
            }else if (status.variable_name == "Questions"){
                questions = status[1]
            }
        }
        println "Uptime for Database: " + uptime
        println "Number of Queries: " + questions
        println "Queries per Minute = " + Integer.valueOf(questions) / Integer.valueOf(uptime)

        int insertnum = 0
        int selectnum = 0
        int updatenum = 0
        sql.eachRow("show status like 'Com_%'){ status ->
            if(status.variable_name == "Com_insert"){
                insertnum = Integer.valueOf(status[1])
            }else if (status.variable_name == "Com_select"){
                selectnum = Integer.valueOf(status[1])
            }else if (status.variable_name == "Com_update"){
                updatenum = Integer.valueOf(status[1])
            }
        }
        println "% Queries Inserts = " + 100 * (insertnum / Integer.valueOf(uptime))
        println "% Queries Selects = " + 100 * (selectnum / Integer.valueOf(uptime))
        println "% Queries Updates = " + 100 * (updatenum / Integer.valueOf(uptime))
    }
}
```

Reviewing today's lesson

In this month's installment of *Practically Groovy*, you've seen how GroovySql can simplify JDBC programming. This nifty API combines closures and iterators with Groovy's relaxed syntax to facilitate rapid database application development on the Java platform. Most powerfully, GroovySql shifts resource management tasks from the developer to the underlying Groovy framework, letting you focus on the more important stuff of queries and results. But don't just take my word for it. Next time you're asked to delve into the drudgery of JDBC, try a little GroovySql magic on it instead. Then send me an e-mail and tell me about your experience.

In next month's installment of *Practically Groovy*, I'll cover the ins and outs of Groovy's template framework. As you'll discover, it's a snap to create the view component of an application with this clever framework.

Downloads

Description	Name	Size	Download method
Sample code	j-pg01115.zip	4KB	HTTP

[Information about download methods](#)

About the authors

Andrew Glover

Andrew Glover is a developer, author, speaker, and entrepreneur. He is the founder of the [easyb](#) Behavior-Driven Development (BDD) framework and is the co-author of three books: [Continuous Integration](#), [Groovy in Action](#), and [Java Testing Patterns](#). He teaches a wide variety of Groovy-, Grails-, and testing-related classes at [ThirstyHead.com](#). You can keep up with Andy at [thediscoblog.com](#) where he routinely blogs about software development.

Scott Davis

Scott Davis is an internationally recognized author, speaker, and software developer. He is the founder of [ThirstyHead.com](#), a Groovy and Grails training company. His books include [Groovy Recipes: Greasing the Wheels of Java](#), [GIS for Web Developers: Adding Where to Your Application](#), [The Google Maps API](#), and [JBoss At Work](#). He writes two ongoing article series for IBM developerWorks: [Mastering Grails](#) and [Practically Groovy](#).